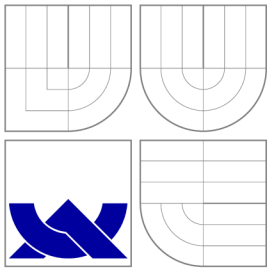
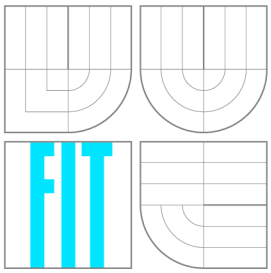


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

INKREMENTÁLNÍ NAČÍTÁNÍ DOKUMENTŮ V ZOBRAZOVACÍM STROJI HTML

INCREMENTAL DOCUMENT PARSING IN THE HTML RENDERING ENGINE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. PAVEL HRABEC

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. RADEK BURGET, Ph.D.

BRNO 2016

Zadání diplomové práce/17077/2015/xhrabe00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2015/2016

Zadání diplomové práce

Řešitel: **Hrabec Pavel, Bc.**
Obor: Informační systémy
Téma: **Inkrementální načítání dokumentů v zobrazovacím stroji HTML**
Incremental Document Parsing in the HTML Rendering Engine
Kategorie: Web

Pokyny:

1. Seznamte se s architekturou experimentálního zobrazovacího stroje CSSBox.
2. Prostudujte existující knihovny pro dekódování dokumentů HTML a jejich podporu pro zpracování dokumentu v průběhu načítání.
3. Navrhněte vhodné modifikace zobrazovacího stroje umožňující zobrazovat zpracovávaný dokument již v průběhu načítání.
4. Implementujte navržené změny v zobrazovacím stroji.
5. Implementujte samostatné moduly pro využití různých existujících dekodérů pro zpracování vstupních dokumentů.
6. Proveďte testování výsledného řešení na reálných dokumentech.
7. Zhodnoťte dosažené výsledky a navrhněte další pokračování projektu.

Literatura:

- Kocman, R.: Podpora dynamického DOM v zobrazovacím stroji HTML, Brno, FIT VUT v Brně, 2014
- Loskot, R.: Podpora JavaScriptu v zobrazovacím stroji HTML, Brno, FIT VUT v Brně, 2014
- Dokumentace projektu CSSBox: http://cssbox.sourceforge.net/documentation.php

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 3

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Burget Radek, Ing., Ph.D.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2015
Datum odevzdání: 25. května 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

Cílem této práce je prozkoumat experimentální zobrazovací stroj CSSBox, prozkoumat možnosti jeho rozšíření o inkrementální načítání dokumentů a poté navrhnout potřebné úpravy. Nejprve je uveden přehled existujících možností, po kterém následuje návrh samotného řešení. Navržené úpravy jsou implementovány, otestovány a je s nimi experimentováno. Závěr práce je věnován zhodnocení výsledků a jsou nastíněny možnosti dalšího rozvoje.

Abstract

The goal of this thesis is to explore the CSSBox experimental rendering engine, to explore the possibility of its expansion on incremental rendering of documents and then to propose the necessary modifications. The opening chapters contain an overview of existing possibilities and subsequently, the solution is proposed. The proposed changes are implemented and tested. Experiments were performed and results evaluated. The conclusion is dedicated to the evaluation of results and options for further development are outlined.

Klíčová slova

CSSBox, Java, HTML, inkrementální načítání

Keywords

CSSBox, Java, HTML, incremental parsing

Citace

HRABEC, Pavel. *Inkrementální načítání dokumentů v zobrazovacím stroji HTML*. Brno, 2016. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Burget Radek.

Inkrementální načítání dokumentů v zobrazovacím stroji HTML

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Radka Burgeta Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Pavel Hrabec
24. května 2016

Poděkování

Chtěl bych poděkovat svému vedoucímu diplomové práce Ing. Radku Burgetovi, Ph.D. za to, že mi umožnil vypracovat toto zadání, za odborné vedení, za pomoc a rady při jeho zpracování.

© Pavel Hrabec, 2016.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

Kapitola 1

Úvod

V současné době považujeme za samozřejmé využití webových prohlížečů pro přístup k nejrůznějším informacím publikovaným na webu. Proto je důležité věnovat se výzkumu a vývoji nástrojů, které pomohou lepšímu strojovému zpracování těchto informací. Prvním stupněm každého takového systému je součást schopná vykreslit daný dokument, resp. vyhodnotit jeho výslednou vizuální podobu, kterou potom předá k dalšímu zpracování. Tato oblast je také oblastí zájmu různých výzkumných projektů, jedním z nich je nástroj CSSBox, vyvinutý na FIT VUT v Brně.

CSSBox je experimentální zobrazovací stroj postavený na platformě Java, určený k analýze webového obsahu (jeho vizuální podoby). V současnosti je vhodný na dávkové zobrazování HTML dokumentů, jeho nevýhodou je však nutnost plně zpracovat vstupní dokument před tím, než je prezentován uživateli. Cílem této práce je tento stav změnit, tak aby zpracování vstupního dokumentu mohlo probíhat již během jeho načítání. V současnosti je tento problém řešen pouze webovými prohlížeči pro koncové uživatele a žádný z existujících experimentálních projektů pro platformu Java touto vlastností nedisponuje.

V kapitole 2 je popsána struktura projektu CSSBox, jehož rozšíření je cílem této práce. Je analyzován postup, který knihovna využívá během zpracování dokumentu a je prozkoumána struktura vykresleného dokumentu z pohledu vnitřních datových struktur. V podkapitole 2.4 jsou popsány konkurenční projekty řešící obdobné problémy.

Následuje popis existujících knihoven pro dekódování HTML dokumentů v kapitole 3, včetně analýzy jejich vhodnosti pro použití při inkrementálním zpracování dokumentů. Je prozkoumáno celkem pět knihoven, ze kterých se tři ukázaly být dobrými kandidáty pro využití v následujících částech práce.

V kapitole 4 následuje návrh řešení, který se zabývá jak požadavky na inkrementální syntaktickou analýzu, tak požadovanými úpravami v jádře CSSBoxu i návrhem modulů využívající různé knihovny pro dekódování HTML dokumentů.

Poté je v kapitole 5 popsána implementace navržených úprav v jádře CSSBoxu, spolu s popisem vytvořených modulů, včetně jejich výhod a případných nedostatků. Jsou diskutovány i dopady provedených úprav na stávající uživatele CSSBoxu. Dále jsou popsány nově vytvořené adaptéry, které slouží jako nástroj pro vylepšení kompatibility a demonstraci možností vytvořeného řešení.

V následující kapitole 6 se nachází popis postupu testování, popis prováděných experimentů a rozbor jejich výsledků. Experimentováno je s různými parametry výsledného řešení a je zkoumán vliv na dobu nutnou k prvnímu viditelnému vykreslení a na celkovou dobu zpracování dokumentu. Analyzovány jsou i další možnosti optimalizací CSSBoxu s ohledem na zlepšení uživatelské odezvy.

Závěrečná kapitola 7 shrnuje dosažené výsledky a jsou v ní nastíněny možnosti dalšího rozvoje projektu.

1.1 Cíle práce

Cíle této práce jsou zejména:

- Analyzovat architekturu experimentálního zobrazovacího stroje CSSBox.
- Analyzovat existující knihovny pro dekódování HTML dokumentů a jejich podporu pro zpracování během načítání.
- Navrhnout modifikace CSSBoxu tak, aby umožňoval zpracování dokumentu během jeho načítání.
- Implementovat navržené změny včetně modulů využívajících různé knihovny pro dekódování dokumentů
- Provést experimenty s implementovaným řešením, zhodnotit jejich výsledky a navrhnout možná pokračování práce

Kapitola 2

CSSBox

Cílem této práce je rozšířit experimentální zobrazovací stroj CSSBox [1], a proto je vhodné nejdříve se seznámit s architekturou tohoto nástroje. V kapitole 2.1 je popsána struktura projektu, v 2.2 postup při zpracování dokumentu, v kapitole 2.3 struktura zpracovaného dokumentu a v kapitole 2.4 ve stručnosti i ostatní zobrazovací stroje HTML pro platformu Java.

CSSBox je HTML/CSS vykreslovací stroj napsaný čistě v jazyce Java bez použití nativních knihoven. Mezi jeho hlavní cíle patří poskytování detailních a dále zpracovatelných informací o vykresleném dokumentu a jeho rozložení. Jeho výstupem je objektová reprezentace vizuální podoby stránky. Tato reprezentace je vhodná jako vstup pro další algoritmy, jako jsou algoritmy pro segmentaci nebo pro extrakci informací. Současně však může sloužit i jako komponenta v Java aplikacích postavených s využitím rámce pro tvorbu uživatelských rozhraní Swing[1], kde může sloužit jako komponenta určená k zobrazování webového obsahu uvnitř existujících desktopových aplikací. Je využíván v různých projektech, které ke své funkčnosti potřebují informace o struktuře webové stránky. Takovým projektem je např. projekt FITLayout, rámec pro segmentaci a analýzu webových stránek, který využívá CSSBox jako výchozí implementaci rozhraní pro vykreslování dokumentů. Vykreslený dokument se pak stává vstupem pro různé algoritmy provádějící analýzu vizuálních vlastností dokumentu. Mezi další využití patří například převod HTML dokumentů na obrázky ve formátech Portable Network Graphics (PNG) a Scalable Vector Graphics (SVG).

CSSBox je knihovna s otevřeným zdrojovým kódem, která je distribuována pod licencí GNU Lesser General Public License (GNU LGPL) 3. Zdrojové kódy knihovny jsou volně dostupné na serveru *github.com*[2] a binární podoba prostřednictvím serveru *sourceforge.net*[3]. Původně byla vytvořena v rámci výzkumného grantu *Výzkum informačních technologií z hlediska bezpečnosti (2007-2013, MSM)*[4]. Celá knihovna je vytvořena s využitím jazyka Java ve verzi 6, který sice nepatří mezi nejmodernější, ale zato ho podporuje široká řada platforem a nástrojů. Díky tomu, že nejsou využity žádné nativní knihovny a je využita pouze platforma Java, je zaručena kompatibilita s celou řadou operačních systému, které platforma Java podporuje.

[1] Součást Java Foundation Classes (JFC), rámce pro tvorbu grafických uživatelských rozhraní na platformě Java obsahujícího komponenty Abstract Window Toolkit (AWT), Swing a Java 2D. http://www.oracle.com/technetwork/java/faqs-140150.html

[2] https://github.com/radkovo/CSSBox/

[3] https://sourceforge.net/projects/cssbox/files/cssbox/

[4] http://www.isvav.cz/researchPlanDetail.do?rowId=MSM0021630528

2.1 Struktura projektu

Projekt se skládá z několika částí, z části pro načítání vstupních dokumentů, součásti převádějící takto získaný vstup na Document Object Model (DOM) strom, části pro zpracování Cascading Style Sheets (CSS) (využívá knihovnu *jStyleParser*), části pro výpočet rozložení dokumentu a z části pro vykreslení výsledného dokumentu.

Část pro načítání se stará o získání obsahu HyperText Markup Language (HTML) dokumentu, typicky na základě Uniform Resource Locator (URL), ale jsou možné implementace, které obsah získávají libovolným jiným způsobem, např. z databáze.

Součást pro výpočet rozložení dokumentu zajišťuje výpočet pozic všech prvků na zobrazovaném dokumentu na základě vypočtených CSS vlastností. Nejprve jsou vypočteny pozice všech elementů v relativních souřadnících, po kterém následuje výpočet rozložení v absolutních souřadnicích.

Část implementující samotné vykreslení výsledku podporuje jak vykreslení obrázku reprezentovaného Java rozhraním `Graphics2D`[5], tak generování SVG dokumentu. Všechny vykreslovací třídy implementují společné rozhraní `BoxRenderer`[6], díky kterému je možné je snadno vzájemně zaměňovat.

Součástí knihovny jsou také pomocné nástroje, které si kladou za cíl usnadnit testování. Nástroj *TestBatch* umožňuje spouštění oficiálních testovacích sad pro CSS[7] a vyhodnocování jejich úspěšnosti. Nástroj dokáže spouštět více testů současně a podporuje i přerušení testu po definovaném časovém limitu pro případ, že by došlo k zamrznutí testovaného kódu.

Spolu s knihovnou jsou distribuovány i příklady použití, jako je např. neinteraktivní prohlížeč *BoxBrowser*, nástroj pro převod HTML dokumentu na obrázek *ImageRenderer* nebo nástroj *ComputeStyles*, který vypočte efektivní CSS styly všech elementů a vypíše modifikovaný HTML dokument, který má u všech elementů v atributy `style` tyto styly uvedené.

Projekt také obsahuje několik podprojektů:

- **jStyleParser** - CSS analyzátor vytvořený čistě v jazyce Java. Je použit v CSSBoxu jako dekodér CSS, má vlastní Application programming interface (API) umožňující efektivní práci se styly a jejich reprezentaci jako Java objekty, podporuje CSS 2.1 a poměrně velkou podmnožinu CSS 3. Dokáže aplikovat zpracované styly na jednotlivé elementy DOM stromu a vypočítat efektivní styly pro tyto elementy. Pro lexikální a syntaktickou analýzu využívá známou knihovnu ANother Tool for Language Recognition (ANTLR).

- **SwingBox** - Swing komponenta navržená jako náhrada `JEditorPane`[8] tak, aby umožňovala zobrazovat obsah odpovídající moderním standardům. Pro vykreslování obsahu využívá CSSBox.

- **Pdf2Dom** - Portable Document Format (PDF) analyzátor, který umožňuje načíst PDF dokument jako DOM strom. Díky tomu je možná konverze z PDF do HTML nebo je ho možné využít jako alternativní `DOMSource` pro CSSBox a pracovat tak nad PDF dokumentem. Takový přístup může být vhodný např. v případě dalšího zpracování výstupu, jako je použití segmentačních algoritmů.

[5] `java.awt.Graphics2D`

[6] `org.fit.cssbox.render.BoxRenderer`

[7] Oficiální testovací sada pro CSS od organizace W3C si klade za cíl zlepšení interoperability různých implementací webových prohlížečů (https://www.w3.org/Style/CSS/Test/).

[8] `javax.swing.JEditorPane`

- **WebVector** - Nástroj pro konverzi HTML dokumentů na obrázky ve formátech PNG a SVG. Takové obrázky (zejména SVG) je možné dále upravovat v grafických editorech nebo je využít pro zobrazování náhledů webových stránek. Na rozdíl od jádra CSSBoxu tato komponenta vyžaduje alespoň verzi Java 7.

2.2 Postup zpracování dokumentu

Vstupem samotného zobrazovacího stroje je DOM strom. Stroj je schopný sám načíst HTML dokument, knihovna za tímto účelem poskytuje dvě abstraktní třídy `DocumentSource`[9] a `DOMSource`[10]. První jmenovaná slouží jako abstrakce nad zdrojem dokumentu, který je schopný získat jeho obsah na základě URL. Druhá třída již provádí samotnou syntaktickou analýzu dokumentu, na jejímž výstupu je objektová reprezentace v podobě DOM stromu.

Knihovna zároveň poskytuje i výchozí implementace obou popsaných tříd, a to `DefaultDocumentSource`[11], `StreamDocumentSource`[12] a `DefaultDOMSource`[13]. Výchozí implementace zdroje dokumentů využívá standardních možností platformy Java, konkrétně třídy `URLConnection`[14] obohacené o zpracování schématu data [9]. Třída `StreamDocumentSource` umožnuje využít již existující vstupní data ve formě standartního vstupního proudu jazyka Java `InputStream`[15]. Výchozí implementace zdroje DOM provádí syntaktickou analýzu pomocí HTML analyzátoru CyberNeko HTML Parser [2].

DOM strom je základem pro samotný výpočet vizuální podoby. Nejprve jsou s pomocí knihovny *jStyleParser* vypočteny efektivní styly všech elementů. Poté následuje výpočet rozložení boxů v relativních souřadnicích a závěrečným krokem je výpočet absolutních pozic všech boxů. Během zpracování je stroj schopný automaticky načíst odkazované stylopisy CSS a obrázky. Všechny tyto operace jsou blokující, takže výpočet rozvržení může proběhnout až ve chvíli, kdy jsou všechny tyto zdroje k dispozici.

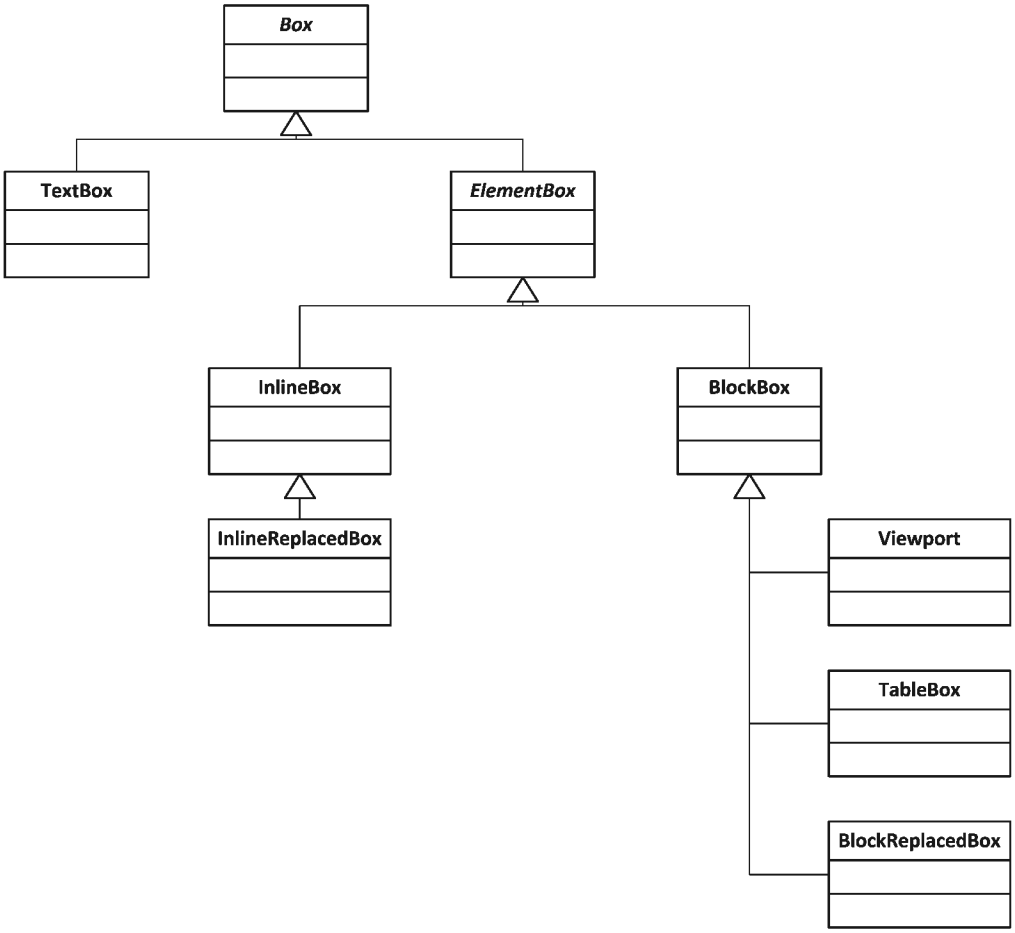
Oblast, do které je dokument vykreslován, je reprezentována pomocí instance třídy `BrowserCanvas`[16], která představuje abstrakci nad vykreslovacím plátnem prohlížeče. Zároveň zapouzdřuje informace o simulovaném výstupním zařízení (rozlišení, viditelná plocha), které jsou nutné pro podporu *CSS Media Queries*[17]. Interně třída funguje na principu vykreslování obrázku s využitím třídy `GraphicsRenderer`. Pokud není nutné vykreslovat dokument jako obrázek (např. z konzolové aplikace nebo pokud je CSSBox využit jako součást serverové aplikace), je to možné. V takovém případě aplikace získá po dokončení instanci třídy `Viewport`, se kterou může dále pracovat, např. použít pro vygenerování SVG dokumentu. Protože v současné době `BrowserCanvas` rozšiřuje třídu `JPanel`[18], je ho možné využít i ve Swing aplikacích ke snadnému zobrazení vykresleného dokumentu. Avšak tato komponenta umožňuje pouze neinteraktivní zobrazení, při potřebě interaktivního zobrazení je možné využít modul *SwingBox* popsaný výše.

[9] `org.fit.cssbox.io.DocumentSource`
[10] `org.fit.cssbox.io.DOMSource`
[11] `org.fit.cssbox.io.DefaultDocumentSource`
[12] `org.fit.cssbox.io.StreamDocumentSource`
[13] `org.fit.cssbox.io.DefaultDOMSource`
[14] `java.net.URLConnection`
[15] `java.io.InputStream`
[16] `org.fit.cssbox.layout.BrowserCanvas`
[17] Umožnují vytvářet záznamy stylopisu, které jsou podmíněně použity při jistých vlastnostech výstupního zařízení (rozměry výstupního zobrazení, rozlišení, obrazovka, tiskárna). Specifikace je součástí CSS 3 (http://www.w3.org/TR/css3-mediaqueries/).
[18] `javax.swing.JPanel`

Jak je vidět, nevýhodou tohoto přístupu je sekvenční zpracování: nejdříve je nutné celý vstupní HTML dokument analyzovat a až poté je možné přejít k výpočtu efektivních stylů atd. A právě řešení tohoto problému je cílem této práce.

2.3 Struktura vykresleného dokumentu

Základní jednotkou reprezentující část vykresleného dokumentu je *Box*. Představuje obdélníkovou oblast odpovídající konkrétnímu HTML elementu. Jeden HTML element může být reprezentován více Boxy, např. víceřádkový odstavec textu v elementu `<p>` je rozdělen na boxy odpovídající jednotlivým řádkům. Boxy tvoří strom, jehož kořenem je vždy objekt `Viewport`[19] reprezentující zobrazovací plochu prohlížeče. Tento uzel má právě jednoho potomka reprezentujícího kořen vykreslovaného dokumentu, obvykle odpovídá elementu `<body>`.



Obrázek 2.1: Zjednodušený diagram tříd (pro přehlednost je vynechána většina tříd týkajících se rozvržení tabulek).

Existuje více druhů Boxů, reprezentujících konkrétní typy obsahu, kořen hierarchie tvoří abstraktní třída `Box`[20]. Na obrázku 2.1 je zjednodušený diagram tříd celé hierarchie.

[19] `org.fit.cssbox.layout.Viewport`
[20] `org.fit.cssbox.layout.Box`

Jednotlivé typy přibližně odpovídají efektivním hodnotám CSS vlastnosti `display`. Každý box má informaci o své absolutní pozici v rámci dokumentu a o své velikosti.

Ke každému boxu je přiřazena instance třídy `VisualContext`, která slouží k uchovávání vizuálních vlastností jako jsou barva, řez písma včetně jeho velikosti, varianty a stylu (např. kurzíva). Zároveň obsahuje vypočtené hodnoty jako je počet pixelů na jeden *em*, *rem* nebo *ex*[21] a počet bodů na čtvereční palec (DPI). Dále si nese informaci o odpovídajícím uzlu DOM stromu, na jehož základě byl box vytvořen, vazbu na rodičovský box (kromě instancí třídy `Viewport`). Informace o vizuálních vlastnostech, tj. jak má nastavenou viditelnost (CSS vlastnost `visibility`), jestli je zobrazený (odpovídající CSS vlastnosti `display`). Ořezávací obdélník (angl. clipping box) určuje, jaká část boxu je zobrazená (`Viewport` nebo nejbližší rodič s CSS vlastností `overflow` nastavenou na `hidden`). Pokud box nezasahuje ani částí do svého ořezávacího obdélníku, není viditelný. V neposlední řadě si nese vypočtenou absolutní pozici na stránce.

2.4 Ostatní zobrazovací stroje

Pro platformu Java vzniklo hned několik zobrazovacích strojů umožňujících zpracování dokumentů HTML. Jejich cíle jsou sice odlišné od CSSBoxu, zaměřují se spíše na vytvoření webového prohlížeče než na tvorbu prostředku vhodného pro vědecké zkoumání. Nicméně by mohly být zajímavé s ohledem na inkrementální zpracování načítaných dokumentů, protože je to předpokládatelná vlastnost u webového prohlížeče. Cílem této části je jejich stručný popis. Tato kapitola je motivována snahou objevit existující přístup využitý pro inkrementální zpracování dokumentů a prozkoumat jestli by bylo možné ho adaptovat pro využití s CSSBoxem. Bohužel se ukázalo, že žádný z analyzovaných zobrazovacích strojů, takové chování neimplementuje.

2.4.1 Lobo, Lobo Evolution a gngr

The Lobo Project[22] vznikl jako jeden z prvních projektů zaměřujících se na vytvoření webového prohlížeče čistě na platformě Java. Jeho cílem byla podpora HTML 4, JavaScriptu a CSS 2. Aktivita projektu ustala v roce 2009 a od té doby vývoj prakticky neprobíhal.

V roce 2014 na něj navázal projekt Lobo Evolution[23], jehož součástí byla migrace ze zastaralého systému správy verzí Concurrent Version System (CVS) na Git[24] a celková modernizace zdrojového kódu. Projekt využívá nejnovější revizi jazyka Java, verzi 8. Dále se modernizovaly cíle projektu, tak aby prohlížeč podporoval moderní webové technologie jako je HTML 5 a CSS 3.

Dalším pokračovatelským projektem je gngr[25], který také vznikl v roce 2014. Provedl obdobnou modernizaci jako Lobo Evolution, ale zaměřuje se spíše na konzervativní uživatele s velkým důrazem na soukromí a bezpečnost.

Vzhledem ke stejnému původu mají i všechny tři projekty stejnou licenci, a to GNU General Public License (GNU GPL) 2. Bohužel mají také ještě jednu společnou vlastnost,

[21] Jednotky CSS definované jako: *em* - relativní velikost definovaná podle velikosti fontu, *rem* - relativní velikost definovaná podle velikosti fontu kořenového elementu, *ex* - na rozdíl od předchozích, které jsou odvozeny od šířky fontu, je tato jednotka odvozena od výšky fontu.

[22] http://www.lobobrowser.org/ - Dnes je již doména mimo provoz (prosinec 2015).

[23] http://sourceforge.net/projects/loboevolution/

[24] Git - svobodný distribuovaný systém pro správy verzí původně vytvořený Linusem Torvaldsem pro vývoj Linuxu jako náhrada za proprietární BitKeeper (https://git-scm.com/).

[25] https://gngr.info/

která je činí nezajímavými pro účely tohoto projektu, a to že nepodporují inkrementální zpracování dokumentů během jejich načítaní. Toto zjištění může být poměrně překvapivé, protože taková vlastnost je u webového prohlížeče určeného pro koncové uživatele poměrně důležitá.

2.4.2 Flying Saucer

Cílem projektu Flying Saucer[26] je vytvoření knihovny čistě v jazyce Java sloužící pro vykreslování dokumentů podle specifikací XML, XHTML a CSS 2.1. Projekt vznikl v červnu roku 2004 a je aktivně vyvíjen do dnešních dní. Je navržený tak, aby umožňoval snadné vkládání webových uživatelských rozhraní do Java aplikací, ale vzhledem k tomu, že nepodporuje HTML, není vhodný pro vytvoření obecně použitelného webového prohlížeče. Má schopnost generovat dokumenty ve formátu PDF, k čemuž využívá volně dostupnou knihovnu iText[27]. S tím také souvisí dobrá podpora pro formátování tiskových výstupů jako jsou záhlaví, zápatí nebo specifikace okrajů. Díky těmto schopnostem je často užívaný v serverových částech Java aplikacích pro generování výstupních dokumentů. Zdrojové kódy je možné distribuovat a upravovat podle podmínek licence GNU LGPL 2.1 nebo vyšší.

Vzhledem ke svému zaměření není jeho ambicí podporovat postupné zpracování dokumentu během načítání, protože potřeba takového zpracování je v případech použití, které se projekt snaží pokrýt, minimální.

[26] https://github.com/flyingsaucerproject/flyingsaucer
[27] http://itextpdf.com/

Kapitola 3

Knihovny pro dekódování dokumentů HTML

Cílem práce je přidat podporu pro inkrementální zpracování dokumentu během načítaní do CSSBoxu, proto je nezbytně nutné, aby knihovna pro dekódování podporovala takové průběžné zpracování. Vzhledem k tomu, že výstupem práce má být více vstupních modulů, které budou využívat různé knihovny. Tato kapitola některé z nich popíše a stručně zhodnotí jejich použitelnost pro implementační část práce. Pro platformu Java vznikla celá řada různých knihoven pro dekódování dokumentů HTML, které se liší jak architekturou, tak kvalitou implementace. Cílem této části je představit některé zajímavé knihovny, ne jejich vyčerpávající výčet.

3.1 CyberNeko HTML Parser

NekoHTML [2] je jednoduchý analyzátor HTML dokumentů založený na standardních rozhraních platformy Java pro práci s Extensible Markup Language (XML) dokumenty. V současnosti jej CSSBox využívá ve výchozí implementaci třídy `DocumentSource`. Dokáže kompenzovat mnoho častých chyb, kterých se dopouštějí autoři těchto dokumentů, jako jsou chybějící rodičovské elementy, chybějící uzavírající části elementů nebo špatně umístěné elementy. Implementuje jak rozhraní umožňující přímo výstup DOM, tak rozhraní umožňující proudové zpracování dokumentů Simple API for XML (SAX). Knihovna je implementována s využitím rozhraní Xerces Native Interface (XNI), které umožňuje přístup k implementačním detailům XML analyzátoru Xerces2[1]. Knihovna podporuje i velmi historické verze Javy, počínaje verzí 1.3. Projekt sice v posledních letech vykazuje minimální aktivitu, ale alespoň nějaká stále je a jsou průběžně opravovány nalezené chyby. Proto tato situace nemusí u zavedeného, široce využívaného projektu představovat problém. Knihovna je publikována pod licencí Apache 2.0.

Knihovna podporuje hned dva přístupy k inkrementálnímu zpracování. První založený na zpracování událostí během sestavování DOM stromu a druhý, nízkoúrovňovější, pomocí SAX událostí.

[1] Knihovna podporuje syntaktickou analýzu, validaci a manipulaci s XML dokumenty (https://xerces.apache.org/xerces2-j/).

3.1.1 Zpracování událostí během sestavování DOM stromu

Metoda využívá zpracování událostí generovaných třídou `DOMParser`[2]. Takové řešení je možné implementovat pomocí vytvoření třídy odvozené od třídy `DOMParser`, která přepíše metody volané při zpracování určitých elementů vstupního dokumentu. Tyto metody jsou definovány v XNI rozhraní `XMLDocumentHandler`[3]. Důležité události a metody, které jsou při jejich výskytu volány, jsou popsány v tabulce 3.1.

Metoda	Událost	Popis
`startDocument`	Začátek dokumentu	První generovaná událost, odpovídá začátku dokumentu.
`startElement`	Začátek elementu	Událost generovaná ve chvíli, kdy se na vstupu objeví začátek nového elementu.
`endElement`	Konec elementu	Událost generovaná při ukončení elementu.
`characters`	Textový obsah	Událost generovaná při výskytu volného textu na vstupu (text mezi elementy).
`endDocument`	Konec dokumentu	Poslední generovaná událost, odpovídá konci dokumentu.

Tabulka 3.1: Důležité události pro zpracování HTML dokumentů.

Na výpisu 3.1 je ukázáno využití zachytávání událostí během sestavování DOM stromu. Nejprve je otevřeno spojení ke zdroji dokumenty a vytvořena instance anonymní třídy rozšiřující třídu `DOMParser`, která při výskytu volného textu provede jeho vypsání na standardní výstup. Následně je už spuštěno samotné zpracování a je i naznačeno získání výsledného DOM stromu.

```
URLConnection connection = url.openConnection();
InputStream input = connection.getInputStream();

DOMParser parser = new DOMParser(new HTMLConfiguration()) {

    @Override
    public void characters(XMLString text, Augmentations augs) throws
        XNIException  {
        super.characters(text, augs);

        System.out.println(text.toString());
    }
};
parser.parse(input);
Document document = parser.getDocument();
```

Výpis 3.1: Příklad použití knihovny CyberNeko HTML Parser s využitím zachytávání událostí během sestavování DOM stromu.

[2] `org.cyberneko.html.parsers.DOMParser`
[3] `org.apache.xerces.xni.XMLDocumentHandler`

3.1.2 Zachytávání SAX událostí

Další možnost využívá plného přístupu založeného na SAX, její nevýhodou je nutnost ručního sestavování DOM stromu. Vzhledem k vnitřní struktuře knihovny je implementace obdobná jako u metody využívající DOM. Využívá třídy `SAXParser`[4], které je pomocí metody `setContentHandler()` nastavena instance třídy implementující SAX2 rozhraní `ContentHandler`[5]. Během zpracování jsou potom volány metody tohoto objektu, který tak může reagovat na obsah vstupního dokumentu a provádět sestavování DOM stromu. Jednotlivé metody a události, při kterých jsou tyto metody volány jsou popsány v tabulce 3.2.

Metoda	Událost	Popis
`startDocument`	Začátek dokumentu	První generovaná událost, odpovídá začátku dokumentu.
`startElement`	Začátek elementu	Událost generovaná ve chvíli, kdy se na vstupu objeví začátek nového elementu.
`endElement`	Konec elementu	Událost generovaná při ukončení elementu.
`characters`	Textový obsah	Událost generovaná při výskytu volného textu na vstupu (text mezi elementy).
`endDocument`	Konec dokumentu	Poslední generovaná událost, odpovídá konci dokumentu.

Tabulka 3.2: Metody třídy `ContentHandler` důležité pro zpracování HTML dokumentů. Mají stejný význam, ale odlišné signatury od těch uvedených v tabulce 3.1.

Výpis 3.2 ukazuje použití knihovny s využitím SAX rozhraní. Nejdříve je otevřeno spojení ke zdroji dokumentu a vytvořena instance anonymní třídy rozšiřující pomocnou třídu `DefaultHandler`[6], která vypisuje všechny texty na standardní výstup. Poté je vytvořena instance třídy `SAXParser` z balíku Xerces-J a spuštěno samotné zpracování vstupního dokumentu.

```
URLConnection connection = url.openConnection();
InputStream input = connection.getInputStream();

ContentHandler handler = new DefaultHandler() {

    @Override
    public void characters(char[] ch, int start, int length) throws
        SAXException {
        String text = new String(ch, start, length);
        System.out.println(text);
    }
};
```

[4] `org.cyberneko.html.parsers.SAXParser`

[5] `org.xml.sax.ContentHandler` - Obdobné rozhraní jako `XMLDocumentHandler`, definuje metody se stejnými názvy, avšak odlišnými parametry. V případě nutnosti je možné místo něj využít metodu `setDocumentHandler()` a rozhraní `DocumentHandler` definované specifikací SAX1, které je dnes již označeno jako zastaralé.

[6] Třída `org.xml.sax.helpers.DefaultHandler` implementuje rozhraní `EntityResolver`, `DTDHandler`, `ContentHandler` a `ErrorHandler`. Všechny implementované metody mají prázdná těla a celá třída tak slouží jako pomocná. Uživatel může přepsat pouze požadované metody a o ostatní se nestarat. Díky tomuto přístupu je také odstíněn od možných budoucích změn v implementovaných rozhraních (přidání nových metod).

```
XMLReader parser = new SAXParser(new HTMLConfiguration());
parser.setContentHandler(handler);
parser.parse(new InputSource(input));
```

Výpis 3.2: Příklad použití knihovny CyberNeko HTML Parser s využitím zachytávání SAX událostí.

3.2 Jericho HTML Parser

Jericho HTML Parser[7] je knihovna vytvořená čistě v jazyce Java, která umožňuje zpracovávat HTML dokumenty ze skutečného světa, tedy takové, které nejsou platné. Umožňuje jak zpracování celého dokumentu v jednom kroku, tak postupné, proudové zpracování pomocí vlastní třídy `StreamedSource`[8]. Podporuje verze Javy 6 a vyšší. Projekt je stále aktivní, vycházejí nové verze obsahující jak opravy chyb, tak nové vlastnosti. Knihovna je publikována hned pod třemi různými svobodnými licencemi, a to Eclipse Public License (EPL) 1.0, GNU LGPL 3.0 a Apache 2.0. Díky tomu je ji možné používat i v programech s uzavřeným zdrojovým kódem.

3.2.1 Podpora inkrementálního zpracování

Výchozím bodem pro inkrementální zpracování je třída `StreamedSource`, která implementuje standardní rozhraní `Iterable<Segment>`, kde `Segment`[9] představuje část vstupního dokumentu. Třída poskytuje několik konstruktorů, které umožňují vstupní dokument definovat pomocí URL adresy, připojení nebo přímo obsahu dokumentu. Poté už stačí získat iterátor (pomocí metody `iterator()`), pomocí kterého je možné iterovat přímo nad jednotlivými komponentami vstupu. Během iterace je možné pomocí operátoru `instanceof` jazyka Java rozlišovat mezi jednotlivými typy komponent. Nad tímto rozhraním je možné vystavět různé abstrakce jako je např. využití návrhového vzoru Návštěvník (Visitor). Význam jednotlivých tříd je uveden v tabulce 3.3.

Třída	**Význam**	**Popis**
`StartTag`	Začátek elementu	Třída reprezentující začátek nového elementu.
`EndTag`	Konec elementu	Třída reprezentující ukončení elementu.
`CharacterReference`	Reference SGML	Reference na znak, popř. entitu definovanou specifikací SGML (jako jsou např. ` `, `&#x263a;` apod.).
`Attribute`	Atribut	Třída reprezentující atribut HTML elementu.
`Segment`	Ostatní obsah	Ostatní nerozlišený obsah, v praxi odpovídá výskytu volného textu na vstupu (text mezi elementy).

Tabulka 3.3: Důležité třídy rozšiřující třídu `Segment`.

Na výpisu 3.3 je ukázka, která vypíše všechny texty obsažené v dokumentu na standardní výstup. Nejprve je vytvořena instance třídy `StreamedSource`, přes kterou je následně

[7] http://jericho.htmlparser.net/docs/index.html
[8] `net.htmlparser.jericho.StreamedSource`
[9] `net.htmlparser.jericho.Segment`

iterováno dle popisu výše. V tomto případě je využit operátor `instanceof` pro přeskočení všech HTML značek.

```
StreamedSource source = new StreamedSource(url);
for (Segment segment : source) {
    if (segment instanceof Tag)
        continue;

    System.out.println(segment.toString());
}
```

Výpis 3.3: Příklad použití knihovny Jericho HTML Parser.

3.3 jsoup

Knihovna jsoup[10] je schopná pracovat s širokou škálou různě poškozených (špatně vytvořených) HTML dokumentů. Cílem autorů je vytvořit knihovnu, která vždy vrátí v rámci možností smysluplný výstup, nikdy chybu. Podporované verze Javy jsou 5 a vyšší. Její velkou výhodou je, že implementované chování odpovídá algoritmu pro zpracování HTML, popsanému ve specifikaci HTML5 [6]. Díky tomu produkuje stejný DOM strom jako moderní webové prohlížeče. Poskytuje jednoduché moderní API, které umožňuje načtení DOM stromu, vyhledávání v něm za pomoci vyhodnocování CSS selektorů, manipulaci se stromem a jeho čištění od nežádoucích konstrukcí (např. jako ochrana před Cross-site scripting (XSS)). Příklad použití knihovny pro načtení dokumentu, extrakci veškerého textu a jeho výpis na standardní výstup je vidět na výpisu 3.4.

```
Document document = Jsoup.connect("http://www.fit.vutbr.cz").get();
String text = document.body().text();
System.out.println(text);
```

Výpis 3.4: Příklad použití knihovny jsoup.

Knihovnu je stále vyvíjena a je publikována pod podmínkami liberální licence MIT[11]. Bohužel však, jako jedinou možnost zpracování nabízí zpracování celého dokumentu v jednom kroku, což ji činní pro tuto práci nepoužitelnou.

3.4 HTML Parser

HTML Parser[12] je knihovna, která umožňuje rychlé zpracování HTML dokumentů pomocí postupného načítání zdroje a postupného zpracovávání. Umožnuje také zpracovat celý dokument v jednom kroku, avšak tato funkce je implementována s využitím inkrementálního zpracování a vytvoření výsledného seznamu, který je vrácen uživateli. Podporuje také standardní rozhraní SAX. Podporovány jsou verze Javy 5 a novější. Projekt bohužel nevykazuje od konce dubna 2011 žádnou aktivitu, což by mohl být problém v případě nalezení nějakých zásadních problémů (chyb) v implementační části práce. Knihovna podléhá podmínkám licence Common Public License (CPL) 1.0[13].

[10] http://jsoup.org/
[11] The MIT License - http://jsoup.org/license
[12] http://htmlparser.sourceforge.net/
[13] Svobodná licence vytvořená společností IBM (http://htmlparser.sourceforge.net/license.html)

3.4.1 Podpora inkrementálního zpracování

Každé zpracování začíná vytvořením instance třídy `Parser`[14]. Konstruktor přijímá právě jeden argument typu `String` - ten je interpretován jako obsah dokumentu nebo jako URL na základě obsahu, `URLConnection` sloužící pro získání obsahu dokumentu nebo vlastní třídu reprezentující zdroj lexémů (lexikální analyzátor) rozšiřující třídu `Lexer`. Následně je možné využívat jednotlivé metody třídy `Parser` jako jsou `elements()` a `visitAllNodesWith()`.

První jmenovaná vrátí instanci třídy implementující rozhraní `NodeIterator`[15], která svému názvu navzdory nerozšiřuje standardní rozhraní `Iterator<T>`, ale specifikuje vlastní metody `hasMoreNodes()` a `nextNode()` se zřejmým významem. Ukázka využití tohoto způsobu je na výpisu 3.5. Ukázka demonstruje průchod celým dokumentem a výpis všech textů obsažených v dokumentu na standardní výstup.

```
Parser parser = new Parser("http://www.fit.vutbr.cz");
for (NodeIterator iterator = parser.elements(); iterator.hasMoreNodes();) {
    Node node = iterator.nextNode();
    String text =
        parser.getLexer().getPage().getText(node.getStartPosition(),
        node.getEndPosition());
    System.out.println(text);
}
```

Výpis 3.5: Příklad použití knihovny HTML Parser.

Druhý postup vytváří abstrakci nad prvním s využitím návrhového vzoru Visitor. Metodě `visitAllNodesWith()` je předána instance třídy rozšiřující abstraktní třídu `NodeVisitor`[16]. Metody volané pro jednotlivé události jsou popsány v tabulce 3.4.

Metoda	Událost	Popis
`beginParsing`	Začátek zpracování	První generovaná událost, vyvolaná na začátku zpracování, odpovídá začátku dokumentu.
`visitTag`	Začátek elementu	Událost generovaná ve chvíli, kdy se na vstupu objeví začátek nového elementu.
`visitEndTag`	Konec elementu	Událost generovaná při ukončení elementu.
`visitStringNode`	Textový obsah	Událost generovaná při výskytu volného textu na vstupu (text mezi elementy).
`finishedParsing`	Konec zpracování	Poslední generovaná událost, vyvolaná při ukončení zpracování, odpovídá konci dokumentu.

Tabulka 3.4: Metody volané během zpracování dokumentu

Oba popsané postupy využívají nízkoúrovňový přístup a nemusí být tedy úplně vhodné, avšak knihovna také nabízí implementaci SAX rozhraní `XMLReader`, která zapouzdřuje výše popsané operace.

[14] `org.htmlparser.Parser`
[15] `org.htmlparser.util.NodeIterator`
[16] `org.htmlparser.visitors.NodeVisitor`

3.5 javax.swing.text.html.parser.DocumentParser

Třída[17], jenž je přímo součástí Java Development Kit (JDK) (konkrétně rámce pro tvorbu grafických aplikací JFC). Součástí Javy je již od verze 1.2 vydané v prosinci roku 1998. Třída umožňuje během zpracování zachytávat události a reagovat na ně. K zachytávání událostí je potřeba vytvořit třídu rozšiřující třídu `ParserCallback`[18] a přepsat poskytované metody. Mapování mezi událostmi a metodami této třídy je uvedeno v tabulce 3.5[19].

Metoda	Událost	Popis
`handleStartTag`	Začátek elementu	Událost generovaná ve chvíli, kdy se na vstupu objeví začátek nového elementu.
`handleEndTag`	Konec elementu	Událost generovaná ve chvíli, kdy se na vstupu objeví začátek nového elementu a značka je zároveň ukončující, v takovém případě není generována událost `handleEndTag`.
`handleSimpleTag`	Jednoduchý element	Událost generovaná při ukončení elementu.
`handleText`	Textový obsah	Událost generovaná při výskytu volného textu na vstupu (text mezi elementy).

Tabulka 3.5: Metody volané během zpracování dokumentu

Díky této podpoře je možné třídu využít v aplikacích vyžadující inkrementální zpracování HTML dokumentů. Výpis 3.6 demonstruje využití třídy pro výpis všech textů ve vstupním dokumentu na standardní výstup. Nejprve je otevřeno spojení ke zdroji dokumentu a vytvořena instance anonymní třídy rozšiřující třídu `ParserCallback`. V ukázce tato třída implementuje chování, které vypisuje všechny textová data na standardní výstup. Následuje získání předdefinované instance třídy `DTD` pro HTML 3.2 a spuštění samotného zpracování. Poslední argument `false` značí, že se nemá ignorovat případná znaková sada definovaná v dokumentu.

```
URLConnection connection = url.openConnection();
InputStream input = connection.getInputStream();
Reader reader = new InputStreamReader(input);

ParserCallback callback = new ParserCallback() {

    @Override
    public void handleText(char[] data, int pos) {
        System.out.println(data);
    }
};

DTD dtd = DTD.getDTD("html32");
DocumentParser parser = new DocumentParser(dtd);
```

[17] https://docs.oracle.com/javase/8/docs/api/javax/swing/text/html/parser/DocumentParser.html

[18] `javax.swing.text.html.HTMLEditorKit.ParserCallback`

[19] Některé méně významné metody nejsou pro přehlednost uvedeny.

```
parser.parse(reader, callback, false);
```

Výpis 3.6: Příklad použití třídy `DocumentParser`.

JDK bohužel ve výchozím stavu podporuje pouze HTML 3.2. Podporu novějších verzí je možné doplnit pomocí externích Document Type Definition (DTD), což ale taky znamená, že není možné podporovat HTML 5. Díky poměrně dobrému návrhu by sice bylo možné nahradit zabudovaný syntaktický analyzátor, ale potom by poněkud ztrácelo využití této třídy smysl, protože v takovém případě by bylo vhodnější využít náhradu přímo. Z tohoto důvodu se tato třída jeví jako nevhodná pro účely této práce.

Kapitola 4

Návrh modifikací zobrazovací stroje

Uvažované modifikace zobrazovacího stroje vyžadují změny v jádře CSSBoxu. V současnosti zpracování probíhá sekvenčně: nejprve dojde k syntaktické analýze vstupního dokumentu, následuje výpočet stylů, rozvržení dokumentu a samotné vykreslení výsledku. Návrh předpokládá změny, které povedou k provádění všech částí tohoto procesu současně. V kapitolách 4.1, 4.2, 4.3 a 4.4 jsou analyzovány nutné úpravy jednotlivých kroků a nakonec v kapitole 4.5 popsány konkrétní navržené modifikace.

4.1 Inkrementální syntaktická analýza

Aby bylo možné provádět zpracování dokumentu už během načítání, je nutné, aby měl prohlížeč přístup k částem analyzovaného dokumentu už během jeho načítání. V současnosti tomu tak ale není, dokument je analyzován celý a je postoupen do další části jako instance třídy implementující rozhraní `Document`[1]. Nabízejí se dvě možnosti jak tomuto problému čelit.

První možností je přechod na plně proudové zpracování založené na SAX. Výhodou tohoto přístupu je menší paměťová náročnost a vyšší výkon zpracování. Zásadní nevýhodou tohoto přístupu by ale byla nutnost provést masivní změny v jádře CSSBoxu, proto byla zvolena druhá cesta.

Druhá možnost je ponechat současnou reprezentaci založenou na DOM a implementovat zpracování událostí z průběhu zpracování. Tato možnost byla zvolena, aby bylo možné projekt dokončit s minimálními dopady na zpětnou kompatibilitu. Analýza dokumentu bude ve své podstatě probíhat proudově, takže bude možné reagovat na události, ale současně bude probíhat výstavba DOM stromu.

Budou implementovány moduly využívající různé knihovny pro dekódování HTML dokumentů, konkrétně byly zvoleny tyto:

- **CyberNeko HTML Parser** - Zvoleno z důvodu zpětné kompatibility se stávající výchozí implementací. Bude využito stejné rozhraní jako doposud, tedy DOM v kombinaci se zpracováním událostí během sestavování tohoto dokumentu. Podrobnější popis v kapitole 3.1.

[1] `org.w3c.dom.Document`

- **Jericho HTML Parser** - Nepodporuje žádná standartní rozhraní, proto se jedná o vhodného kandidáta pro demonstraci nezávislosti navržených úprav na použitých knihovnách. Podrobnější popis v kapitole 3.2.
- **HTML Parser** - Podporuje pouze rozhraní SAX a vlastní, tvoří tedy vhodný protipól řešení založenému na DOM. Podrobnější popis v kapitole 3.4.

Jde tedy o všechny popisované knihovny v kapitole 3, které mají nějakou podporu pro zpracování již během načítání dokumentu.

4.2 Přepočet stylů

V případě, že během načítání dokumentu dojde k přidání dalšího stylopisu, je nutné přepočítat efektivní styly pro všechny elementy[2]. V případě CSSBoxu znamená pouze zavolání metody `recomputeStyles()` třídy `DOMAnalyzer`. Takový postup ale způsobí přepočet stylů pro celý dokument, což nemusí být ideální. Lepší by mohlo být vypočítat změny pouze pro přidané elementy a jejich sousedy, stejně jako to implementují moderní prohlížeče. Avšak pro provedení takových úprav by bylo nutné provést i změny v knihovně jStyleParser, což je mimo záběr této práce. Jedná se o oblast, která se i ve světě velkých moderních prohlížečů stále vyvíjí a např. jádro Blink[3] ještě v roce 2014 [10] při každé změně elementu `<body>` provádělo přepočet celého dokumentu, a to jde o moderní pokročilé jádro, které je využíváno v prohlížečích Chrome a Opera.

4.3 Přepočet rozvržení dokumentu

Při některých změnách dokumentu je nutné provést přepočet rozložení dokumentu (angl. layout). Typickým příkladem je změna DOM stromu (přidání nebo odebrání uzlů) nebo změna efektivních stylů. Vzhledem k tomu, že by nebylo příliš efektivní provádět přepočet rozvržení při každé změně dokumentu, bude implementován stejný postup, jaký provádějí moderní prohlížeče. Ten spočívá ve shlukování změn do skupin, které jsou asynchronně zpracovávány jednou za čas pomocí odložených úloh naplánovaných časovačem. Hodnota délky časového intervalu, ve kterém bude překreslování probíhat, bude stanovena až během experimentů v implementační fázi práce, ale dá se předpokládat, že bude nabývat hodnot v řádu stovek milisekund. Taková délka intervalu je dostatečně velká, aby snížila zátěž, a přitom dostatečně malá, aby minimalizovala latenci z pohledu uživatele.

Moderní prohlížeče umožnují i synchronní přepočet v případě, že kód JavaScriptu se snaží přistoupit k vlastnostem rozvržení dokumentu[4] např. voláním `window.getComputedStyle()` nebo např. v případě změny velikosti okna prohlížeče. V případě navržených úprav CSSBoxu bude probíhat při dokončení načítání celého dokumentu.

[2] CSSBox neuvažuje atribut `scoped` elementu `<style>` (https://html.spec.whatwg.org/multipage/semantics.html#attr-style-scoped)

[3] https://www.chromium.org/blink

[4] JavaScript sice není CSSBoxem podporován, ale podpora této vlastnosti by mohla být vhodná pro projekty, jako jsou ScriptBox [8] (https://github.com/ITman1/ScriptBox) nebo JSDOMBox [7] (https://github.com/Gals42/JSDOMBox).

4.4 Externí zdroje

HTML dokumenty obsahují také externí zdroje, které je potřeba uvažovat, např. obrázky, webové fonty, CSS stylopisy, JavaScriptové soubory neoznačené atributem `async`[5].

V případě CSS stylopisů (pokud jsou v hlavičce) může být účelné počkat na jejich načtení před začátkem vykreslování výsledku. Je to způsobeno tím, že jejich absence by stejně vynutila přepočet stylů i rozvržení dokumentu.

Načítání obrázků by nemělo blokovat proces zobrazení celého dokumentu, proto bude načítání obrázků probíhat asynchronně a po jejich načtení dojde k přepočtu rozvržení dokumentu[6].

4.5 Nejvýznamnější změny jádra CSSBoxu

Pro podporu inkrementálního zpracování bude nutné provést i úpravy v jádru CSSBoxu, tak aby výše popsané operace bylo možné provádět najednou, na rozdíl od současného přístupu, který využívá sekvenčního zpracování.

Bude zavedeno nové rozhraní `StreamingSource`, které bude specifikovat jedinou metodu `void parse(ContentHandler)`. Bude tedy sloužit jako abstrakce nad různými implementaci dekodérů HTML dokumentů. Rozhraní `ContentHandler` pochází ze specifikace SAX a význam jednotlivých metod specifikovaných tímto rozhraním je uveden v tabulce 3.2. Výhodou tohoto řešení je to, že nebude mít dopad na existující uživatele CSSBoxu.

Aby bylo možné sloučit všechny předchozí kroky, bude nutné vytvořit novou obalující třídu `DocumentProcessor`, která bude řídit celý proces načítání a vykreslování dokumentu. Konstruktor třídy přijímá tři parametry, a to instanci třídy `URL` reprezentující základní adresu dokumentu, vůči které jsou vyhodnocovány relativní odkazy, instanci třídy rozšiřující třídu `DocumentSource` a instanci třídy `BrowserCanvas`[7]. Třída zároveň bude definovat metodu `Document process()`, která spustí zpracování dokumentu, blokuje další vykonávání volajícího kódu, dokud není celé zpracování dokončeno a následně vrátí finální podobu DOM stromu. Interně bude třída udržovat veškerou nutnou infrastrukturu pro implementaci níže popsaného algoritmu, jako jsou časovače a pracovní vlákna, a bude zajišťovat nutnou synchronizaci pro udržení korektnosti a bezpečnosti více vláknového zpracování.

Postup zpracování dokumentu třídou `DocumentProcessor` bude následující:

1. Získání vstupního proudu pomocí metody `getInputStream()` dodaného zdroje dokumentu.

2. Předání získaného vstupního proudu dodanému zdroji DOM stromu pomocí volání metody `parse()`, což zahájí samotné získávání a dekódování dokumentu.

3. Při volání některé z popsaných metod:

   (a) Zpracování dokumentu ještě nebylo dokončeno, ale již byla zpracována část dokumentu až po otevírající značku `<body>` a nastane některá z těchto událostí:

      i. Uplynula definovaná doba od posledního překreslení.

      ii. Bylo dokončeno zpracování dokumentu.

[5] CSSBox JavaScript ani webové fonty prozatím nepodporuje, takže tuto situaci návrh neřeší.

[6] Pokud element `<img>` nemá uvedené rozměry pomocí atributů `width` a `height`.

[7] Třídu `BrowserCanvas` bude nutné upravit tak, aby ve svém konstruktoru nevyžadovala instanci třídy `DOMAnalyzer` a kořen DOM stromu. Oba tyto objekty budou vloženy později třídou `DocumentProcessor`.

nebo

(b) Zpracování dokumentu již bylo dokončeno a nastane některá z těchto událostí:

i. Dojde k načtení externího zdroje podle popisu v kapitole 4.4.

Dojde k:

(a) Přepočtu stylů pomocí volání metod na instanci třídy `DOMAnalyzer`:

i. Konverzi atributů na CSS styly voláním metody `attributesToStyles()`.

ii. Přidání nově načtených stylopisů CSS (pokud nějaké jsou) voláním metody `addStyleSheet()`, pro každý takový stylopis.

iii. Vlastní přepočet stylů pomocí volání metod `recomputeStyles()` a `getStyleSheets()`.

(b) Výpočtu rozvržení dokumentu v relativních pozicích, výpočtu pozic elementů v absolutních souřadnicích a finálnímu vykreslení pomocí volání metody `createLayout()` na instanci třídy `BrowserCanvas`.

4. V okamžiku, kdy je dokončeno zpracování vstupního dokumenty, jsou zpracovány všechny externí zdroje, je provádění asynchronního načítání dokumentu úspěšně dokončeno.

Pokud během zpracování dojde k chybě, ze které se není možné zotavit, je proces zpracování zastaven s chybou, tj. vyvoláním výjimky.

Kapitola 5

Implementace

Součástí práce bylo i provedení nutných úprav v jádru CSSBoxu popsaných v kapitole 4 a vytvoření vstupních modulů s využitím knihoven popsaných v kapitole 3. Tato kapitola se zabývá vysvětlením podrobností implementace provedených úprav. Implementace se sestávala z několika fází, nejprve byly provedeny potřebné úpravy v jádru CSSBoxu (kapitola 5.1). Poté následovala implementace modulů využívající různé knihovny pro zpracování HTML dokumentů (kapitola 5.2). V kapitole 5.3 jsou shrnuty nástroje využité při implementaci.

5.1 Úpravy CSSBoxu

Pro podporu inkrementálního načítání dokumentů bylo nutné provést několik úprav v jádře CSSBoxu. Mezi nejvýznamnější změny patří zavedení nového rozhraní `StreamingSource`, které slouží jako adaptér umožňující využít různé knihovny pro dekódování HTML dokumentů. Dalším významným dodatkem je třída `DocumentProcessor`, jenž implementuje nutnou logiku pro provádění vykreslování dokumentu. Dále bylo nutné doplnit možnost asynchronního vykonávání úloh do potomků třídy `Box`, typickým příkladem je třída `ContentImage` reprezentující element `<img>` ve zdrojovém dokumentu, která asynchronní zpracování využívá k podpoře neblokujícího načítání obrázků.

Rozhraní `StreamingSource`[1] reprezentuje ekvivalent třídy `DOMSource` umožňující využít možností inkrementálního zpracování vstupních HTML dokumentů. S jeho pomocí je možné implementovat moduly využívající různé knihovny pro syntaktickou analýzu a zároveň se vyhnout úpravám jádra specifickým pro jednotlivé knihovny. Rozhraní specifikuje jedinou metodu `parse()`. Metoda přijímá jediný argument, a to instanci třídy implementující rozhraní `ContentHandler` specifikované v rozhraní SAX. Implementace metody potom během zpracování vstupního dokumentu provádí zpětná volání pomocí metod specifikovaných tímto rozhraním.

Výpis 5.1 ukazuje strukturu tohoto rozhraní. Metoda `parse()` deklaruje dvě výjimky. První je `SAXException`[2] a slouží k informování volajícího, že během analýzy vstupního dokumentu došlo k fatální chybě kvůli které nelze pokračovat (např. vnitřní chyba implementace). Druhá výjimka `IOException` indikuje chybu při získávání obsahu dokumentu, který se má zpracovávat (např. přerušené síťové spojení, nedostatečná oprávnění pro čtení z pevného disku apod.). V klasických implementacích rozhraní budou sloužit k propagaci této výjimky z volání metody `getInputStream()` třídy `DocumentSource` nebo z následných volání metody

[1] `org.fit.cssbox.io.StreamingSource`
[2] `org.xml.sax.SAXException`

`read()` na vrácené instanci třídy implementující rozhraní `InputStream`, prováděných během analýzy dokumentu.

```
public interface StreamingSource {

    void parse(ContentHandler contentHandler) throws SAXException,
        IOException;
}
```

Výpis 5.1: Rozhraní `StreamingSource`

Třída `DocumentProcessor`[3] implementuje samotný algoritmus inkrementálního zpracování navržený v kapitole 4.5. Během konstrukce instance třídy jsou připravena pracovní vlákna, která později, během zpracování zajišťují načítání externích zdrojů a provádění opakované úlohy popsané dále. Jako výchozí počet vláken bylo zvoleno číslo pět a s touto hodnotou je experimentováno v kapitole 6. Při zahájení zpracování dojde k naplánování periodicky se opakující úlohy zajišťující výpočet aktualizovaného rozležení a vykreslení výsledku. Výchozí hodnota pro periodu byla zvolena 200 ms a experimenty s touto hodnou se opět zabývá kapitola 6.

Následuje volání implementace rozhraní `StreamingSource` pro zahájení načítání a analýzu vstupního dokumentu. Jako parametr volání je předána speciální implementace rozhraní `ContentHandler`[4], která z generovaných událostí vytváří odpovídající instance tříd implementujících interní rozhraní `ContentEvent`[5]. Toto rozhraní specifikuje jedinou metodu `visit(ContentEventVisitor visitor)`. Další interní rozhraní `ContentEventVisitor`[6] obsahuje jednu přetíženou metodu `visit()` pro každou konkrétní implementaci rozhraní `ContentEvent`. Spolu tak tyto dvě rozhraní tvoří implementaci návrhového vzoru Návštěvník (Visitor) [4]. V tabulce 5.1 je uvedeno mapování mezi událostmi SAX a konkrétními třídami využitými pro jejich vnitřní reprezentaci. Všechny uváděné třídy jsou umístěny v balíku `org.fit.cssbox.incremental`. Význam jednotlivých událostí SAX je popsán v tabulce 3.2. Události vkládá do fronty, ze které jsou asynchronně odebírány periodickou úlohou a tvoří tak implementaci návrhového vzoru Producent–konzument (Producer—consumer) [11].

Událost SAX	**Reprezentující třída**
`startElement`	`ElementStart`
`endElement`	`ElementEnd`
`characters`	`Characters`
`startDocument`	`DocumentStart`
`endDocument`	`DocumentEnd`

Tabulka 5.1: Mapování událostí SAX na vnitřní třídy prováděné třídou `ParserEventHandler`.

Při každém opakování periodické úlohy dochází k vyprázdnění fronty událostí a aktualizaci pracovního DOM stromu, odpovídajícím přepočtům efektivních stylů, relativních / absolutních souřadnic a samotnému vykreslení. Úloha je reprezentována vnitřní třídou `RenderTask`[7], která implementuje rozhraní `Runnable`, aby bylo možné úlohu spouštět asyn-

[3] `org.fit.cssbox.incremental.DocumentProcessor`

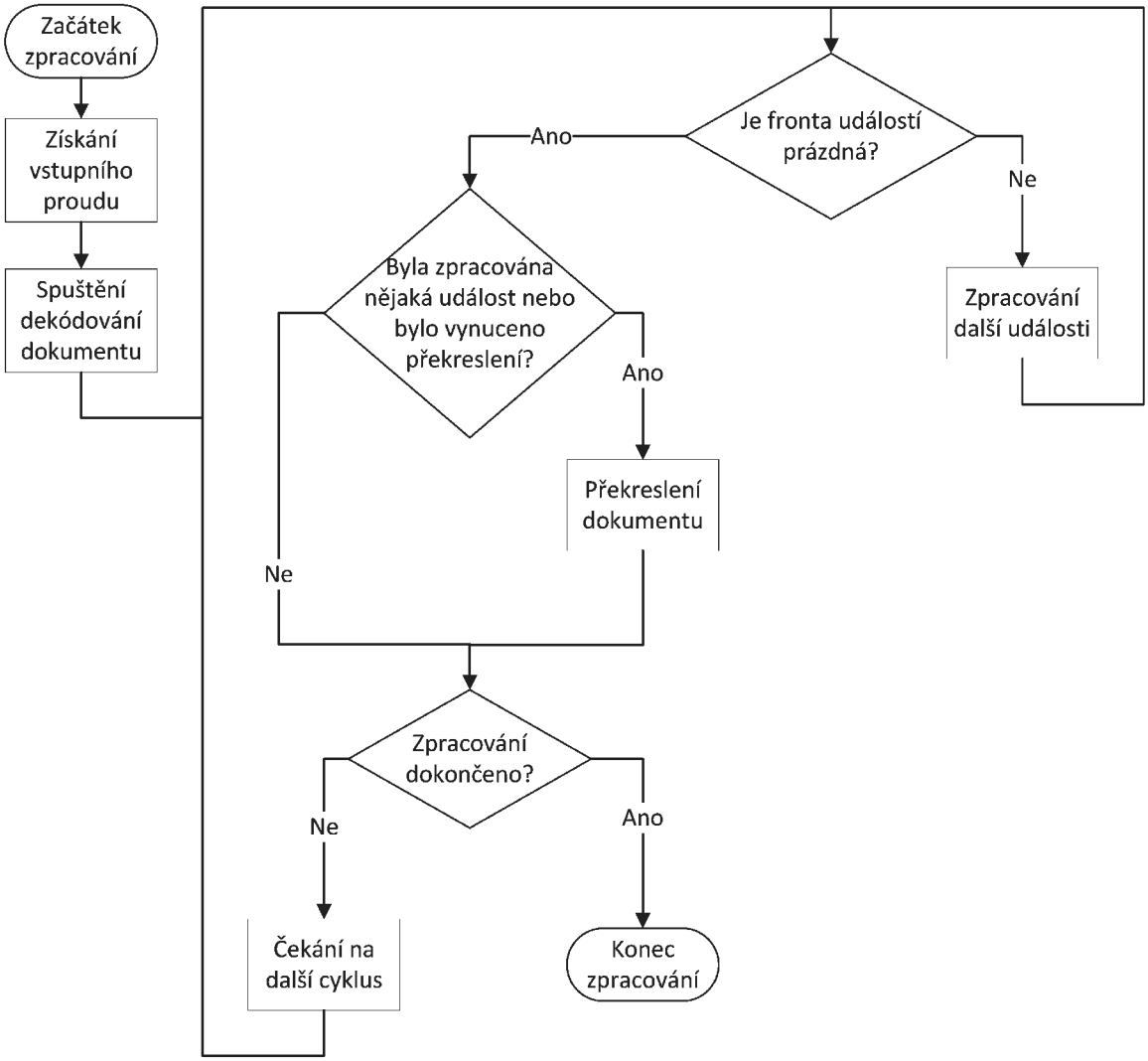
[4] Iterní třída `org.fit.cssbox.incremental.ParserEventHandler`.

[5] `org.fit.cssbox.incremental.ContentEvent`

[6] `org.fit.cssbox.incremental.ContentEventVisitor`

[7] Třída `org.fit.cssbox.incremental.DocumentProcessor.RenderTask` je vnitřní třídou třídy `DocumentProcessor`.

chronně pomocí tříd implementující rozhraní `Executor`[8] a zároveň implementuje rozhraní `ContentEventVisitor` popsané výše. Celý proces zpracování dokumentu je schématicky naznačen na obrázku 5.1.



Obrázek 5.1: Proces vykresolování řízený třídou `DocumentProcessor`.

Na výpisu 5.2 jsou nejdůležitější veřejné metody definované třídou `DocumentProcessor`. Konstruktor vyžaduje tři parametry. Prvním je základní URL, která slouží pro převod relativních odkazů na absolutní[9], který je prováděn třídou `DOMAnalyzer`. Druhým parametrem je instance třídy implementující rozhraní `StreamingSource`, která, jak bylo popsáno výše, slouží ke komunikace s odpovídajícím analyzátorem HTML dokumentu. Poslední parametr je instance třídy `BrowserCanvas` sloužící k vykreslování výsledného dokumentu.

Volání metody `process()` zahajuje zpracování dokumentu podle algoritmu popsaného v kapitole 4.5. Před samotným provedením algoritmu zajistí vytvoření všech potřebných pracovních vláken a naplánuje periodickou úlohy v nastaveném intervalu. Práce s pracovními

[8] `java.util.concurrent.Executor`

[9] Základní URL není využita, pokud HTML dokument obsahuje v hlavičce element `<base>`, který v takovém případě dostane přednost.

vlákny je zapouzdřena ve třídě `TaskExecutor`[10], která udržuje jedno vyčleněné vlákno pro provádění periodické úlohy a definovaný počet pracovních vláken pro využití potomky třídy `Box`. Po ukončení zpracování jsou všechna pracovní vlákna ukončena a zdroje uvolněny.

Metoda `requestRender()` slouží k vynucení překreslení dokumentu během dalšího vykreslovacího cyklu, přestože nedošlo od posledního cyklu k modifikaci DOM stromu. Poslední významná metoda `scheduleTaskWithRender()` slouží k provedení úlohy asynchronně a k provedení překreslení dokumentu po jejím dokončení – ekvivalentně k provedení úlohy a následnému volání metody `requestRender()`.

Třída dále obsahuje metody `set` a `get` pro nastavení a získání počtu pracovních vláken, zpoždění před prvním spuštěním periodické úlohy a nastavení a získání intervalu jejího opakování.

```
public class DocumentProcessor {

    public DocumentProcessor(URL baseUrl, StreamingSource domSource,
        BrowserCanvas browserCanvas);

    public Document process() throws IOException;

    public void requestRender();

    public <T> Future<T> scheduleTaskWithRender(Callable<T> task);

    ...
}
```

Výpis 5.2: Rozhraní třídy `DocumentProcessor`

V případě, že některý z potomků abstraktní třídy `Box` vyžaduje provedení asynchronní akce, jako je např. načtení externích zdrojů, předá tento požadavek třídě `DocumentProcessor` pomocí volání metody `scheduleTaskWithRender()`. Metoda zajistí provedení úlohy některým volným pracovním vláknem a po jejím dokončení naplánuje překreslení zobrazeného dokumentu tak, aby byly reflektovány právě provedené změny. Pro tyto účely byla rozšířena třída `BrowserCanvas` o metodu `getProcessor()` umožňující přístup k právě využívané instanci třídy `DocumentProcessor`.

Dále byla vytvořena pomocná třída `CachedNetworkProcessor`[11], která implementuje návrhový vzor dekorátor [4] a obaluje existující implementace rozhraní `NetworkProcessor`[12] z projektu jStyleParser. Cílem takového obalení je, aby nedocházelo k opakovanému přenosu CSS stylopisů při přepočtu stylů na stránce a vede tak ke zvýšení celkového výkonu. Implementace pracuje tak, že si udržuje vnitřní mapování mezi adresami URL jednotlivých stylopisů a jejich skutečným obsahem. V případě, že přijde požadavek, pro který ještě neexistuje záznam, je tento požadavek delegován na obalenou instanci třídy `NetworkProcessor` a uložen do vyrovnávací paměti. V opačném případě je okamžitě vrácen jeho obsah bez nutnosti síťových akcí. Stejně jako již existující třída `ImageCache` z CSSBoxu, ani tato třída neimplementuje podporu možností protokolu HTTP pro správu vyrovnávacích pamětí podle standardu RFC 2616 [3].

[10] `org.fit.cssbox.incremental.TaskExecutor`
[11] `org.fit.cssbox.css.CachedNetworkProcessor`
[12] `cz.vutbr.web.css.NetworkProcessor`

Vzhledem k tomu, že překreslování lze v AWT aplikacích (což jsou i Swing aplikace) provádět bezpečně pouze v hlavním vlákně, tj. vlákně zpracovávajícího události AWT, byla třída `BrowserCanvas` upravena tak, aby bylo možné bezpečně volat metodu `createLayout()` i z jiných vláken než z hlavního aplikačního vlákna. Toho bylo dosaženo přidáním detekce, zda je aktuální vlákno hlavní a pokud ne, je veškeré překreslování prováděno pomocí metody `invokeAndWait()` třídy `EventQueue`[13].

Výpis 5.3 demonstruje použití nového rozhraní pro spuštění inkrementálního zpracování. Je zde využit modul využívající knihovnu CyberNeko, interval zpracování je nastaven na 200 ms a příklad využívá 5 pracovních vláken. Další možnosti pro výběr implementace rozhraní `StreamingSource` jsou `JerichoStreamingSource` a `HtmlParserStreamingSource`, tyto knihovny ovšem mají problematickou funkčnost popsanou v kapitolách 5.2 a 6.1.

```
DocumentSource source = new DefaultDocumentSource(url);
StreamingSource parser = new CyberNekoStreamingSource(source);
BrowserCanvas canvas = new BrowserCanvas(url);

DocumentProcessor processor = new DocumentProcessor(url, parser, canvas);
// volitelná nastavení
processor.setQueueProcessDelay(200);
processor.setQueueProcessPeriod(200);
processor.setThreadCount(5);

processor.process();
```

Výpis 5.3: Příklad použití inkrementálního zpracování

5.1.1 Kompatibilita

Při implementaci navržených úprav byl kladen důraz na maximální zpětnou kompatibilitu, tak aby provedené změny neměly dopad na existující uživatele CSSBoxu, jako je např. FitLayout[14]. Vzhledem k tomu, že nedošlo k žádným zpětně nekompatibilním úpravám částí, které provádějí samotné vykreslování, ani k zásahům do částí, které provádějí zpracování dokumentu v jednom kroku, neměly by mít provedené úpravy vliv na uživatele. Tento cíl se tedy podařilo splnit. Úpravy byly provedeny tak, že novou třídu `DocumentProcessor` není nutné využívat a všechny upravené třídy, které využívají jejích služeb to provádějí na základě detekce, zda je aktuálně využívána[15]. Např. v případě úprav abstraktní třídy `ContentImage` to znamená, že pokud třída `DocumentProcessor` není aktuálně využívána, provede se načtení obrázku klasickým – blokujícím způsobem.

5.1.2 Demonstrační aplikace

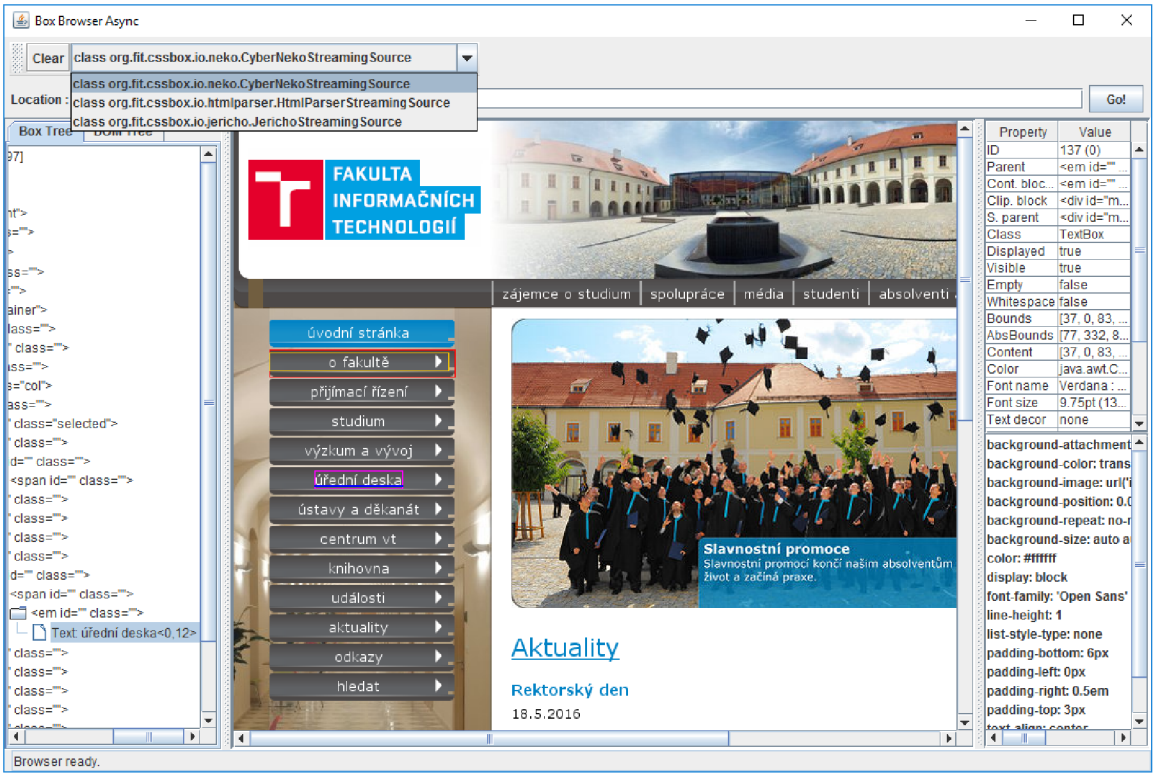
Pro demonstrační účely byla vytvořena varianta již existující demo aplikace *Box Browser* využívající nově implementované inkrementální zpracování. Je možné zvolit jeden ze tří implementovaných modulů pro zpracování HTML dokumentů a následně spustit vykreslování stránky, při kterém je možné pozorovat inkrementální zpracování. V ostatních ohledech je

[13] `java.awt.EventQueue`

[14] Rámec pro analýzu webových stránek vyvinutý na FIT VUT (http://www.fit.vutbr.cz/~burgetr/FITLayout/)

[15] To že není aktuálně využívána je indikováno tím, že metoda `getProcessor()` třídy `Viewport` vrací hodnotu `null`.

aplikace stejná jako již existující příklad *Box Browser*. Na obrázku 5.2 je snímek obrazovky zachycující demo aplikaci, kde je v levém horním rohu možné vidět výběr modulu pro zpracování HTML dokumentu. Zpracování je zahájeno stiskem tlačítka, po kterém je už možné proces vizuálně pozorovat.



Obrázek 5.2: Snímek obrazovky demo aplikace *Box Browser Async*.

5.2 Implementované moduly

Součástí zadání práce byl i požadavek na implementaci modulů využívajících různé knihovny pro zpracování HTML dokumentů. Ve výsledku byly na základě návrhu v kapitole 4.1 implementovány tři. Jejich podrobnějším popisem se zabývají následující kapitoly. Názvy kapitol reflektují knihovnu využívanou popisovanými moduly.

Každý modul je reprezentován jednou třídou a vzhledem k jejich jednoduchosti, nebylo účelné jejich vyčlenění do samostatných *JAR*[16] archivů. Proto jsou všechny třídy ve výsledku součástí základního balíku CSSBoxu a jejich závislosti jsou označené jako volitelné v nastavení sestavovacího nástroje Maven[17], který je využíván při CSSBoxu. Díky deklaraci závislostí jako volitelných, nejsou uživatelům, kteří nemají o tyto moduly zájem, zbytečně přidávány tyto tranzitivní závislosti. Způsob využití modulů v kombinaci s nástrojem Maven je popsán v příloze B.

Během implementace se ukázalo, že některé knihovny pro syntaktickou analýzu HTML dokumentů jsou vhodnější než jiné, a to zejména kvůli problémům při zpracování HTML dokumentů, které ale nejsou zároveň platnými XML dokumenty. Problémy jednotlivých

[16] **J**ava **AR**chive - souborový formát určený k distribuci přeložených Java tříd.
[17] Apache Maven - https://maven.apache.org/

knihoven jsou probrány v rámci popisů jednotlivých modulů, ale dá se říct, že jediná použitá knihovna, která byla zcela bezproblémová, je CyberNeko HTML Parser.

5.2.1 CyberNeko HTML Parser

Současná výchozí implementace třídy `DOMSource` využívá knihovnu CyberNeko HTML Parser, více informací o ní je v kapitole 3.1. Knihovna poskytuje jak DOM, tak i SAX rozhraní, díky tomu byla implementace snadná. Jedinou komplikací byla potřeba obejít jednu z chyb v knihovně, ale ta je již řešena ve dříve zmíněné třídě `DefaultDOMSource`. Implementace tohoto modulu se nachází ve třídě `CyberNekoStreamingSource`[18]. Jak bylo uvedeno v kapitole 3.1, knihovna je vytvořena s využitím rozhraní XNI. Implementace modulu tedy využívá implementaci rozhraní `XMLReader` poskytovanou knihovnou Xerces spolu s implementací rozhraní XNI `XMLPullParserConfiguration` určenému pro zpracování HTML dokumentů v podobě třídy `HTMLConfiguration`.

SAX rozhraní v podání této knihovny bylo funkční a nevykazovalo oproti DOM rozhraní žádné další problémy. Produkovaný proud SAX událostí bylo možné bez dalších úprav použít ke konstrukci identického DOM stromu jako v případě přímého využití DOM rozhraní. Knihovna provádí správné automatické uzavírání elementů, ať už v případě, kdy specifikace nevyžaduje výskyt uzavírající značky, nebo pokud je uzavírající značka na chybném místě. Nemá problémy s nevalidními dokumenty a je v současnosti využívána CSSBoxem, proto se jeví jako vhodná pro účely této práce.

5.2.2 HTML Parser

HTML Parser (více v kapitole 3.4) poskytuje nativní SAX API. Rozhraní `StreamingSource` počítá s přímým využitím syntaktických analyzátorů se SAX rozhraním. Díky tomu se implementace tohoto modulu stala velmi jednoduchou. Tato knihovna na rozdíl od předchozí, poskytuje vlastní implementace rozhraní `XMLReader`. Avšak třída se jmenuje stejně jako toto rozhraní, což může být matoucí. Modul je reprezentován třídou `HtmlParserStreamingSource`[19].

Přestože implementace modulu využívajícího tuto knihovnu byla snadná, ke vzniku použitelného to bohužel nevedlo. Knihovna nesprávně implementuje zpracování vstupního HTML dokumentu v případě využití SAX rozhraní. Není implementována podpora automatického uzavírání elementů v případech, kdy není podle standartu nutné ukončující značku uvádět[20]. Díky tomuto problému dojde v případě ne-XHTML dokumentů využívajících možnosti automatického uzavírání elementů k chybné konstrukci DOM stromu, který je ve výsledku hlubší (nevhodně zanořené elementy) a k chybnému vykreslení stránky nebo v případě, že se takový element vyskytuje v elementu `<head>`, např. `<link>` nedojde k vykreslení vůbec. Problém by bylo možné obejít v kódu adaptéru, ale takové řešení by nejspíše nebylo příliš vhodné už kvůli velkému množství dalších chyb[21], které vzhledem k minimální aktivitě projektu už nikdy nikdo řešit nebude.

Kvůli výše uvedeným problémům se tato knihovna jeví jako nevhodná pro použití s CSSBoxem při využití inkrementálního zpracování (a tedy s využitím SAX rozhraní).

[18] `org.fit.cssbox.io.neko.CyberNekoStreamingSource`

[19] `org.fit.cssbox.io.htmlparser.HtmlParserStreamingSource`

[20] *XMLReader doesn't match start and end tags* - problém neřešený od roku 2007 (https://sourceforge.net/p/htmlparser/bugs/242/).

[21] V době psaní práce 47 nevyřešených problémů, z nich se část datuje až do roku 2007 (https://sourceforge.net/p/htmlparser/bugs/).

Jiná situace by mohla nastat při klasickém využití jako další potomek abstraktní třídy `DOMSource`.

5.2.3 Jericho HTML Parser

Jericho HTML Parser (více v kapitole 3.2) na rozdíl od ostatních knihoven neposkytuje žádné standartní rozhraní, ale místo toho má vlastní API pro proudové zpracování `StreamedSource`. Z tohoto důvodu bylo vhodné volání knihovny obalit s využitím návrhového vzoru Adaptér [4]. Implementace adaptéru se nachází ve třídě `JerichoStreamingSource`[22].

Třída `StreamedSource` implementuje rozhraní `Iterable<Segment>`, kde `Segment`[23] reprezentuje část HTML dokumentu (deklaraci typu dokumentu, začátek / konec elementu, text mezi elementy apod.). Při volání `next()` na poskytnutém iterátoru je vrácen další dostupný segment, pokud je k dispozici nebo volání blokuje dokud k dispozici není. Díky vlastnostem jazyka Java, zejména rozšířená konstrukce `for`, není nutné obstarávat volání `hasNext()` a `next()` iterátoru ručně, ale je možné elegantně využít cyklu a skrýt vnitřní implementaci, což přispívá k čitelnosti výsledného kódu. Podle typu segmentu (konkrétního potomka třídy `Segment`) popř. jeho dalších vlastností je vygenerována odpovídající událost (zavolána příslušná metoda rozhraní `ContentHandler`).

Knihovna nemá problémy s deformovanými dokumenty, má poměrně dobrou dokumentaci, ale s jejím API se místy špatně pracuje. Ale i tak je její funkčnost dobrá a je vhodná pro použití při inkrementálním zpracování.

5.2.4 Adaptéry

Pro demonstraci interoperability mezi asynchronním (inkrementálním) a klasickým zpracováním dokumentu v CSSBoxu vznikly dva adaptéry. Umožnují využít vstupní modul pro inkrementální zpracování pro zpracování v jednom kroku a opačně. Zejména pro první variantu je možné najít praktické využití, přičemž druhá slouží spíše k demonstraci možností.

Prvním z adaptérů je `StreamingSourceDOMAdapter`[24], který obaluje poskytnutou implementaci `StreamingSource` jako potomka abstraktní třídy `DOMSource`. Pracuje tak, že získá strom DOM a následně jej rekurzivně prochází a generuje odpovídající události. Praktické využití takového adaptéru bude nejspíše minimální a slouží zejména k demonstračním účelům.

Pro konverzi v opačném směru slouží třída `DOMSourceStreamingAdapter`[25]. Při volání obalené implementaci rozhraní `StreamingSource` jí poskytne speciální implementaci rozhraní `ContentHandler`, která na základě přijímaných událostí sestavuje výsledný DOM strom. Po dokončení zpracování celého dokumentu je takto sestavený strom vrácen a je tak splněn kontrakt metody `parse()`. Na rozdíl od předchozího adaptéru, je možné pro tento nalézt praktické využití. A to možnost využít nově implementované vstupní moduly i v aplikacích, kde není potřeba inkrementálního zpracování a jeho implementace by vedla pouze ke zvýšené režii bez zjevného opodstatnění.

[22] `org.fit.cssbox.io.jericho.JerichoStreamingSource`
[23] http://jericho.htmlparser.net/docs/javadoc/net/htmlparser/jericho/Segment.html
[24] `org.fit.cssbox.io.adapter.StreamingSourceDOMAdapter`
[25] `org.fit.cssbox.io.adapter.DOMSourceStreamingAdapter`

5.3 Použité nástroje

CSSBox je vytvořen s využitím jazyka Java SE 6 [5], proto i praktická část této práce byla vytvořena s jejím využitím. Verze 6 je sice již staršího data a nepodporuje nejnovější novinky jako jsou lambda výrazy, ale disponuje širokým rozšířením a jedná se o již dobře odladěnou platformu a zajišťuje tak dobrou kompatibilitu pro uživatele.

Při práci na projektu byla využita studentská licence integrovaného vývojového prostředí (IDE) IntelliJ IDEA.

Kapitola 6

Testování a dosažené výsledky

V první části se tato kapitola zabývá testováním vytvořených modifikací zobrazovacího stroje, včetně vytvořených modulů. Protože mezi hlavní cíle implementovaným modifikací patří záměr zmenšit dobu mezi zahájením zpracování a začátkem vykreslování z pohledu uživatele, zabývá se část této kapitoly experimenty s různými nastaveními pro implementované modifikace, tak aby byla doba odezvy z pohledu uživatele minimalizována. Je také experimentováno s vlivem úprav na celkový čas nutný k dokončení zpracovávání celého dokumentu. Nakonec se tato kapitola zabývá nalezením dalších slabých míst CSSBoxu z hlediska výkonu a návrhy na budoucí úpravy s cílem zmírnění jejich dopadu.

6.1 Testování

Testování probíhalo formou vizuálního porovnání vykreslené stránky jak s využitím původního přístupu, tak s využitím nově implementovaného řešení. V případě využití knihovny CyberNeko byly výsledky vykreslování vizuálně shodné - knihovna se používala už dříve, jen se s ní pracovalo jiným způsobem, takže se ukázalo, že provedené úpravy neměly dopad na kvalitu zpracování (výstupy jsou na pixel shodné). Porovnání je vidět na obrázku 6.1.

Obrázek 6.1: Porovnání před provedením úprav (vlevo) a po úpravách s využitím modulu využívajícího knihovny CyberNeko HTML Parser (vpravo).

V případě ostatních modulů byla situace o poznání horší, stránky byly chybně vykreslené kvůli chybám při dekódování HTML dokumentů. O konkrétních problémech jednotlivých knihoven pojednávají podkapitoly věnované jednotlivým knihovnám v kapitole 5.2. Na obrázku 6.2 je porovnání výstupu při využití knihovny HTML Parser. Jsou zde jasně viditelné

chyby v zobrazení způsobené chybnou konstrukcí DOM stromu, která je zapříčiněná chybou v knihovně, jenž je podrobněji popsána v kapitole 5.2.2.

Obrázek 6.2: Porovnání před provedením úprav (vlevo) a po úpravách s využitím modulu využívajícího knihovny HTML Parser (vpravo).

Další prováděné testy byly ve formě testování subjektivního pocitu z plynulosti a rychlosti práce. V tomto případě došlo k velmi výraznému vlivu na pocitovou rychlost demonstrační aplikace. V případě rychlého připojení je vliv citelný, ale opravdu velký vliv se projeví při použití pomalého připojení k internetu, kdy jde o rozdíly mezi desítkami sekund a téměř okamžitým zobrazením alespoň částí dokumentu.

6.2 Experimenty

Pomocí experimentů byl ověřován vliv rychlosti připojení, počtu vláken a intervalu zpracování na:

- Dobu do prvního viditelného vykreslení pro uživatele
- Dobu na dokončení vykreslování celé stránky

Výsledky ukazují, že došlo ke zvýšení výkonu ve všech testovaných scénářích nebo případně pouze nedošlo ke zlepšení, ani jeden z testů neprokázal významnější zhoršení výkonu. Pozitivní efekt je viditelný zejména při testování odezvy na uživatelské akce, ale v menší míře je přítomen i v případě testování doby na načtení celé stránky.

Na celkovou dobu načítání má v případě rychlého připojení vliv zejména zvyšující se počet vláken. Je to způsobené tím, že se minimalizuje vliv latence a plně se využije potenciál linky pro přenos většího množství malých souborů.

Interval zpracování nemá významný vliv (pokud se pohybuje v řádu stovek milisekund) ani na dobu do prvního vykreslení, ani na celkovou dobu vykreslování. Má však významný vliv na subjektivní vnímání plynulosti zobrazování uživatelem. Při malých hodnotách má uživatel dojem plynulosti, ale delší intervaly vytvářejí dojem „poskakování", který má negativní vliv na pocitové vnímání aplikace uživatelem.

6.2.1 Testovací prostředí a metodika

Testování probíhalo na stroji s operačním systémem Windows 10, procesorem Intel Core 2 Quad Q6600 s 8 GB operační paměti. Rychlost připojení k internetu byla 30 Mbps.

Pro testy využívající omezené rychlosti přenosu dat byl využit proxy server Squid [12]. Tento nástroj umožňuje snadno omezovat rychlost přenosu pro jednotlivé klienty. Adresa

proxy serveru byla pomocí parametrů příkazové řádky nastavena přímo Java Virtual Machine (JVM)[1] a díku tomu nebylo nutné provádět žádné změny přímo v CSSBoxu.

Všechny pokusy byly opakovány desetkrát. Takový počet opakování by měl být dostatečný, aby eliminoval vliv Just-in-time (JIT) překladače, krátkodobé výkyvy rychlosti připojení k internetu apod. Výsledné hodnoty zobrazené v grafech jsou aritmetickým průměrem těchto hodnot, navíc jsou v grafech vyznačeny směrodatné odchylky zobrazovaných veličin. Navíc před spuštěním testu bylo provedeno jedno spuštění všech testů navíc, aby se zajistilo, že všechny potřebné Java třídy budou načtené, a ty často využívané z nich i zpracovány JIT překladačem.

6.2.2 Čas do viditelného vykreslení

Cílem tohoto experimentu bylo zjistit vliv provedených úprav na čas potřebný k tomu, aby uživatel viděl „pohyb". Tedy aby neměl z aplikace pocit, že nereaguje – „zamrzla". Stejně významné jsou i intervaly překreslování, které mají na subjektivní pocit uživatele úplně stejný vliv, které ale na rozdíl od času do viditelného vykreslení byly zkoumány jen subjektivně. Experimenty byly provedeny na celkem devíti kombinacích - interval zpracování 100 ms, 200 ms, 500 ms a 1, 3 a 5 pracovních vláknech. V grafech jsou pro přehlednost uvedeny jen některé kombinace.

Na grafu 6.3 jsou vyneseny výsledky experimentu, při rychlosti připojení 30 Mbps. Takový test odpovídá využití s rychlým připojením k internetu. Je zde vidět mírný výkonnostní zisk implementovaného řešení oproti původnímu stavu. Díky vysoké rychlosti připojení v tomto případě není úzkým hrdlem samotná rychlost přenosu HTML dokumentu nebo dalších externích zdrojů, ale fakt, že není se samotným vykreslováním nutné čekat na dostupnost všech obrázků s tím, že stránka je překreslena ihned, jak jsou obrázky dostupné. Z výsledků vyplývá, že zvyšující se počet pracovních vláken má v tomto případě malý vliv, stejně tak vliv intervalu překreslování je zanedbatelný.

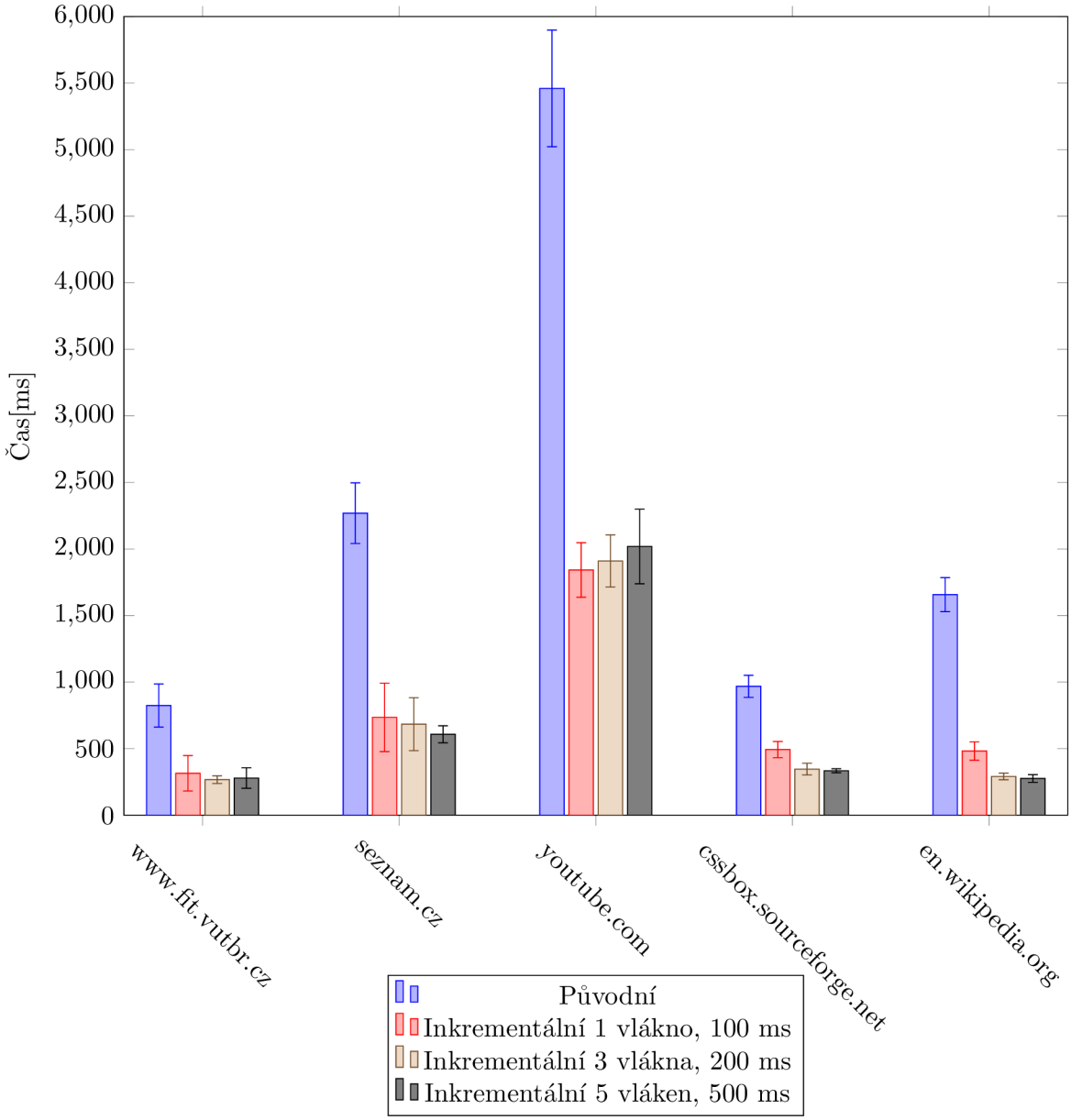
Na grafu 6.4 jsou výsledky experimentu, při rychlosti připojení 450 kbps. Takový test odpovídá využití s mobilním připojením s technologií Enhanced Data Rates for GSM Evolution (EDGE). V tomto případě jsou rozdíly výraznější než při použití rychlého připojení. Při takto nízkých rychlostech je snadno vidět velký výkonnostní zisk, který je zapříčiněn zejména inkrementálním zpracováním HTML dokumentu. Tím, že není nutné čekat na načtení celého dokumentu, které může při takto nízkých rychlostech zabrat značné množství času, je možné rychle vykreslit alespoň malou část dokumentu.

V obou případech je tedy jasně vidět výrazné zlepšení oproti výchozímu stavu. Při zkoumání vlivu počtu pracovních vláken se ukázalo, že rozdíl mezi jedním a třemi je jasně zřetelný, ale rozdíl mezi třemi a pěti už není téměř měřitelný. Vliv různých intervalů zpracování byl v tomto případě zanedbatelný, a to i ze subjektivního pohledu - čas do prvního zobrazení lišící se v řádu malých stovek milisekund uživatel téměř nezaznamená.

6.2.3 Čas do dokončení zpracování

Druhý experiment se zabýval zkoumáním doby nutné k vykreslení celého dokumentu a zjištění, zda provedené úpravy přinesly zrychlení vykreslování stránek díky paralelnímu získávání externích zdrojů nebo spíše, zda jejich režie neměla negativní dopad na výkonnost zobrazovacího stroje.

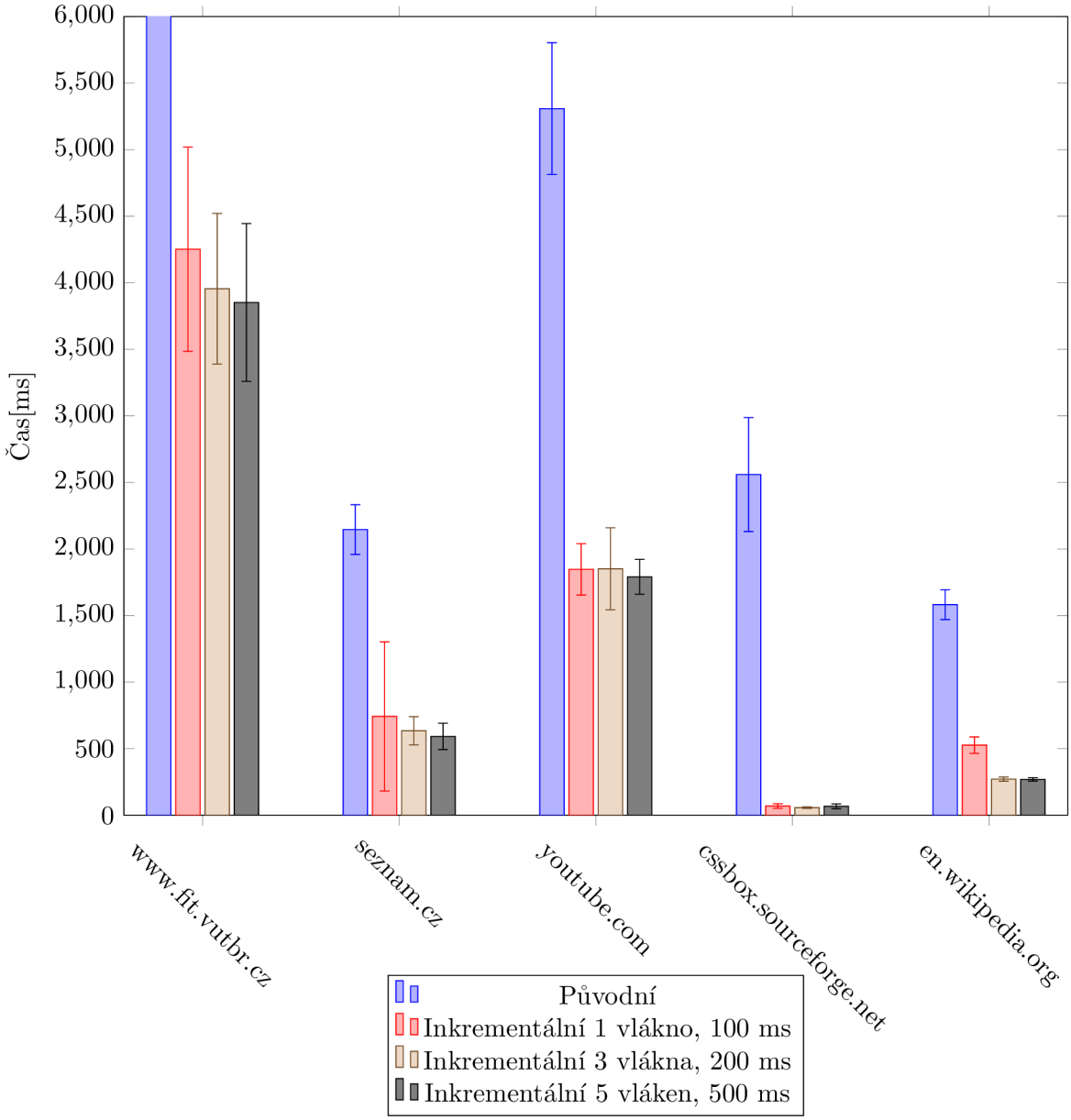
[1] Vlastnosti `http.proxyHost` a `http.proxyPort` (https://docs.oracle.com/javase/1.5.0/docs/guide/net/proxies.html)



Obrázek 6.3: Doba nutná pro prvotní vykreslení při využití rychlého připojení k internetu.

Na grafu 6.5 jsou zobrazeny výsledky prvního testování, které bylo prováděno při rychlosti připojení 30 Mbps. Díky vysoké rychlosti připojení v tomto případě není úzkým hrdlem samotná rychlost přenosu dat, ale spíše latence linky. Hlavní výkonnostní zisk je zde tedy zapříčiněn paralelním získáváním odkazovaných zdrojů (obrázky), které jsou v případě využití původního řešení získávány sekvenčně, což zvýrazňuje vliv latence na celkovou dobu zpracování.

Výsledky druhého případu se sníženou rychlostí na úroveň EDGE jsou zobrazeny v grafu 6.6. Při takto nízké rychlosti se začíná v některých případech projevovat nově přidaná režie a dochází k situacím, kdy provedené úpravy způsobí propad výkonu nebo alespoň zanedbatelný výkonnostní zisk za cenu zvýšené komplexnosti řešení.

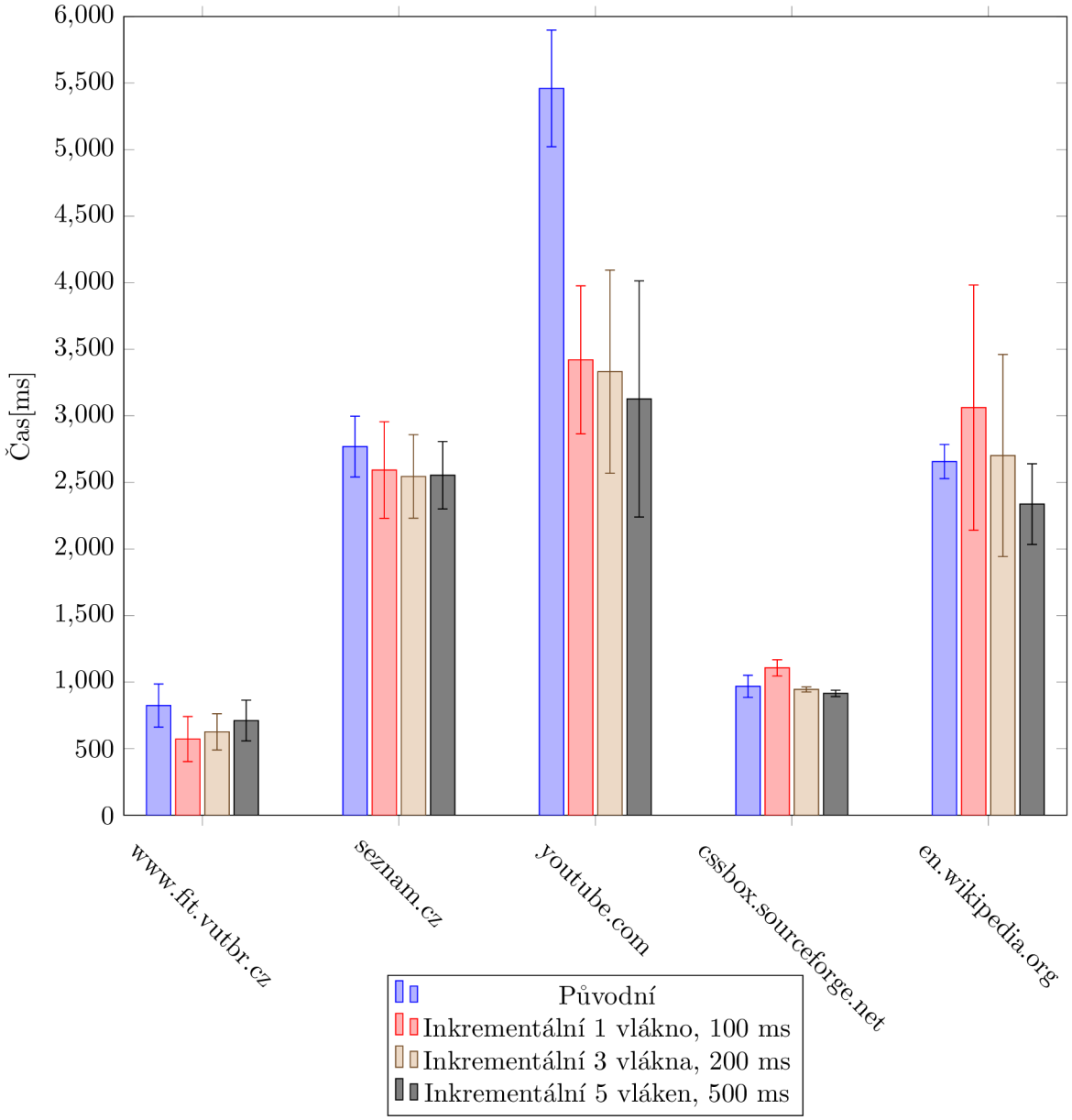


Obrázek 6.4: Doba nutná pro prvotní vykreslení při simulaci připojení přes EDGE.
Pozn.: Skutečná hodnota v prvním sloupci je přibližně 40 s.

6.3 Další slabá místa z hlediska výkonu

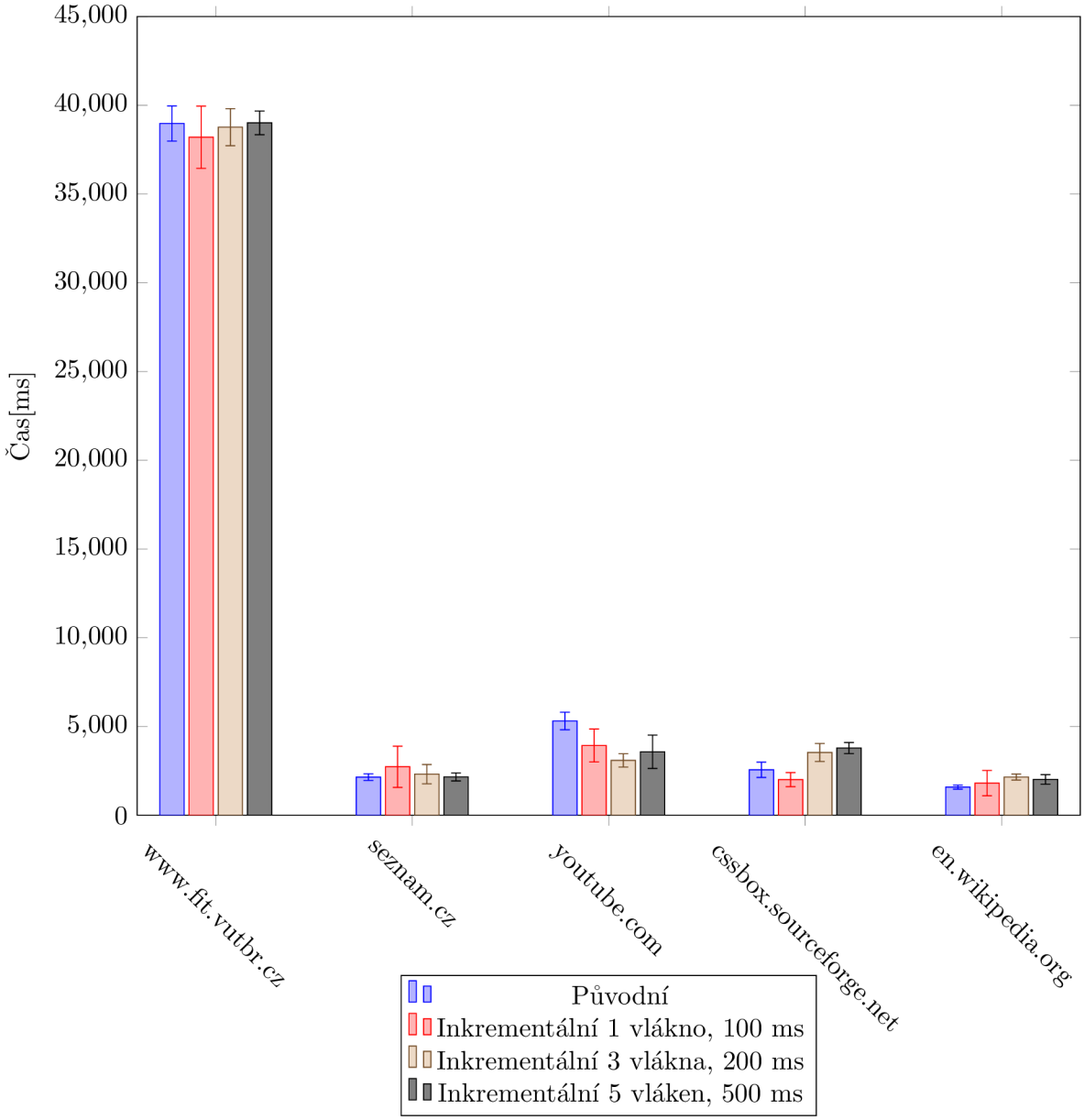
Pro zjištění dalších možností optimalizace výkonu byly využity možnosti profilování pomocí nástroje Java VisualVM[2], který je součástí JDK. Nástroj umožňuje analyzovat řetězce volání a zpracovávat doby vykonávání jednotlivých funkcí. Tyto data dokáže zobrazovat v agregované podobě, ze které je možné snadno získat představu o výpočetní náročnosti jednotlivých částí programu. Výsledky práce s profilerem ukázaly, že v rámci CSSBoxu a návazných knihoven existuje prostor pro další optimalizace.

[2]Grafický nástroj pro analýzu běžící JVM (https://visualvm.java.net).



Obrázek 6.5: Doba nutná pro vykreslení celého dokumentu při využití rychlého připojení k internetu.

Prostor pro vylepšení je v oblasti rychlosti syntaktické analýzy HTML dokumentu. A vzhledem k nedostatku kvalitních knihoven pro provádění dekódování HTML dokumentů (problém diskutovaný v kapitole 5.2), by mohlo řešení spočívat ve vytvoření nové Java knihovny pro zpracování HTML dokumentů podle algoritmu specifikovaného v kapitole 8.2 standardu HTML5 [6]. Při správné implementaci by mohlo dojít ke značnému zvýšení výkonu při analýze dokumentu a zároveň k odstranění nedostatků současných knihoven, které jsou často založené na upraveném XML analyzátoru a ne připravené specificky pro HTML, u kterého jsou pravidla volnější. Specifikace HTML přesně definuje jak přesný postup lexikální a syntaktické analýzy, ale i například algoritmus pro detekci znakové sady dokumentu nebo správnou konstrukci DOM stromu. Řešeny jsou i případy, kdy jsou elementy



Obrázek 6.6: Doba nutná pro vykreslení celého dokumentu při simulaci připojení přes EDGE.

špatně zanořeny nebo se nacházejí na nepovolených pozicích (např. nepovolené elementy přímo v elementu `<table>`). V takových případech je popsané správné chování, jehož cílem je maximalizovat schopnost správného zobrazení chybného dokumentu prohlížečem. Taková specifikace je nutná, aby bylo chování všech prohlížečů jednotné i v případě chybných dokumentů a ne jen v případě správných.

V současné době využívá CSSBox pro zpracování CSS knihovnu jStyleParser, která využívá syntaktický analyzátor založený na knihovně ANTLR. Tato knihovna umožňuje snadno vytvářet komplexní syntaktické analyzátory. V tomto případě výsledky profileru ukázaly, že samotná syntaktická analýza je výpočetně poměrně náročná a byla by vhodná její optimalizace, např. s využitím novějších vydání ANTLR, které obsahují výkonnostní vylepšení.

Značný dopad na výkon má i vyhodnocování CSS selektorů, které je nutné pro určení pravidel aplikovatelných na daný HTML element. Stejným problémem trpí i moderní prohlížeče, např. ve vykreslovacím jádře Blink používaném v prohlížečích Chrome a Opera toto vyhodnocování tvoří až 50% času pro výpočet efektivních stylů [10]. Proto každé zlepšení v oblasti efektivity vyhodnocování selektorů bude mít výrazný dopad na celkový výkon výpočtu stylů.

Dále existuje prostor ve formě více drobných zlepšení, která by mohli mít pozitivní dopad na výkon. Takovým příkladem je metoda `getFontName()` ve třídě `VisualContext`, která má za úkol vrátit název prvního dostupného fontu na základě CSS deklarace, která může uvádět více fontů v pořadí jejich preference[3]. Současná implementace při každém volání získá názvy všech dostupných fontů v hostitelském systému a v tomto seznamu následně sekvenčně vyhledává, dokud některá se zadaných možností není nalezena nebo nejsou vyčerpány všechny možnosti, v takovém případě vrátí výchozí hodnotu. Výsledky profileru ukazují, že největší výkonnostní problém je právě získávání všech systémových fontů. Vzhledem k tomu, že je tato funkce volána často, mělo by smysl ji optimalizovat využitím vyrovnávací paměti obsahující dostupné fonty a zlepšením vyhledávání v ní využitím vhodné datové struktury, např. třídy `HashMap`.

[3] Vlastnost `font-family` (https://www.w3.org/TR/CSS1/#font-family)

Kapitola 7

Závěr

Cílem práce bylo prozkoumat strukturu projektu CSSBox, prozkoumat existující knihovny pro dekódování HTML dokumentů včetně jejich podpory pro zpracování během načítání. A konečně navrhnout úpravy, které by umožnily CSSBoxu zobrazovat dokumenty již během jejich načítání. Těchto cílů bylo dosaženo, v kapitole 2 byl popsán CSSBox, v kapitole 3 byly popsány možné knihovny pro dekódování HTML dokumentů a v kapitole 4 bylo navrženo řešení zadaného problému. V kapitole 5 byla popsána implementace změn v jádru CSSBoxu podle návrhu popsaného v kapitole 4, včetně implementace modulů využívajících různé knihovny pro dekódování HTML dokumentů. V předposlední kapitole 6 bylo popsáno testování a prozkoumány další možnosti výkonnostních vylepšení a konečně byly zhodnoceny výsledky tohoto testování a diskutovány možnosti budoucího rozvoje projektu.

7.1 Budoucí vývoj

Prostor pro další rozvoj projektu vidím v následujících vylepšeních pro podporu inkrementálního zpracování, jde zejména o:

- *HTML5 analyzátor* - vytvoření syntaktického analyzátoru podle specifikace HTML 5, který odstraní problémy současných knihoven, jako jsou chybná konstrukce DOM stromu a nízký výkon
- *Asynchronní zpracování CSS* - v současnosti jsou všechny operace s CSS blokující (jak získání stylopisu, tak jeho zpracování), ale pokud by bylo možné provádět tyto operace vůči hlavnímu aplikačnímu vláknu asynchronně, snížilo by to zamrzání uživatelského rozhraní a zlepšilo uživatelský komfort
- *Částečný přepočet stylů* - implementace mechanismů diskutovaných v kapitole 4.2, jejichž cílem by bylo snížení výpočetní náročnosti odstraněním redundantních operací a tím zmenšení prodlev v zobrazení.

Další rozvoj projektu je možné také pojmout jako snahu o zvýšení celkového výkonu CSSBoxu optimalizací problematických částí popsaných v kapitole 6.3 nebo implementací nových vlastností jako je podpora protokolu HTTP/2, který snižuje vliv síťových latencí.

Literatura

[1] Burget, R.: Dokumentace projektu CSSBox [online]. 2014 [cit. 2015-12-06].
URL http://cssbox.sourceforge.net/documentation.php

[2] Clark, A.; Guillemot, M.: Dokumentace projektu NekoHTML [online]. [cit. 2015-12-25].
URL http://nekohtml.sourceforge.net/

[3] Fielding, R.; Gettys, J.; Mogul, J.; aj.: Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), Červen 1999.
URL http://www.ietf.org/rfc/rfc2616.txt

[4] Gamma, E.; Helm, R.; Johnson, R.; aj.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, 1994, ISBN 9780201633610.

[5] Gosling, J.: *The Java Language Specification*. Addison-Wesley Java series, Addison-Wesley, 2005, ISBN 9780321246783.

[6] Hickson, I.; Berjon, R.; Faulkner, S.; aj.: HTML5. Recommendation, W3C, Říjen 2014.
URL https://www.w3.org/TR/2014/REC-html5-20141028/

[7] Kocman, R.: *Podpora dynamického DOM v zobrazovacím stroji HTML*. Diplomová práce, FIT VUT v Brně, Brno, 2014.

[8] Loskot, R.: *Podpora JavaScriptu v zobrazovacím stroji HTML*. Diplomová práce, FIT VUT v Brně, Brno, 2014.

[9] Masinter, L.: The "data"URL scheme. RFC 2397 (Proposed Standard), Srpen 1998.
URL http://www.ietf.org/rfc/rfc2397.txt

[10] Opera Software AS: Style Invalidation in Blink [online]. [cit. 2016-05-02].
URL https://docs.google.com/document/d/1vEW86DaeVs4uQzNFI5R-_xS9TcS1Cs_EUsHRSgCHGu8

[11] Peierls, T.; Goetz, B.; Bloch, J.; aj.: *Java Concurrency in Practice*. Pearson Education, 2006, ISBN 9780132702256.

[12] Wessels, D.: *Squid: The Definitive Guide*. Definitive Guides, O'Reilly Media, 2004, ISBN 9780596550530.

Přílohy

Seznam příloh

Příloha A

Obsah CD

- Zdrojové kódy implementovaného řešení
- Dokumentace automaticky vygenerovaná na základě JavaDoc komentářů
- Text práce a jeho zdrojové kódy

Příloha B

Maven

Jak bylo uvedeno v kapitole 5.2, jsou všechny implementované moduly součástí hlavní distribuce CSSBoxu, ale zároveň CSSBox nutně nezávisí na všech potřebných knihovnách. Vzhledem k tomu, že CSSBox využívá nástroj pro správu závislostí Maven, jsou závislosti deklarovány v souboru `pom.xml`. Ve výpisu B.1 je ukázka deklarace závislosti nutné pro použití modulu využívajícího Jericho. Je také možné deklarovat závislost na více knihovnách současně jednoduchým opakováním elementu `<dependency>`.

```
<dependency>
    <groupId>net.sf.cssbox</groupId>
    <artifactId>cssbox<artifactId>
    <version>1.0-SNAPSHOT</version>
    <dependencies>
        <!-- Použité dekodéry HTML dokumentů -->
        <dependency>
            <groupId>net.htmlparser</groupId>
            <artifactId>jericho-html<artifactId>
            <version>2.3</version>
        </dependency>
        <dependency>
            ...
        </dependency>
    </dependencies>
</dependency>
```

Výpis B.1: Příklad zápisu závislosti na konkrétním dekodéru HTML dokumentů v `pom.xml`.

Hodnoty pro deklarace ostatních závislostí, včetně jejich testovaných verzí, jsou uvedeny v tabulce B.1. Pozn.: závislost na CyberNeko není nutné uvádět explicitně, protože CSSBox na něm závisí vždy, kvůli podpoře klasického zpracování - třída `DefaultDOMSource`.

Knihovna	**groupId**	**artifactId**	**version**
CyberNeko	net.sourceforge.nekohtml	nekohtml	1.9.22
HTML Parser	org.htmlparser	htmlparser	2.1
Jericho	net.htmlparser	jericho-html	2.3

Tabulka B.1: Hodnoty pro jednotlivé knihovny