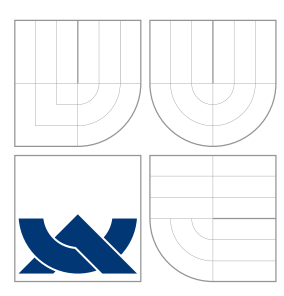
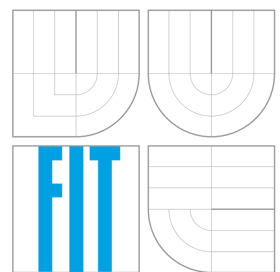


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

APLIKACE NA PLATFORMĚ MOZILLA

APPLICATIONS ON MOZILLA PLATFORM

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. JAN KUPČÍK
AUTHOR

VEDOUCÍ PRÁCE — Ing. RADEK BURGET, Ph.D.
SUPERVISOR

BRNO 2007

Zadání diplomové práce

Řešitel: **Kupčík Jan, Bc.**
Obor: Informační systémy
Téma: **Aplikace na platformě Mozilla**
Kategorie: Web

Pokyny:

1. Seznamte se s aplikační platformou Mozilla
2. Prostudujte technologie RDF, XUL a JavaScript v kontextu vývoje aplikací na této platformě
3. Navrhněte rozšíření prohlížeče Firefox pro sledování změn na webových stránkách využívající popsaných technologií
4. Implementujte navržené rozšíření na platformě Mozilla
5. Proveďte shrnutí výsledků a navrhněte další možné úpravy

Literatura:

- Dokumentace dostupná na Mozilla.org

Při obhajobě semestrální části diplomového 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 ročníkového a semestrálního projektu (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 paměťovém médiu (disketa, CD-ROM), 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: 28. února 2006
Datum odevzdání: 22. května 2007

Licenční smlouva

Licenční smlouva v kompletním znění je uložena v archivu Fakulty informačních technologií Vysokého učení technického v Brně.

Výňatek z licenční smlouvy:

Článek 2
Udělení licenčního oprávnění

1. Autor touto smlouvou poskytuje nabyvateli oprávnění (licenci) k výkonu práva uvedené dílo nevýdělečně užít, archivovat a zpřístupnit ke studijním, výukovým a výzkumným účelům včetně pořizování výpisů, opisů a rozmnoženin.

2. Licence je poskytována celosvětově, pro celou dobu trvání autorských a majetkových práv k dílu.

3. Autor souhlasí se zveřejněním díla v databázi přístupné v mezinárodní síti:

   ■ ihned po uzavření této smlouvy

   □ 1 rok po uzavření této smlouvy

   □ 3 roky po uzavření této smlouvy

   □ 5 let po uzavření této smlouvy

   □ 10 let po uzavření této smlouvy

   (z důvodu utajení v něm obsažených informací).

4. Nevýdělečné zveřejňování díla nabyvatelem v souladu s ustanovením § 47b zákona č. 111/1998 Sb., v platném znění, nevyžaduje licenci a nabyvatel je k němu povinen a oprávněn ze zákona.

Abstrakt

Cílem této diplomové práce je seznámit se s aplikační platformou Mozilla – její strukturou, technologiemi, které využívá, a způsobem vývoje samostatně běžících i rozšiřujících aplikací, které tuto platformu využívají (např. webový prohlížeč Firefox, poštovní klient Thunderbird). Práce zahrnuje relevantní informace o využívaných jazycích, kterými jsou XUL, CSS, JavaScript, RDF/XML a další. Jsou zde také rozebrány objektově orientované principy použitelné v jazyce JavaScript v.1.7. Další části se věnují problematice tvorby a využívaní komponent platformy i aplikací. Informace o platformě jsou završeny uvedením možností ladění a následného převodu do distribuovatelné podoby. Tyto znalosti jsou následně použity při tvorbě aplikace umožňující sledování změn dokumentů v síťovém prostředí. Lze zde najít její návrh a některé detaily implementace vztahující se k obecnému vývoji aplikací na uvedené platformě.

Klíčová slova

Mozilla, XUL, XBL, JavaScript, XPCOM, IDL, RDF, XML, XPI

Abstract

The goal of the thesis is to study the Mozilla application platform – its structure, used technology, and the ways of development of standalone applications and extensions for the applications based on this platform (e.g. the Firefox web browser, the Thunderbird e-mail client). The thesis also contains relevant information about the used programming languages such as XUL, CSS, JavaScipt, RDF/XML and others. It describes the object oriented principles available in the JavaScript v.1.7 language. Next parts are dedicated to creating and using the platform components and the applications. The information about the platform is concluded by a presentation of the debugging and deployment possibilities. This knowledge is used to create an application able to watch changes of documents in a network environment. The thesis describes the application design and some details related to the general development of applications based on the discussed platform.

Keywords

Mozilla, XUL, XBL, JavaScript, XPCOM, IDL, RDF, XML, XPI

Citace

Jan Kupčík: Aplikace na platformě Mozilla, diplomová práce, Brno, FIT VUT v Brně, 2007

Aplikace na platformě Mozilla

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.

....
Jan Kupčík
22. května 2007

© Jan Kupčík, 2007.
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

1 Úvod 2
- 1.1 Přehled následujících kapitol . . . 3
- 1.2 Formátování textu . . . 5

2 Současný stav 6

3 Architektura platformy Mozilla 8

4 Tvorba uživatelského rozhraní 10
- 4.1 Obsah XUL . . . 11
  - 4.1.1 Určení vzhledu . . . 12
  - 4.1.2 Podpora pro definici chování . . . 12
- 4.2 Kaskádové styly . . . 14
- 4.3 Definice externích řetězců . . . 15
  - 4.3.1 DTD entity . . . 15
  - 4.3.2 Textové konstanty . . . 16
  - 4.3.3 Volba aktivního jazyka . . . 16

5 Programování chování 17
- 5.1 Programovací jazyky . . . 17
  - 5.1.1 Objektové programování v jazyce JavaScript . . . 18
- 5.2 XPCOM . . . 22
  - 5.2.1 Možné problémy . . . 23
  - 5.2.2 Vytváření komponent . . . 24
  - 5.2.3 Bezpečnost . . . 27

6 Datová úložiště, RDF 29
- 6.1 Ukládání znalostí . . . 29
- 6.2 Dotazování . . . 31
- 6.3 Podpora v XPCOM . . . 31

7 Ladění a testování aplikací 33
- 7.1 Konfigurace prohlížeče Firefox . . . 33
- 7.2 Úprava aplikace . . . 34
- 7.3 Ladění uživatelského rozhraní . . . 35
- 7.4 Ladění kódu aplikace . . . 35
- 7.5 Ladění komponent . . . 36
- 7.6 Ladění RDF . . . 38

8 Distribuce aplikací **39**

9 Aplikace DocWatcher **41**

9.1 Návrh aplikace . . . 42
9.1.1 Stanovení požadavků . . . 42
9.1.2 Architektura aplikace . . . 42
9.1.3 Ukládání entit do databáze . . . 43
9.2 Detaily implementace . . . 45
9.2.1 Datová vrstva . . . 45
9.2.2 Uživatelské rozhraní . . . 46
9.2.3 Hlavní okno aplikace . . . 46
9.2.4 Dialogové okno vlastností odkazu . . . 47
9.2.5 Proces kontroly . . . 48
9.3 Závěrečné poznámky . . . 51

10 Závěr **52**

Kapitola 1

Úvod

Vývoj každého softwarového produktu začíná stanovením požadavků, které by měl splňovat. Mezi tyto požadavky také bezesporu patří výběr operačních systémů a hardwarových architektur, na kterých musí výsledný produkt fungovat. Na základě tohoto výběru a stanovených cenových nákladů je pak potřeba zvolit, zda aplikace bude využívat pouze aplikační programové rozhraní podporovaných operačních systémů, zda bude postavena na nějaké multiplatformní programové knihovně, nebo bude pracovat v určitém aplikačním prostředí, jehož služby bude využívat. Toto rozhodnutí je klíčové, protože má vliv na budoucí architekturu aplikace, může omezovat množinu vhodných programovacích jazyků, ve fázi provozu může ovlivnit výkonnost, použitelnost a další parametry.

Výběr vhodné programové základny pro nový software byl řešen i roku 1998 v *Mozilla Organization*. Došlo totiž k uvolnění většiny zdrojových kódů prohlížeče Netscape Communicator 5.0 pod licenci *Open source* a následně bylo rozhodnuto, že zdrojový kód je již natolik neudržovatelný, že je potřeba jej celý přepsat. Celý projekt byl však tak ambiciózní a rozsáhlý, že až po čtyřech letech byl vydán nový webový prohlížeč Mozilla 1.0. Důvodem zdržení byl fakt, že jako základ nebyla použita žádná stávající knihovna, avšak byla vyvinuta zcela nová základna pro internetové aplikace s plně programovatelným uživatelským rozhraním a modulární architekturou – vznikla aplikační platforma Mozilla.

Postupem času přibyly nové aplikace, které tuto platformu využívají. Jde např. o SeaMonkey (nástupce Mozilla Suite, k jehož ukončení došlo v roce 2003), velice úspěšný webový prohlížeč Firefox, poštovní klient Thunderbird, HTML editor Nvu, hudební přehrávač Songbird, SIP klient ZAP a další.

Ačkoli se aplikační platforma Mozilla dobře hodí pro vytváření nových volně šiřitelných desktopových aplikací v síťovém prostředí, je vývojáři dost přehlížena. Důvodem může být neznalost – existuje málo těch, kteří ví, co všechno se nachází za prohlížečem Firefox, a také větší nároky na studium této problematiky – v síti Internet lze najít nemalé množství návodů, jak vytvořit jednoduchou aplikaci, avšak aktuální ucelený popis vytváření složitější aplikace nepřináší žádný z nich. Jeden ze dvou cílů této diplomové práce je proto navázat na publikaci [1] a doplnit shromážděné informace a odkazy na zdroje platné pro vývoj aplikací na platformě Mozilla s jádrem odpovídajícím XULRunner v.1.8, upozornit na chyby v jádře, které dosud nebyly opraveny a nekorektní informace uvést na správnou míru. Práce tedy rozebírá problematiku vytváření samostatně běžících aplikací pro XULRunner i rozšiřujících aplikací (označované jako *extensions*), ukazuje některé principy vytváření uživatelského rozhraní, programování chování, tvorbu komponent systému a práci s datovými úložišti ve formátu RDF. Nemalá část je věnovaná ladění aplikací a vše je završeno informacemi o možnostech převodu do distribuovatelné podoby a správě aktualizací.

Druhým cílem práce je využít nabyté znalosti a vytvořit rozšiřující aplikaci pro vybrané aplikace založené na platformě Mozilla, která má řešit neexistenci volně šiřitelného multiplatformního nástroje pro sledování změn dokumentů v síťovém prostředí. Použití tohoto rozšíření si lze ve světě elektronických dokumentů představit téměř kdekoli – chtěli bychom si udržovat přehled o cenách určitých produktů, hlídat si vydání nových verzí programů, zjišťovat, zda se na fórech hovoří o tématech, která nás zajímají. Studenty může zajímat, zda přibyly nebo byly změněny informace na stránkách předmětů, je-li již opravena zkouška, které nové kulturní akce škola připravuje. Diplomová práce tedy dále obsahuje návrh a některé důležité aspekty implementace z hlediska obecného vývoje aplikací pro tuto platformu. Nástroj pro sledování změn dokumentů byl pojmenován DocWatcher.

Diplomová práce navazuje na předchozí semestrální projekt, ve kterém byly uvedeny základní informace o vývoji aplikací na platformě Mozilla. Tato práce uvedené informace rozšiřuje především o poznatky z praktického používání, přidány jsou kapitoly o XPCOM a ladění aplikací. Druhá část, návrh aplikace DocWatcher, je ze semestrálního projektu přebrán a mírně upraven tak, aby vyhovoval vlastnostem platformy Mozilla. Doplněny jsou pak informace týkající se implementace aplikace a pár závěrečných poznámek, které mohou vývojáři pomoci.

1.1 Přehled následujících kapitol

- Současný stav – uvádí programové knihovny a softwarové platformy, které jsou dostupné pro návrh nových aplikací.
- Architektura platformy Mozilla – zabývá se architekturou platformy Mozilla, uvádí zjednodušený diagram návaznosti jednotlivých vrstev a vysvětluje jejich funkci.
- Tvorba uživatelského rozhraní – obsahuje informace o způsobu vytváření uživatelských rozhraní využívajících jazyk XUL, odlišnostech v definicích kaskádových stylů a popisuje, jak aplikaci připravit pro provoz ve vícejazyčném prostředí.
- Programování chování – vypisuje seznam podporovaných jazyků, které lze využívat při programování a následně se detailně zabývá možnostmi jazyka JavaScript, především způsobem realizací objektově orientovaných principů. Následně se věnuje využívání a tvorbě vlastních komponent.
- Datová úložiště, RDF – představuje principy jazyka RDF/XML, uvádí, jakým způsobem lze zapisovat znalosti a jaké prostředky platforma pro práci s nimi poskytuje.
- Ladění a testování aplikací – informuje o způsobech ladění a testování aplikací využívajících jazyk JavaScript, které prostředky lze využít, nutné úpravy kódu a nastavení prostředí.
- Distribuce aplikací – popisuje adresářovou strukturu, kterou je vhodné dodržovat a ukazuje, jak převést sadu souborů do podoby, kterou lze předat uživatelům.
- Aplikace DocWatcher – stanovuje požadavky kladené na tuto aplikaci, uvádí její strukturu, pojednává o způsobu řešení uživatelského rozhraní a některých aspektů chování aplikace a popisuje vybrané aspekty implementace, které jsou důležité z pohledu obecného vývoje aplikací na této platformě.

- Závěr – hodnotí výsledky této práce, její přínos a další možnosti rozvíjení v budoucnu.

1.2 Formátování textu

Pro lepší čitelnost jsou v textu použity různé typy písma.

- `identifikátory, hodnoty konstant, klíčová slova a výpisy kódu`
- *slovní vyjádření v angličtině*
- označení souborů a adresářů

Kapitola 2

Současný stav

Existuje několik možných cest, kterými se může ubírat vývoj nových softwarových projektů. První z nich je vhodná pro aplikace, na které jsou kladeny zvláštní požadavky z hlediska funkčnosti, výkonnosti nebo třeba grafického uživatelského rozhraní (GUI). Tyto aplikace využívají přímo služeb operačního systému, na kterém běží, pomocí volání funkcí aplikačního programového rozhraní (API). Uvedený způsob je využíván například u systémových komponent daného operačního systému, kde má programátor možnost detailně řídit všechny procesy, které se zde provádějí. Obecně je na této úrovni také využíván neobjektový jazyk C, jehož překladem a optimalizací se dá dosáhnout rychlejší odezvy a vyššího výpočetního výkonu. Nevýhodou ovšem může být větší časová náročnost tvorby projektu způsobená vyšším množstvím zdrojového kódu, který požadovanou funkčnost implementuje. S tím následně také souvisí zvýšené množství úsilí, které je potřeba vynaložit k odladění a otestování tohoto kódu a zároveň díky použití jazyka C může struktura programu při větších změnách rychleji degradovat.

Uvedené nevýhody můžeme částečně eliminovat využitím programových knihoven, například nabízejících podporu pro snadnější přístup k uživatelskému rozhraní. Toto je druhá cesta. Knihovny, v operačním systému Windows jmenujme MFC[1], pak zapouzdřují volání API funkcí do logických celků, vytvářejí tedy vyšší úroveň abstrakce. Díky toho, že se zde již uplatňují objektové jazyky, jako je C++, je proces vývoje rychlejší a kód aplikace robustnější, než kdyby byla použita přímo volání API funkcí.

Třetí varianta, která stále více získává na oblibě, je vystavění aplikace na vhodné aplikační platformě. V tomto případě program již ke své činnosti používá API zcela minimálně, nebo vůbec, protože platforma abstrahuje většinu potřebných částí operačního systému – nabízí vlastní třídy zapouzdřující GUI, práci se soubory, přístup k síťovým prostředkům, správu paměti a procesů, komunikaci mezi procesy a další. Aplikační platforma tedy tvoří vrstvu, která odděluje aplikaci od operačního systému. Tato vrstva se skládá ze dvou částí – rozhraní, které je dostupné aplikaci a které je zároveň neměnné, a dále vlastní kód platformy, který implementuje nabízené služby. Jestliže je platforma označena jako multiplatformní, lze hostující aplikace provozovat na podporovaných operačních systémech a hardwarových architekturách. V tomto případě je pouze jiná implementace této platformy, avšak díky nezměněného rozhraní není nutné měnit zdrojový kód aplikace.

Dle typu programovacích jazyků lze následně rozdělit softwarové platformy na tři skupiny. První tvoří platformy, kde jsou hostitelské aplikace psány v přeložitelném jazyce do binárního kódu procesoru, na kterém bude program spuštěn – například C++. Pro

[1] http://msdn2.microsoft.com/en-us/library/d06h2x6e(VS.80).aspx

přenos mezi různými systémy není potřeba zasahovat do zdrojových kódů, avšak je nutná jejich kompilace pomocí kompilátoru určeného pro výslednou hardwarovou architekturu a operační systém. Aplikace jsou kompatibilní pouze na úrovni API. Do této skupiny patří například QT[2], wxWidgets[3] a další.

Dále existují platformy, kde zdrojový kód hostitelské aplikace není kompilován do nativního kódu procesoru, ale do speciálního mezikódu, který je distribuován. Uživatel si nainstaluje platformu určenou pro jeho systémovou konfiguraci a ta po spuštění aplikace automaticky za běhu překládá mezikód do nativního kódu procesoru, přičemž ten je následně spuštěn. Proces, který tuto činnost provádí, se nazývá *Just-In-Time* (JIT) kompilátor. Výhodou je tedy jediná spustitelná verze softwarového produktu pro podporované systémové konfigurace, avšak za cenu nutnosti mít nainstalované dané běhové prostředí na počítači a pomalejších reakcí programu při jeho spuštění. Sem patří Java[4], .NET Framework[5].

Do poslední skupiny se řadí platformy využívající jazyky, které nejsou před distribucí kompilovány. Aplikace jsou tedy šířeny s kódem v otevřené textové podobě. Po spuštění aplikace je zdrojový kód v případě potřeby vybalen a běhové prostředí platformy následně provádí interpretaci kódu. Toto řešení je nejpomalejší ze všech výše uvedených a hodí se pro nenáročné aplikace, které jsou volně šiřitelné a prozrazení kódu je irelevantní. Podstatnou navýhodou je pak problematická kontrola nejen syntaktických chyb, protože kód je vyhodnocován až v průběhu interpretace, a tak na chyby je programátor upozorněn až v okamžiku jejich potencionálního spuštění. Zástupcem je například aplikační platforma Mozilla[6]. Obecně sem můžeme zařadit i další skriptovací jazyky; pak je jen otázka, jak bohaté aplikační rozhraní daný interpret kódu nabízí.

Ačkoli by se mohlo zdát, že platforma Mozilla díky výše uvedeným vlastnostem není příliš vhodná pro širší použití, opak je pravdou. Důkazem jsou například úspěšný webový prohlížeč Firefox nebo poštovní klient Thunderbird. Tato platforma nabízí jednoduchý způsob programování založený na jazyku XML, který definuje uživatelské rozhraní, a jazyku JavaScript, jenž je používán pro definování chování aplikace. Výsledek lze beze změny spustit na více než 10 operačních systémech, viz [1, 16]. Hlavní výhodou je však přítomnost komponent umožňujících zobrazit (X)HTML dokument, aktivně se na něj napojit a pracovat s jeho obsahem. Právě tato vlastnost a vysoký počet podporovaných systémů vyzdvihují platformu Mozilla nad ostatní.

Bohatý seznam dostupných podpůrných knihoven i softwarových platforem lze najít v [25].

[2] `http://www.trolltech.com/`
[3] `http://www.wxwidgets.org/`
[4] `http://java.sun.com/`
[5] `http://msdn2.microsoft.com/en-us/netframework/default.aspx`
[6] `http://xulplanet.com/`

Kapitola 3

Architektura platformy Mozilla

Zjednodušené schéma architektury je uvedeno na obrázku 3.1. Podrobnější pohled lze najít v [1]. Komponenty můžeme rozdělit do několika vrstev. Nejníže položená, která komunikuje s operačním systémem, se nazývá *back-end*. Nachází se zde velké množství komponent, což jsou třídy implementované v programovacím jazyce C++ a přeložené pro konkrétní operační systém. Tyto komponenty pak tvoří jádro celé aplikační platformy. Mezi ně patří například:

- Komponenty jádra – definice datových typů, správce komponent a rozhraní, komponenty umožňující práci se soubory a datovými toky, řízení procesů, vláken, časovačů, výjimek a událostí.
- Komponenty pro práci v síťovém prostředí – podpora známých síťových protokolů, soketů, řízení vyrovnávacích pamětí, proxy, správce cookies, kooperace s webovými službami, služby pro stahování obsahu, možnost otevírání ZIP a JAR archívů.
- Komponenty pro podporu vývoje aplikací – správce oken, vestavěný webový prohlížeč, přístup k historii a profilům, RDF databáze, nativní podpora XML, DOM, HTML, XUL, SVG.
- Komponenty související s elektronickou poštou – odesílání, přijímání a správa pošty a diskuzních skupin, podpora adresáře, protokolů LDAP, IMAP, POP3.
- A další komponenty zahrnující například řízení a interpretace kódu v jazyce JavaScript, správa bezpečnosti a napojení na vyšší vrstvu XPCOM.

Nad tímto blokem leží systém zvaný *Cross Platform Component Object Model* (XPCOM). Ten zpřístupňuje komponenty vyšším vrstvám platformy tak, že přístup k nim již není závislý na použitém operačním systému. Protože XPCOM je přímo dosažitelné pouze pomocí programovacích jazyků C, C++, existuje zde ještě tenká vrstva XPConnect poskytující rozhraní pro JavaScript.

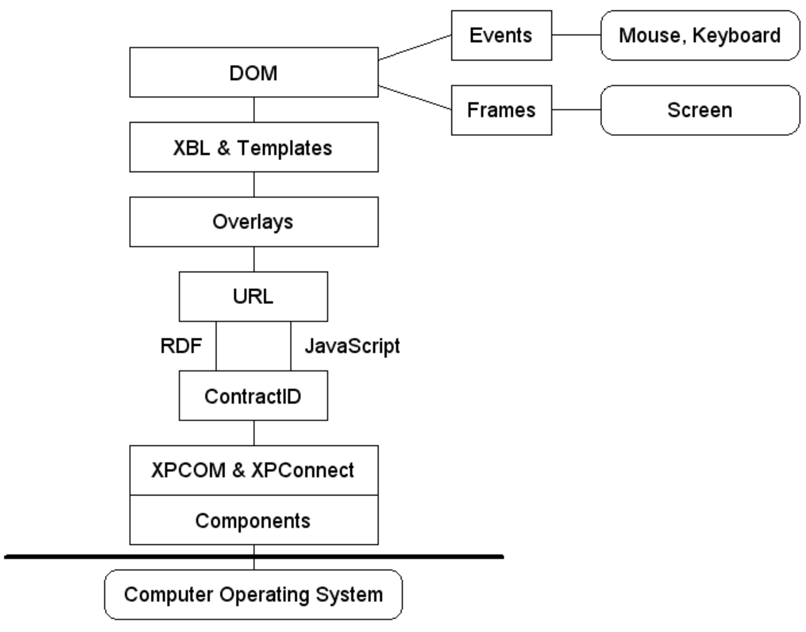
Pro zcela přenositelný kód je potřeba jistým způsobem adresovat požadované komponenty. Proto má každá komponenta díky XPCOM svůj jedinečný identifikátor, tzv. ContractID. Je ve tvaru `@mozilla.org/component-name;1`. Ten je mapován přes objekt `URL` do prostoru, ve kterém se již nachází aplikace napsané v jazyce JavaScript. Kromě tohoto propojení existuje ještě druhý typ spojení využívající identifikátorů RDF, které mapují URL na určitý záznam v otevřené RDF databázi.

Vrstva označená *Overlays* zodpovídá za rozšíření funkčností aplikací jako je Firefox nebo Thunderbird. Takto lze například přidat novou položku do menu, informaci do stavového řádku, vytvořit další panel atd. bez toho, aniž by se musel jakkoli měnit kód hostitelské aplikace. Všechny informace, jaké nové prvky mají být přidány, jaké jsou jejich vlastnosti a umístění, jsou součástí distribuce rozšiřující aplikace.

Následující vrstva je určena pro vytváření nových grafických komponent. Je založena na jazyku *XML Binding Language*, XBL, díky němuž lze skládat existující prvky uživatelského rozhraní do větších celků a definovat jim požadované chování. Příkladem je prvek `Tree`, ve kterém se zobrazuje historie navštívených dokumentů v prohlížeči Firefox.

Další část této architektury, *Templates*, řídí propojení některých prvků uživatelského rozhraní a zdrojů v RDF databázi. Takto lze bez jediného řádku kódu naplnit daty například strom zobrazující historii odkazů. Tato vrstva také umožňuje vícenásobné pohledy na data řízené společnou aktualizací. Jestliže je ve více oknech zobrazen obsah nějaké strukturované informace uložené v RDF databázi, její změnou v jednom okně může automaticky dojít k aktualizaci těchto dat i v ostatních oknech.

Nejvýše je položená vrstva podporující standardy W3C a zajišťující interakci s uživatelem. Na této úrovni jsou například načítány a za účasti nižších vrstev zpravovávány (X)HTML/XUL dokumenty, kaskádové styly, prováděny XSL transformace a další. Data, která mají být prezentována uživateli, jsou vykreslována renderovacím jádrem Gecko, akutální verze je 1.8.1 [32].



Obrázek 3.1: Zjednodušený pohled na architekturu platformy Mozilla

Kapitola 4

Tvorba uživatelského rozhraní

K programování uživatelského rozhraní je určen jazyk *XML User-Interface Language* (XUL). Pomocí textového zápisu lze hierarchicky skládat komponenty, které vytvoří požadovanou podobu GUI. Platforma nabízí velké množství uživatelských prvků:

- boxy, panely a mřížky, které lze použít pro pozicování dalších komponent,
- standardní ovládací prvky, jako jsou různé typy tlačítek, výběrových nabídek, prvků pro vložení a zobrazení textů a obrazků, apod.,
- prvek zobrazující stromovou strukturu v kombinaci s dalšími sloupci,
- realizace záložek s příslušejícími panely,
- hlavní i kontextové nabídky, nástrojová lišta, stavový řádek, bublinová nápověda,
- ovládací prvky pro realizaci průvodců,
- sofistikované komponenty, které dokáží zobrazit obsah webových stránek a případně jejich obsah vizuálně upravovat a
- další pomocné prvky pro připojení k datovému zdroji a podobně.

Obecná struktura zdrojového kódu okna aplikace vypadá následovně:

```
<?xml version="1.0"?>
<?xml-stylesheet href="chrome://global/skin/" type="text/css"?>
<?xml-stylesheet href="chrome://docwatcher/skin/main.css"
                 type="text/css"?>

<!DOCTYPE window [
  <!ENTITY % mainDTD SYSTEM "chrome://docwatcher/locale/main.dtd">
  %mainDTD;
]>
```

```
<window
 xmlns="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul">
  <script type="application/x-javascript" src="main.js"/>

  <commandset>
    <command id="cmd_newLink" />
  </commandset>

  <keyset>
    <key id="key_newLink" command="cmd_newLink" keycode="VK_INSERT" />
  </keyset>

  <menupopup id="cmenuFolders">
    <menuitem label="&cmd.newLink;" accesskey="&cmd.newLink.accesskey;"
              key="key_newLink" command="cmd_newLink"/>
  </menupopup>

  <description flex="1">Content</description>

<script type="application/x-javascript"><![CDATA[
]]></script>
</window>
```

Soubor s příponou .xul se skládá ze dvou částí. Nejprve se uvádí deklarační část, ve které se nachází seznam importovaných stylů a DTD souborů s řetězcovými konstantami. Následuje blok definující hierarchické uspořádání prvků uživatelského rozhraní.

4.1 Obsah XUL

Jak již bylo řečeno, XUL soubor je ve formátu XML, takže kromě daných pravidel syntaktického zápisu vyžaduje mimojiné i kořenový element. Podporováno je těchto pět:

- `window` – představuje standardní okno aplikace, které se neliší od ostatních oken daného operačního systému.
- `prefwindow` – speciální typ okna používaný pro nastavování parametrů aplikace. Umožňuje přímé napojení na datový zdroj.
- `dialog` – reprezentuje dialogové okno. Automaticky přidává tlačítka OK a Storno, přičemž chování chování a vzhled okna jsou upraveny tak, aby odpovídaly vlastnostem dialogových oken daného systému.
- `wizard` – typ okna usnadňující tvorbu průvodců.
- `overlay` – speciální element, který uvozuje obsah upravující některou část stávajícího uživatelského rozhraní platformy. Další informace jsou uvedeny níže.

Nezbytnou součástí kořenového elementu je také určení jmenných prostorů, které budou používány. Všechny XUL elementy leží v prostoru

`http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul`. Většinou je XUL dokument tvořen pouze XUL elementy, a proto se identifikátor jmenného prostoru neuvádí. Přesto ale existují situace, kdy je potřeba do XUL také vložit HTML elementy. V takovém případě je potřeba uvést jmenný prostor, ve kterém se tyto elementy nachází a určitý identifikátor:

```
xmlns:html="http://www.w3.org/1999/xhtml"
```

HTML elementy jsou pak vkládány běžným způsobem, jejich jméno je navíc uvozeno identifikátorem příslušného jmenného prostoru. Například `<html:p>`.

Elementy, které mohou být umístěny v daném kořenovém XUL elementu, lze rozdělit do dvou skupin.

4.1.1 Určení vzhledu

První skupina je tvořena značkami, které utvářejí vlastní uživatelské rozhraní, mají nějakou vizuální podobu. Tyto elementy jsou dobře popsány v [37].

Při programování složitějších aplikací však nezřídka nastávají situace, kdy je standardní funkčnost prvků nedostatečná a je potřeba ji modifikovat. V takovém případě je nezbytné pochopit jejich implementaci. Všechny XUL prvky jsou definovány pomocí jazyka *XML Bindings Language* (XBL), který umožňuje skládat již existující elementy uživatelského rozhraní do větších celků, definovat jejich chování, rozhraní pro komunikaci s vnějším světem a vzhled. Protože známe název elementu, jehož implementaci potřebujeme prozkoumat, vycházíme z CSS souboru získaného z adresy chrome://global/content/xul.css, jehož skutečné umístění se nachází v firefox_dir/chrome/toolkit.jar – content/global/xul.css, kde firefox_dir je adresář s nainstalovanou aplikací Firefox. V xul.css nalezneme požadovaný název elementu a jemu příslušející hodnotu vlastnosti `-moz-binding`, která obsahuje URL s XBL definicí požadované XUL značky. Pokud potřebujeme ještě podrobnější informace, které lze získat pouze ze zdrojových kódů platformy, je v [38] k příslušné značce uveden název odpovídající třídy. S těmito znalostmi lze ve vlastní aplikaci následně vytvářet zcela nové prvky na základě již existujících a nebo pouze upravovat chování a vzhled stávajících.

4.1.2 Podpora pro definici chování

Do druhé skupiny značek, které lze nalézt v XUL dokumentu, pak patří ty, jenž nedefinují vizuální podobu, avšak nějakým způsobem určují chování, jenž se přímo týká uživatelského rozhraní. Bývá totiž zvykem, že veškerý kód definující chování aplikace není míchán s vizuálním obsahem, máme snahu tyto dvě oblasti od sebe co nejvíce oddělit. Výjimkou může být krátký kód, který se například stará o správné pozicování prvků při změně velikosti okna. Tento kód je většinou zapisován do prvku `<script>...</script>` umístěného na konci XUL.

V aplikacích lze obvykle některé funkce vyvolat z více míst, například z hlavní nebo kontextové nabídky, nástrojové lišty nebo pomocí klávesové zkratky. Místa, kde všude lze zvolit danou funkci, by ale měla být propojena pouze na úrovni uživatelského rozhraní, ideálně přímo v XUL. Aplikačního programátora již nezajímá, jak lze aktivovat danou funkci, požaduje pouze jediný bod, který mu umožní připojit se a reagovat na tento požadavek, centrálně zakázat funkci nebo například upravit popisek. Stejně tak definice klávesových

zkratek by měla být umístěna pouze v XUL nebo souborech s řetězcovými konstantami (viz dále).

Pro tyto účely XUL nabízí několik elementů:

- `command` – definuje výchozí místo spouštění určité funkce. Prvky umožňující výběr této funkce (například `menuitem`, `button`) pak musí mít hodnotu atributu `command` rovnou hodnotě atributu `id` prvku `command`. Jakmile se vybere libovolným způsobem daná funkce, je vyvolána událost `oncommand` tohoto centrálního místa. Tato událost je právě místem propojení uživatelské rozhraní – aplikační úroveň.

- `broadcaster` – umožňuje určit jediné místo pro nastavování atributů. Jestliže potřebujeme například nastavovat hodnotu nějakého atributu pro více značek najednou, není potřeba nastavovat je odděleně, můžeme je propojit stejným způsobem jako u elementu `command`. Místo atributu `command` je však používán atribut s názvem `observes`. Jakmile je určitému elementu `broadcaster` přiřazena nebo změněna hodnota libovolného atributu mimo `id`, jsou okamžitě nastaveny patřičným způsobem i atributy připojených elementů. Tuto vlastnost má i předchozí značka `command`, u které se běžně používají atribut pro zakazování dané funkce (`disabled`) a její popisek (`label`).

- `key` – definuje klávesovou zkratku. Způsob napojení na danou funkci se provádí prostřednictvím již uvedeného atributu `command`. Bohužel zde existuje jeden z problémů, který není v žádných zdrojích jasně uveden. Pokud je jistý `<key>` propojen s `<command>`, nemůže být registrace události `oncommand` u elementu `command` provedena pomocí metody `addEventListener()`, je nutné využít `setAttribute()`. V opačném případě klávesová zkratka není funční.

Aby program mohl zachytávat reakce uživatele, jakou může být třeba poklepání na položku ve stromě, je potřeba přiřadit k požadované události, kterou platforma generuje, kód zpracovávající požadavek. Obecně je možné tento kód vepsat přímo do atributu, který odpovídá názvu události. Tento způsob však většinou vede k nepřehlednému a neudržovatelnému kódu. Zřejmě nejlepší řešení se skládá ze dvou kroků:

1. Hodnotu atributu `onload` kořenového elementu, s výjimkou `overlay`, nastavit na název inicializační funkce, která se nachází v příslušném souboru s JavaScript kódem. V případě, že se v této funkci alokují prostředky, které je nutné před ukončením práce uvolnit, provádí se tato činnost ve funkci, jejíž název se staticky definuje pomocí atributu `onunload`.

2. V inicializační funkci připojit k událostem jednotlivých prvků oblužné funkce pomocí volání metody `nsIDOMEventTarget::addEventListener()` (výjimku tvoří výše uvedený problém s elementem `key`).

S prvky uživatelského rozhraní se pracuje vždy až v okamžiku, kdy je vytvořen DOM daného XUL dokumentu, není tedy nutné zabývat se umístěním prvku `script` v XUL dokumentu. Pokud by totiž byl zpracováván nějaký kód na začátku dokumentu, který by pracoval s prvkem uživatelského rozhraní umístěného níže ve výpisu zdrojového kódu, prvek by nikdy nebyl nalezen, protože v daném okamžiku interpretace tohoto kódu ještě neexistuje.

Použití metody `nsIDOMEventTarget::addEventListener()` je výhodné ze dvou důvodů. Jednak lze určit fázi zachycení události a také můžeme k jedné události přiřadit

více obslužných funkcí. Ty lze dynamicky za běhu přidávat i odebírat. Přímý zápis kódu do atributu s názvem události toto neumožňuje.

4.2 Kaskádové styly

Kaskádové styly jsou umístěny v souborech s příponou .css. Určují vzhled a umístění jednotlivých elementů i celých hierarchií definovaných v XUL dokumentech. Protože je vykreslovací jádro pro XUL a HTML stejné, lze pro úpravu vzhledu používat běžné CSS vlastnosti a každému prvku přiřadit atribut `class`. Přesto však byly pro XUL přidány nové vlasnosti, jejich seznam s vysvětlením je uveden v [23].

Zde uveďme dvě vlasnosti, které mají speciální význam. První vlastnost je `-moz-binding`, tato již byla uvedena výše v souvislosti s XUL elementy. Spojuje název značky uváděné ve zdrojovém kódu XUL s XBL definicí.

Druhá vlastnost se již týká vzhledu, její název je `-moz-appearance`. Jestliže prvek tuto hodnotu nemá nastavenou, hodnota je rovna `none`, nebo je platforma provozována na jiném operačním systému než je Windows XP a vyšší a Mac OS X, pak se element chová a vypadá přesně takovým způsobem, jak jej určují XBL a CSS. Jestliže je ale tato vlastnost nastavena na neprázdnou platnou hodnotu, například `button`, chování a vzhled prvku je řízeno interně dle nastavené hodnoty. Důvodem je nová vlastnost vykreslovacího jádra Gecko, které v operačních systémech Windows XP a Mac OS X podporuje nativní vzhled a chování grafických komponent. Interně se vytvoří prvek uživatelského rozhraní operačního systému a ten je vložen na požadované místo v dokumentu, čímž se ignoruje nastavení vzhledu uvedeného v CSS. Seznam povolených hodnot se nachází v souboru nsCSSKeywordList.h. Protože většina XUL prvků má tyto vlastnosti implicitně nastaveny, změna vzhledu částí prvků není možná. Jedinou možností pak bývá vlastnost `-moz-appearance` vypnout a je na vývojáři, aby se vizuální podoba upravovaného XUL prvku co nejvíce blížila nativnímu vzhledu prvků daného operačního systému.

Kromě nových vlasností přidává Mozilla do CSS ještě jeden speciální zápis selektorů, který se uplatňuje při upravování vzhledu řádků, sloupců a obsahu buněk prvku `tree`. Klasickým způsobem lze měnit pouze vizuální podobu okrajů stromu a záhlaví sloupců. Důvodem je odlišný způsob reprezentace generovaného obsahu stromu, jenž nedovoluje použití atributu `class`. Ten nahrazuje nový atribut s názvem `properties`. Odpovídající nastavení v CSS pak vypadá následovně:

```
selektor_stromu treechildren::název_vlastnosti(hodnoty) {
  CSS definice
}
```

`selektor_stromu` je nepovinný selektor vybírající požadovaný strom v DOM, `treechildren` označuje regulární název elementu umístěného ve stromu, který je vždy přítomen a ohraničuje datovou oblast, `název_vlastnosti` je povinný a udává, která vnitřní oblast stromu bude upravována a seznam hodnot uvedených v závorce vybírá obsah, na něhož bude vzhled aplikován. Tyto hodnoty se vyhledávájí v hodnotách atributu `properties`. Kromě hodnot, které definuje uživatel, jsou přítomny i platformou definované, jako je například sudý/lichý řádek, seřazený sloupec a podobně. Kompletní seznam je uveden v [24].

Pro připojení kaskádových stylů k XUL nebo XBL je používána notace platná pro XML:

```
<?xml-stylesheet href="chrome://docwatcher/skin/main.css"
                 type="text/css"?>
```

Význam URL, který je zde uveden, vysvětluje kapitola 8. Zde je nutné všimnout si, že skutečná fyzická cesta, která by v tomto případě byla content/skin/classic/docwatcher/main.css neodpovídá výše uvedené. Důvodem je fakt, že platforma umožňuje vytvářet více typů vzhledu aplikace a přepínat se mezi nimi. Zde `classic` označuje název výchozího vzhledu, který bude aplikován, pokud není nalezen adresář s názvem aktuálně zvoleného skinu pro celou platformu.

Proto, aby každý XUL dokument vypadal stejným způsobem, je potřeba připojit systémovou složku se styly chrome://global/skin/, z níž platforma automaticky vybírá potřebné kaskádové styly. Bez těchto souborů by vhled aplikace připomínal obyčejnou neupravenou HTML stránku. Poté již mohou být připojeny další styly aplikace.

4.3 Definice externích řetězců

Jeden ze standardních požadavků na moderní aplikace je podpora více jazykových verzí a jejich snadná správa. Tohoto cíle by bylo problematické dosáhnout, pokud by řetězce byly v příslušných místech přímo vepsány, byly by tedy součástí zdrojového kódu. Z tohoto důvodu jsou pro účely lokalizace určeny dva typy souborů.

4.3.1 DTD entity

První typ externě deklaruje textové entity, které mohou být připojeny k defici typu dokumentu (DTD), soubory mají příponu .dtd. Příklad jednoho řádku z tohoto souboru:

```
<!ENTITY cmd.newLink "Nov&#x00FD; odkaz">
```

Za klíčovým slovem `ENTITY` je uveden název entity, pomocí něhož se v XUL dokumentu odkazujeme na skutečný obsah. Notace zápisu pro vložení obsahu textové entity při interpretaci dokumentu je `&nazev_entity;`. Způsob připojení souboru DTD s řetězci je ukázán v příkladu na začátku této kapitoly.

Vlastní textový obsah, který bude vložen do objektového modelu dokumentu (DOM), je uveden v uvozovkách za názvem entity. Jestliže využíváme znaky národní abecedy, tzn. jejich ASCII hodnota je vyšší než 127, musíme si uvědomit, že se také uplatňuje vliv kódování souboru. Pokud například ve Windows uložíme soubor, který bude obsahovat písmeno "Ř", bude implicitně uložen v kódování CP1250 a toto písmeno bude mít určitou hodnotu. Pokud stejný soubor vytvoříme a uložíme v Linuxu, bude použito pravděpodobně jiné kódování (ISO8859-2) a daný znak bude mít odlišnou hodnotu. Při načtení renderovacím jádrem Mozilly pak dochází k problému – je-li použit výchozí typ kódování uložený v nastaveních a pokud se neshoduje s kódováním, ve kterém byl soubor uložen, budou znaky s diakritikou zobrazeny chybně, nebo XML parser nebude schopen vůbec daný element načíst a interpretace končí chybou.

Nejspolehlivějším řešením je nahrazovat všechny znaky s ASCII hodnotou větší než 127 jejich hodnotami dle 16bitového Unicode. Tyto hodnoty se zapisují v hexadecimálním

tvaru, příklad je uveden výše, kde je nahrazován znak "ý". Pro texty v českém jazyce platí tabulky označené *Latin-1* a *Latin Extended A* nacházející se v [28].

4.3.2 Textové konstanty

Druhý způsob uchovávání řetězcových konstant mimo kód aplikace spočívá v použití souborů s příponou .properties. K extrakci požadovaného textu z tohoto zdroje je však nutné použít jeden z jazyků popisujících chování aplikace, viz. kapitola 5.1. Řádek vypadá například takto:

```
confirmDelete=Opravdu chcete vybran\u00E9 odkazy (celkem %S) smazat?
```

Každý řádek obsahuje identifikátor řetězce a příslušející text. Stejně jako u externích textových entit, i zde je nejlepší cesta zápisu znaků národní abecedy přes odpovídající 16bitovou Unicode hodnotu. Dále zde fungují escape sekvence známé z jazyka C, což je také jediný způsob, jak vložit do textu znak nového řádku.

A konečně, řetězce mohou být parametrizované, tj. na místa, která jsou označena pomocí znaku procento následovaného řetězcem reprezentujícím požadovaný formát, se vloží vstupní hodnoty. Podporované formáty zřejmě odpovídají formátům funkce `printf` jazyka C, i když tato skutečnost není v žádné oficiální dokumentaci explicitně uvedena. Na stránkách [37] je k elementu `stringbundle` uveden pro vložení textového řetězce pomocí jazyka JavaScript parametr `%s`, což je ovšem chyba. Metoda třídy `nsStringBundle` určená pro analýzu řetězcových konstant nepřímo volá statickou metodu `nsTextFormatter::dosprintf`, ve které lze najít podporované formáty. Z tohoto kódu je patrné, že `%s` je pouze pro 8bitové ASCII řetězce, zatímco `%S` je určeno pro 16bitové Unicode řetězce. Protože JavaScript pracuje pouze s 16bitovými znaky, správný identifikátor formátu je `%S`, `%s` způsobí chybný výstup.

4.3.3 Volba aktivního jazyka

Jazyk, ve kterém jsou řetězce psány, není explicitně uveden v souborech, je však dán jménem adresáře, ve kterém se tyto soubory nacházejí. Název je ve tvaru ln nebo ln-CT, kde ln je kód jazyka malými písmeny dle ISO 639.2 a CT je kód země velkými písmeny dle ISO 3166 [8]. Nadřazený adresář těchto jazykových adresářů se pak jmenuje locale. Pro Českou republiku je kód cs-CZ.

Je důležité všimnout si, že ve zdrojovém kódu uvedeném na začátku této kapitoly není v cestě chrome://docwatcher/locale/main.dtd konkrétní název jazykové varianty. Obecně všechny soubory, které se nacházejí v podhierarchii adresáře locale, nemají uvedený kód jazyka. Mozilla sama tuto cestu upravuje a doplňuje na základě nejlepší shody s aktuálním nastavením operačního systému. Jestliže nalezne adresář s jazykovým kódem ekvivalentním kódu, který poskytuje operační systém, jsou použity řetězce z tohoto adresáře. V případě neshody je vyhledán adresář en-US. Jestliže ani ten neexistuje, je vybrán první zaregistrovaný adresář v průběhu procesu instalace rozšíření, viz. kapitola **??**.

Kapitola 5

Programování chování

Před vlastním obsahem této kapitoly je vhodné zmínit dostupné zdroje, odkud lze získávat mnoho informací o API, funkčních postupech i chybách platformy. Při vývoji musíme brát totiž ohled na to, že se jedná o otevřenou aktivně se rozvíjející aplikační platformu, na jejímž vytváření se podílí hodně programátorů z celého světa a ne každý dodržuje všechna pravidla zápisu zdrojových kódů, apod. Z tohoto důvodu jsou pak některé části dokumentace dostupné v [37] neúplné, v některých místech popis zcela chybí. V takovém případě lze zkoumat přímo zdrojové kódy, které jsou přístupné na adrese `http://lxr.mozilla.org`, je zde i funkce vyhledávání. Bohužel zde však nejsou zahrnuty zdrojové kódy aplikace Firefox 2.0, a proto je potřeba stáhnout si zdrojové kódy. Bližší informace jsou uvedeny v [17]. Jen upozorňuji, že rozbalený archiv zabírá na disku přes 250MB, obsahuje více než 40000 souborů a nalezení potřebné informace může být časově dosti náročné.

Dalším zdrojem informací je samotná instalace zvoleného produktu Mozilla, hlavně pak archivy v adresáři chrome. Ty jsou potřeba především pro nalezení idenfikátorů za účelem rozšiřování stávajících aplikací pomocí *overlays*. Jestliže vytváříme aplikaci, jejíž nějaká funkce je podobná činnosti již existujícího rozšíření nebo části produktu Mozilla, je občas nutné přistoupit k reverznímu inženýrství a zjistit, jakým způsobem fungují. Je totiž pravděpodobné, že se vývojáři potýkali se stejnými problémy. Dostupná rozšíření lze získat například z adresy `https://addons.mozilla.org`. Zbývá upozornit na adresu `https://bugzilla.mozilla.org`, na níž je dostupná Bugzilla, což je databáze s nalezenými chybami platformy Mozilla – ne vždy je totiž chyba v kódu vývojáře.

5.1 Programovací jazyky

Pro programování aplikací jsou na této platformě primárně dostupné dva jazyky. Prvním je C++, kterým lze implementovat požadované chování na úrovni komponent. Nevýhodou je, že tento kód bude po překladu závislý na HW architektuře a případně i na operačním systému. Hodí se tam, kde potřebujeme zrychlit výpočet, skrýt nějaké implementační detaily nebo využít přímého napojení na kokrétní operační systém.

Druhým jazykem je JavaScript, který je nejčastěji využíván. Jeho použití je vpodstatě nezbytné i v předchozím případě, kde je potřeba vytvořené komponentě předat řízení. Vlastnosti jazyka shrnuje následujících několik bodů:

- Vychází ze standardu ECMAScript (ECMA-262 Edition 3) a přidává některá rozšíření. Aktuální verze je 1.7 [18].

- Je to skriptovací jazyk, což znamená, že zdrojový kód se syntaxí podobnou syntaxi jazyka C, je při běhu programu čten a interpretován. Tento fakt má tři nevýhody: Je snížena rychlost provádění kódu, což je znatelné při složitějších výpočtech. Ladění je obtížnější díky toho, že chyby jsou většinou odhaleny až v okamžiku, kdy mají být zpracovány. Poslední nevýhoda se týká otevřené podoby kódu – ten je snadno čitelný, pokud není z nějakého důvodu možné použít nástroj pro zatemnění kódu, tzv. *obfuscator*.
- Typy proměnných se explicitně neuvádějí, jsou zjišťovány za běhu. Objekty se vytvářejí pomocí operátoru `new`, destruktory zde neexistují. O uvolňování paměti se automaticky stará *Garbage collector*.
- Podporováno je objektově orientované programování, u tříd lze rozlišovat privání a veřejné atributy a metody. Lze využívat polymorfismu (volání jedné konkrétní metody objektů různých tříd) a jednoduchá dědíčnost (odvozování třídy od jiné třídy a přebírání jejich veřejných atributů a metod). Objekt a asociativní pole si vpodstatě odpovídají.
- JavaScript umožňuje i funkcionální programování.
- Je možné také využívat iterátory, getter/setter metody známé např. z jazyka C#, funkce může vracet více hodnot najednou a další vlastnosti, které se stále rozvíjí.

V poslední době se také vyvíjí dva nové projekty, které se snaží propojit jádro platformy s jazyky Python a Java. Názvy příslušných projektů jsou PyXPCOM[1] a JavaXPCOM[2].

5.1.1 Objektové programování v jazyce JavaScript

Jak již bylo uvedeno výše, JavaScript podporuje objektově orientované programování, čímž lze dosáhnout lepší struktury a udržovatelnosti kódu. Příklad definice třídy vypadá následovně:

```
function classA(parameter)
{
  const _self = this;
  var private_attribute = parameter;
  var private_method = function() {
    _self.public_attr1 = null;
  }
  this.public_attr1 = null;
  this.public_method1 = function() {}
}
```

[1] http://developer.mozilla.org/en/docs/PyXPCOM
[2] http://developer.mozilla.org/en/docs/JavaXPCOM

```
classA.prototype.__defineGetter__("public_attr2", function()
  { return this.public_attribute; });
classA.prototype.__defineSetter__("public_attr2", function(val)
  { this.public_attribute = val; });
classA.prototype = new function() {
  var private_static_attr = null,
  this.public_attr2 = null;
  this.public_method2 = function() {}
}
```

Funkce i třídy sdílejí v jazyku JavaScript jediné klíčové slovo `function`. V příkladu je definována třída `classA`, tělo funkce představuje její konstuktor, jehož parametrem mohou být libovolné proměnné. Vlastnosti, které jsou dostupné vně třídy, mají před názvem uvedeno klíčové slovo `this` oddělené tečkou. V tomto případě se jedná o všechny položky s prefixem `public`. `this` funguje stejným způsobem jako v jiných objektových jazycích, označuje tedy referenci na aktuální objekt. Dále se zde nachází atribut a metoda s prefixem `private`. Ty jsou přístupné pouze v rámci těla konstruktoru a protože jsou dostupné po celou dobu existence objektu, lze je považovat za privátní. V Mozille však existuje chyba, která způsobuje, že v těle privátních metod není přes klíčové slovo `this` dostupná reference na aktuální objekt. Reference je nastavena na globální objekt, který je většinou `window`. Způsob řešení tohoto problému spočívá v zavedení konstantní proměnné, která udržuje referenci na aktuální objekt a v těle privátních metod se pak lze odkázat pomocí této proměnné na obsah objektu.

V kódu se nachází statická vlastnost třídy nazvaná `prototype`. Jedná se o systémem vytvořený objekt dostupný v každé funkci, který lze použít pro přidávání/odebírání/úpravu veřejných atributů a metod. Pro odebírání vlastností slouží speciální operátor `delete` [6]. Díky této vlastnosti můžeme rozšiřovat funkčnost existujících tříd, ale také všech již vytvořených instancí dané třídy. Tak lze například přidat metody do systémového objektu `Date`.

Metody `__defineGetter__()` a `__defineSetter__()` jsou dostupné pouze v rámci objektu `prototype` a umožňují vytvářet gettery a settery. V příkladu je uvedena vlastnost `public_attr2`, který se syntaxí a použitím tváří jako atribut, avšak při čtení jeho hodnoty je volána metoda getteru a při zápisu nové hodnoty je volána metoda setteru. Vynecháním definice setteru můžeme získat atribut určený pouze pro čtení, který je dostupný vně objektu.

Protože má každá třída jediný statický objekt `prototype`, je všemi instancemi této třídy sdílená. Atribut, který je zde definován a není uvozen klíčovým slovem `this`, je pak interně dostupný všem těmto instancím. Představuje tedy privátní statický atribut. Největší problém uvedeného zápisu je fakt, že privátní atributy definované v konstruktoru a privátní statické atributy obsažené v objektu `prototype` nejsou nikdy viditelné z jediné metody.

V příkladu je dále znázorněn způsob vytváření instance anonymní třídy. Znamená to, že zde existuje jediný objekt definované třídy, která však není dostupná a nelze vytvořit její další instanci. Následující kódy uvádějí dva způsoby vytvoření diskutovaných typů instancí:

```
var object1 = new function()
{
  var private_attr = null;
  var private_method = function() {}
  this.public_attr = null;
  this.public_method = function() {}
  this.__defineGetter__("public_attr2", function()
    { return null; });
}

var object2 = {
  public_attr: null,
  public_method: function() {},
  get public_attr2() { return null; }
}
```

Druhý způsob je jednodušší pro zápis, syntaxe getterů a setterů je zde více zjednodušena. Nevýhodou může být nemožnost definovat privátní vlastnosti, protože všechny identifikátory jsou automaticky viditelné vně objektu.

V jazyce JavaScript je možná také jednoduchá dědičnost. Jestliže využijeme předchozího příkladu, kde je uvedena třída `classA`, pak její potomek, třída `classB`, bude zapsána takto:

```
function classB()

  classA.apply(this);
  // ... zbývající tělo konstruktoru ...

classB.prototype.__proto__ = classA.prototype;
/* Méně vhodné:
classB.prototype = new classA();
classB.prototype.constructor = classB.
*/
```

Dědění probíhá ve dvou krocích. Nejprve je proveden řádek za definicí třídy `classB`. Jak již víme, `prototype` je objekt sdílený všemi instancemi dané třídy, avšak tento identifikátor je dostupný pouze třídám. Instance mohou využívat platformou definovaný atribut `__proto__`, do něhož se po zavedení objektu do paměti, ale ještě před voláním konstruktoru, automaticky přiřadí reference na objekt `prototype` [5]. Jestliže interpret jazyka JavaScript narazí při provádění nějakého kódu na místo, kde se pracuje s vlastností instance objektu, tzn. volá se metoda instance nebo se čte/zapisuje z/do atributu hodnota, je neprve zjištěno, zda tato vlastnost existuje přímo v daném objektu. Jestliže ne, vyhledává ji v objektu `__proto__` dané instance. Pokud ani zde není, vyhledává se v objektu `__proto__` příslušného objektu `__proto__`. Dochází tedy k rekurzivnímu procházení až do okamžiku, kdy je vlastnost nalezena nebo je hodnota `__proto__` rovna `null`. Uvedený zápis v příkladu tedy říká, že vlastnost atributu `__proto__` všech instancí třídy potomka bude obsahovat referenci na objekt `prototype` rodičovské třídy. Jestliže pak pracujeme s vlastností instance třídy potomka a ta zde není uvedena, vyhledává se ve vlastnostech objektu `prototype` třídy rodiče.

Zde je ještě potřeba uvést, jakým způsobem se interně provádí přiřazování nových hodnot vlastnostem. Tato problematika je detailně i s příklady rozebrána v [27]. Jestliže je například definována hodnota veřejného atributu v objektu `prototype` rodičovské třídy, pak při jejím čtení v potomkovi třídy je vrácena hodnota, která uložená v rodiči. Při zápisu nové hodnoty tohoto atributu ale již dochází k vytvoření nového atributu v paměťovém prostoru potomka. V tomto okamžiku tedy existují dvě hodnoty, přičemž ta v rodiči je aktuálně neviditelná, protože při vyhledávání je nalezena a vrácena hodnota umístěná v potomkovi. Jestliže byl tento atribut definován v `prototype`, lze pro získání hodnoty atributu instance přímého rodiče provést následující kód:

```
var valueOfParent = classAInstance.__proto__.__proto__.public_attr2;
```

Je-li atribut definován v těle konstruktoru rodičovské třídy, nelze tuto hodnotu jakýmkoli způsobem získat. Stejné tvrzení pak platí v případě volání metod.

V komentáři je uveden ještě jeden způsob tvorby vazeb dědičnosti. Ten není doporučován, protože v okamžiku načítání scriptů do XUL nebo HTML dokumentu dochází k vytváření instance rodičovské třídy, která však může požadovat parametry konstruktoru a ty v tomto okamžiku ještě nemusí být dostupné. Druhý problém souvisí se zdržením zavádění následných částí dokumentu způsobeného vytvářením instance. Při použití tohoto způsobu je vhodné nastavit zpět hodnotu atributu `constructor`, který je automaticky vytvářen platformou, na referenci na funkci, která vždy představuje konstruktor dané třídy.

Druhým nezbytným krokem je zavolání systémové metody `apply()`, jenž je dostupná v každé funkci. Prvním parametrem je vždy reference na objekt, v jehož kontextu má být spuštěna, a následuje pole s parametry předávaných rodičovské třídě. `call()` se chová stejně, avšak místo pole parametrů se uvádějí jednotlivé parametry zvlášť. Tyto metody spustí kontruktor dané třídy, avšak reference udané klíčovým slovem `this` jsou nahrazeny referencí, která je předávána jako první parametr. To znamená, že všechny veřejné atributy a metody, které se nacházejí v konstruktoru rodiče budou zkopírovány do paměťového prostoru potomka. Pokud by k volání `apply()` nedošlo, potomci rodičovské třídy by neměli položky z konstruktoru k dispozici, protože tyto objekt `prototype` nezahrnuje. Z tohoto faktu také vyplývá, že vývojář má dvě možnosti navrhování tříd a je na něm, aby zvolil vyhovující. Pokud bude využívat lokální proměnné a funkce v těle konstruktoru, se kterými bude pracovat jako s privátními položkami třídy, může sice lépe ukrývat informace před okolním světem, avšak interpret jazyka vytvoří pro každou novou instanci třídy znova všechny metody nacházející se v těle konstruktoru, takže se zvyšují nároky na paměť a rychlost vytváření objektů klesá. Druhá možnost je opačná – do těla konstruktoru umístit pouze atributy a všechny metody definovat do objektu `prototype`. Tím sice příjdeme o možnost ukrývání položek, avšak odstraníme dříve uvedené nevýhody. V tomto případě může pomoci zavedení vlastních pravidel pro označování položek s různými úrovněmi přístupu (*private*, *protected*, *public*) a ty pak dodržovat.

Pro přístup k položkám třídy se používá standardní tečková notace známá např. z jazyka C. Díky to, že jazyk JavaScript chápe objekty jako asociativní pole, ve kterém je klíč tvořen názvem položky, můžeme pro přístup použít i operátor pole. Jestliže do tohoto pole zapíšeme hodnotu na místo dané klíčem, jenž neexistuje, vytvoří se nová vlastnost příslušné instance třídy. Následující dva řádky jsou tedy identické:

```
new classA().public_method()
new classA()["public_method2"]()
```

Nová instance třídy vzniká operátorem `new`. Součástí interpretu jazyka JavaScript je *garbage collector*, který se automaticky stará o uvolňování paměti, pokud počet referencí na daný objekt klesne na nulu. Neexistují zde destruktory, operátor `delete` ale může být použit pro odstranění elementu z pole, vlastnosti z objektu nebo dokonce celého objektu [6]. Další operátory pro účely objektově orientovaného programování jsou tyto: `in` může být použit pro testování existence určité vlastnosti v instanci třídy, `instanceof` udává, zda je objekt na levé straně operátoru instancí třídy uvedené na straně pravé a operátor `typeof` vrací pro třídu řetězec `"function"` a pro instanci třídy `"object"`.

5.2 XPCOM

Obecně programování chování aplikace probíhá stejně, jako u (X)HTML stránek. Objekty `window`, `document` a další běžné jsou dostupné standardní cestou, i přes příslušná rozhraní. S výjimkou prvků uživatelského rozhraní, které jsou ve speciálním režimu budovány přímo z datového RDF zdroje, je možné se všemi prvky v okně manipulovat prostřednictvím DOM operací a na reakce uživatele jsou používány události. Uvedené možnosti jsou však pro implementaci pokročilých aplikací nedostatečné. V běžném JavaScriptu nelze pracovat se soubory na disku, spouštět další aplikace, přistupovat k oknům prohlížeče, vytvářet libovolná síťová spojení a podobně.

Platforma Mozilla proto poskytuje mechanismy, které umožňují XUL aplikacím připojit se k požadovaným komponentám a voláním jejich služeb provádět potřebné operace. Získání reference na požadovaný objekt je provedeno jedním z následujících příkladů volání, jehož výběr je závislý na typu komponenty:

```
Components.classes["@mozilla.org/file/directory_service;1"]
          .getService(Components.interfaces.nsIProperties)
```

nebo

```
Components.classes["@mozilla.org/rdf/container;1"]
          .createInstance(Components.interfaces.nsIRDFContainer)
```

První případ pouze získá referenci na instanci třídy, která je singleton. V celém paměťovém prostoru platformy je tedy pouze jediná instance této služby. Druhý případ již vytváří novou instanci požadované třídy. V hranatých závorkách se uvádí ContractID požadované komponenty, což je jedinečný řetězec, kterým se každá komponenta platformy identifikuje. Parametrem metod `getService()` a `createInstance()` je pak rozhraní, které chceme získat. Volání operací, které získaná komponenta nabízí, se již provádí klasickým způsobem volání metody. Poznamenejme, že ne všechny metody lze volat, některé jsou v dokumentaci [37] označeny `[noscript]` a tyto lze spouštět pouze z jazyka C++ na úrovni komponent.

Před uvedením následujících částí je potřeba také upozornit na dokument [29], s jehož obsahem je vhodné seznámit se, protože obsahuje informace týkající se správných postupů používání XPCOM komponent tak, aby byly eliminovány úniky paměti.

5.2.1 Možné problémy

Při práci s komponentami se můžeme setkat s několika problémy. První vzniká u metod, které mají u nějakého parametru uveden výraz `out` a jeho typ je základní, neobjektový (viz [12], seznam *Built-in Types*). Znamená to, že daná proměnná vyžaduje volání odkazem, tj. v těle metody bude nastavena na novou hodnotu. Řešením je předání objektu, jehož atribut `value` bude po vykonání metody obsahovat návratovou hodnotu.

Druhý problém nastává u parametrů některých metod, které vyžadují objekt s definovaným rozhraním. Jestliže bychom například chtěli použít metodu `Serialize()` komponenty s rozhraním `nsIRDFXMLSource`, musíme jí předat objekt s rozhraním `nsIOutputStream`. V tomto případě je na vývojáři, aby objekt vytvořil a implementoval požadované rozhraní, nelze jej totiž získat pomocí volání `getInstance()`:

```
var serializer = Components.classes["@mozilla.org/rdf/xml-serializer;1"]
           .createInstance(Components.interfaces.nsIRDFXMLSerializer);
serializer.init(ds);
var outputStream = {
  data: "",
  close: function() {},
  flush: function() {},
  write: function (buffer,count) {
    this.data += buffer;
    return count;
  },
  writeFrom : function (stream,count) {},
  isNonBlocking: false
};
serializer.QueryInterface(Components.interfaces.nsIRDFXMLSource)
          .Serialize(outputStream);
```

Protože jsou rozhraní uspořádána hierarchicky, tj. podporují pouze jednoduchou dědičnost, mají všechny jediného předka `nsISupports`. Toto rozhraní implementuje tři metody, z toho dvě jsou dostupné pouze v jádře pod vrstvou XPCOM a slouží pro zvýšování/snižování počítadla referencí na daný objekt. Poslední metoda `QueryInterface()` je již viditelná a umožňuje získat určité rozhraní komponenty, což by mohlo být částečně přirovnáno k přetypování objektu. V předchozím příkladu je voláním `createInstance()` vytvořena komponenta pro serializaci a vráceno jedno z rozhraní, které implementuje. Jestliže je následně potřeba zavolat metodu jiného rozhraní stejné komponenty, musí být před vlastním voláním toto rozhraní obdrženo prostřednictvím metody `QueryInterface()`.

Jestliže potřebujeme nějakou hodnotu nebo objekt předat jiné části kódu pomocí XPCOM nebo si je někam uložit a později si vyzvednout, zjistíme, že potřebná metoda rozhraní má parametr typu `nsISupports`. V případě, že se jedná o hodnotu, nelze ji přímo předat, ale musí být zabalena do objektu. Zde jsou dvě možnosti – buď vytvoříme běžný objekt v JavaScriptu a zvolený atribut nastavíme na požadovanou hodnotu, nebo vytvoříme instanci komponenty odpovídajícího typu s prefixem ContractID `@mozilla.org/supports-` (viz rozhraní `nsISupportsPrimitive`). Druhá možnost je pomalejší, protože dochází ke komunikaci s XPCOM. V tomto okamžiku už by bylo

možné objekt předat, protože XPConnect automaticky převede JavaScript objekt na `nsISupports`. Jakmile získáme objekt zpět, vrátí se jako `nsISupports`. Jestliže jsme předtím vytvářeli komponentu s rozhraním `nsISupportsPrimitive`, zjistíme dle atributu `type`, jaký datový typ tato komponenta představuje, a dle získané hodnoty vyžádáme příslušné rozhraní. Problém ovšem nastává, pokud jsme předávali JavaScript objekt. Zde neexistuje cesta, jak "přetypovat" získaný objekt na původní a tím se dostat k uloženým hodnotám. Proto musí do objektu přidán atribut `wrappedJSObject`, který obsahuje referenci na tento objekt. Poté, co je objekt vrácen z XPCOM, lze k původním datům přistupovat prostřednictvím vlastnosti `wrappedJSObject`. Implementační detaily jsou uvedeny ve zdrojových kódech platformy v souboru nsIXPConnect.idl a další informace jsou v [15, 35].

5.2.2 Vytváření komponent

Vytváření nových komponent může je výhodné zejména tehdy, pokud aplikace potřebuje vlastní služby, tj. singletony v paměťovém prostoru platformy, objekty, které budou pracovat s těmito službami nebo chceme rozšířit stávající komponenty. Programování komponent v jazyce C++ detailně popisuje [2]. Zde se opět zabýváme pouze komponentami v jazyce JavaScript.

Dle [13] bývá kód komponenty obvykle stejný a liší se jen v hodnotách konstant a definici třídy, která implementuje požadované chování:

```
const Cc = Components.classes;
const Ci = Components.interfaces;
const Cr = Components.results;

const CLASS_ID = Components.ID("{1E60EA22-D89D-7FCB-F78A-9FAB5AD6A01F}");
const CLASS_NAME = "DocWatcher: Controller service.";
const CONTRACT_ID = "@docwatcher.jk.cz/controller;1";

function Controller()
{
  /* Implementace chování komponenty... */
}
Controller.prototype.QueryInterface = function(IID) {
  if (!IID.equals(Ci.dwIController) &&
      !IID.equals(Ci.nsISupports))
    throw Cr.NS_ERROR_NO_INTERFACE;

    return this;
}

const Factory = {
  createInstance: function (outer, IID) {
    if (outer != null) throw Cr.NS_ERROR_NO_AGGREGATION;
    return (new Controller()).QueryInterface(IID);
  }
};
```

```
const Module = {
  registerSelf: function(compMgr, fileSpec, location, type) {
    compMgr = compMgr.QueryInterface(Ci.nsIComponentRegistrar);
    compMgr.registerFactoryLocation(CLASS_ID, CLASS_NAME, CONTRACT_ID,
                                    fileSpec, location, type);
  },
  unregisterSelf: function(compMgr, location, type) {
    compMgr = compMgr.QueryInterface(Ci.nsIComponentRegistrar);
    compMgr.unregisterFactoryLocation(CLASS_ID, location);
  },
  getClassObject: function(compMgr, CID, IID) {
    if (!IID.equals(Ci.nsIFactory)) throw Cr.NS_ERROR_NOT_IMPLEMENTED;
    if (CID.equals(CLASS_ID)) return Factory;
    throw Cr.NS_ERROR_NO_INTERFACE;
  },
  canUnload: function(compMgr) {
    return true;
  }
};
function NSGetModule(compMgr, fileSpec) {
  return Module;
}
```

Při inicializaci kódu platformy Mozilla jsou prohledávány adresáře components/ všech nainstalovaných rozšíření, zda se zde nachází nějaké soubory s JavaScriptem, které exportují symbol `NSGetModule()`. Jestliže ano, je pomocí této funkce získána reference na objekt s rozhraním `nsIModule`. Pokud nemá platforma ve své databázi k tomuto souboru žádný záznam, zřejmě se jedná o novou komponentu a je spuštěna metoda `registerSelf()`, která provede zaregistrování komponenty. K tomu jsou potřeba mimojiné tři řetězce – CLASS_ID udává jedinečný identifikátor dle *Universally Unique Identifier* (UUID) ve standardním hexadecimálním formátu, CONTRACT_ID definuje ContractID, kterým bude možné odkazovat se na tuto komponentu pomocí pole `Components.classes` a CLASS_NAME uchovává název komponenty.

V okamžiku, kdy je poprvé žádaná instance třídy prostřednictvím metod `createInstance()` nebo `getService()`, platforma volá metodu `getClassObject()`, která vrací objekt s rozhraním `nsIFactory`, a následně je pomocí něj vytvořena instance třídy, která provádí požadovanou činnost komponenty. Pokud je znovu spuštěna metoda `createInstance()`, proces v tomto odstavci se opakuje. Volání `getService()` naproti tomu ihned vrací poprvé vytvořenou instanci, která byla před návratem uložená do vyhrazeného prostoru uvnitř platformy. Nakonec je potřeba ještě uvést, že každá třída, která implementuje chování komponenty, musí exportovat metodu `QueryInterface()`. Jestliže její parametr odpovídá rozhraní, které tato komponenta implementuje, vrací referenci na instnci třídy. V opačném případě požadované rozhraní není implementováno a je vyhozena výjimka.

Aby byly veřejné položky instance třídy dostupné vně komponenty, musíme dále vytvořit rozhraní. Jazyk, který jej popisuje, se nazývá *Interface Description Language* (XPIDL) [36].

Rozhraní může být definováno například takto:

```
#include "nsISupports.idl"

[scriptable, uuid(1E60EA22-D89D-7FCB-F78A-9FAB5AD6A01F)]
interface dwIController : nsISupports
{
  const short NULL_ITEM = 0;
  readonly attribute nsISupports data;
  wstring getLngStringWA(in string str,
                         [array, size_is(paramsCount)] in string params,
                         in unsigned long paramsCount);
};
```

Nejpve je potřeba připojit definice všech rozhraní, která se zde vyskytují. V hranaté závorce musí být vždy klíčové slovo `scriptable` z důvodu výběru implementačního jazyka komponenty a uvedené UUID se musí shodovat s `CLASS_ID`. Následuje již samotná definice rozhraní, přičemž se uvádí název rozhraní a nepovinný název jeho předka. V případě, že rodič rozhraní není uveden, je implicitní `nsISupports`. V těle lze definovat konstanty, které mohou být pouze typu `short` nebo `long`. Řetezcové konstanty uvedeným způsobem definovat nelze, musí být implementovány jako atribut určený pouze ke čtení. Metody rozhraní mají svůj název, u každého parametru je uveden směr, povolené hodnoty jsou `in`, `out`, `inout`. V tomto příkladu je uvedeno použití klíčového slova `array`, které v dokumentaci [36] chybí. Toto klíčové slovo umožňuje danému parametru předat pole jazyka JavaScript vytvořené pomocí `new Array()` nebo `[]`. Musí být však uvedena také jeho velikost jak pomocí hodnoty klíčového slova `size_of`, tak i jako parametr metody. Seznam typů a jejich ekvivalentů v C++, je uveden v [12].

Pokud vývojář zná objektové jazyky jako je např. C++ nebo Java, je zvyklý na to, že děděním přebírá potomek nejen rozhraní, ale také metody a atributy, jenž jsou součástí rodičovské třídy. V případě kombinace tříd v JavaScriptu a IDL k tomu ale nedochází. Při dědění v IDL sice komponenta přebírá vlastnosti rozhraní rodiče, avšak daná třída musí obsahovat všechny veřejné metody a atributy, které se nacházejí nejen v rozhraní dané komponenty, ale i všech jeho předcích (kromě `nsISupports`). V XUL dokumentu by se tato situace řešila snadno – i když by byla každá třida umístěná ve svém souboru, použitím elementů `script` by došlo k jejich načtení a vytvoření vazeb dědičnosti mezi nimi. Na úrovni komponent ale tento mechanismus nefunguje a bylo by nutné celou hierarchii tříd umístit do jediné komponenty, což je velice nevýhodné. Z tohoto důvodu lze použít následující úsek kódu:

```
function appendBaseClasses() {
  const BASE_CLASS_PATH = "chrome://docwatcher/content/base/Base.js";
  const jsloader = Cc["@mozilla.org/moz/jssubscript-loader;1"]
                   .getService(Ci.mozIJSSubScriptLoader);
  var classHolder = {};
```

```
  jsloader.loadSubScript(BASE_CLASS, classHolder);
  BaseClass = classHolder.BaseClass;

  Controller.prototype.__proto__ = BaseClass.prototype;
}
```

Tato funkce by měla být volána v metodě `nsIFactory::createInstance()` a to pouze při prvním vyžádání instance třídy. Kód načte obsah souboru umístěného na URL daného konstantou `BASE_CLASS_PATH` a jednotlivé položky umístí do objektu `classHolder`. Jestliže je v tomto souboru definována třída `BaseClass`, pak ji získáme z `classHolder` pod tímto jménem. Poslední řádek již provede vytvoření vazby rodič – potomek. Díky tohoto způsobu lze mít všechny třídy implementující požadované chování komponent ve zvláštních souborech, bázové třidy jsou umístěny v chrome, odkud je lze snadno získat, a všechny třídy, tj. bázové i jejich potomci mají definované rozhraní.

Soubor, který obsahuje definici rozhraní, má obvykle příponu .idl. Před použitím je však potřeba přeložit jej do binární podoby, vznikne soubor s příponou .xpt. Nástroj xpidl určený k překladu je součástí vývojového balíku Gecko SDK [10]. V uvedeném zdroji je i odkaz na jeho stáhnutí. V operačním systému Windows však pravděpodobně dojde k chybě způsobené nepřítomnou dynamickou knihovnou libIDL-0.6.dll. Získat ji lze z adresy http://ftp.mozilla.org/pub/mozilla.org/mozilla/source/wintools.zip [9].

Po instalaci Gecko SDK je překlad vybraného rozhraní spuštěn následujícím příkazem [11]:

```
{sdk_dir}/bin/xpidl -m typelib -w -v -I {sdk_dir}/idl -e Controller.xpt
   Controller.idl
```

V adresáři idl/ však nejsou všechna rozhraní platformy, ale pouze ta s označením `FROZEN`. Tato rozhraní už by se v budoucnu neměla měnit. Při vývoji aplikací jsou však většinou vyžadovány i další rozhraní a je nutné je ze zdrojových kódů Firefoxu zkopírovat do uvedeného adresáře idl/.

5.2.3 Bezpečnost

Protože lze XUL aplikaci spouštět i na dálku tak, že je umístěna na vzdáleném serveru jako běžné HTML dokumenty, mohlo by být díky nepozornosti uživatele používajícího například Firefox stažen do počítače kód, který by měl kompletní přístup k celému systému. Proto platforma Mozilla rozeznává přístupová práva. Jestliže se XUL dokument nenalézá v chrome lokaci, tedy nebyl nainstalován jako rozšíření aplikace, je veškerý přístup k XPCOM zakázán a lze pracovat pouze s DOM. Aplikace si však může o tato práva zažádat a pokud uživatel odsouhlasí okno s žádostí, může aplikace dále pokračovat. Vyžádání potřebných práv se provádí například následujícím voláním:

```
window.netscape.security.PrivilegeManager.enablePrivilege(
  "UniversalXPConnect");
```

Parametr metody `enablePrivilege` je řetězec složený z názvů požadovaných bezpečnostních práv, které se oddělují mezerou. Zde je uveden název důležitého

přístupového práva, jenž umožňuje přístup k XPCOM. Mezí další patří čtení a zápis citlivých dat prohlížeče (např. uložená hesla), čtení obsahu souborů a další. Pro bližší podrobnosti je potřeba nahlédnout do [1], nebo prostudovat zdrojový kód třídy `nsScriptSecurityManager`.

Kapitola 6

Datová úložiště, RDF

Existuje více způsobů, jak lze ukládat textová data. Mohou být uchovávány v textových souborech s různou interní strukturou, v dokumentech s tabulkami, ve vyspělých databázových serverech. Pokud je požadováno nenáročné lokální skladiště dat, mohou textové soubory plně postačovat. Dále je potřeba definovat strukturu takového souboru. S použitím značkovacích jazyků, jako je například XML, lze přehledně ukládat požadovaná data včetně jejich hierarchické struktury. XML sám však nedefinuje, jaké prvky mají být používány. Proto je vhodné použít nějakou aplikaci XML – RDF.

6.1 Ukládání znalostí

Resource Description Framework, zkráceně RDF, je standard W3C konsorcia využívající značkovacího jazyka XML (*Extensible Markup Language*) ke kódování znalostí do textového formátu [19, 34, 20]. RDF je založen na zápisu tvrzení o realitě ve formě trojic (subjekt – predikát – objekt). Subjekt představuje hlavní entitu, o které se něco tvrdí, predikát vyjadřuje vztah a objekt druhou část tvrzení. Znalosti v RDF lze také modelovat pomocí orientovaného ohodnoceného grafu.

Jako příklad byla zvolena stromová struktura složek a odkazů, které jsou zapsány v RDF/XML. Odpovídající graf je na obrázku 6.1[1]. Ovál zde představuje subjekt, obdelník objekt a popisy hran reprezentují predikáty.

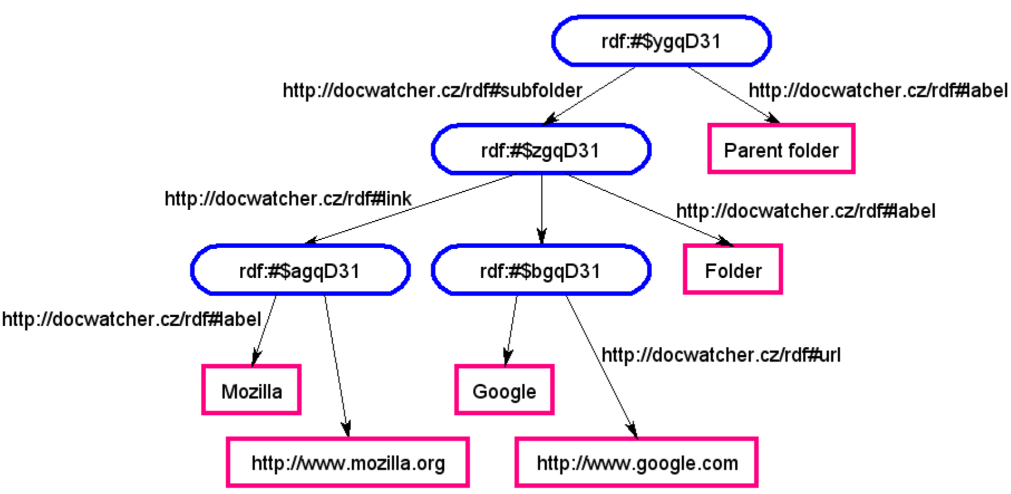
Na začátku je potřeba kromě jmenného prostoru RDF nadefinovat také vlastní jmenné prostory, které budou sloužit pro odlišení názvů predikátů od RDF entit. Trojice (subjekt, predikát, objekt) je zapisována ve tvaru

```
<RDF:Description RDF:about="název subjektu" NS1:predikát="objekt">
```

kde NS1 je název vlastního jmenného prostoru. V příkladu je uvedená také druhá možná notace, kde predikát netvoří atribut, jak je tomu tady, ale vlastní značku, ve kterém se nachází objekt jako textová entita. V tomto případě lze uvést pro RDF procesor typ hodnoty pomocí `NC:parseType`. Standard definuje hodnoty `Literal` (implicitní hodnota označující řetězec) a `Resource` (řetězec je URI). Mozilla kromě toho ještě podporuje `Integer` (řetězec je celé 32-bitové číslo se znaménkem) a `Date` (řetězec je datum).

Oba způsoby zápisu trojice je možné kombinovat až na jedinou výjimku, kterou je odkaz na jiný subjekt pomocí `RDF:resource`. Ten může existovat pouze jako atribut.

[1] Tento obrázek je upravený graf, který vygeneroval RDF validátor na http://www.w3.org/RDF/Validator/



Obrázek 6.1: Graf trojic odpovídající příkladu v RDF

```
<?xml version="1.0"?>
<RDF:RDF xmlns:NS1="http://docwatcher.cz/rdf#"
         xmlns:NC="http://home.netscape.com/NC-rdf#"
       xmlns:RDF="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <RDF:Description RDF:about="rdf:#$ygqD31"
                   NS1:label="Parent folder">
      <NS1:subfolder RDF:resource="rdf:#$zgqD31"/>
  </RDF:Description>
  <RDF:Description RDF:about="rdf:#$zgqD31"
                 NS1:label="Folder">
      <NS1:link>
          <RDF:Description RDF:about="rdf:#$agqD31">
              <NS1:label>Mozilla</NS1:label>
              <NS1:url NC:parseType="Literal">
                  http://www.mozilla.org
              </NS1:url>
          </RDF:Description>
      </NS1:link>
      <NS1:link RDF:resource="rdf:#$bgqD31"/>
  </RDF:Description>
  <RDF:Description RDF:about="rdf:#$agqD31"
                   NS1:label="Google"
                   NS1:url="http://www.google.com"/>
</RDF:RDF>
```

Z příkladu je dále patrný způsob uchovávání znalostí. Místo toho, aby bylo řečeno: “Složka obsahuje odkazy na Google a Mozillu”, je každému výrazu přiřazen jedinečný identifikátor. Teprve mezi nimi se tvoří vazby. Původní větu musíme tedy rozdělit na více

částí: "Existuje zroj s ID rovným rdf:#$zgqD31. Tento zdroj má název složka a obsahuje reference na odkazy s ID rdf:#$agqD31 a rdf:#$zgqD31. Zdroj rdf:#$agqD31 má název Mozilla a URL http://www.mozilla.org. Zdroj rdf:#$bgqD31 má název Google a URL http://www.google.com."

Hierarchickou strukturu lze stavět dvěma způsoby. Potomky nadřazené entity je možné vkládat přímo do rodičovského subjektu. Příklad této možnosti je ukázán zápisem vztahu Složka – odkaz Mozilla. Druhá možnost spočívá v umístění potomka kdekoli do RDF souboru a následně odkázání se na něj přes RDF:resource. Díky tohoto atributu lze vytvářet propojení mezi libovolnými subjekty.

Pro vytváření seznamů je možné použít tyto speciální tagy: <Seq> vytváří uspořádaný seznam, <Bag> je prostá kolekce položek, kde může dojít k záměně jejich pořadí a <Alt> uvozuje seznam, z něhož je vybrána vždy pouze jediná položka. Ve všech třech případech jsou položky ohraničeny elementem <li>. V takovém seznamu je každá položka chápána jako trojice (ID seznamu, automaticky generovaný anonymní název predikátu ve tvaru rdf:_číslo, položka).

6.2 Dotazování

Díky toho, že znalosti jsou modelovány jako trojice, lze snadno provádět dotazování podobně, jako v jazyce Prolog. Tohoto faktu využívají také projekty pod označením Sémantický Web, které si kladou za cíl převést informace z dokumentů na Internetu do vhodné počítačové reprezentace za účelem vytvoření jednotného způsobu výměny a zpracovávání informací.

Softwarová platforma Mozilla poskytuje tzv. *Templates*, díky nichž lze přímo připojit datový RDF zdroj k prvkům uživatelského rozhraní. V XUL se nadefinují dotazy, které RDF procesor zpracovává, výsledky ukládá do požadovaných proměnných a tyto proměnné se přiřazují požadovaným atributům UI prvků. Podporovány jsou i hierarchické struktury, takže je možné tímto způsobem plnit stromy i rozsáhlé víceúrovňové nabídky. U těchto složitějších komponent je dostupné tzv. pozdní přiřazení, které zajišťuje nahrávání dat až v okamžiku, kdy jsou potřeba. Jestliže má zdroj například 100000 položek a nultá úroveň obsahuje jen deset položek, jsou načteny pouze tyto položky a další až v okamžiku otevření nějaké větve.

Proto, aby tento systém fungoval, musí být určen datový zdroj a vstupní bod, což je název subjektu v hierarchii RDF. Dále je určena množina trojic, které mohou obsahovat identifikátory s otazníkem na začátku, do kterých se bude ukládat při zpracování zjištěná hodnota. Tyto proměnné se mohou nacházet na místě subjektu i objektu, predikát musí být vždy zadán. Procesor prohledává stavový prostor od definovaného vstupního bodu do úrovně, která je daná počtem trojic v množině podmínek. Jestliže všechny trojice vyhovují datovým závislostem, je nalezeno řešení, zjištěné hodnoty jsou přiřazeny příslušným atributům XUL prvkům a jsou zařazeny do DOM stromu. Jestliže alespoň jedna trojice nevyhovuje, je nalezené řešení neplatné. Proces se opakuje tak dlouho, dokud nejsou všechny kombinace v daném stavovém prostoru prozkoumány.

6.3 Podpora v XPCOM

Abychom mohli pracovat s RDF, musíme znát několik základních komponent. Datový zdroj reprezentuje komponenta s rozhraním nsIRDFDataSource. Obsahuje především metody pro

ukládání, mazání a úpravu znalostí. Při ukládání trojice pomocí XPCOM se navíc určuje, zda se jedná o pravdivou nebo nepravdivou informaci, což je dáno čtvrtým parametrem metody `Assert()`. Téměř vždy je hodnota nastavena na `true`. Komponenta dále disponuje metodami pro procházení hierarchií trojic daného zdroje.

Jestliže jsou data měněna z více míst, může být pro transformaci pohledů vhodné definovat jediné místo, kterým může být objekt s rozhraním `nsIRDFObserver`. Tento objekt nabízí 6 různých událostí, na které lze reagovat: vkládání a mazání znalosti, změna hodnoty objektu ve trojici, změna hodnoty predikátu trojice a začátek a konec hromadných úprav. Tyto poslední dvě události mohou být používány například v případě, že program hodlá vykonat více změn v datovém zdroji a reagování na všechny tyto úpravy by činnost zbytečně zdržovalo. Datový zdroj umožňuje tyto objekty s událostmi dynamicky přidávat a odebírat.

Z rozhraní `nsIRDFDataSource` jsou dále odvozeny některé specifičtější typy, například datový zdroj umístěný v paměti (`nsIRDFInMemoryDataSource`) a datový zdroj složený z více jiných RDF zdrojů (`nsIRDFCompositeDataSource`), který lze přiřadit všem prvkům uživatelského rozhraní v XUL.

Platforma zpřístupňuje službu s rozhraním `nsIRDFService`. Pomocí ní lze synchronně nebo asynchronně z daného URL získat datový zdroj `nsIRDFDataSource`. Služba se sama postará o načtení dat, jejich rozparsování a uložení do paměťové reprezentace. Tato komponenta také exportuje metody, které slouží k vytváření instancí reprezentujících subjekty, predikáty a objekty. V Mozille je subjekt a predikát ve trojici vždy tvořen komponentou s rozhraním `nsIRDFResource`, jehož atributem URI představuje jejich identifikátor. Objekt v trojici může být buď další idenfikátor, nebo kokrétní hodnota typu řetězec, celé číslo, nebo datum. Implementace ukládání typu datum reprezentovaného 64bitovým číslem však nefunguje správně, pravděpodobně jde o Bug 302387, viz Bugzilla. Poznamenejme, že identifikátory lze vytvářet buď ručně, tj. vložením určitého URI, nebo automaticky pomocí metody `GetAnonymousResource()`, která vytvořila například `rdf:#$7G1Zn1`.

Seznam položek realizuje komponenta s rozhraním `nsIRDFContainer`. Typ seznamu je určen metodou služby `nsIRDFContainerUtils` použité pro jeho vytvoření – `MakeAlt()`, `MakeBag()` a `MakeSeq()`. Seznam nabízí metody pro vložení a odstranění položky, zjištění jejich aktuálního počtu a umožňuje jimi procházet.

Kapitola 7

Ladění a testování aplikací

Výběr prostředí, které budeme využívat pro ladění, závisí na požadované cílové platformě, na které bude aplikace pracovat, a na dostupnosti pomocných rozšíření. Protože je tato práce zaměřena na aplikace založené na jádře XULRunner 1.8, nejsou zde zmíněny způsoby ladění a testování pro produkty SeaMonkey[1], dnes již zastaralý Mozilla Suite a dřívější verze Firefox a Thunderbird před verzí 1.5.

V počátečních fázích vývoje cílové aplikace je pro ladění nejvýhodnější Firefox 1.5 nebo 2.0. Nižší verze je vhodná v případě, že výsledná aplikace má pracovat s produktem Firefox 1.5 a výše, Thunderbird 1.5 a výše, XULRunner 1.8, nebo s některým dalším, jako je Flock, Sunbird, Songbird a další. Pro Firefox totiž existuje nejvíce potřebných ladících nástrojů a disponuje možností zjednodušeného připojování nových softwarových částí. Jakmile se však dostane vývoj do fáze, kdy se již využívají nestandardní komponenty dostupné pouze v daném cílovém produktu, je potřeba provádět testování ve všech produktech, které mají být rozšířením podporovány. Je také potřeba uvědomit si, že na funkčnost mohou mít vliv i minoritní aktualizace stávající aplikace. Vývojář se pak setkává s problémy, kde v jedné verzi nějaký potřebný kód nepracuje správně, a proto se pokusí chybu obejít například pomocí jiné XPCOM komponenty. V další verzi je pak chyba opravena, avšak původně využitá XPCOM komponenta doznala změn, které způsobí jiné chování než vývojář zamýšlel. Dostává se tímto do situace, kdy jeden kód nepracuje ve dvou různých verzích. Ano, platforma se stále dynamicky vyvíjí, s těmito problémy je potřeba počítat.

7.1 Konfigurace prohlížeče Firefox

Jestliže je Firefox na počítači vývojáře používán jako standardní prohlížeč, je rozumné vytvořit nový profil pro účely programování a testování, který nebude sdílet nastavení, programové doplňky, ale i chyby, které se určitě budou objevovat. Pokud Firefox běží, je potřeba jej zavřít. Nyní spustíme Firefox s parametrem `-P`, čímž se zobrazí okno "Výběr profilu uživatele", kde vytvoříme nový profil. Aby se původní profil určený pro prohlížení webových stránek implicitně načítal při startu Firefoxu, je nutné vybrat jej ze seznamu a mít zaškrtnuté tlačítko "Neptat se při startu". S takto nastaveným prohlížečem nedošlo k žádné viditelné změně, původní programoví zástupci fungují stejně.

[1] I když aktuální verze SeaMonkey i Firefox 2.0 využívají Gecko 1.8, mají odlišný způsob registrace nových rozšíření a strukturu adresářů.

Pro spuštění Firefoxu s vývojovým profilem nazvaným dev je vhodné použít následující dávkový soubor (určený pro OS Windows):

```
set MOZ_NO_REMOTE=1
"C:\Program Files\Mozilla Firefox\firefox.exe" -console -P dev
```

První příkaz nastavující proměnnou daného prostředí slouží k zajištění vytvoření nové instance Firefoxu. Pokud by Firefox byl již spuštěn s jiným profilem než dev a tento příkaz zde chyběl, byly by všechny parametry následujícího řádku ignorovány a bylo by pouze otevřeno další okno již běžícího profilu. Parametr -console zajišťuje zobrazení konzolového okna, do kterého lze pak zapisovat a -P dev způsobí načtení profilu pro vývoj [21].

Další krok spočívá v nastavení parametrů profilu dev. Jednou z možných cest je otevřít si stránku about:config a na ní nastavit binární hodnoty následujících položek na true. Pokud položky chybí, musí být vytvořeny:

- javascript.options.showInConsole – chyby, které se vyskytnou v chrome oblasti, budou logovány do Chybové konzoly.
- nglayout.debug.disable_xul_cache – vypíná ukládání XUL dokumentů, kódů v JavaScriptu a dalších do vyrovnávací paměti. Bez tohoto nastavení by se např. XUL dokument při jeho prvním načtení uložil do paměti, a jakákoli změna v jeho zdrojovém kódu by se až do dalšího restartu Firefoxu neprojevila. Se zapnutou volbou lze zdrojový kód změnit a znovu načtením daného XUL okamžitě spatřit nový výsledek.
- browser.dom.window.dump.enabled – aktivuje funkci dump(str), která způsobí výpis řetězce str do konzole, která je zobrazena díky parametru -console.
- javascript.options.strict – zapíná striktní kontrolu jazyka JavaScript, při kterém vypisuje varování, která jsou běžně ignorována.

7.2 Úprava aplikace

Protože je aplikace tvořena soustavou vzájemně propojených souborů, které tvoří jediný celek, není většinou možné testovat je samostatně. Před verzí Firefox 1.5 bylo možné testovat aplikace pouze tak, že se vytvořil instalační balíček rozšíření, a ten se nainstaloval do prohlížeče. Jestliže zde byla objevena chyba, bylo potřeba upravit zdrojové kódy, vytvořit nový balíček, předchozí odinstalovat a nainstalovat tento nový. Celý proces je pak neúměrně náročný nejen na čas.

Verze 1.5 pak přišla s novým způsobem registrace rozšíření, takže při ladění není nutné kód balit do XPI. Jestliže je dodržována struktura aplikace dle kapitoly 8, je potřeba provést úpravu souboru chrome.manifest spočívající v nahrazení řetězců “jar:chrome/package_name.jar!” za “chrome”. Nyní vytvoříme v adresáři profile_dir/extensions/ textový soubor, přičemž profile_dir označuje cestu k adresáři s používaným vývojovým profilem, v OS Windows XP např. c:\Documents and Settings\Uživatel\Data aplikací\Mozilla\Firefox\Profiles\wvs5kema.dev. Název nového souboru musí být shodný s identifikátorem vyvíjeného rozšíření, tj. viz hodnota objektu v trojici (urn:mozilla:install-manifest, id, ?) umístěné v install.rdf. Obsahem tohoto

souboru je jediný řádek s cestou, kde sídlí rozšíření. Firefox musí být schopný v tomto adresáři najít soubory install.rdf, chrome.manifest a adresář content. Po restartu aplikace Firefox by mělo dojít nalezení rozšíření a jeho instalaci.

Protože procházení chrome oblasti není možné, můžeme použít rozšíření *Chrome List*[2]. Díky něj lze v případě problémů s instalací naší aplikace zjistit, zda se nepodařilo aplikaci zaregistrovat (vlivem chyby v některém z konfiguračních souborů), a nebo zda je aplikace v systému přítomná, avšak díky uvedení chybné cesty je dostupná pouze její část.

7.3 Ladění uživatelského rozhraní

Syntaktická stránka XUL dokumentu je kontrolována automaticky při analýze XML. Pokud se zde vyskytne chyba, je označeno místo chyby a zobrazen její popis. Výjimku ovšem tvoří *overlays*, které v příslušném upravovaném dokumentu zobrazí šedý pruh a část chybného kódu, již však bez bližších podrobností.

Správnost sémantiky XUL lze ověřit pouze vizuální kontrolou vzhledu. Jestliže je například omylem napsán chybně název atributu nějakého elementu, bude tento atribut tiše ignorován a k zobrazení chyby nedojde.

Velkým pomocníkem je DOM inspektor, který bývá součástí téměř všech aplikací Mozilla, je-li nainstalován. Umožňuje přehledné procházení hierarchickou strukturou objektového modelu libovolného okna aplikace, nastavování atributů elementů, zobrazit použité kaskádové styly, pracovat s DOM objekty a JavaScriptem.

Kromě dvou výše uvedených parametrů příkazového řádku existuje ještě volba -chrome, která spolu s uvedením URL odkazujícího na XUL dokument v chrome umístění spustí instanci prohlížeče, kde je zobrazen pouze XUL dokument. Takto lze například testovat samostatně vzhled a chování aplikace.

7.4 Ladění kódu aplikace

Pro výpis hodnoty konkrétní proměnné je většina vývojářů webových aplikací zvyklá používat metodu `Window::alert()`. Bylo-li provedeno nastavení dle kapitoly 7.1 a je-li otevřeno konzolové okno Firefoxu spuštěné parametrem -console, je druhou možností výpisu libovolného řetězce metoda `dump()`, která zapisuje výstup do tohoto okna.

Požadujeme-li vypsat obsah celého objektu, tj. všechny názvy vlastností, konstant a jejich akutálních hodnot a všech dostupných metod, můžeme toho dosáhnout tímto cyklem:

```
for (var name in object) {
  try {
    var value = object[name];
    if (typeof(value) == "function")
      dump(name + "()\n");
    else
      dump(name + " = " + value + "\n");
  }
  catch(ex) { dump(name + " - nelze získat hodnotu\n"); }
}
```

[2] https://addons.mozilla.org/firefox/4453/

Protože se objekt chová v jazyce JavaScript jako asociativní pole, lze pomocí uvedeného cyklu procházet všechny jeho páry název – hodnota. Pokud je prvkem proměnná nebo konstanta, získaná hodnota je řetězec reprezentující skutečnou hodnotu dané vlastnosti. Výhodou, například oproti prohlížeči Internet Explorer, je skutečnost, že u objektů je jako hodnota vypsán název třídy, ze které je instance vytvořena, nikoli pouze `[object]`. V některých případech při volání z DOM objektů (např. okna) může načítání hodnot některých vlastností způsobit vyhození výjimky. Proto se zde musí nacházet `try/catch` blok. Poslední připomínka je k metodám – protože je hodnota metody buď výpis její implementace, je-li známa, nebo systémový řetězec `[native code]`, je výstup pro snadnější čitelnost upraven.

Aplikace Firefox nabízí nástroj nazvaný "Chybová konzola", avšak pro ladění je vhodné nahradit jej rozšířením *Console*2 [3]. Narozdíl od přítomné chybové konzoly umožňuje také filtrování zpráv a vyhodnocování vloženého kódu.

Nejpohodlnější způsob ladění však nabízí *Venkman Debugger*[4], což je komplexní grafické ladící prostředí určené pro jazyk JavaScript. Mezi podporované funkce patří například podpora zarážek (tzv. *breakpoints*), zobrazení zásobníku volání, dynamické prohlížení hodnot proměnných a procházení hierarchií objektů [31]. Taktéž je k dispozici konzola, do které lze interaktivně zadávat příkazy řídící běh programu a vyhodnocovat vložený kód.

Dalším užitečným rozšířením je *Developer's Extension*[5]. Nabízí interaktivní testování XUL kódu, takže při psaní kódu ihned vidíme výsledek, konzoli pro testování kódu v jazyce JavaScript s přístupovými právy k XPCOM, nástroj pro vyhodnocování regulárních výrazů a další funkce.

Zbývá ještě upozornit na knihovnu *XULUnit*[6] určenou pro vytváření testovacích případů a následné automatické testování JavaScriptového kódu. Aby bylo možné testovat aplikaci zevnitř, musí být tato knihovna součástí chrome oblasti. Lze ji například umístit do adresáře `content/test` a tím získá přístup ke všem komponetám aplikace. V implementaci této knihovny je však chyba, protože využívá vlastnosti `opener`, jenž v chrome není dostupná. Náprava spočívá v úpravě dvou řádků:

- xulunit.js:
  `this._window = window.open("testrunner.xul", ...)` změnit na
  `this._window = window.openDialog("testrunner.xul", ..., 400, _testRunner)`

- testrunner.xul:
  všechny výskyty `opener._testrunner` změnit na `window.arguments[0]`

7.5 Ladění komponent

Ladění komponent je poněkud komplikovanější než v předchozím případu. Není zde dostupné přímé volání metody `alert()`, proto se nejčastěji používá `dump()`. Ladící nástroj *Venkman Debugger* je možné použít, ovšem vstupní bod ladění bývá v kódu aplikace, nikoli uvnitř komponenty, a bohužel nezvládá přechod do asynchronně volaného kódu.

[3] http://console2.mozdev.org/
[4] https://addons.mozilla.org/firefox/216/
[5] http://ted.mielczarek.org/code/mozilla/extensiondev/
[6] http://xulunit.mozdev.org/

Problematiku ladění lze rozdělit do dvou skupin dle typu komponenty, tj. zda jde o službu jedinečnou v celém paměťovém prostoru platformy, nebo zda se jedná o běžnou komponentu s libovolným počtem instancí, jenž je vytvářena pomocí `createInstance()`.

Začněme jednodušším případem, komponentami získanými pomocí `createInstance()`. Jestliže kompenta byla již načtena do paměti platformy a vývojář nalezne a opraví chybu v implementaci, nedojde při dalším volání `createInstance()` k vytvoření instance s opravenou implementací. Při každé změně kódu komponent by tedy bylo nutné platformu restartovat. Tuto situaci lze řešit podobným způsobem, jako je řešena dědičnost uvnitř komponent:

```
const SOURCE = "chrome://docwatcher/content/compdevel/Controller.js";
const loader = Cc["@mozilla.org/moz/jssubscript-loader;1"]
                .getService(Ci.mozIJSSubScriptLoader);
const BaseClass = null;

createInstance: function (outer, IID) {
  if (outer != null)  throw Cr.NS_ERROR_NO_AGGREGATION;

  appendBaseClasses();

  var obj = {};
  loader.loadSubScript(SOURCE, obj);
  obj.Controller.prototype.__proto__ = BaseClass.prototype;
  return (new obj.Controller()).QueryInterface(IID);
}
```

Při každém volání `createInstance()` jsou znovu načteny zdrojový kód dané třídy komponenty i všech jeho předchůdců. Funkce `appendBaseClasses()` odpovídá funkci uvedené jako příklad v kapitole 5.2.2. Tento způsob má dvě menší nevýhody – vytváření instancí těchto komponent je pomalé a platforma má někdy problém s uvolňováním paměti. Druhý problém spočívá v tom, že pokaždé, když je vytvářena nová instance, jsou zároveň vytvářeny i nové bázové třídy. V ostrých verzích software však bude existovat pouze jediná bázová třída. Jestliže je chyba součástí bázových tříd, nemusí se to v některých případech při ladení uvedeným způsobem projevit. V ostré verzi pak může být pozorováno vzájemné ovlivňování atributů instancí odvozených tříd. Proto je nezbytné i důkladné testování ostrých verzí, zde už však pomáhá jen `dump()` a restarty platformy.

Jestliže potřebujeme odladit službu, je předchozí řešení nefunkční, protože instance komponenty se vytváří pouze jednou a pak už je vrácena jen její reference. Řešení spočívá v několika krocích. Je potřeba vytvořit XPCOM službu, která bude schopná uchovávat reference na objekty. Službu nazvěme `ObjectStorageService`, příklad její implementace:

```
function ObjectStorageService() {
  var objects = [];

  this.getObject = function(objName) {
    return (objects[objName]);
  }
```

```
  this.setObject = function(objName, instance) {
    objects[objName] = instance;
  }

  this.REINIT = function() {
    var controller = Cc["@docwatcher.jk.cz/controller;1"]
                       .createInstance(Ci.dwIController);
    objects["Controller"] = controller;
  };

  this.REINIT();
}
```

Aplikace už nesmí volat metodu `getService()` pro obdržení laděných služeb, místo toho jsou tyto služby získávány voláním metody `ObjectStorageService::getObject()`. Při jejím prvním volání je také spuštěna metoda `REINIT()`, která vytvoří instance všech laděných služeb aplikace a uloží je do svého pole. V okamžiku, kdy změníme implementaci některé třídy reprezentující službu, spustíme nějakým způsobem znovu metodu `REINIT()`, která upravené třídy načte. Součástí úprav je tedy i změna implementace metod `createInstance()` podle předchozího příkladu, aby docházelo k automatickému načítání nových zdrojových kódů.

Pro úplnost je potřeba uvést, že v případě změny rozhraní komponenty bývá nejjistější cestou odinstalace komponenty přes Správce doplňků, restartování Firefoxu, čímž dojde k odebrání záznamů o tomto rozšíření, a po zavření spuštěného prohlížeče lze opět v adresáři s rozšířeními vytvořit soubor odkazující na místo s testovanou aplikací.

Mezi nástroje, které je možné používat při vývoji XPCOM komponent patří *XPCOMViewer*[7], jenž zobrazuje seznam dostupných komponent, rozhraní a chybových konstant a nabízí možnost jejich filtrování. Díky něj můžeme snadno zjistit, zda instalace nové komponenty proběhla bez chyb a zda jsou všechny vlastnosti dostupné. Druhé rozšíření má název *Leak Monitor*[8], který je schopen detekovat úniky paměti a upozornit na ně.

7.6 Ladění RDF

Pro ověření správnosti zápisu v jazyce RDF/XML využíváme službu validátoru, jenž je dostupná na stránkách W3C[9]. Ze zápisu dokáže získat dostupné trojice znalostí a případně je znázornit ve formě grafu.

V případě, že požadujeme kontrolu obsahu datového zdroje, který se nachází pouze v paměti, lze získat aktuální stav ve formě textu pomocí metody `nsIRDFXMLSource::Serialize()`. Kód, kterým lze tento převod provést, je uveden jako příklad v kapitole 5.2.1. Vstupní proměnná `ds` musí obsahovat referenci na daný datový zdroj s rozhraním `nsIRDFDataSource`. Výsledný textový řetězec je uložen v `outputStream.data`.

[7] http://xpcomviewer.mozdev.org/
[8] https://addons.mozilla.org/firefox/2490/
[9] http://www.w3.org/RDF/Validator/

Kapitola 8

Distribuce aplikací

Aby bylo možné aplikaci distribuovat a následně provozovat, je potřeba dodržovat určitou strukturu adresářů. Kořen obsahuje tyto položky:

- components/* – adresář se soubory XPT, které obsahují přeložená IDL rozhraní komponent a jejich implementace v souborech .js.
- defaults/preferences/*.js – implicitní nastavení aplikace uložená v souborech s příponou .js. Řádek z tohoto souboru vypadá například takto:
  `pref("extensions.docwatcher.enabled", true);`
  Po instalaci rozšíření lze tyto hodnoty měnit na speciální stránce `about:config` a programový přístup k nim zajištuje XPCOM komponenta s rozhraním `nsIPrefService`.
- chrome/ – adresář, který po instalaci představuje kořen tzv. chrome oblasti, ve které jsou zdrojové kódy aplikaci, textové konstanty, soubory s kaskádovými styly a další položky, např. obrázky. Programové soubory získávají v této oblasti automaticky přístupová práva k XPCOM. Obvykle obsahuje tři podadresáře:
  - content – obsahuje soubory XUL, XML, JS.
  - locale – nachází se zde jeden nebo více adesářů s názvem udávájícím jazyk textových konstant uložených v souborech, které obsahuje. Soubory .dtd jsou spojovány s XUL, k řetězcům ze souborů .properties je možný přístup z kódu aplikace, viz kapitola 4.3.
  - skin – zde bývají kaskádové styly, obrázky a další soubory pro potřeby úprav vzhledu.

  Cesta `chrome://docwatcher/content/*.xul` umožňuje zpřístupnit soubory XUL, přičemž `docwatcher` označuje název aplikace, který je uveden v chrome.manifest.
- platform/ – adresář s kódem závislým na operačním systému a hardwarové platformě. Podrobné informace lze nalézt v [22].
- install.rdf – soubor používaný při instalaci rozšíření, skládá se ze dvou částí – informací o produktu, tj. jeho jedinečný identifikátor dle UUID, jméno, verze, adresa URL a další a seznamu aplikací daných jejich UUID a rozsahem verzí, do kterých lze toto rozšíření instalovat. Podrobné vysvětlení všech položek je uvedeno v [14] a seznam platných verzí a identifikátorů produktů Mozilla se nachází v [30].

- chrome.manifest – soubor, jenž identifikuje obsah adresáře chrome. Zároveň také nahrazuje staré registrační soubory contents.rdf, které byly používány před verzí Firefox/Thunderbird 1.5, avšak jsou stále nutné pro fungování rozšíření v aplikaci SeaMonkey. Podrobný popis uvádí [4].

Existují dvě možnosti, jak lze aplikace provozovat – jako rozšíření již existující aplikace nebo samostatně spustitelný program. První možnost spočívá v integraci nové aplikace do některé hostitelské aplikace využívající platformu Mozilla. Do této varianty spadají rozšíření s příponou .xpi, které lze stahovat například na webových stránkách MozDev.org[1].

Rozšíření .xpi je zabalený archiv algoritmem ZIP, který obsahuje výše uvedenou adresářovou strukturu, až na obsah adresáře chrome. Ten je zabalen opět algoritmem ZIP do archivu s příponou .jar. a tento archiv se jako jediný nachází ve složce chrome.

Druhou možností je vytvoření samostatně spustitelné aplikace. Její vývoj se po programové stránce téměř neliší od implementace rozšíření. Jediný rozdíl je v tom, že je nutné určit hlavní okno aplikace, a XUL elementy `overlay` tedy postrádají smysl.

V předchozím případě byla aplikační platforma součástí hostitelské aplikace. Zde se uplatňuje XULRunner, což je platforma přeložená pro konkrétní operační systém do spustitelné podoby [39]. Uživateli se tedy dodává instalátor obsahující XULRunner a zabalenou aplikaci. Při instalaci do systému uživatele jsou tyto dvě části zkopírovány na požadované místo a následně je spuštěna registrace aplikace do běhového prostředí XULRunner. Tím dojde k jejich propojení.

Platforma také nabízí možnost automatických aktualizací aplikací, které ji využívají. V souboru install.rdf je potřeba uvést i parametr `updateURL`, jenž obsahuje adresu RDF/XML dokumentu udávajícího označení dostupných verzí a pravidla, jakým způsobem se mají aplikace aktualizovat. Podrobný popis s vysvětlením všech polí je dostupný v [7].

[1] http://www.mozdev.org/projects/active.html

Kapitola 9

Aplikace DocWatcher

Nástrojů pro sledování změn dokumentů v síťovém prostředí není mnoho. V dubnu 2005 byl na [3] proveden test těchto aplikací. Výsledky lze rozdělit do čtyř skupin:

- Vítězem se stal WebSite-Watcher společnosti Aignes[1].
- Další příčku obsadily ostatní komerční programy nabízející nižší množství funkcionality: Check&Get[2], Copernic Tracker[3], TimelyWeb[4], Right Web Monitor[5] a Wysigot[6].
- Následují volně dostupné programy, jejichž možnosti jsou jen základní: WebMon[7], Check4Me[8] a InfoIC[9].
- Poslední skupinu tvoří webové služby hlídající změny v dokumentech na Internetu: WatchThatPage[10], TrackEngine[11].

Uvedené programy se liší především možnostmi výběru částí dokumentu, které budou porovnávány, a které budou naopak ignorovány. Třetí skupina je zcela nepoužitelná pro webové stránky, ve kterých se např. dynamicky generuje aktuální datum. Čtvrtá skupina – webové služby – má nevýhodu v podobě slabých možností reakce na změnu. I když je služba propracovaná, většinou nabízí jen možnost odeslání e-mailové zprávy. Naproti tomu lokální aplikace mohou provést na počítači nějakou akci, např. spustit program. Vhodné pro aktivní používání jsou tedy jen programy první a druhé skupiny, které jsou však komerční a jejich volné verze jsou časově a nebo funkčně omezené. Navíc, všechny aplikace, kromě webových služeb, jsou dostupné pouze pro operační systém Windows. Zatím tedy chybí volně dostupná multiplatformní aplikace s otevřeným kódem.

Tuto mezeru se bude snažit zaplnit nový program nazvaný DocWatcher, který je postaven na diskutované aplikační platformě Mozilla. Nyní je hlavním cílem vytvořit základní aplikaci, kterou bude možné dále upravovat a rozvíjet.

[1] http://www.aignes.com
[2] http://activeurls.com/en/
[3] http://www.copernic.com/en/products/tracker/index.html
[4] http://www.timelyweb.com/index.html
[5] http://www.right-soft.com/webmon/
[6] http://www.wysigot.com/
[7] http://www.markwell.btinternet.co.uk/webmon/
[8] http://www.neogie.com/check4me/
[9] http://www.infoic.com/Index.asp
[10] http://www.watchthatpage.com/
[11] http://www.trackengine.com/

9.1 Návrh aplikace

9.1.1 Stanovení požadavků

Požadavky zahrnující podporu více operačních systémů a HW architektur a volnou dostupnost již byly zmíněny. Nyní se proto zaměříme na funkční požadavky, tedy na to, co program bude uživateli nabízet. Většina uvedených požadavků vychází z funkcí, které nabízí WebSite-Watcher [33].

- Testování změn dokumentů v síťovém prostředí – podpora protokolů HTTP(S), FTP i lokálních dokumentů dostupných přes file:///.
  - Binární formáty – pouze informace o změně, kontrola dle parametru délka souboru, čas modifikace nebo obsah.
  - Textové formáty – možnost nastavit, které části dokumentu mají být sledovány nebo ignorovány, které výskyty řetězců mají být hlídány. Jestliže není uplatněn filtr, je dokument kontrolován některým z parametrů uvedených u binárních formátů.
- Podpora autentizace pomocí SSL.
- Nastavení intervalu kontrol – kontrolování dokumentů při spuštění programu, v zadaném intervalu, manuálně v libovolný okamžik.
- Ukládání předchozích verzí dokumentů a možnost pohybovat se historií změn, zadání hloubky historie.
- Notifikace: zobrazení informačního okna, zvukové upozornění a spuštění externí aplikace

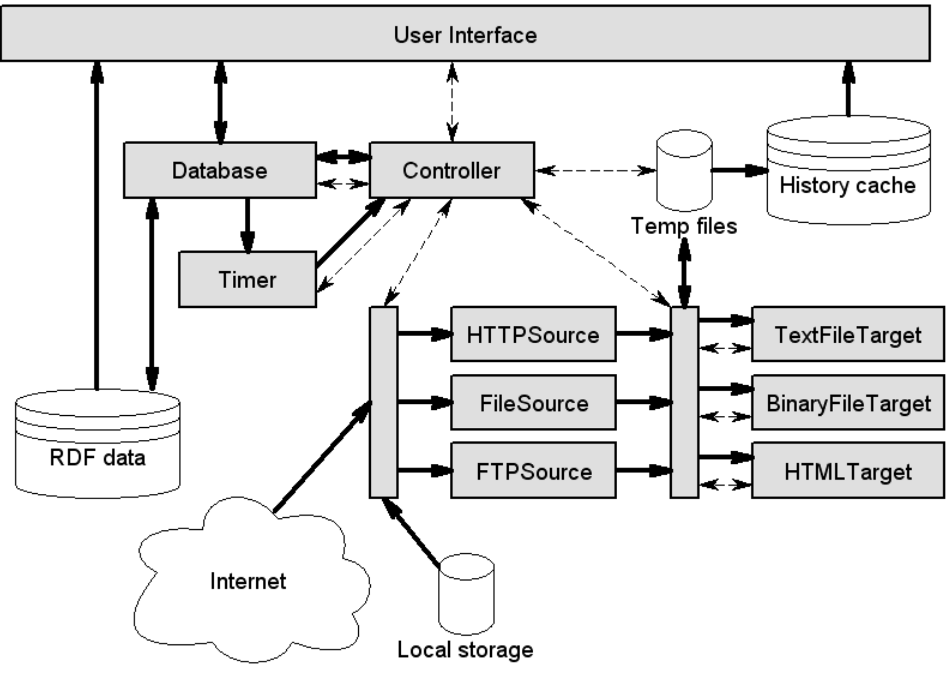
Mezi požadavky je ještě vhodné uvést, že výsledná aplikace bude primárně fungovat jako rozšíření produktů Firefox a Thunderbird, avšak kód by měl být psán tak, aby bylo možné vytvořit i samostatně spustitelnou aplikaci. DocWatcher pak bude dostupný v nabídce "Nástroje".

9.1.2 Architektura aplikace

Celkový pohled na propojené komponenty této aplikace je uveden na obrázku 9.1. Tlustá čára označuje hlavní datový tok, kde se přenáší větší množsví dat, čárkovaná tenká čára odpovídá řídícímu toku, kde se volají operace a prochází události. V tomto modelu jsou zahrnuty některé charakteristické vlastnosti aplikační platformy Mozilla.

Aplikace je složena z několika hlavních komponent. První s názvem `Database` zodpovídá za načítání a ukládání nastavení jednotlivých složek a odkazů do RDF uložiště dat, jehož struktura je popsána v kapitole 9.1.3. Uživatelské rozhraní načítá data z `Database` v případě, že uživatel chce zobrazit vlastnosti složky nebo odkazu a ukládání dat probíhá v okamžiku vytvoření nových nebo upravení již existujících dat entit. Kromě toho je uživatelské rozhraní přímo napojeno na datový zdroj pomocí *Templates*, seznamy složek a odkazů jsou generovány přímo z něj.

Další významná komponenta je `Timer`. Jestliže uživatel zvolí automatické kontrolování odkazů v určitých intervalech, tato komponenta si automaticky ve spolupráci s `Database` zjišťuje, kdy má být příští kontrola a jakmile tento čas nastane, předá daný odkaz ke kontrole komponentě `Controller`.



Obrázek 9.1: Celkový pohled na propojené komponenty aplikace DocWatcher

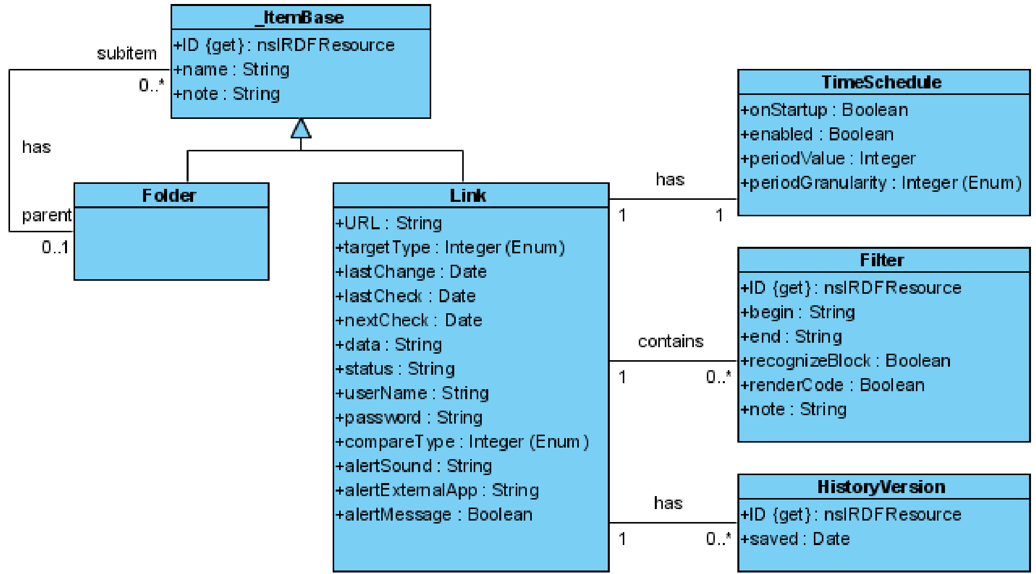
`Controller` je řídící komponenta celé aplikace. Kromě podpory ostatním částem aplikace slouží také ke spouštění kontroly vybraných odkazů. Podle umístění dokumentu je odkaz poslán k načtení do příslušné komponenty, například dokumenty s URL začínající `https://` jsou zaslány ke zpracování komponentě `HTTPSource`. Jestliže je vybráno kontrolování dle obsahu, nebo je-li zapnuto ukládání do historie, je při načítání dokumentu datový tok ukládán do dočasného souboru. Později může být přesunut do úložiště verzí dokumentu. Jestliže je vybráno kontrolování dokumentu dle jeho obsahu, zpracovává jej příslušná komponenta s postfixem `Target`, v opačném případě údaje o velikosti souboru nebo datumu poslední změny zjišťuje načítající komponenta. Jestliže je zjištěna odlišná hodnota od té, jenž je uložena u příslušného odkazu, je zahlášena změna.

9.1.3 Ukládání entit do databáze

ER diagram zobrazující perzistentní vztah složek, odkazů a dalších podpůrných entit je zobrazen na obrázku 9.2.

Oproti původnímu návrhu, jenž byl součástí semestrálního projektu, byl tento návrh značně upraven. Původně byla zamýšlena možnost hierarchicky definovat společné vlastnosti. Například uživatelské jméno a heslo bylo možné nastavit na úrovni složky a u všech odkazů, které by měly tyto vlastnosti nevyplněné, by došlo k jejich převzetí. Po zkušební implementaci tohoto způsobu se však ukázalo, že uživatelé jsou spíše zmateni a tuto možnost nevyužívají. Také ji téměř plně může nahradit funkce, která umožňuje vybrat více odkazů najednou a nastavit v jednom okamžiku vybrané hodnoty na jednu konkrétní hodnotu.

Diagram obsahuje pět typů entit. `_ItemBase` je bázová entita v hierarchii dědičnosti, ze kterých dědí společné vlastnosti entity odkaz (`Link`) a složka (`Folder`). Jedna složka může



Obrázek 9.2: ER diagram zobrazující perzistentní vztah odkazů a složek

obsahovat více zanořených složek a odkazů a také všechny tyto entity mají, s výjimkou kořenové složky, nadřazenou složku. Význam většiny atributů je jasný, proto je zde uvedeno jen několik podrobností:

- `targetType` – cíl odkazu, dokument může být v textovém, binárním formátu, HTML atd.
- `compareType` – způsob porovnávání verzí dvou dokumentů, je možná jedna z výčtových hodnot: velikost souboru, datum a čas změny souboru, kontrola obsahu. S tím také souvisí atribut `data` v entitě odkaz, který obsahuje hodnotu závislou na typu porovnávání. Je zde tedy uložena buď velikost poslední verze dokumentu, nebo konkrétní datum a čas poslední změny a nebo otisk získaný algoritmem MD5.
- `alertMessage` – udává, zda bude vyskakovat okno se zprávou.

Každý odkaz má informace o tom, kdy má být kontrolován. Tyto údaje jsou uloženy v entitě `TimeSchedule`. Vlastnost `periodGranularity` může nabývat jednu z výčtových hodnot sekunda, minuta, hodina, den, měsíc.

Fitrům je věnována vlastní kapitola 9.2.5, kde je také vysvětlen význam atributů entity.

Poslední entita `HistoryVersion` slouží k udržování přehledu o uložených verzích jednoho dokumentu, přičemž je uchováván unikátní identifikátor kopie a časové razítko jejího pořízení.

Poznámka – jestliže uvažujeme realizaci této perzistence na platformě Mozilla, zjistíme, že všechny tyto záznamy budou uloženy v jednom XML souboru. Hesla se zde budou zřejmě nacházet v otevřené textové podobě, což může být značné bezpečnostní riziko. Proto v dalších verzích tohoto nástroje bude potřeba vyřešit způsob zabezpečení a šifrování těchto citlivých dat, bude také nutné detailně prozkoumat implementaci správce hesel, jenž je součástí platformy Mozilla. Asi jediným dostatečně univerzálním řešením pak bude nabídnout uživateli ochranu citlivých údajů pomocí společného hesla, které bude muset

vkládat při startu aplikace. V takovém případě je pak možné šifrovat data některým ze symetrických kryprovacích algoritmů.

9.2 Detaily implementace

Při vývoji aplikace DocWatcher došlo k nutnosti přepracovat větší část zdrojového kódu. První verze implementovala třídy `Database`, `Controller`, `Timer` jako běžné třídy na úrovni XUL dokumentu. Protože uživatel mohl spustit více oken webového prohlížeče a každé okno si vytváří nové instance všech definovaných proměnných a tříd, došlo by k tomu, že v systému by fungovalo více těchto objektů. Dokumenty by mohly být kontrolovány vícekrát za sebou, více instancí `Database` by způsobilo problémy v datovém úložišti. Proto byla v XPCOM vytvořena služba `ObjectStorageService`, která udržovala jediné instance uvedených tříd v celém prostoru platformy. V okamžiku, kdy bylo nainstalováno rozšíření *Leak Monitor*, viz kapitola 7.5, uživatel otevřel nové okno prohlížeče a předchozí okno zavřel, bylo zobrazeno hlášení o úniku paměti. První okno, které uživatel zavřel, totiž vytvořilo objekty v paměti a jejich reference byly předány službě `ObjectStorageService` k uchování. Protože je objekt vytvořený v okně prohlížeče na něj interně vázán, nedošlo k uvolnění paměti zabírané již zavřeným oknem. Tato skutečnost byla odstraněna přesunutím těchto tříd do XPCOM, byly tedy vytvořeny služby s rozhraními `dwIDatabase`, `dwIController`, `dwITimer`. Třídy reprezentující odkazy, složky a další, které s nimi spolupracují, bylo také nutné převést do XPCOM. Došlo tím vpodstatě také k dokonalejšímu odstínění aplikační logiky od datové vrstvy reprezentované `dwIDatabase` a dalších komponent aplikace.

9.2.1 Datová vrstva

Datová vrstva je tvořena komponentou s rozhraním `dwIDatabase`, která zajišťuje přístup k datovým zdrojům ve formátu RDF/XML a následníky rozhraní `dwIPropertyBase`. Toto rozhraní je abstraktní a poskytuje operace pro práci s atributem perzistentního objektu. Musíme si uvědomit, že je-li nějaký atribut třídy číselného typu, například `PRInt16`, a i když do vnitřní řeprezentace můžeme zapsat hodnotu `null`, nelze takovou hodnotu předat přes tento typ rozhraní. Proto dvě z metod, které rozhraní `dwIPropertyBase` nabízí, jsou `isNull()` a `setNull()`.

Odvozenou třídou je `_ItemBase`, která obsahuje společné položky odkazů a složek, což jsou jméno, poznámka a nadřazená složka. Z ní jsou pak již odvozeny komponenty s rozhraním `dwILink` a `dwIFolder`. Ve všech třech třídách je implementováno načítání hodnot na vyžádání, což znamená, že hodnoty jsou z RDF zdroje čteny až v okamžiku, kdy jsou potřeba. Každý atribut deklaruje místo pro uchování hodnoty, jejího typu daného potomkem rozhraní `nsISupportsPrimitive` a v případě potřeby je připraven mechanismus transformace načtených hodnot z RDF zdroje. Hodnoty nejsou do atributů přiřazovány přímo, avšak přes obalující typ s rozhraním `dwIType`. Po vytvoření instance třídy jsou totiž všechny atributy nastaveny na hodnotu `null` a tím indikují, že hodnota ještě nebyla získána ze zdroje. Pokud by ovšem tato získaná hodnota byla opět `null`, nebylo by možné odlišit stav "nenačteno" od "hodnota = null".

Po načtení hodnoty atributu z databáze se automaticky vytváří instance třídy `Type` a do její vlastnosti `value` se přiřadí buď `null` nebo nová instance třídy s příslušným rozhraním následníka `nsISupportsPrimtive` a typ rozhraní je uložen do proměnné `VIID`. Tím získáváme silnou typovou kontrolu, jenž lze předávat přes IDL a zároveň také prostor pro parametrizování hodnot atributů. Této vlastnosti bylo využito v předešlé verzi aplikace,

kde ještě fungovala možnost hierarchického nastavování hodnot. Takovým parametrem byl `isParentValue` signalizující, zda je hodnota definována u dané položky, nebo zda tuto hodnotu pouze přebírá od rodičů.

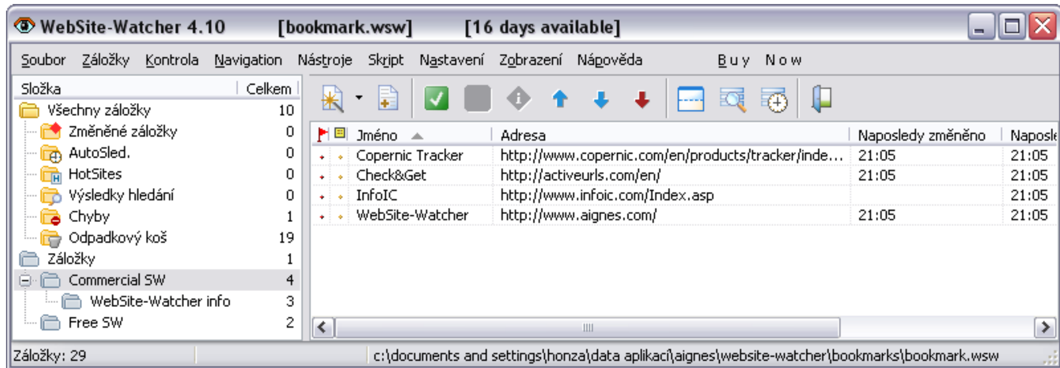
Třída s `dwIType` je používaná také k předávání libovolné hodnoty přes rozhraní typově bezpečným způsobem. Tím nedochází k problémům, kdy přes číselný atribut rozhraní potřebujeme dostat `null`.

9.2.2 Uživatelské rozhraní

Klíčovými prvky uživatelského rozhraní jsou hlavní okno aplikace a dialogové okno vlastností odkazu, které je zobrazováno při přidávání nebo úpravě informací o dokumentu určeném ke kontrole.

9.2.3 Hlavní okno aplikace

Aplikace WebSite-Watcher a TimelyWeb používají dva různé způsoby zobrazování položek ve stromové struktuře. Lze je vidět na obrázcích 9.3 a 9.4.

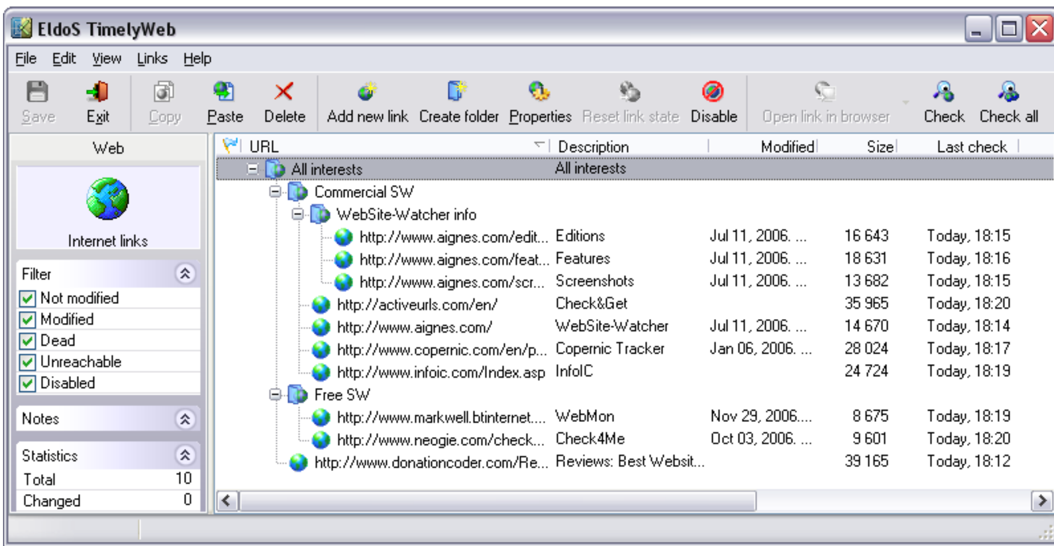


Obrázek 9.3: Hlavní okno aplikace WebSite-Watcher

V obou programech se používá stromová struktura, která zobrazuje složky a v případě TimelyWeb také odkazy, které se v nich nacházejí. Jestliže je ale strom hodně rozvětvený a obsahuje velké množství složek a odkazů na dokumenty, je způsob zobrazení používaný v aplikaci TimelyWeb velice nepřehledný. Smíchání složek a odkazů do jedné struktury působí poměrně chaoticky.

Proto byl zvolen oddělený pohled na strom složek a na seznam odkazů, které vybraná složka, případně složky, obsahují. Takový způsob zobrazení je také známý z aplikace Explorer v operačním systému Windows. Zde je možno spatřit první vylepšení oproti uvedeným programům – lze vybrat najednou více složek, přičemž se všechny jejich položky zobrazí v pravé části okna. Díky toho lze skupinově měnit vlastnosti odkazů.

Hlavní okno aplikace se tedy sestává ze dvou částí – v levé se nachází strom s uživatelskými složkami. Kromě nich se zde nacházejí i tři systémové, jenž informují o právě kontrolovaných dokumentech, chybách, které vznikly během posledních kontrol a změněných odkazech. V pravé části okna je umístěn seznam odkazů, který zobrazuje tyto detaily: Název odkazu, URL, datum/čas poslední změny, datum/čas poslední kontroly, datum/čas příští kontroly a stav poslední kontroly.



Obrázek 9.4: Hlavní okno aplikace TimelyWeb

Oba XUL elementy, které zobrazují složky a odkazy, jsou přímo napojeny na dva datové zdroje. Prvním je externí RDF, jenž je umístěn v adresáři docwatcher/data.rdf v daném profilu uživatele. O jeho případné vytvoření a načtení se stará služba `Database`. Jsou zde umístěny všechny údaje o odkazech a složkách, které je potřeba uchovávat. Ještě poznamenejme, že vytvoření hierarchické struktury složek je umožněno skládáním identifikátorů pomocí kontejnerů realizovaných `dwIRDFContainer`. Jiný způsob ukládání znalostí do RDF by pravděpodobně způsobil nemožnost přímého zobrazení stromové struktury. Druhým datovým zdrojem je interní RDF, které je vytvořeno dynamicky při startu aplikace. Používá se z několika důvodů. Prvním je potřeba vytvořit dvouúrovňovou hierarchii složek, kde budou vloženy složka "Přehled" spolu se třemi podsložkami a k nim přiřadit odpovídající odkazy. K této hierarchii se pak ještě přidává kořenová složka uživatelských složek, jenž jsou vzápětí také připojeny. Druhým důvodem je interpretace hodnot uživateli. V externím RDF zdroji je například datum poslední změny uloženo jako 64bitové číslo a tuto hodnotu nelze uživateli vypsat. Proto jsou u odkazů definovány metody, které vytváří novou trojici, kde subjekt odpovídá identifikátoru odkazu, predikát uvozuje čitelnou podobu konkrétní hodnoty a objekt je řetězec, zde například datum a čas. Tyto trojice jsou vytvářeny pouze tehdy, jestliže uživatel daný odkaz vybral a může jej vidět, a opět rušeny, byl-li odkaz skryt.

9.2.4 Dialogové okno vlastností odkazu

Druhým důležitým prvkem uživatelského rozhraní je dialogové okno s vlastnostmi odkazu. Skládá se z několika záložek, na jejichž panelech lze zadat následující údaje:

- Obecné – název dokumentu, adresa URL a typ souboru (textový, binární).
- Nastavení – přihlašovací jméno a heslo a způsob porovnávání dokumentu (dle obsahu, velikosti, datumu poslední změny).
- Filtr – záložka je dostupná pouze u textových souborů. Umožňuje omezit oblast, ve které budou vyhledávány změny.

- Historie – nastavení parametrů ukládání historie změn dokumentů.
- Časy kontroly – možnost zaškrtnutí volby “Kontrola při startu”, výběr manuálních/automatických kontrol. Ve druhém případě lze určit, jak často má být dokument kontrolován.
- Akce – způsob upozornění uživatele při změně dokumentu a případně volba akce. Zahrnuje výběr zvukového souboru, název externí aplikace, která bude spuštěna a možnost určit, zda bude změna signalizovaná vyskakujícím oknem.
- Poznámka – karta s editačním polem umožňující zadat libovolnou textovou poznámku.

Při výběru více odkazů mohou být vlastnosti nastavovány v jeden okamžik na stejnou hodnotu. Aplikace WebSite-Watcher tuto vlastnost také zvládá, avšak používá jiný typ dialogového okna než je zobrazen při vytváření nového odkazu, což může uživatele mást. DocWatcher využívá jediné okno. Jestliže je daná vlastnost ve všech odkazech nastavena na stejnou hodnotu, je tato hodnota uvedena. Pokud jsou hodnoty různé, je zobrazen obsah textové konstanty `moreValues` umístěné v souboru `app.properties`, případně, jedná-li se o zaškrtávací tlačítko, je zobrazen třetí nerozhodný stav, který byl pro tento účel implementován. Pravé tlačítko myši vyvolává kontextové menu, které umožní návrat na původní hodnotu.

9.2.5 Proces kontroly

Po spuštění aplikace je pomocí metody `dwIDatabase::getNextLinksToCheck()` nalezen nejnižší čas, kdy se má/měla spustit kontrola dokumentu. Je-li nalezený čas menší nebo roven aktuálnímu času, je prozkoumána databáze odkazů a vyhledány ty, které mají být zkontrolovány. V opačném případě je vrácen odkaz, který má být zkontrolován nejdříve. Průběh kontroly textových a binárních dokumentů je zobrazen na obrázku 9.5.

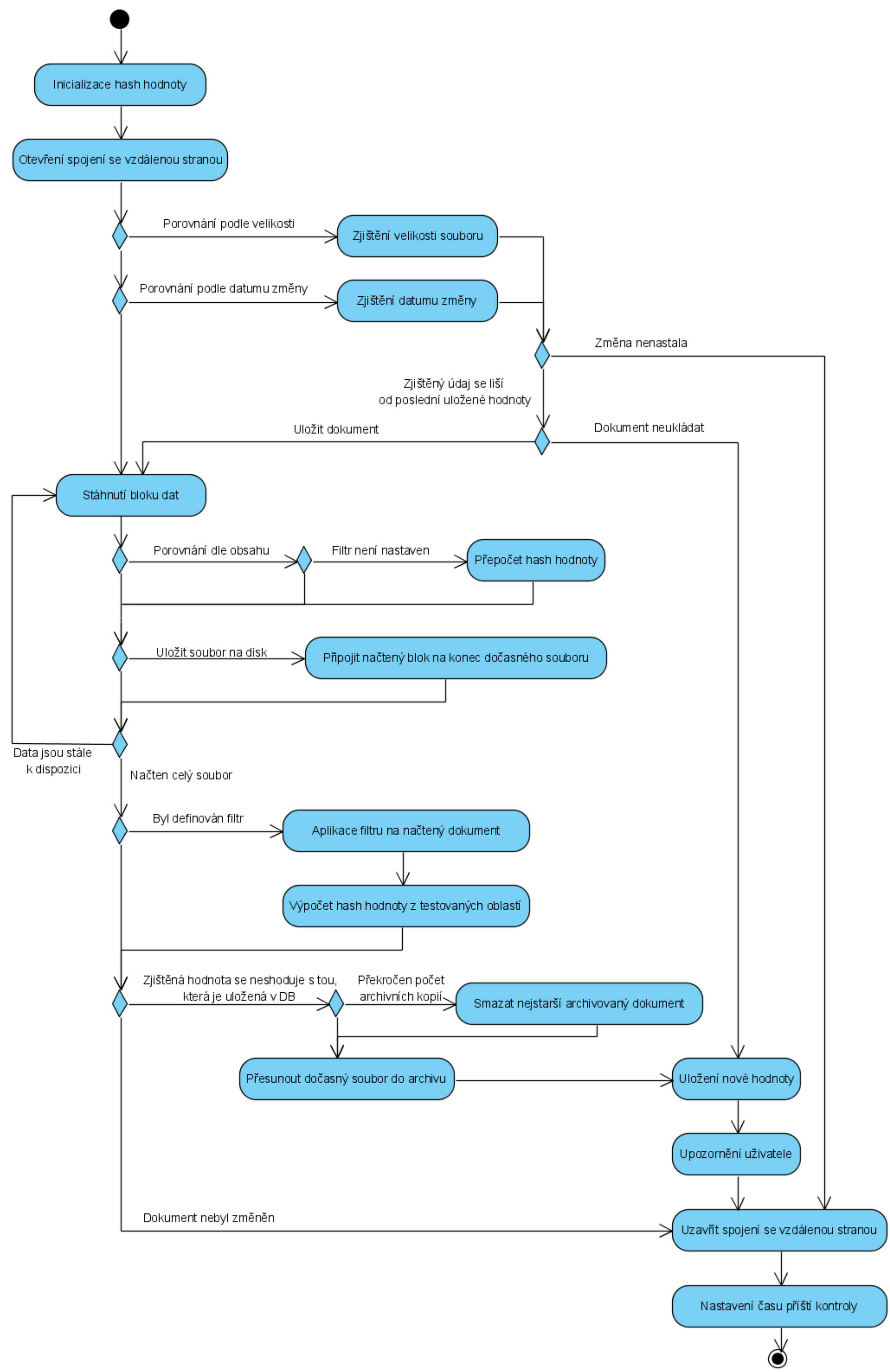
Po odstartování procesu kontroly dokumentu je navázáno spojení se serverem, na kterém je zkoumaný dokument uložen, nebo je otevřen lokálně dostupný soubor. Také je inicializovaná hodnota otisku dat (*hash*). Jestliže byla zvolena možnost porovnávání dle velikosti nebo času změny souboru, je tento parametr ihned vyhodnocen a v případě neshody s uloženou hodnotou je zahlášena změna.

Soubor je čten a kopírován do dočasného souboru po blocích, jejichž velikost určuje vzdálený server nebo operační systém. Bloky jsou načítány asynchronně, tj. čtení neblokuje hlavní vlákno aplikace. Jestliže uživatel nezadal filtr, je při každém načteném bloku upravena hodnota otisku dat. Tento proces je opakován až do načtení celého souboru. V případě úspěšného dokončení stahování je aplikován filtr a vypočítán otisk vyfiltrovaných oblastí, pokud uživatel tuto volbu vyžaduje. Následuje kontrola množství předešlých načtených verzí souboru a při překročení hraniční hodnoty je nejstarší verze smazána.

V tomto okamžiku je k dispozici *hash*, který charakterizuje oblast dat, o kterou se uživatel zajímá. Jestliže tato hodnota je odlišná s tou, která je uložená v databázi, je dokument změněn, uživateli se aktivují vybrané způsoby notifikace a je uložená nová hodnota. Celý proces končí uzavřením spojení nebo lokálního souboru.

Filtrování obsahu textového dokumentu

U textových formátů je možné využít funkci filtru. Právě tato funkce podstatným způsobem určuje kvalitu aplikací zaměřujících se na detekci změn obsahu v dokumentech. Bez ní nelze



Obrázek 9.5: Diagram aktivit znázorňující průběh kontroly dokumentu.

detekovat změny v dokumentech, do kterých se například automaticky generuje datum a čas. Při každém načtení by byl takovýto dokument označen jako změněný.

Každý textový soubor může obsahovat hierarchickou posloupnost filtrů. U každého filtru se definují následující vlastnosti:

- ohraničení pomocí dvou řetězců, které nejsou zahrnuty do testování a
- typ bloku, tj. zda má být vyznačená část dokumentu ignorována nebo sledována

Pravidla definice filtrů:

1. Filtrování je neaktivní v případě, že je zobrazen pouze jediný filtr, který je určen začátkem a koncem souboru a jehož obsah je sledován. Tento filtr nelze smazat.
2. Do filtru, jehož obsah je sledován, lze vložit filtry obou typů:
   (a) Vložen filtr pro sledování bloku – rodičovský filtr pozbývá funkce sledování, již pouze vyznačuje část v dokumentu. Nový filtr provedl upřesnění místa sledování.
   (b) Vložen filtr pro ignorování obsahu bloku – rodičovský filtr stále určuje oblast sledování výskytu změn s výjimkou oblastí, které mají být ignorovány.
3. Do filtru, jehož obsah má být ignorován, již nelze vložit další filtr.
4. Filtry nelze překrývat.

Aplikace filtrů je činnost systému, při kterém je ve vstupních textových datech vyhledávána skupina řetězců, které budou následně testovány na shodu s minulou verzí. Filtry jsou uspořádány hierarchicky, postupuje se od globálního filtru (viz bod 1 definice pravidel) až k nejvíce zanořeným filtrům. Je nalezen začátek a konec bloku, který filtr definuje, a v něm jsou aplikovány zanořené filtry. Jestliže je definováno více filtrů na stejné úrovni v hierarchii, pak začátek nového bloku je vyhledáván od konce bloku předchozího. Je-li nalezen sledovací filtr, který již neobsahuje další zanořený sledovací filtr, pak je řetězec daný jeho ohraničením a vyříznutím všech částí, které definují ignorující filtry, uložen do paměti.

Jestliže existuje alespoň jeden řetězec udávající hranici bloku filtru, který není v dokumentu nalezen, pak byla zřejmě změněna struktura dokumentu a filtr nelze aplikovat. V takovém případě je zahlášena "Chyba filtru".

V původním návrhu v semestrálním projektu byla možnost provádět filtrování nad elementy HTML dokumentu. Aby to bylo možné, musela by existovat nějaká třída, která je schopná vykreslit v paměti danou HTML stránku, tj. vytvořit DOM model, spustit JavaScript kód, který tato stránka obsahuje a aplikovat CSS styly (které například ukryjí nějaké informace nebo změní jejich pořadí). Taková třída dostupná v jazyce JavaScript ale neexistuje. Z popisu procesu renderování uvedeného v [26] a příspěvků v několika diskusních skupinách vyplývá, že vykreslování HTML dokumentu musí vždy probíhat v kontextu nějakého okna. Tím se dostáváme do problematické situace – DocWatcher pracuje většinu svého času pouze v paměti a okno je vytvářeno až v okamžiku kliknutí na volbu "DocWatcher" v menu "Nástroje". Řešením by mohlo být spuštění procesu renderování v kontextu nějakého otevřeného okna prohlížeče. Jestliže však uživatel otevře nové okno a původní okno zavře, dojde k chybě a "stěhování" na jiné okno nelze provést. Druhá možnost je využití skrytého okna, které je potřeba na operačním systému Mac OS X, okno je dostupné přes atribut `nsIAppShellService::hiddenDOMWindow`. Vzhledem k tomu, že

by mělo být toto okno z implementací pro Windows XP a Linux časem odebráno, opět to není vhodné řešení. Z tohoto důvodu zůstává funkce filtrování dle prvků DOM daného dokumentu nefunkční a v budoucnu bude snaha vytvořit komponentu v jazyce C++, která se přímo napojí na renderování, sama v paměti vytvoří skryté okno a dodá potřebný kontext.

9.3 Závěrečné poznámky

Při ladění zdrojových kódů byl kromě principů uvedených v kapitole 7 také vytvořen jednoduchý systém komentářů pro označení oblastí, které jsou určeny pouze pro ladění, které mají být pouze v ostré verzi, a zbývající kód. Speciální komentář byl také vytvořen pro vkládání souboru na dané místo. Příklad:

```
/* INCLUDE "chrome/content/compdevel/Controller.js" */
/* DEBUG B */
function Controller()
{
}
/* DEBUG E */

/* RELEASE B *
dump("Controller OK\n");
/* RELEASE E */
```

Pro převod do XPI balíku můžeme mít vytvořenou dávku, jejíž součástí je spuštění například skriptu v jazyce Perl, který projde všechny soubory, komentáře `/* DEBUG B */` ... `/* DEBUG E */` odstraní včetně jejich obsahu, smaže také řádky `/* RELEASE B *` (zde opravdu chybí lomítko) a `/* RELEASE B */` a konečně řádky `/* INCLUDE "cesta" */` nahradí skutečným obsahem uvedeného souboru. Příklad tohoto dávkového souboru a příslušných skriptů je uveden na doprovodném DVD.

Druhá poznámka se týká komentování kódu v JavaScriptu a automatické vytváření dokumentace. Zdrojové kódy aplikace DocWatcher jsou komentovány dle mírně upravených požadavků aplikace *Natural Docs*[12], která umožňuje převod jasně čitelných komentářů do moderní dokumentace v HTML. Pro správný výstup je potřeba všechny zdrojové kódy ještě zpracovat pomocí skriptu umístěného na DVD.

Alternativou je *JSDoc*[13], který je vytvářen pouze pro jazyk JavaScript, avšak jeho výstupy už nejsou tak hezké a u složitějších konstrukcí má problémy s rozpoznáním kontextu.

[12] http://naturaldocs.org/
[13] http://jsdoc.sourceforge.net/

Kapitola 10

Závěr

Uživatelé, kteří používají pro prohlížení webových stránek aplikaci nazvou Firefox a nebo čtou svou poštu e-mailovým klientem Thunderbird většinou nemají ani tušení, jak mocný nástroj používají a co vše je ukryto pod povrchem uživatelských rozhraní, se kterými pracují. Tato diplomová práce má snahu toto tajemství odkrýt a představit světu volně šiřitelnou aplikační platformu Mozilla.

Její výhodou je velké množství podporovaných operačních systémů, kterých je více než deset. Této vlastnosti je dosaženo vyspělou architekturou platformy, jejíž struktura je uvedena na začátku této práce. Hlavní předností je však nativní podpora zobrazování (X)HTML dokumentů a možností aktivně s nimi pracovat. Je možné upravovat jejich objektový model, aniž by bylo potřeba psát složitý kód.

Avšak platforma má bohužel i stinné stránky. Největší nevýhodou je nedostupnost kvalitních ucelených informací, které se zabývají vývojem aplikací na nejnovějších verzích této platformy. Cílem této práce je proto navázat na knihu Rapid Application Development with Mozilla od N. McFarlanea, uvést na pravou míru informace, které nejsou platné pro jádro běhového prostředí XULRunner v.1.8. a zároveň upozornit vývojáře na chyby, které existují ve stávající implementaci jádra platformy. Tato diplomová práce zároveň obsahuje kapitoly v pořadí, v jakém se obvykle vytváří aplikace – programátor je seznámen s vytvářením uživatelského rozhraní v jazyce XUL a jeho možnostmi, dále je proveden objektově orientovanými principy dostupných v jazyce JavaScript, jenž je hlavní programovací jazyk nad vrstvou XPCOM. O této vrstvě, jenž odděluje jádro platformy od prostoru, ve kterém se pohybuje vývojář, jsou uvedeny důležité informace a dále se zde můžeme seznámit s problematikou vývoje vlastních komponent. Nechybí ani vysvětlení ukládání znalostí a dotazování v jazyce RDF/XML. Větší část práce je věnovaná ladění aplikací, jenž je poněkud komplikovanější než u jiných kompilovaných jazyků a platforem. Poslední kapitola této části uvádí možnosti distribuce hotových aplikací a způsoby jejich provozu.

Získané znalosti byly následně využity prakticky při vytváření volně šiřitelného multiplatformního nástroje zvaného DocWatcher, jenž využívá ke svému fungování aplikační platformu Mozilla. Cílem kapitoly zabývající se touto aplikací je uvést další konrétnější obecně platné informace a nabídnout příklad praktické implementace. Vzhledem k faktu, že byl o tuto aplikaci projeven zájem z řad dalších uživatelů, bude její vývoj pokračovat dále, jen se přesune na server `http://mozdev.org` nebo `https://addons.mozilla.org`.

Další vývoj se bude snažit vyřešit problém s filtrováním dle DOM elementů, který je nyní nefunkční z důvodu nedostatečné podpory ze strany platformy a přidání filtrování binárních souborů. Uživatelé by určitě také přivítali barevné odlišení změn mezi uloženými verzemi dokumentů, pokročilejší metody přihlašování se na server a opětovné odhlášení,

testování celých adresářových struktur, RSS, ... Požadovaných vlastnostní je určitě hodně a DocWatcher by se jednou mohl kvalitou a funkcemi příblížit špičkám, jako je WebSite-Watcher. Nespornou výhodu má však DocWatcher již nyní – je to první aplikace tohoto zaměření, která funguje v operačních systémech Windows, Linux, Mac OS X a dalších, což pro uživatele znamená stejné nastavení doma, ve škole i v práci.

Použití tohoto nástroje již záleží na představivosti a potřebách uživatele, avšak díky toho, že svět kolem nás je velmi dynamický a mění se každým okamžikem, můžeme s jeho pomocí pohodlněji sledovat vydání nových článků na téma, které nás zajímá, může nás informovat o vydání nové verze softwarového produktu, nebo jen prostě s jeho pomocí nezmeškáme oblíbený oběd v jídelně s on-line jídelním lístkem.

Literatura

[1] McFarlane, N.: *Rapid Application Development with Mozilla.* P T R Prentice-Hall, 2003. ISBN 0-13-142343-6
http://www.informit.com/content/downloads/perens/0131423436_pdf.zip.

[2] Turner, D., Oeschger, I.: *Creating XPCOM Components.* Brownhen Publishing, 2003. http://www.mozilla.org/projects/xpcom/book/cxc/.

[3] WWW stránky. Best web site monitoring tool.
http://www.donationcoder.com/Reviews/Archive/WebWatchers/index.html.

[4] WWW stránky. Chrome Registration.
http://developer.mozilla.org/en/docs/chrome.manifest.

[5] WWW stránky. Core JavaScript 1.5 Guide:Inheritance.
http://developer.mozilla.org/en/docs/
Core_JavaScript_1.5_Guide:Inheritance.

[6] WWW stránky. Core JavaScript 1.5 Guide:Operators:Special Operators.
http://developer.mozilla.org/en/docs/
Core_JavaScript_1.5_Guide:Operators:Special_Operators.

[7] WWW stránky. Extension Versioning, Update and Compatibility.
http://developer.mozilla.org/en/docs/
Extension_Versioning%2C_Update_and_Compatibility.

[8] WWW stránky. File-naming convention for XUL toolkit applications.
https://bugzilla.mozilla.org/attachment.cgi?id=157639&action=view.

[9] WWW stránky. Find xpidl.exe in Gecko SDK for compiling IDL files.
http://mozilla-firefox-extension-dev.blogspot.com/2004/11/
find-xpidlexe-in-gecko-sdk-for.html.

[10] WWW stránky. Gecko SDK. http://developer.mozilla.org/en/docs/Gecko_SDK.

[11] WWW stránky. How to Build an XPCOM Component in Javascript.
http://developer.mozilla.org/en/docs/
How_to_Build_an_XPCOM_Component_in_Javascript.

[12] WWW stránky. IDL Author's Guide – Rules and Syntax.
http://www.mozilla.org/scriptable/xpidl/idl-authors-guide/rules.html.

[13] WWW stránky. Implementing XPCOM components in JavaScript.
http://kb.mozillazine.org/Implementing_XPCOM_components_in_JavaScript.

[14] WWW stránky. Install Manifests.
http://developer.mozilla.org/en/docs/Install_Manifests.

[15] WWW stránky. JavaScript XPCOM Components Status – Draft 2.
http://www.mozilla.org/scriptable/js-components-status.html.

[16] WWW stránky. Mozilla FAQ.
http://mozilla.gunnars.net/mozfaq_general.html.

[17] WWW stránky. Mozilla Source Code (HTTP/FTP).
http://developer.mozilla.org/en/docs/Download_Mozilla_Source_Code.

[18] WWW stránky. New in JavaScript 1.7.
http://developer.mozilla.org/en/docs/New_in_JavaScript_1.7.

[19] WWW stránky. Resource Description Framework.
http://en.wikipedia.org/wiki/Resource_Description_Framework.

[20] WWW stránky. Resource Description Framework (RDF). http://www.w3.org/RDF/.

[21] WWW stránky. Setting up extension development environment.
http://kb.mozillazine.org/Setting_up_extension_development_environment.

[22] WWW stránky. Structure of an Installable Bundle.
http://developer.mozilla.org/en/docs/Bundles.

[23] WWW stránky. Style Properties.
http://xulplanet.com/references/elemref/ref_StyleProperties.html.

[24] WWW stránky. Styling a Tree.
http://www.xulplanet.com/tutorials/xultu/treestyle.html.

[25] WWW stránky. The GUI Toolkit, Framework Page.
http://www.free-soft.org/guitool/.

[26] WWW stránky. The Life Of An HTML HTTP Request.
http://www.mozilla.org/docs/url_load.html.

[27] WWW stránky. The Prototype.
http://www.debreuil.com/docs/ch01_Prototype.htm.

[28] WWW stránky. The Unicode Character Code Charts By Script.
http://www.unicode.org/charts/.

[29] WWW stránky. Using XPCOM in JavaScript without leaking.
http://www.mozilla.org/scriptable/avoiding-leaks.html.

[30] WWW stránky. Valid Application Versions.
https://addons.mozilla.org/en-US/firefox/pages/appversions.

[31] WWW stránky. Venkman Introduction.
http://developer.mozilla.org/en/docs/Venkman_Introduction.

[32] WWW stránky. Versions of Gecko.
http://developer.mozilla.org/en/docs/Gecko.

[33] WWW stránky. WebSite-Watcher features. http://www.aignes.com/features.htm.

[34] WWW stránky. What Is RDF. http://www.xml.com/pub/a/2001/01/24/rdf.html.

[35] WWW stránky. XPCNativeWrapper.
http://developer.mozilla.org/en/docs/XPCNativeWrapper.

[36] WWW stránky. XPIDL. http://developer.mozilla.org/en/docs/XPIDL.

[37] WWW stránky. XUL Planet. http://xulplanet.com/.

[38] WWW stránky. XUL Tag Implementation.
http://wiki.mozilla.org/XUL:XUL_Tag_Implementation.

[39] WWW stránky. XULRunner.
http://developer.mozilla.org/en/docs/XULRunner.