FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY
UNIVERZITY KOMENSKÉHO
BRATISLAVA

Renderovanie XML dokumentov pri WYSIWYG editovaní

Diplomová práca

Martin Kollár **2007**

Renderovanie XML dokumentov pri WYSIWYG editovaní

DIPLOMOVÁ PRÁCA

Martin Kollár

UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY
KATEDRA INFORMATIKY

Informatika

Vedúci diplomovej práce
prof. RNDr. Branislav Rovan, PhD

BRATISLAVA 2007

Čestne prehlasujem, že túto diplomovú prácu som vypracoval samostatne len s použitím uvedenej literatúry a s odbornou pomocou vedúceho práce.

Bratislava, máj 2007 Martin Kollár

V prvom rade by som chcel poďakovať svojmu školiteľovi prof. RNDr. Branislavovi Rovanovi, PhD za vedenie, konzultácie a inšpiráciu.

Tiež by som sa chcel poďakovať Tomášovi Studvovi za jeho plodné otázky návrhy a komentáre.

Veľká vďaka patrí Monike a rodine za podporu a strpenie.

Abstrakt

XML jazyk sa stáva univerzálnym komunikačným formátom. S jeho rozšírením vznikla požiadavka aj na zobrazenie dát v tomto formáte a neskôr aj na ich editovanie. V súčasnosti existuje množstvo rendererov mnohých XML jazykov, vzniká však potreba tieto XML dokumenty editovať WYSIWYG technikou. Táto práca sa zaoberá návrhom frameworku pre tvorbu rendererov XML dokumentov, vhodných pre WYSIWYG editáciu. Náš framework zahŕňa širokú podporu pre efektívne riešenie problémov a požiadaviek vznikajúcich pri WYSIWYG editovaní XML dokumentov. Veľký dôraz sa kladie na rozšíriteľnosť, flexibilitu, modularitu a intuitívnu implementáciu, čo vedie implementátora k správnemu použitiu.

Kľúčové slová : XML, WYSIWYG renderer, framework, XML editovanie, MVC

Predhovor

Cieľom tejto práce bolo navrhnúť a implementovať framework pre tvorbu rendererov, vhodných pre WYSIWYG editáciu XML dokumentov. Tento framework má byť všeobecný a robustný, čím sa vytvorí široká podpora pre efektívne riešenie najčastejších problémov a požiadaviek vznikajúcich pri WYSIWYG editovaní XML dokumentov.

Prínos tejto práce vidíme v navrhnutí a implementovaní nového prístupu a nových procesov pri tvorbe WYSIWYG XML rendererov. Tento framework nie je viazaný na konkrétny XML jazyk ani na konkrétnu grafickú knižnicu, preto je použiteľný na vytvorenie rendereru, ktorý vykresľuje ľubovolný XML jazyk, spôsobom, ktorý je vhodný pre WYSIWYG editáciu. Tento framework má otvorený kód čím prispieva rozširovať počet open source WYSIWYG XML editorov, ktorý je v súčasnosti nízky a ich kvalita je nedostačujúca. Prínosom tejto práce je aj to, že pomáha splniť cieľ projektu Euromath2 [6], ktorým je vytvoriť platformu pre tvorbu WYSIWYG XML editorov.

Vychádzali sme zo skúseností v oblasti tvorby XML rendererov a poznatkov o nových XML formátoch pre kancelárske dokumenty. Vychádzali sme tiež z literatúry, ktorá popisuje najčastejšie návrhové vzory, ktoré sú určené pre tvorbu modulárnych, flexibilných a rozšíriteľných softvérových produktov.

Obsah

Úvod 7

Cieľ 10

1 Existujúce XML renderery 11
- 1.1 Formatting Objects Processor (FOP) 11
  - 1.1.1 Spracovanie XSL-FO od súboru k obrazovke 11
  - 1.1.2 Poučenia pre návrh nášho frameworku 15
- 1.2 Batik 17
  - 1.2.1 Spracovanie SVG od súboru k obrazovke 17
  - 1.2.2 Poučenia pre návrh nášho frameworku 19

2 XML formáty pre dokumenty 20
- 2.1 Open XML 20
  - 2.1.1 Formát a štruktúra súboru Open XML 20
  - 2.1.2 WordprocessingML 22
  - 2.1.3 Custom schema 22
- 2.2 Open Document 23
  - 2.2.1 Formát a štruktúra súboru 24
  - 2.2.2 XForms 24

3 Framework 26
- 3.1 Prehľad procesov vo frameworku 26
- 3.2 Architektúra frameworku a základné pojmy 30
- 3.3 NodeTree 31
- 3.4 Atribúty a Vlastnosti 32
  - 3.4.1 Prínos konverzie atribútov na vlastnosti 32
  - 3.4.2 Tvorba vlastností z atribútov 33
  - 3.4.3 Tvorba atribútov z vlastností 34
  - 3.4.4 Aktualizácia vlastností 34
  - 3.4.5 Popis implementácie modulu Property 35
  - 3.4.6 Použitie modulu Property 36
- 3.5 Šetrenie pamäte v module NodeTree 37
  - 3.5.1 Zahadzovanie 37
  - 3.5.2 Nahrávanie 37
  - 3.5.3 Nepriamy prístup k uzlom stromu NodeTree 38
- 3.6 Zmeny dokumentu vo fáze NodeTree 39
  - 3.6.1 Editačná požiadavka 40
  - 3.6.2 Použitie editačných požiadaviek 40
  - 3.6.3 Príjem editačnej požiadavky 41
  - 3.6.4 Príjem NodeTree editačnej požiadavky 41

3.6.5 Príjem XML editačnej požiadavky 42
3.6.6 Poznámky pre implementátora frameworku 43
3.6.7 Zmena XML dokumentu 43
3.6.8 Aktualizácia NodeTree 47
3.6.9 Aktualizácia NodeTree a nahrávanie 47
3.6.10 Informovanie o nahratej zmene 48
3.7 Rozmiestnenie 49
3.7.1 Základný koncept rozmiestňovania 49
3.7.2 Statické rozmiestňovanie 50
3.7.3 Dynamické rozmiestňovanie a vykresľovanie 52
3.7.4 Aktualizácia rozmiestnenia 54
3.7.5 Dirty layout a dirty rendering 57
3.7.6 Napojenie modulu LayoutManager na modul NodeTree 59
3.7.7 Šetrenie pamäte v module AreaTree 59
3.8 Renderovanie a začlenenie do MVC 60

4 Záver **63**

Zoznam skratiek **64**

Literatúra **66**

Prílohy **67**
Príloha 1 67
Príloha 2 68
Príloha 3 69
Príloha 4 70
Príloha 5 71
Príloha 6 72
Príloha 7 73
Príloha 8 74
Príloha 9 75
Príloha 10 76
Príloha 11 77

Zoznam obrázkov

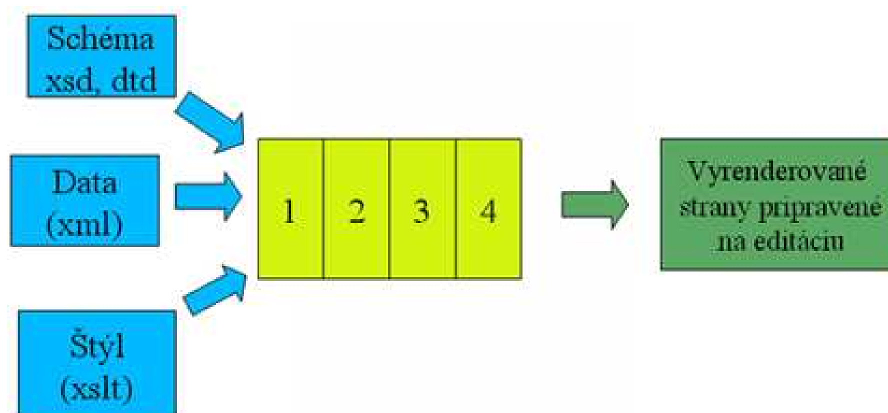
1 Fázy spracovania XML dokumentu . . . 7
2 XSLT transformácia . . . 8
3 Pageregions [2] . . . 12
4 Flows a spans [2] . . . 13
5 Batik High-Level Architecture [1] . . . 17
6 Štruktúra WordprocessingML súboru sample.docx . . . 21
7 Rendering process . . . 27
8 Rendering process 2 . . . 27
9 Rozmiestňovanie a vykresľovanie . . . 28
10 NodeTree pre WordprocessingML dokument . . . 31
11 Brzdiaci element . . . 34
12 Zahadzovanie podstromu NodeTree . . . 39
13 Dirty editovanie na úrovni NodeTree . . . 44
14 Editovanie XML dokumentu s XSLT transformáciou . . . 45
15 Aktualizácia NodeTree s čiastočnou XSLT transformáciou . . . 46
16 Chyba pri aktualizácii NodeTree po nahratí . . . 48
17 Správne vymazávanie zahodeného uzla . . . 49
18 Statický layout . . . 51
19 Dynamický layout . . . 53
20 Dynamický layout . . . 55
21 Lokálny layout . . . 55
22 Inteligentný layout . . . 56
23 Dirty rendering . . . 57
24 Dirty layout . . . 58
25 Zahadzovanie podstromov AreaTree . . . 60
26 MVC s klasickým návrhom komunikácie . . . 61
27 MVC s upraveným návrhom komunikácie . . . 61
28 Traverzovanie XML dokumentu . . . 67
29 Komponentový diagram frameworku . . . 69
30 Jadro modulu Property . . . 70
31 Vytváranie zoznamu vlastností s default hodnotami . . . 71
32 Konverzia atribútov na vlastnosti . . . 72
33 Balík org.apache.batik.parser . . . 73
34 Zahadzovanie a nahrávanie uzlov stromu NodeTree . . . 74
35 Zmeny dokumentu vo vrstve NodeTree . . . 75
36 Editačné požiadavky na zmenu XML dokumentu . . . 76
37 Zmeny dokumentu vo vrstve NodeTree . . . 77

Úvod

V tejto kapitole stručne uvedieme dôvody rozšírenia XML jazyka a vysvetlíme základný proces pri vykresľovaní XML dokumentu.

XML (Extensible Markup Language) je značkovací jazyk štandardizovaný konzorciom W3C. Tento jazyk slúži najmä na ukladanie, popis a posielanie dát. Rozšírenie tejto technológie nastalo vďaka jej výhodám ako sú jednoduchosť, čitateľnosť, prehľadnosť a oddelenie dát od ich prezentácie. K jej rozšíreniu prispeli aj kvalitné voľne dostupné nástroje umožňujúce spracovanie XML dokumentov (Xalan, Xerces, Forrest, Ant), a tým sa umožnil vývoj ďalších aplikácií pracujúcich s touto technológiu.

V súčasnosti existujú na nástroje, ktoré dokážu vykresliť mnohé XML jazyky. Kostra procesu pri vykresľovaní XML dokumentov je zhruba nasledovná :



Obrázok 1: Fázy spracovania XML dokumentu

Vstupom sú samotné XML dáta, ďalej schéma, vzhľadom na ktorú sa tieto dáta kontrolujú (DTD [1] [2] alebo XSD [3] [4]) a štýl, teda transformácia, ktorá povie ako majú byť tieto dáta zobrazené (napríklad ich transformuje do jazyka XSL-FO [5] [6]). Proces sa dá rozdeliť do štyroch krokov :

1 Príprava na spracovanie

[1]**DTD** (Document Type Definition) - Značkovací jazyk, ktorý sa používa na automatickú kontrolu štruktúry XML alebo SGML dokumentu

[2]**SGML** (Standard Generalized Markup Language) - Značkovací jazyk, na popis informácii. Tak isto je to aj ISO štandard. Je na ňom založený napríklad jazyk HTML

[3]**XSD** (XML Schema Definition) - Inštancia XML Schémy

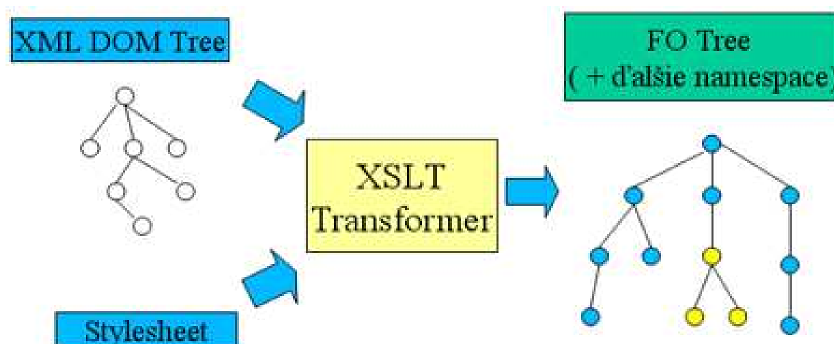
[4]**XML Schema** - Jazyk vyvíjaný W3C konzorciom na popis štruktúry XML dokumentov, ktorý nemá nedostatky DTD

[5]**XSL-FO** (XSL Formatting Objects) - XML jazyk na formátovanie dokumentov, je časťou jazyka XSL

[6]**XSL** (eXtensible StyleSheet Language) - Rodina jazykov, ktoré určujú ako majú byť dokumenty v XML formáte formátované alebo transformované. Patria sem jazyky XSLT, XSL-FO a XPath

V tejto fáze sa parsuje XML súbor, pričom mu môžu byť pridané nedátové informácie, napríklad ak je z XML súboru vytvorený jeho DOM [7] , tak sa ku každému elementu pridá atribút s identifikačným číslom tohto elementu. Ďalej sa v tejto fáze inicializujú XML schémy a validujú sa dáta vzhľadom k týmto schémam.

2 Transformácia



Obrázok 2: XSLT transformácia

XML dokument môže byť počas parsovania alebo po jeho skončení transformovaný napríklad pomocou jazyka XSL. Úlohou transformácie je upraviť XML tak, aby obsahovalo dostatok informácii o svojom vzhľade a aby sa preniesli nedátové informácie pridané v predchádzajúcich krokoch. Pri stránkových dokumentoch sa často využíva transformácia do XSL-FO, ale napríklad prezentačná časť jazyka MathML [8] už nepotrebuje ďalšie informácie o svojom vzhľade.

3 Rozmiestnenie (Layout)

V tejto fáze sa zisťujú veľkosti a rozmiestňujú sa grafické prvky. Rozmiestňovanie a zisťovanie veľkosti sú prepojené úlohy, tento princíp sa využíva v každom nástroji na vizualizáciu. Zohľadňujú sa atribúty, ktoré môžu byť špecifikované v štýle (centrovanie, veľkosť okrajov, font, . . .)

4 Vykresľovanie (Renderovanie)

Výstupom sú dáta vypočítané pri rozmiestňovaní. Vytvoria sa grafické entity (slová, riadky, bloky textu, matematické a iné znaky). Pri vytváraní týchto znakov sa berú do úvahy atribúty, ktoré môžu byť špecifikované v štýle (font, farba písma a pozadia, dekorácia okrajov, . . .)

Po týchto fázach je XML dokument vykreslený na obrazovku alebo exportovaný do rôznych formátov, napríklad PDF, DVI, SVG alebo TeX. Tieto nástroje sú statické v

[7] **DOM** (Document Object Model) - Spôsob prístupu k XML dokumentu, každá entita v XML dokumente je reprezentovaná jedným uzlom v stromovej štruktúre.

[8] **MathML** (Mathematical Markup Language) - XML jazyk na zápis matematických symbolov, vzorcov a formúl

zmysle, že vykreslia dokument, ale neumožňujú jeho WYSIWYG [9] editáciu. V súčasnosti existuje málo XML jazykov, pre ktoré existujú WYSIWYG editory s otvoreným kódom a navyše kvalita týchto editorov je často malá, vyskytujú sa v nich chyby, nepodporujú editovanie všetkých čŕt jazyka, sú zle zdokumentované a neexistuje k nim podpora . V komerčnej sfére sa tieto nedostatky nevyskytujú, ale tvorcovia týchto editorov si komplikovaný proces editácie strážia. V komerčných editoroch aj v editoroch s otvoreným kódom sú procesy WYSIWYG editácie naviazané na konkrétny XML jazyk, pretože *"neexistuje WYSIWYG editačná technika pre všeobecné XML, vždy sa dá takýmto spôsobom editovať iba konkrétny XML jazyk"* (Vyletel, 2004, s. 15) [10] , napríklad ChemML [11] .

Softwarový tím, ktorý chce vytvoriť WYSIWYG editor pre konkrétny XML jazyk, musí vytvoriť program kde jedna jeho časť sa stará o aplikačnú logiku editora a druhá časť, ktorú nazývame renderer, ktorá tento konkrétny jazyk vykresľuje spôsobom vhodným pre WYSIWYG editáciu. Open source projekt Euromath 2 [6] si dáva za cieľ vytvoriť platformu pre tvorbu WYSIWYG XML editorov. Táto diplomová práca výrazne pomôže splniť cieľ tohto projektu. Vytvorili sme framework [12] pre tvorbu rendererov, vhodných pre WYSIWYG editáciu XML dokumentov. Tento framework nie je naviazaný na konkrétny XML jazyk ani na konkrétnu grafickú knižnicu, preto môže byť použitý pre tvorbu ľubovoľného rendereru, ktorý vykresľuje ľubovoľný XML jazyk. Náš framework je voľne dostupný a má otvorený kód. Framework je modulárny a renderer, ktorý je pomocou neho vytvorený je flexibilný a ľahko rozšíriteľný, vhodný pre stranovo orientované dokumenty, vhodný pre nie stranovo orientované dokumenty, vhodný pre tvorbu rendererov malých dokumentov, kde sa kladie dôraz najmä na rýchlosť a aj veľkých dokumentov, kde sa kladie dôraz na šetrenie pamäte. Framework podporuje výskyt viacerých menných priestorov v jednom dokumente. Framework sa skladá z dvoch hlavných modulov, NodeTree a AreaTree, slúžiacich na zachytenie XML dát do objektovej podoby, ktorá zohľadňuje zložitosť konkrétneho XML jazyka a je vhodná na ďalšie spracovanie a menenie a vytvorenie a rozmiestnenie virtuálnych grafických oblastí, ktoré sú pripravené na prevedenie do reálnych pomocou konkrétnej grafickej knižnice. Jeho výhody sú dosiahnuté pomocou techník ako je čiastočné rozmiestňovanie a vykresľovanie, dynamické rozmiestňovanie a renderovanie, zahadzovanie častí interných štruktúr, dirty editing a rozmiestňovanie a vykresľovanie objektov (napríklad strán) mimo poradia v akom sa nachádzajú v XML dokumente. Jednotlivé pojmy a techniky sú vysvetlené v kapitole 3. Pri návrhu frameworku nám pomohli poznatky, ktoré sme získali z existujúcich rendererov. Tieto poznatky sú popísané v kapitole 1. Študovaním najnovších a najrozšírenejších XML formátov pre kancelárske dokumenty sme získali poznatky, ktoré prispeli k širšej použiteľnosti nášho frameworku. Tieto poznatky sú popísané v kapitole 2.

[9] **WYSIWYG** (What You See Is What You Get) - Spôsob editácie dokumentov, pri ktorom je verzia zobrazená na obrazovke vzhľadom totožná s výslednou verziou dokumentu

[10] Tibor Vyletel, WYSIWYG editácia XML dát, 2004 [17]

[11] **ChemML** (Chemical Markup Language)- XML jazyk na zápis chemických symbolov, vzorcov a formúl

[12] **Framework** je množina kooperujúcich tried, ktoré tvoria návrh aplikácie a jej čiastočnú implementáciu. Na framework sa dá pozerať ako na kostru, ktorá je spoločná pre istú triedu softwarových produktov. Framework sa na rozdiel od knižnice vyznačuje obráteným riadením, teda riadiacu časť aplikácie obsahuje framework, ktorý volá špecifický kód tried, ktoré predstavujú zvyšnú časť aplikácie.

Cieľ

Cieľom tejto diplomovej práce je vytvoriť framework pre tvorbu renderov, vhodných pre WYSIWYG editáciu XML dokumentov.

Študovaním rendererov a XML jazykov sme dospeli k nasledujúcim požiadavkám na náš framework :

- Pomocou frameworku sa dá vytvoriť renderer vhodný pre veľké dokumenty, kde sa kladie dôraz na šetrenie pamäte, ale aj renderer vhodný pre malé dokumenty, kde sa kladie dôraz na rýchlosť (napríklad matematické rovnice a vzorce)
- Framework je použiteľný pre tvorbu rendererov stranovo orientovaných XML dokumentov aj nie stranovo orientovaných XML dokumentov.
- Framework podporuje výskyt viacerých menných priestorov v jednom dokumente
- Framework je modulárny a rozšíriteľný
- Renderer, ktorý je pomocou frameworku vytvorený je modulárny, flexibilný a rozšíriteľný
- Framework nie je viazaný na konkrétnu grafickú knižnicu alebo framework
- Framework je dostatočne všeobecný a zároveň robustný, čím sa minimalizuje množstvo implementácie, ktorú musí používateľ frameworku [13] urobiť.

[13]Programátora, ktorý bude používať náš framework za účelom vytvorenia renderera budeme v tejto práci nazývať používateľ frameworku alebo implementátor frameworku.

1 Existujúce XML renderery

Táto kapitola prináša informácie o existujúcich XML redereroch. Popíšeme dátový model, ktorý sa vytvorí pri parsovaní XML dokumentov, jeho úpravy, techniky a algoritmy, pomocou ktorých sa rozmiestni a vykreslí dokument. Upozorňuje na problémy nedostatky, ktoré môžu pri návrhu WYSIWYG XML editora vzniknúť. Na základe týchto informácií navrhneme a implementujeme všeobecný framework na tvorbu WYSIWYG XML rendererov. Ako referenčné renderery sme vybrali dva nástroje s otvoreným kódom s veľmi odlišnými črtami.

1.1 Formatting Objects Processor (FOP)

FOP je nástroj od spoločnosti Apache, ktorý vie vstupné XSL-FO vykresliť pomocou grafických objektov knižnice AWT [14] . XSL-FO je komplexný jazyk, štandardizovaný konzorciom W3C [15] na popis toho, ako má byť dokument zobrazený. FOP implementuje väčšinu čŕt jazyka XSL-FO. Medzi základné patrí rozdelenie na strany. Strany sa ďalej delia na 5 regiónov region-before, region-start, region-body, region-end a region-after ako môžme vidieť na obrázku 3.

Region-body je ďalej rozdelený na oblasti before-float-reference, main reference a footnote-reference. Text je ďalej rozdelený v každom z regiónov do stĺpcov (flows) , cez ktoré môžu viesť vodorovné oblasti (spans). Situáciu znázorňuje obrázok 4. V stĺpcoch sa text rozdeľuje do blokov. V blokoch sa pomocou riadkových formátovacích objektov môžu špecifikovať ďalšie vlastnosti.

1.1.1 Spracovanie XSL-FO od súboru k obrazovke

Hlavný proces spracovania dokumentu v jazyku XSL-FO nástrojom FOP sa dá rozdeliť do nasledovných fáz :

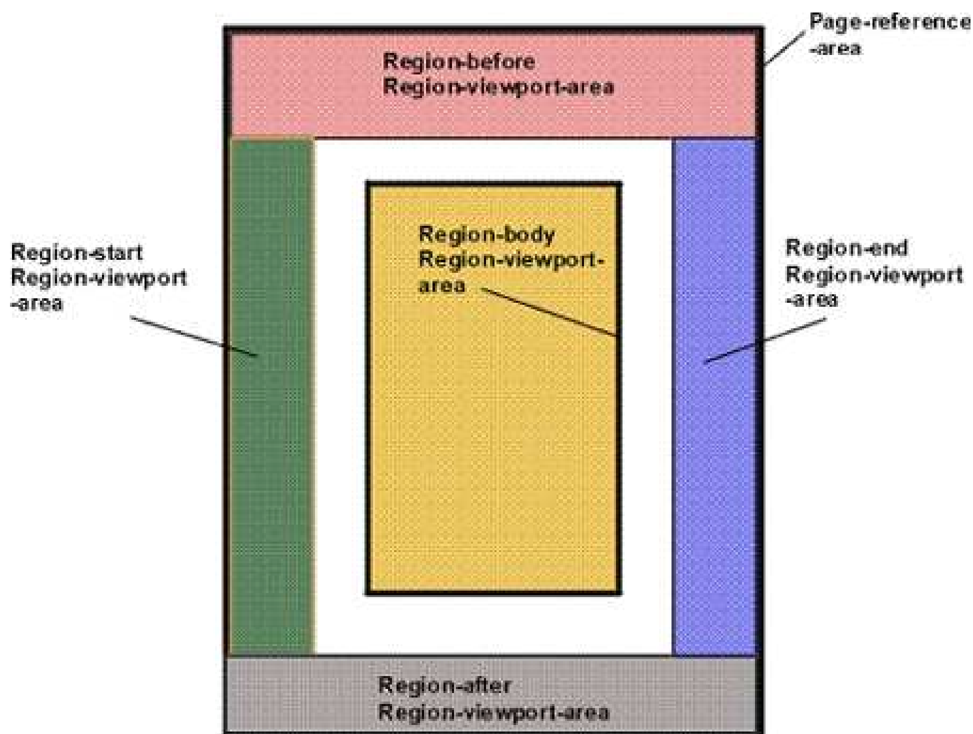
- Parsovanie

  XML dokument môže byť spracovaný dvoma spôsobmi (SAX [16] alebo DOM). V prvom prípade sa dokument prechádza sériovo, nič sa interne neukladá a v každom okamihu je k dispozícii len istá časť dokumentu, pri tomto prechádzaní sa vyvolávajú SAX udalosti. Tento prístup je rýchlejší ako DOM a je nenáročný na pamäť. Pri DOM je vytvorený strom objektov, ktoré reprezentujú jednotlivé elementy v XML. Zdalo by sa, že pre WYSIWYG editovanie XML dokumentov je výhodnejší DOM, ale pri veľkých dokumentoch (500 strán) je zbytočné uložiť celý dokument do pamäte. FOP navyše nepotrebuje mať opätovný prístup k renderovanému XML dokumentu

[14] **AWT** (Abstract Window Toolkit) - Štandardne aplikačné rozhranie pre Java programátorov, ktoré poskytuje grafické užívateľské rozhrania (GUI)

[15] **W3C** (World Wide Web Consorcium) - Konzorciu, ktoré vyvíja spolupracujúce technológie, špecifikácie, príručky, software a nástroje, ktoré vedú k zlepšovaniu internetu

[16] **SAX** (Simple API for XML) - Spôsob prístupu k XML dokumentu, dokument je čítaný sériovo, pričom sú volané udalosti



Obrázok 3: Pageregions [2]

kvôli jeho zmene (editácii), používa preto SAX a pri prechádzaní dokumentu si tvorí vlastný strom objektov, FOTree.

- Vytváranie stromu FOTree

  FOTree je budovaný pomocou SAX udalostí, ktoré sú vyvolávané napríklad pre začiatok elementu, koniec elementu a textové dáta. Je vytvorené mapovanie medzi atribútmi elementu a vlastnosťami vo FONode (uzol stromu FOTree). FOP vie spracovať aj elementy, ktoré nie sú z XSL-FO menného priestoru. Sú to buď elementy z iného známeho menného priestoru, tieto sú uložené vo forme DOM (napríklad SVG [17]) alebo sú to elementy predstavujúce rôzne riadiace inštrukcie napríklad pre renderer, tieto elementy sa zohľadnia v interných objektoch FOPu a nakoniec môže ísť o nerozpoznané elementy, pre ktoré FOP vytvorí generický DOM alebo jednoduchý objekt s minimom informácii a funkcionality (dummy object).

  FOTree sa postupne pretransformuje do inej stromovej štruktúry, ktorú nazývame AreaTree a bližšie ju popíšeme neskôr. O tvorbu AreaTree sa stará trieda `AreaTreeHandler` tak, že pri budovaní FOTree sa po vytvorení niektorých uzlov stromu FOTree vyvolajú udalosti, ktoré sú delegované do tejto triedy a spúšťajú ďalšie procesy vo FOPe. Napríklad koniec elementu page-sequence deleguje túto informáciu do triedy `AreaTreeHandler` a ten spúšťa rozmiestňovací proces, ktorého výsledkom je

[17] **SVG**(Scalable Vector Graphics) - Jazyk pre dvojdimenzionálnu grafiku vo formáte XML

Obrázok 4: Flows a spans [2]

AreaTree. FOP si nepamätá celý strom FOTree, ale vždy len elementy prislúchajúce jednej sekvencii strán. Ďalším príkladom je udalosť, ktorá je vyvolaná po dokončení stromu NodeTree. Táto informácia je tiež delegovaná do triedy `AreaTreeHandler`, a tá zavolá metódu renderera, ktorá dokončí výslednú prezentáciu dokumentu.

- Vytváranie vlastností v uzloch stromu FOTree

  V tejto fáze sa transformujú atribúty elementu na vlastnosti v objektoch FONode (uzol stromu FOTree). Do úvahy treba brať :

  - Aká je základná množina vlastností
  - Dedičnosť : Niektoré vlastnosti môžu byť zdedené
  - Adopcia : Niektoré objekty sa môžu stať rodičmi iných objektov, napríklad Markers [XSL-FO]
  - Viaceré menné priestory : Treba zohľadniť vlastnosti cudzích menných priestorov a ich konverziu pri dedičnosti

- ◦ Výrazy : Treba vyrátať XSL-FO výrazy

  FOP obsahuje základné typy vlastností ako Number, String, ColorType, Length, Space, ..., od ktorých sú odvodené všetky ostatné. Ku každému atribútu sa nájde trieda `PropertyMaker` a pomocou nej sa vyrobí vlastnosť. Nie všetky vlastnosti môžu byť v tejto fáze úplne určené lebo závisia na informáciách o rozmiestnení. Príkladom môže byť situácia keď na párnych stranách je text čierny a na nepárnych modrý. Vlastnosť uzla farba, teda nemôže byť vypočítaná v tejto fáze.

- Rozmiestňovanie (layout) :

  V tejto fáze sa na základe uzlov stromu FOTree vytvoria a rozmiestnia oblasti (Areas). Oblasť je objekt, ktorý obsahuje všetky informácie o tom, ako a kde má byť zobrazený. Za vytvorenie a rozmiestnenie oblastí zodpovedajú rozmiestňovací manažéri (layout managers). Rozmiestňovací manažér je zviazaný s FONode, ale nie vždy (napríklad tabuľky). Úlohou rozmiestňovacích manažérov je vytvoriť z FOTree AreaTree. AreaTree je strom objektov dvoch typov, prvým sú strany a druhým oblasti (Areas). Rozmiestňovací manažéri počas rozmiestňovania vytvárajú uzly stromu AreaTree, pričom majú informáciu o lokálnom kontexte, zalamujú jednotlivé oblasti (breaking), udržujú medzery medzi oblasťami (spacing) a vkladajú oblasti do strán.

  Jedným typom oblastí sú bloky, tieto sa skladajú z riadkových oblastí (LineAreas) a tie z vnútroriadkových oblastí (InlineAreas) a tie obsahujú textové oblasti (TextAreas), môže však dôjsť aj k jemnejšiemu deleniu a to na slová a znaky. Oblasti obsahujú informácie o zviazaní s ostatnými oblasťami a medzery od ostatných oblastí, tieto informácie sa uvoľnia z pamäte, keď sa dokončí rozmiestnenie strany.

  Princíp rozmiestňovania si vysvetlíme na práci rozmiestňovacieho manažéra pre bloky (BlockManager). BlockManager prislúcha uzlu FOBlock stromu FOTree. Tento uzol prislúcha elementu `<xsl-fo:block>`. BlockManager si vyžiada oblasti, ktoré vytvorili manažéri, ktorí rozmiestňujú deti uzla FOBlock. Tieto oblasti postupne pridáva do bloku až kým nie je plný (veľkosť bloku môže byť obmedzená napríklad veľkosťou strany), potom urobí rozhodnutie o zalomení. Na zalamovanie používa implementáciu abstraktnej triedy `AbstractBreaker`, pri čom sa používa implementácia abstraktnej triedy `BreakingAlgorithm`, ktorý na svoju prácu môže využívať informácie o miestach, kde môže dôsť k zalomeniu (break posibility). Rovnaký princíp sa používa napríklad pri rozmiestňovaní strán alebo pri rozmiestňovaní riadkov, kde sa berie do úvahy aj delenie slov (hyphenation). Na rozmiestnenie a určenie všetkých vlastností niektorých oblastí je nutné, aby sa najskôr rozmiestnili strany, tieto oblasti budeme nazývať nedoriešené. Sú to napríklad poznámku pod čiarou, odkazy alebo citácie, ktorých súčasťou je číslo strany kde sa daný odkaz alebo citácia nachádza. Pre nedoriešené oblasti je vyhradený predpokladaný priestor. Po dokončení každej strany sa kontroluje, či nemôže byť nejaká nedoriešená oblasť dokončená.

- Hľadanie možností zalomenia (break posibilities) :

Rozmiestňovací manažéri sú zodpovední za zalamovanie. Jeden z možných algoritmov je založený na možnostiach zalomenia (ďalej len MZ). Rozmiestňovací manažér si potom vyberie najlepšiu z nich. Vo všeobecnosti manažér požiada svojich detských manažérov o MZ. MZ obsahujú rôzne príznaky, napríklad smer ukladania oblastí, či išlo o donútené zalamovanie alebo či ide o poslednú oblasť. Tento proces sa šíri smerom dole až kým sa nenarazí na atomické oblasti ako znak abecedy alebo grafický objekt. Rozmiestňovací manažéri sa teda budujú od hora dole (top-down) a možnosti zalomenia zdola hore (bottom-up).

- Generovanie čŕt uzlov stromu AreaTree :

  Ako sme už spomenuli, AreaTree je výsledkom práce rozmiestňovacích manažérov, ide o internú reprezentáciu výsledného dokumentu. Koncept tejto reprezentácie je zhodný s oblasťami XSL-FO popísanými konzorciom W3C. Dáta v AreaTree sú minimálne v zmysle, že obsahujú len informácie o zobrazení a nie informácie o tom, ako sa vypočítali a nie sú medzi nimi žiadne závislosti. Tieto dáta v oblastiach sa nazývajú črty (napríklad farba, dekorácie alebo umiestnenie v závislosti od rodičovskej oblasti) a sú určené príslušným rozmiestňovacím manažérom na konci procesu rozmiestňovania.

- Vykresľovanie :

  Úlohou rendereru je z AreaTree vytvoriť výsledný formát dokumentu, napríklad PDF [18] alebo vyprodukovať grafický obsah pomocou konkrétnej grafickej knižnice alebo frameworku. Rendereru sú posielané strany z AreaTree buď hneď, ako sú vytvorené a po ich vytvorení sú odstránené z pamäte, čo šetrí pamäťové nároky alebo až po tom, čo je vytvorený celý AreaTree. Tieto strany môžu byť posielané rendereru aj mimo poradia v akom sa nachádzajú vo výslednom dokumente, ak to renderer podporuje.

1.1.2 Poučenia pre návrh nášho frameworku

V tejto časti popíšeme niektoré konštrukcie a procesy vyskytujúce sa v nástroji FOP. Časti týchto konštrukcii a procesov sú po úprave a doplnení použiteľné pre náš framework.

Študovaním tohto nástroja sme dospeli k nasledujúcim poučeniam :

- UserAgent : Framework, ktorý chceme vytvoriť má byť ľahko rozšíriteľný a flexibilný, preto je výhodné vytvoriť triedu `UserAgent`, kde sú registrované triedy zodpovedné za štandardné správanie sa frameworku. Ak používateľ frameworku potrebuje pre jazyk, ktorý chce vykresliť a editovať špecifické správanie sa frameworku alebo ďalšiu funkčnosť, tak vytvorí novú triedu (alebo triedy) dediacu od štandardnej a zaregistruje ju v triede `UserAgent` namiesto pôvodnej. Medzi triedy zabezpečujúce štandardné správanie patrí napríklad `TreeBuilder`, ktorý zabezpečuje vytvorenie stromu FOTree z elementov XML dokumentu alebo už spomínaná trieda `AreaTreeBuilder`, ktorá zabezpečuje spúšťanie ďalších fáz spracovania dokumentu.

[18] **PDF**(Portable Document Format) - je súborový formát vyvinutý firmou Adobe na ukladanie dokumentov obsahujúcich text aj obrázky nezávisle od softwaru a hardwaru, na ktorom boli vytvorené.

- Reprezentácia XML : Kvôli nevýhodám oboch štandardný reprezentácii dokumentu (DOM a SAX udalosti), je výhodné ich skombinovať a počas zachytávania SAX udalostí vytvárať vlastnú reprezentáciu dát v XML (FOTree). Nerozpoznané XML elementu sú v strome uložené vo forme DOM.

- Vytváranie stromu FOTree : Štandardná trieda `TreeHandler`, ktorá je zaregistrovaná v triede `UserAgent`, obsahuje mapovanie medzi menom elementu a inštanciu triedy `NodeMaker`, ktorá vytvára uzol stromu FOTree. Ak chceme zmeniť správanie sa uzla FONode alebo podporovať nové črty XML jazyka, stačí upraviť alebo pridať triedy NodeMaker do tohto mapovania.

- AreaTreeHandler : Táto trieda zachytáva špeciálne situácie pri tvorbe stromu FOTree, napríklad začiatok a koniec tvorby niektorých uzlov (koreň, sekvencia strán, blok). Na základe týchto situácii spúšťa proces rozmiestňovania a vykresľovania. Tiež slúži na šetrenie pamäte spôsobom popísaným v časti 1.1.1 vykresľovanie. `AreaTreeHandler` eviduje nevyriešené oblasti a zodpovedá za ich správne rozmiestnenie a dokončenie. Táto trieda je tiež registrovaná v triede `UserAgent` ako štandardná a jej nahradením sa dá zabezpečiť iný spôsob rozmiestňovania a vykresľovania.

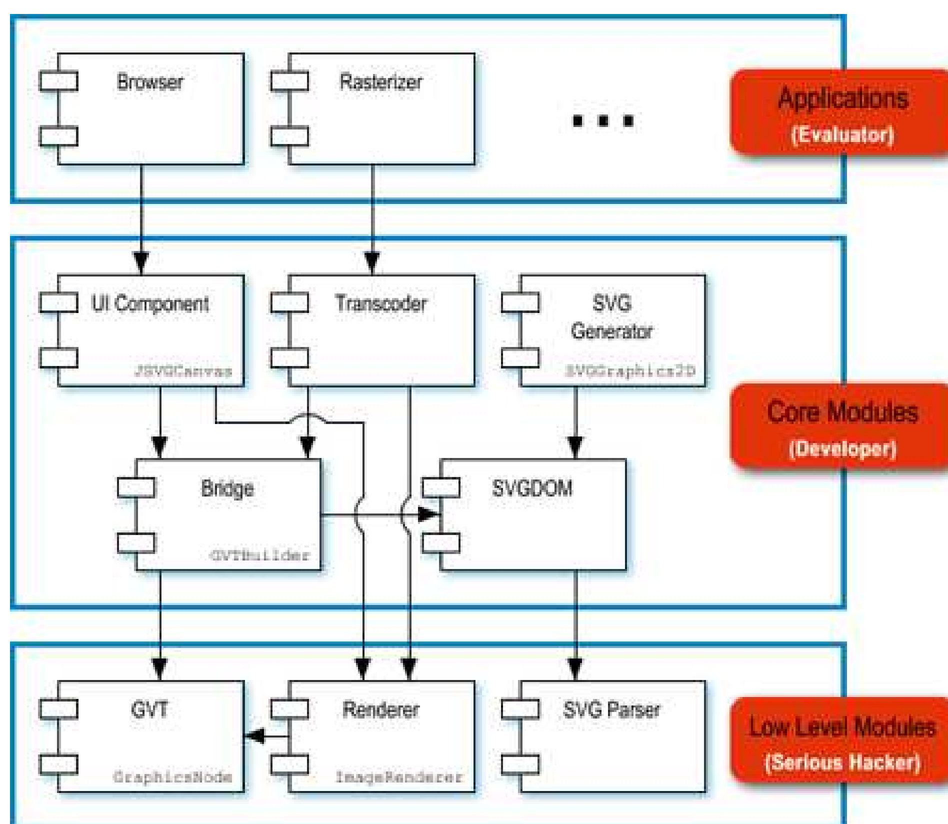
- Generovanie vlastností : Uzol FONode môže obsahovať vlastnosti (Properties). Vlastnosť je trieda zabezpečujúca objektový prístup k atribútom XML elementu. Každá vlastnosť v sebe obsahuje statickú podtriedu `PropertyMaker`, ktorá zabezpečuje transformovanie hodnôt atribútu (alebo atribútov) a ich uloženie do vlastnosti. Vo FOPe existuje mapovanie medzi identifikátorom vlastnosti a inštanciou triedy `PropertyMaker`, ktorý ju vytvorí a tiež mapovanie medzi identifikátorom triedy PropertyMaker, ktorý je totožný s menom atribútu a identifikátorom vlastnosti, ktorú vytvára. Vlastnosti môžu mať aj podvlastnosti, ktoré zodpovedajú podatribútom (napríklad space-before.minimum). Medzi menami podatribútov a identifikátormi príslušných podvlastností je ďalšie mapovanie.

- LayoutManagerMapping : `LayoutManagerMapping` je trieda, ktorá zabezpečuje mapovanie medzi uzlami stromu FOTree a triedami typu `LayoutManagerMaker`. `LayoutManagerMaker` vytvorí rozmiestňovacieho manažéra. `LayoutManagerMapping` je súčasťou triedy `AreaTreeHandler`. Takýto dizajn je výhodný lebo je ľahké nájsť miesto v kóde, kde treba urobiť zmenu, ak chceme aby boli oblasti rozmiestnené iným spôsobom. Stačí nahradiť `LayoutManagerMaker` takým, ktorý vytvorí rozmiestňovacieho manažéra, ktorý toto rozmiestnenie zabezpečí.

- AbstractBraker a AbstractBrakingAlgorithm : Použitím takejto separácie istých algoritmov v rámci rozmiestňovania oblastí je uľahčená ich modifikácia alebo nahradenie. `AbstractBreakingAlgortihm` je trieda implementujúca Knuthov zalamovací algoritmus [5].

1.2 Batik

Batik je sada nástrojov vyvinutá spoločnosťou Apache. Tieto nástroje boli vyvinuté pre aplikácie a aplety napísané pomocou technológie Java, ktoré pracujú s obrázkami vo formáte SVG (Scalable Vector Graphics). Nasledujúci obrázok zobrazuje architektúru nástroja Batik.



Obrázok 5: Batik High-Level Architecture [1]

Pre návrh nášho frameworku sa môžme inšpirovať návrhom modulov GVT, Renderer, SVGDOM a Bridge. Hlavné koncepty a procesy sú popísané v nasledujúcej podkapitole.

1.2.1 Spracovanie SVG od súboru k obrazovke

V tejto časti sa budeme zaoberať spracovaním a postupným transformovaním SVG dokumentu od XML súboru až ku grafickým elementom, ktoré sú zobrazené na obrazovke.

Spracovanie SVG dokumentu v nástroji Batik prebieha v nasledujúcich fázach :

- Parsovanie
  Batik používa na načítanie SVG dokumentu triedu `SAXSVGDocumentFactory`, ktorá je rozšírením org.xml.sax.helpers.DefaultHandler. `SAXSVGDocumentFactory používa` "mikro parsery"na parsovanie komplexných SVG atribútov, napríklad farba alebo

font. Výsledkom tejto fázy je SVG dokument transformovaný do modelu `SVGOMDocument`, čo je implementácia org.w3c.dom.SVGDocument.

- Transformovanie SVGDocument do GVTTree
  GVTTree je strom vytvorený z objektov triedy `GVTNode`, ktoré patria do modulu GVT (Graphics Vector Toolkit). GVTNode v sebe obsahuje informácie o rozmiestnení a ak predstavuje zobraziteľný element, tak obsahuje aj grafický objekt knižnice AWT.

  K väčšine elementov z modelu `SVGDocument` existuje trieda spĺňajúca rozhranie `Bridge` (neskôr popíšeme ako sa k elementom viaže). Táto trieda je mostom medzi SVG elementom a uzlom GVTNode v strome GVTTree. Tento most je jednosmerný a slúži na vytvorenie GVTNode. Ak most implementuje navyše rozhranie `BridgeUpdateHandler`, tak slúži aj na aktualizáciu uzla GVTNode.

  V procese transformácie modelu SVGDocument na GVTTree sa používa trieda `BridgeContext`, ktorá zaobaľuje rôzne väzby medzi elementami modelu SVGDocument a uzlami stromu GVTTree. `BridgeContext` obsahuje informácie potrebné pre rozmiestňovanie, napríklad aktuálnu oblasť, do ktorej sa kreslí, ďalej mapu, kde kľúčmi sú elementy a hodnoty sú jednotlivé oblasti kde sú tieto elementy kreslené, ďalej obsahuje mapu, ktorá mapuje URI [19] menného priestoru na mapu, kde kľúčmi sú lokálne mená elementov v danom mennom priestore a hodnoty sú inštancie mostov, ďalej mapovanie elementov na GVTNode a mapovanie GVTNode na element. `BridgeContext` obsahuje triedu `UpdateManager`, ktorá sa stará o zmeny, ktoré nastali v SVGDocument. `UpdateManager` používa triedu `UpdateTracker`, ktorá na základe zmeny v GVTNode zistí oblasti, ktoré treba aktualizovať a triedu `RepaintManager`, ktorá dá rendereru vedieť, čo sa má prekresliť.

  Transformovanie SVGDocument na GVTTree zabezpečuje trieda `GVTTreeBuilder`, ktorá vytvorí a inicializuje triedu `GVTBuilder`, ktorá rekurzívne prechádza stromom SVGDocument. `GVTTreeBuilder` zachytáva rôzne stavy počas vytvárania GVTTree. `GVTBuilder` ku každému elementu nájde most, pomocou ktorého vytvorí GVTNode, potom sa rekurzívne spracujú deti elementu a ku každému sa nájde most a pomocou neho GVTNode. Pri vynáraní sa z rekurzie sa pre každý element pomocou už nájdeného mostu dokončí tvorba uzla GVTNode. Takto je umožnené spracovanie GVTNode po tom, čo už boli vytvorené a spracované detské uzly. Uzly stromu GVTTree obsahujú informácie o rozmiestnení po tom, čo je GVTTree celý vybudovaný. Ak uzlol stromu GVTTree implementuje rozhranie `GraphicsNode`, tak obsahuje aj vytvorený grafický objekt knižnice AWT.

- Vykresľovanie
  O proces vykresľovania sa stará trieda `StaticRenderer`, ktorá je vytvorená pomocou triedy `ImageRendererFactory`. `StaticRenderer` prehľadáva strom GVTTree do

[19] **URI** (Uniform Resource Identifier) - Reťazec slúžiaci na identifikovanie alebo pomenovanie zdroja. Jeho súčasťou sú URL (Uniform Resource Locator) a URN (Uniform Resource Name)

hĺbky a ak narazí na uzol GVTNode, ktorý implementuje rozhranie `GraphicsNode`, tak zavolá metódu `paint()` z tohto rozhrania, čím sa pomocou grafickej knižnice AWT vykreslí na obrazovku grafický objekt, ktorý tento uzol obsahuje.

1.2.2 Poučenia pre návrh nášho frameworku

V tejto časti popíšeme niektoré konštrukcie a procesy vyskytujúce sa v nástroji Batik. Časti týchto konštrukcii a procesov sú po úprave a doplnení použiteľné pre náš framework.

Študovaním tohto nástroja sme dospeli k nasledujúcim poučeniam :

- Mikro parsery : Mikro parsery slúžia na zjednodušenie čítania zložitých atribútov, akými sú napríklad zoznamy čísiel alebo zoznamy bodov. Je vytvorená hierarchia parserov. Uvedieme si príklad časti tejto hierarchie : `AbstractParser` (stará sa o lokalizáciu a spracovanie chýb), od neho dedí `NumberParser` (stará sa o parsovanie reálneho čísla), od neho dedí `PointsParser`, ktorý používa triedu `PointsHandler`, ktorá zachytáva udalosti ako začiatok budovania bodov, nový bod a koniec budovania bodov. Táto hierarchia parserov je dobre navrhnutá, ľahko rozšíriteľná a je pomocou nej ľahké previesť aj komplikované XML atribúty, ktoré sú v textovej forme, do objektovej formy. Rozhodli sme sa použiť celý tento balík v našom frameworku.
- Bridge : Myšlienka jednosmerných mostov a `BridgeContext`, je ťažkopádna pre dokumenty, ktoré potrebujú komplikovanejšie rozmiestňovacie mechanizmy. Ideu mostov môžme pri návrhu nášho frameworku využiť a upraviť tak, že budú poskytovať funkcionalitu v oboch smeroch, teda od XML k obrazovke a naopak, napríklad vytváranie a aktualizácia pohľadu na XML dokument a vykonanie zmeny modelu XML dokumentu na základe zmeny v pohľade na dokument. Úlohou mostov pri WYSIWYG editovaní by mohlo byť aj zahadzovanie a nahrávanie častí modelu XML dokumentu.

2 XML formáty pre dokumenty

V tejto kapitole sa budeme rozoberať dva konkurenčné XML formáty pre kancelárske dokumenty (OpenXML [10] a OpenDocument [15]). Zameriame sa na ich štruktúru a črty, ktoré sú podstatné pre návrh nášho frameworku pre tvorbu rendererov, vhodných pre WYSIWYG editáciu XML dokumentov.

2.1 Open XML

Cieľom projektu je vytvoriť XML formáty pre kancelárske dokumenty - Office Open XML file formats. Za účelom tohto vývoja bola 21. Marca 2006 vytvorená komunita vývojárov Open XML Formats Developer Group. Medzi 40 zakladateľov patria také firmy ako Apple, Document Sciences Corp., Microsoft, Toshiba a iné. O štandardizáciu týchto formátov sa stará spoločnosť Ecma International. V tejto komunite sa vyvíjajú referenčné schémy ako WordprocessingML [20] , SpreadsheetML [21] a PresentationML [22] , ktoré sú použité v kancelárskom balíku Office12 od firmy Microsoft v programoch Word, Excel a Powerpoint. Tieto formáty sú zamerané na zobrazenie (display-oriented), sú navrhnuté tak, aby mali dlhotrvácnu archivačnú schopnosť a schopnosť spolupráce s ostatnými aplikáciami. Nedajú sa však ľahko štrukturovať a potom použiť vlastnú sémantiku, čo je potrebné pre biznis procesy. Tento problém vyriešila komunita pomocou vlastných schém (custom schema) a možnosťou vytvoriť napríklad WordprocessingML dokument zobrazujúci vlastné XML bez použitia XSLT, a to pomocou integrovania vlastného XML dokumentu do `XML store` a použitím "content controls". Bližšie sa tejto problematike budeme venovať v časti 2.1.3, kde si aj vysvetlíme jednotlivé pojmy. Na popis týchto pojmov a pochopenie toho ako vlastné schémy fungujú, je potrebné najskôr popísať základné vlastnosti Open XML dokumentov.

2.1.1 Formát a štruktúra súboru Open XML

Open XML súbor je ZIP balík, ktorý obsahuje viacero XML súborov, ktoré dohromady formujú dokument. Balík môže obsahovať aj binárne súbory, napríklad obrázky. Základný Open XML súbor obsahuje v koreňovom adresári súbor `[ContentTypes.xml]` a väčšinou tri (môže obsahovať aj viac podadresárov) podadresáre: `_rels`, `docProps` a podadresár špecifický pre typ dokumentu (pre wordprocessing súbore je to adresár `word`).

Vysvetlíme si teraz význam niektorých častí tejto štruktúry.

[Content_Types].xml súbor
Tento súbor popisuje obsah ZIP balíku a obsahuje mapovanie pre súborové rozšírenia a preťaženia špecifických URI

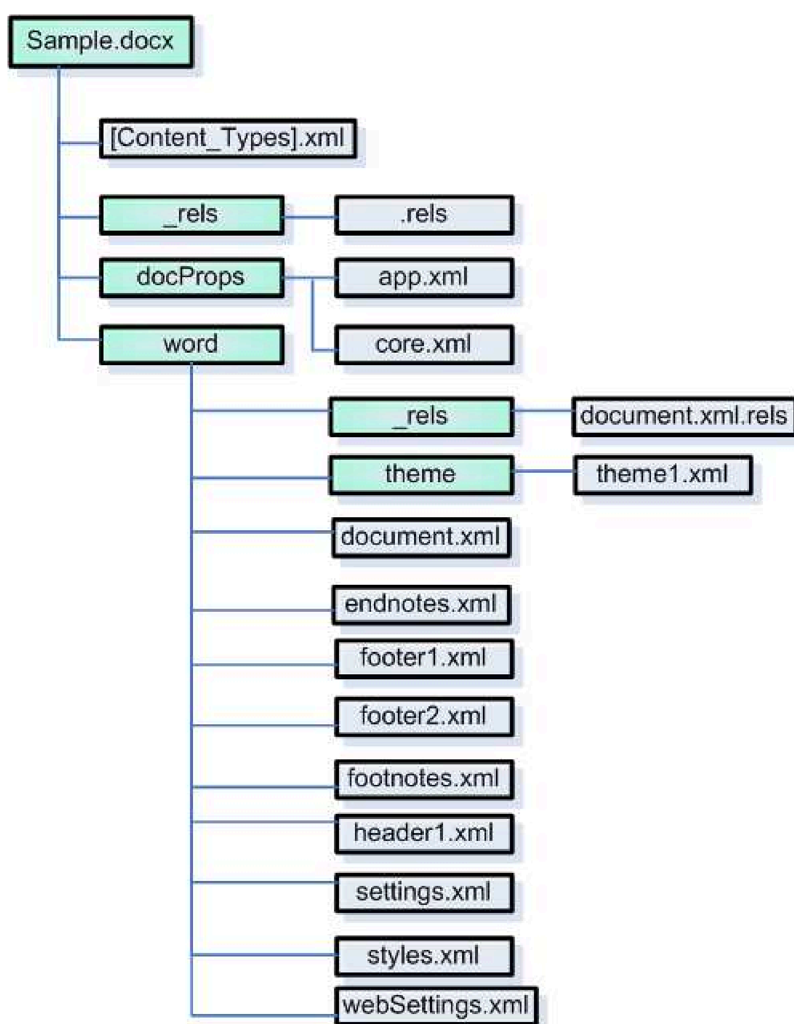
_rels adresár
Do tohto adresára sa ukladajú vzťahy pre akúkoľvek časť v rámci ZIP balíka.

[20]**WordprocessingML** (Wordprocessing Markup Language) - Súborový formát pre ukladanie textových dokumentov, napríklad knihy, články alebo poznámky
[21]**SpreadsheetML** (Spreadsheet Markup Language) - Súborový formát pre ukladanie tabuľkových dát
[22]**PresentationML** (Presentation Markup Language) - Súborový formát pre ukladanie prezentácii

Na nájdenie vzťahu pre špecifickú časť (napríklad document.xml) sa treba pozrieť do adresára _rels, ktorý je súrodencom k danej časti v adresárovom strome. Ak táto časť má vzťah k inej, tak v adresári _rels bude XML súbor s názvom, ktorý sa skladá z originálneho názvu časti a pridaná je súborová prípona .rels (napríklad document.xml.rels). Situáciu vidieť na obrázku 6.

_rels/.rels súbor
V adresári _rels, ktorý je v koreni súborového systému v balíku ZIP sa vždy nachádza súbor .rels, kde sú uložené vzťahy týkajúce sa balíka.

Dobrým príkladom toho ako Open XML formáty vyzerajú sú word processing dokumenty. Pre word sa vyvíja schéma nazývaná WordprocessingML, ktorá plne reprezentuje všetky informácie o rozmiestnení a formátovaní ako XML.



Obrázok 6: Štruktúra WordprocessingML súboru sample.docx

2.1.2 WordprocessingML

V tejto časti sa budeme venovať štruktúre Open XML súboru, ktorá prislúcha WordprocessingML schéme.

Obsah dokumentu sa skladá zo šiestich poddokumentov, ktoré sa nazývajú príbehy (stories). Tieto poddokumenty sú uložené ako samostatné XML súbory v adresári `word` v adresárovej štruktúre Open XML dokumentu. Ide o nasledujúce poddokumenty :

- Hlavný príbeh - hlavné telo dokumentu, je to jediná povinná časť dokumentu
- Poznámky pod čiarou a koncové poznámky(Footnotes a Endnotes)
- Hlavičky a Pätky strany(Headers a Footers)
- Rámce(Frames)
- Poznámky(Comments)
- Poddokumenty - dokument sa môže skladať z viacerých poddokumentov.

Príbehy v dokumente väčšinou zdieľajú tú istú sadu vlastností, napríklad informácie o štýloch, odrážkach a číslovaní, font a nastaveniach dokumentu (napríklad stav zväčšenia (zoom state), nastavenie kompatibility alebo optimalizácia pre web), tieto informácie sú uložené v častiach `styles.xml` a `settings.xml`. Štruktúra WordprocessingML dokumentu môže vyzerať tak, ako je znázornené na obrázku 6.

Pre návrh nášho frameworku je dôležité uvedomiť si, že XML dokument, ktorý chceme editovať sa môže skladať z viacerých XML dokumentov, ktoré sa môžu navzájom referencovať a že pre správne parsovanie takéhoto zloženého XML dokumentu je potrebné čítať niektoré jeho časti skôr ako iné (napríklad súbor `[Content_Types].xml`, ktorý informuje o obsahu celého dokumentu).

2.1.3 Custom schema

V tejto časti sa budeme venovať novej funkcii v kancelárskom balíku Office 12, vlastná schéma (custom schema).

Vlastná schéma je prístup akým sa môžu editovať XML dokumenty. Cieľom tohto prístupu je, aby sa k dobre štrukturovanému XML dokumentu, ktorý obsahuje len informácie o dátach a nie je povedané ako sa majú zobraziť, dali ľahko pripojiť informácie o tom, ako majú byť tieto dáta prezentované a aby sa dali editovať WYSIWYG technikou. Dátový XML dokument spolu s informáciami o prezentácii formuje jeden Open XML dokument. Tento prístup vyžaduje, aby užívateľ definoval dokument pomocou XML Schema syntaxe, ďalej musí do ZIP balíka pridať samotný XML súbor a vytvoriť vzťah z hlavnej časti dokumentu k XML súboru.

Nová vlastnosť Office12 nazývaná XML Data Store [10] zabezpečí, že dátové XML bude vždy súčasťou celého dokumentu (bude uložené ako samostatný súbor). Takto je umožnený prístup k dátam a je možné ich modifikovať. Tieto dáta sú dostupné aj keď

je dokument načítaný. To znamená, že môžu byť kedykoľvek čítané a modifikované. Je vytvorený model udalostí, ktoré zabezpečia, že keď iný proces zmení dáta, bude o tom renderer informovaný.

Oddelenie dát od dokumentu má prínos v tom, že užívateľ nemôže priamo editovať dáta, môže editovať len zobrazovací komponent (content control), ktorý obsahuje biznis logiku, ktorá dovoľuje len validnú editáciu. Ku každému elementu v dátovom XML môže byť pridelený zobrazovací komponent. Neskôr si povieme ako. Je k dispozícii množstvo zobrazovacích komponentov. Medzi základné patria:

1 Plain Text - Tomuto komponentu môže byť nastavené formátovanie, o ktoré sa môže starať dizajnér alebo grafik. Normálny používateľ aplikácie Office 12 bude môcť len meniť text.

2 Drop Down List - Tento komponent slúži na výber možností z povolených hodnôt.

3 Kalendár

4 Combo Box - Je podobný komponentu Drop Down List, ale užívateľ môže navyše zadať vlastnú hodnotu.

5 Rich Text - správa sa ako akýkoľvek text vo aplikácii Word

Každý komponent má svoje nezávislé nastavenia ako napríklad editovateľnosť, či môže alebo nemôže byť vymazaný alebo text, ktorý sa má vypísať na jej miesto ak je prázdny (placeholder text), napríklad "sem napíš dátum".

Najväčšou prednosťou prístupu vlastná schéma (custom schema) je, že zobrazovacie komponenty (content controls) sa dajú namapovať na XML schému, podľa ktorej sa validuje dátový XML dokument. Takto je dosiahnutá separácia dát od prezentácie (data-view separácia).

2.2 Open Document

OpenDocument Format (ODF) [15] je súborový formát pre kancelárske dokumenty, ktoré obsahujú text, tabuľky, diagramy, grafy a grafické elementy. Štandard sa snaží o znovu používanie existujúcich štandardov (SVG, XSL, SMIL, XLink, XForms, Dublin Core, . . .) kdekoľvek je to možné, čo umožňuje ľahšiu transformáciu do iných formátov. O štandardizáciu tohto formátu sa stará neziskové konzorcium OASIS, ktoré má na starosti vývoj, reguláciu a schvaľovanie štandardov pre e-bussines. ODF sa stal OASIS štandardom 1.5.2005 a o rok neskôr 8.5.2006 bol schválený ako ISO a IEC International Standard ISO/IEC 26300. Konzorcium OASIS 25.10.2006 schválilo špecifikáciu OpenDocument v1.1 . Za týmto formátom stoja spoločnosti ako Adobe Systems, IBM, Intel, Novell, Oracle a Sun Microsystems.

V nasledujúcich dvoch kapitolách popíšeme štruktúru ODF súboru a ako môže byť zabezpečená separácia dát od prezentácie.

2.2.1 Formát a štruktúra súboru

OpenDocument súbory sú uložené v archíve JAR [23] . Tento archív tak isto ako OpenXML balík môže obsahovať binárne súbory, napríklad obrázky, všetky ostatné súbory sú XML súbormi.

Funkciu niektorých súborov v balíku si teraz popíšeme.

mimetype
Tento súbor obsahuje jediný riadok hovoriaci o MIME type dokumentu. Napríklad dokument slúžiaci na prezentáciu má súborovú príponu `odt` a MIME typ je `application\vnd.oasis.opendocument.presentation`

content.xml
Súbor predstavuje hlavný obsah dokumentu

styles.xml
Súbor obsahuje informácie o použitých štýloch. Takto sú oddelené informácie o štýle od samotného obsahu, čo umožňuje väčšiu flexibilitu.

meta.xml
Súbor obsahuje meta informácie o obsahu dokumentu, napríklad meno autora, dátum poslednej zmeny a podobne.

META-INF\manifest.xml
Súbor obsahuje zoznam všetkých ostatných súborov v archíve a aj informáciu o tom, akého je daný súbor typu.

settings.xml

V tomto súbore sú uložené informácie špecifické pre aplikáciu, napríklad veľkosť okna, zväčšenie (zoom), kde v dokumente bola posledná zmena a podobne.

2.2.2 XForms

Analógiou vlastnej schémy (custom schema) a zobrazovacích komponentov (content controls), ktoré sa používajú v OpenXML je v črta XForms. Separácia dát od prezentácie je možná aj použitím súborového formátu OpenDocument, ktorého súčasťou je aj XML jazyk XForms, ktorý je štandardizovaný konzorciom W3C. XForms sa skladá z dvoch sekcií, a to XForms model a XForms user interface.

XForms model slúži na popis štruktúry dát, kde sa skrýva celá logika napríklad firemného dokumentu. Na popis tejto štruktúry sa využívajú XML jazyky ako napríklad XPath [24] (na popis výrazov) alebo XML Schema (na popis dátových typov).

[23]**JAR** (Java Archive) - Súborový formát, ktorý je ZIP balíkom, ktorý obsahuje aspoň jeden súbor MANIFEST.MF, ktorý hovorí o tom, ako bude tento archív použitý

[24]**XPath** (XML Path Language) - XML jazyk obsahujúci výrazy pomocou ktorých sa dajú adresovať časti XML dokumentu alebo sa pomocou nich dajú vypočítať hodnoty, ktoré závisia na rôznych častiach dokumentu

XForms user interface slúži na prezentáciu dát. Jednotlivé elementy sa nazývajú aj XForms Controls. XForms Controls sú nezávislé na zariadení (device independent), na ktorom sa používajú a nehovoria ako presne majú byť zobrazené. Presné zobrazenie závisí od internetového prehliadača alebo renderera.

Na zviazanie týchto sekcií sa používa jazyk XPath.

3 Framework

V tejto kapitole sa venujeme návrhu a implementácii nášho frameworku pre tvorbu rendererov, vhodných pre WYSIWYG editáciu XML dokumentov. Framework sme navrhli a implementovali tak, že je vhodný pre veľké aj malé dokumenty, stranovo orientované aj nie stranovo orientované dokumenty. Framework podporuje viacero menných priestorov, pri jeho návrhu sme kládli dôraz na jeho modularitu, rozšíriteľnosť a flexibilitu. Framework nie je naviazaný na konkrétny XML jazyk ani na konkrétnu grafickú knižnicu alebo framework, preto je použiteľný pre tvorbu akéhokoľvek rendera ľubovoľného XML jazyku. Ciele, ktoré náš framework spĺňa sú bližšie rozobraté v časti Cieľ tejto práce a motivácia, ktorá nás viedla k stanoveniu týchto cieľov je popísaná v úvode tejto práce.

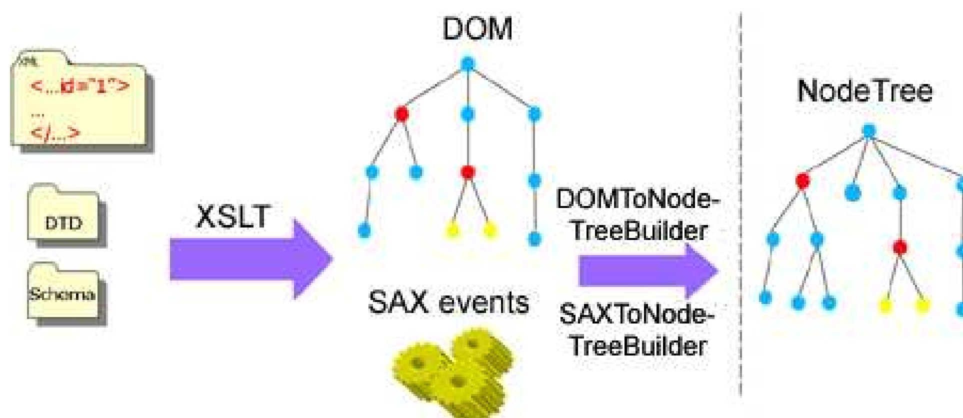
Na dosiahnutie cieľov sme použili sofistikované postupy, techniky a procesy, ktorých prehľad sa nachádza v podkapitolách 3.1 a 3.2. Postupnosť ďalších podkapitol a ich častí nezodpovedá postupnosti krokov, ktoré treba urobiť pri transformácii XML dokumentu na obrazovku lebo na vysvetlenie ďalších krokov je nutné pochopiť zložitosť viacerých aspektov WYSIWYG editovania. Rozhodli sme sa preto usporiadať podkapitoly a ich časti v takom poradí, v akom by mali byť čítané pre správne pochopenie návrhu nášho frameworku. Framework sa skladá z dvoch hlavných modulov, NodeTree a AreaTree (komponentový diagram nášho frameworku sa nachádza v prílohe 3). Podkapitoly 3.3, 3.4, 3.5 a 3.6 sa týkajú modulu NodeTree a podkapitoly 3.7 a 3.8 sa týkajú modulu AreaTree.

3.1 Prehľad procesov vo frameworku

V tejto kapitole je popísaný frameworkom podporovaný proces, ktorým XML dokument prejde, kým je vykreslený na obrazovku. Tiež sú tu popísané techniky na aktualizáciu XML dokumentu na základe zmeny, ktorá pochádza od užívateľa, ktorý edituje pohľad na dokument. Ďalej sa budeme v tejto kapitole zaoberať tým ako na základe aktualizácie XML dokumentu čo najlepšie aktualizovať pohľad na dokument. Kvôli prehľadnosti sú tu zámerne zatajené detaily týchto procesov. Budeme sa nimi zaoberať v jednotlivých podsekciách, popisujúcich časti frameworku. Oboznámime sa so základnými pojmami, ktoré budeme potrebovať v neskorších častiach tejto práce.

Na začiatku máme XML dokument a DTD alebo schému, na základe ktorej budeme XML dokument validovať. Budeme sledovať element s atribútom id="1", ktorý je na obrázku 7 označený červenou farbou. Po validácii môže prebehnúť XSLT transformácia, výsledkom ktorej je transformovaný XML dokument, ktorý môže navyše obsahovať ďalšie informácie, napríklad informácie o svojom vzhľade. Tento dokument je k dispozícii ako DOM alebo sú volané SAX udalosti. Modré uzly predstavujú prvý menný priestor, žlté druhý menný priestor a červené uzly sú v prvom mennom priestore a vznikli zo sledovaného elementu s id="1". V DOM sa vyskytujú dva červené uzly lebo XSLT transformácia mohla sledovaný element zduplikovať, prípadne zduplikovať a zmeniť niektoré atribúty (napríklad z kapitol sa vytvoril obsah, editovaním obsahu sa mení názov kapitol a naopak menením názvu kapitol sa mení obsah).

Ďalším krokom je vytvorenie vnútornej stromovej objektovej reprezentácie XML dokumentu. Túto reprezentáciu XML dokumentu budeme volať **NodeTree**. Za vytvorenie

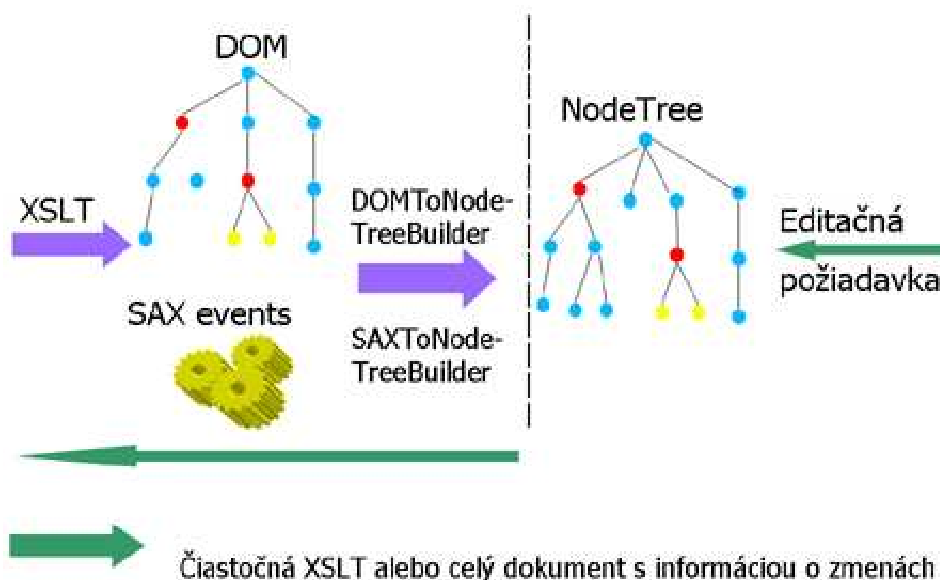


Obrázok 7: Rendering process

stromu NodeTree zodpovedá buď trieda `DOMToNodeTreeBuilder`, ktorá vykoná traverz stromom DOM alebo `SAXToNodeTreeBuilder`, ktorá zachytáva SAX udalosti.

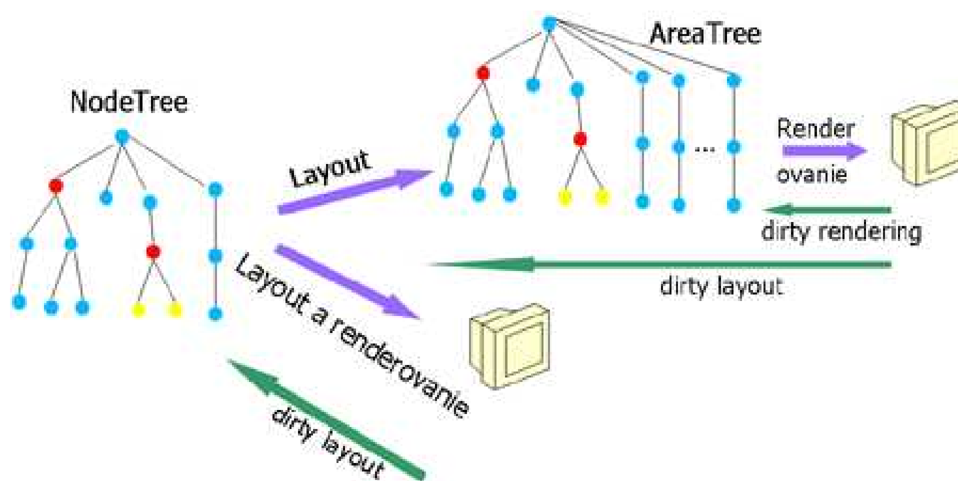
Vertikálna čiara reprezentuje hranicu, kde napravo je to, čo framework zahŕňa a naľavo je to, čo framework predpokladá. Prostredie, ktoré je naľavo od tejto vertikálnej čiary budeme nazývať **Files And Transformations System** (**FATS**).

`SAXToNodeTreeBuilder` aj `DOMToNodeTreeBuilder` poskytujú atribúty XML elementov v primitívnej textovej forme. Táto forma nezachytáva hierarchiu vlastností (vlastnosti sa skladajú z podvlastností), dedenie a komplexnosť zápisu núti k ich parsovaniu. Preto je vhodné transformovať tieto atribúty do omnoho výhodnejšieho **objektového modelu vlastností**.



Obrázok 8: Rendering process 2

NodeTree môže byť transformovaný do ďalších štruktúr a tie sú vykreslené na obrazovku alebo sa priamo z neho robí rozmiestňovanie a vykresľovanie súčasne. Na základe akcie užívateľa môže vzniknúť **editačná požiadavka** (napríklad dopísanie textu). NodeTree musí byť schopný túto požiadavku zachytiť a poslať do FATS. Framework predpokladá, že FATS zistí, či je táto zmena možná napríklad validáciou XML alebo inými (rýchlejšími) technikami nahrádzajúcimi validáciu. V prípade, že je požiadavka na zmenu akceptovaná, FATS poskytne iba tú časť dokumentu, ktorá bola zmenená. Ak FATS túto možnosť nemá, tak poskytne celý zmenený dokument, vo forme DOM alebo SAX udalostí s pridanou informáciou o zmenách pomocou umelých uzlov alebo atribútov. Ak ani túto funkcionalitu FATS nepodporuje, tak poskytne celý dokument akoby bol čítaný prvý krát. Bližšie túto situáciu rozoberieme v časti 3.6. Kvôli šetreniu pamäte sa môžu v ďalších fázach, niektoré podstromy NodeTree zahodiť [25] . Môže nastať situácia, kedy ich budeme potrebovať (napríklad keď budeme chcieť zmeniť uzol v zahodenej časti), preto kladieme požiadavku na FATS, aby vedel na požiadanie nahrať ľubovoľný podstrom NodeTree, ak takúto možnosť nepodporuje, tak poskytne celý NodeTree. **Zahadzovaniu a nahrávaniu podstromov** stromu NodeTree sa budeme venovať v podkapitole 3.5.



Obrázok 9: Rozmiestňovanie a vykresľovanie

Vnútorná reprezentácia pôvodného XML dokumentu NodeTree model dát, na základe ktorého sa vytvárajú, rozmiestňujú a vykresľujú grafické prvky (layout a rendering). **Rozmiestňovanie a vykresľovanie** sa dá robiť naraz alebo oddelene. Pri obidvoch sa NodeTree dá spracovať

- staticky : NodeTree je spracovaný až keď je celý vybudovaný
- dynamicky : NodeTree je spracovávaný počas budovania, napríklad po vytvorení špeciálneho uzla sa spustí spracovávanie jeho časti

[25] Pod pojmom zahodiť máme na mysli odstránenie objektov, ktoré predstavujú uzly v tomto podstrome z pamäte.

AreaTree je ďalšou vnútornou stromovou reprezentáciou XML dokumentu. Strom AreaTree obsahuje všetky informácie o rozmiestnení dokumentu a zároveň všetky grafické informácie, napríklad farba, priesvitnosť, font a podobne. Z tohto stromu sa vytvoria jednotlivé grafické elementy, takzvaný pohľad na dokument. Tvorba týchto grafických elementov môže byť robená staticky, teda po vytvorení AreaTree alebo dynamicky počas jeho vytvárania (napríklad po rozmiestnení strany sa táto vykreslí).

Zelené šípky na obrázku 9 opäť predstavujú editačnú požiadavku, ktorá bola vytvorená na základe akcie užívateľa v pohľade na dokument. Táto požiadavka je poslaná do príslušného uzlu v strome AreaTree a ten ju pošle do príslušného uzlu v strome NodeTree alebo je priamo poslaná priamo do uzla stromu NodeTree a ten ju v oboch prípadoch pošle do FATS. Návrhu **rozhraní na príjem editačných požiadaviek** sa venujeme v častiach 3.6.2, 3.6.3 a 3.8.

Na základe editačnej požiadavky sa XML dokument zmení a ak je táto zmena validná, tak sa aktualizuje strom NodeTree a pri oddelenom rozmiestňovaní a vykresľovaní sa aktualizuje aj strom AreaTree. Aktualizáciou vnútorných reprezentácii XML dokumentu sa budeme zaoberáme v častiach 3.6.8, 3.6.9, 3.7.4 a 3.8

Požiadavka môže byť však aj typu **dirty editing**. Tento pojem označuje editovanie dokumentu bez nutnosti synchronizácie zmeneného XML dokumentu s vnútornými štruktúrami do, ktorých je transformovaný. Táto požiadavka je volaná len v prípade, že systém si je istý, ako bude dokument po danej zmene vyzerať, čo je vo všeobecnosti ťažké lebo XSLT transformácia má príliš veľkú výpočtovú silu. Vo väčšine prípadov sa XSLT transformácia používa na jednoduché úpravy XML dokumentu, a preto je v mnohých prípadoch predvídateľné, ako bude XML dokument po zmene a transformácii vyzerať. Napríklad pri dopísaní písmenka môže byť framework nastavený tak, že odhadne, že sa príslušne zmení text v uzle stromu NodeTree a nie je nutné ho synchronizovať s XML dokumentom. Ak však ide o červený uzol, kde jeho zmena má nepredvídateľný vplyv na to, čo XSLT transformácia vyprodukuje, tak sa nesmie použiť technika dirty editing. Medzi dirty editing, patrí dirty rendering a dirty layout.

Pri **dirty rendering** sa nerobí synchronizácia NodeTree so zmeneným dokumentom a ani sa nesynchronizuje rozmiestnenie so zmeneným stromom NodeTree. Táto technika sa používa ak systém vie, že daná zmena nespôsobí chyby v rozmiestnení grafických prvkov. Napríklad ak ide o zmenu farby alebo ak prvky v AreaTree majú nastavené rozmery minimum, optimum, maximum a danou zmenou sa neprekročia povolené hranice. Táto zmena sa pošle do NodeTree a ten ju prepošle do FATS, ale NodeTree nespúšťa podnet na aktualizáciu rozmiestnenia.

Pri **dirty layout** sa tiež nerobí synchronizácia NodeTree so zmeneným XML dokumentom. Keď NodeTree dostane dirty layout editačnú požiadavku, prepošle ju do FATS ako dirty, zaktualizuje na jej základe svoje uzly a pri tom sa spúšťa proces aktualizácie rozmiestnenia alebo proces aktualizácie rozmiestnenia a vykresľovania. Táto problematika je rozobratá v častiach 3.6.2 a 3.7.5.

V prípade, že sa pamätá iba tá časť AreaTree, ktorá sa napríklad práve edituje (je v nej kurzor alebo focus), tak môže nastať situácia kedy je nutné z NodeTree vytvárať zodpovedajúce časti AreaTree a následne k nim vytvárať grafické entity. V prípade, že

ani časti NodeTree, z ktorých sa majú vytvoriť časti AreaTree, nie sú nahraté v pamäti, tak sa tieto najskôr nahrajú a následne sa vytvoria zodpovedajúce časti AreaTree. Touto problematikou sa zaoberáme v častiach 3.7.6 a 3.7.7.

Doteraz sme nespomenuli, ako sa užívateľova editačná požiadavka, ktorá vznikne na pohľade na dokument prepošle či už do AreaTree pri oddelenom rozmiestňovaní a vykresľovaní alebo do NodeTree pri spojenom rozmiestňovaní a vykresľovaní. Cieľom tejto práce nie je vytvoriť prostredie na zachytávanie užívateľských akcii v pohľade na dokument pomocou vstupných zariadení, ako sú myš a klávesnica, na to slúži grafická knižnica alebo prostredie. Takéto prostredia už existujú, jedným z nich je napríklad GEF [26] . Napájaniu nášho frameworku na takéto prostredia sa venujeme v podkapitole 3.8.

3.2 Architektúra frameworku a základné pojmy

V tejto časti sa oboznámime s pojmami, ktoré budeme potrebovať v ďalších častiach tejto práce, architektúrov frameworku a výhodami, ktoré náš návrh prináša.

Framework slúži na vytvorenie renderera, vhodného pre WYSIWYG editáciu XML dokumentov a predpokladá, že tento renderer bude použitý ako súčasť nejakého XML editora, ktorý sa stará o aplikačnú logiku a prácu s XML súborom. Medzi našim frameworkom a XML editorom je rozhranie `IFATS`, ktoré hovorí o požiadavkách, ktoré kladie náš framework na FATS, ktorý je súčasťou aplikačnej logiky XML editora. Rozhranie medzi našim frameworkom a grafickou knižnicou alebo frameworkom je popísané v časti 3.8.

Framework kladie veľký dôraz na rozšíriteľnosť, flexibilitu, modularitu a intuitívnu implementáciu, o čom svedčí komponentový UML diagram, ktorý sa nachádza v prílohe 3.

`UserAgent` je centrálne miesto, kde sa registrujú všetky triedy, ktoré rozširujú funkčnosť frameworku a preťažujú jeho štandardné správanie. Sú to triedy `EventHandlerRegistry`, `ElementToNodeMappingsRegistry`, `BridgeMappingRegistry`, `AttributeHandlerRegistry`, ...

Nasledujúci príklad ukazuje akým spôsobom je náš framework rozšíriteľný a flexibilný. Trieda `EventHandler` definuje správanie sa frameworku v špeciálnych prípadoch počas parsovania XML dokumentu. Napríklad začiatok a koniec parsovania dokumentu, začiatok alebo koniec spracovania nejakého špeciálneho elementu. `EventHandler` je špecifický pre menný priestor a v triede `EventHandlerRegistry` je spravované mapovanie medzi menným priestorom a `EventHandlerom`. `EventHandlerRegistry` spolupracuje s FATS a jeho `SAXToNodeTreeBuilder` alebo `DOMToNodeTreeBuilder` a tiež s uzlami stromu NodeTree. V týchto špeciálnych prípadoch sa štartujú ďalšie fázy spracovania XML dokumentu alebo zahadzovanie podstromov `NodeTree`. Napríklad v prípade renderovania jazyka XSL-FO, odvodený `FOEventHandler` môže zachytávať ukončenie parsovania strany v metóde `endPage()` a v nej spustí rozmiestňovací proces a po jeho skončení zahodí príslušnú časť stromu `NodeTree`. Situáciu objasňuje activity diagram, ktorý sa nachádza v prílohe 1.

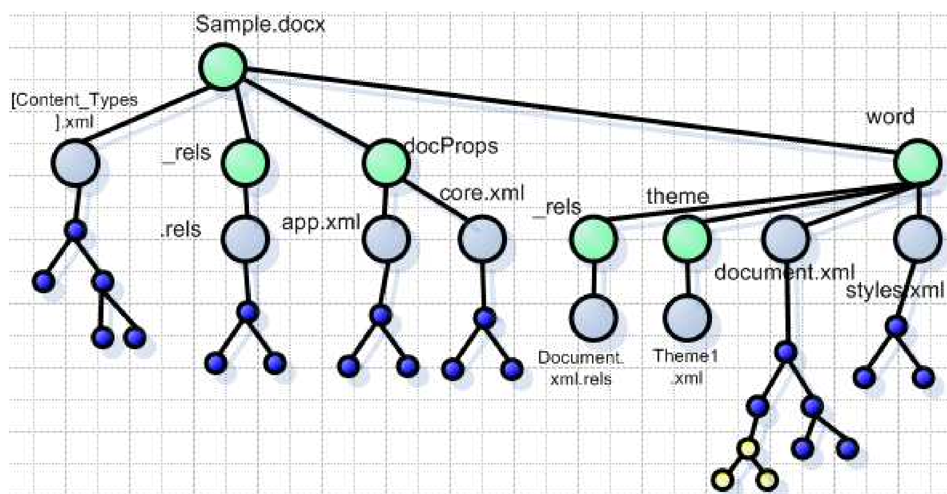
[26] **GEF** (Graphical Editing Framework) - Prostredie na tvorbu grafických editorov [7]

3.3 NodeTree

V tejto podkapitole popíšeme ako sme navrhli základné rozhrania, pomocné triedy a metódy modulu Node potrebné pri vytváraní uzla. Situáciu objasňuje UML diagram tried, ktorý sa nachádza v prílohe 2.

Strom NodeTree pozostáva z uzlov, ktoré implementujú rozhranie `INode`. Toto rozhranie musí byť dostatočne všeobecné, aby pokrylo všetky špecifiká jazyka, ktorý ideme vykresľovať a zároveň komplexné, aby čo najviac operácií nad týmto stromom vykonával framework.

Čoraz častejšie sa XML využíva na prezentovanie a editovanie firemných dát, ktoré sú vo formáte XML. Môžu to byť napríklad XML súbory vygenerované z databázy. Tieto dáta neobsahujú žiadne prezentačné informácie. FATS tieto informácie môže pridať napríklad pomocou XSLT transformácie alebo technikou vlastná schéma (custom schema)(2.1.3). Najjednoduchší prípad nastane, keď FATS poskytuje XML dokument predstavujúci jeden súbor [27] , ktorý na jednom mieste obsahuje všetky informácie o tom, ako má byť dokument zobrazený (napríklad XSL-FO). Môže však nastať situácia, keď informácia o tom, ako má byť dokument zobrazený je roztrúsená a NodeTree je komplikovanejší (napríklad WordprocessingML (2.1.2)) V predchádzajúcom príklade si môžeme všimnúť, že mnohé



Obrázok 10: NodeTree pre WordprocessingML dokument

uzly stromu nie je možné alebo vhodné editovať WYSIWYG technikou, a teda tieto uzly sú v návrhu odlíšené cez metódu `isEditable()`, dokonca niektoré uzly sú v jazyku z iných ako zobrazovacích dôvodov (napríklad informácie o právach na dokument alebo o mieste kde bol naposledy editovaný, rôzne väzby medzi časťami dokumentu) alebo nie sú priamo premietnuté do grafických prvkov a slúžia ako pomocné informácie pre renderer (napríklad štýly), tieto sú v návrhu odlíšené metódou `isVisible()`. Tento poznatok vyústil do návrhu dvoch základných rozhraní pre uzly stromu NodeTree, a to `INode` a `IEditableNode`

[27] V skutočnosti nejde o súbor, transformácia XML dokumentu prebehne z DOM/SAX do DOM/SAX. Transformovaný dokument je teda držaný v operačnej pamäti ako DOM alebo je poskytnutý ako SAX udalosti a teda nie je uložený ako súbor na pevnom disku

a ich predvolených implementácií `XMLNode` a `EditableXMLNode`. Kde rozhranie `INode` sa stará o prepojenie na zvyšok frameworku, navigáciu v strome, pridávanie uzlov do stromu, identifikácia uzla menom, menným priestorom a miestom v XML dokumente, spracovanie atribútov, vyvolanie ďalšej fázy spracovanie XML dokumentu (rozmiestňovanie a vykresľovanie), zahadzovanie vetiev a ďalšie funkcie popísané v ďalších kapitolách. Rozhranie `IEditableNode` sa stará o spracovanie požiadavky na zmenu uzla, jej preposlanie do FATS a tiež na príjem požiadavky na aktualizáciu NodeTree z FATS. Detailnejšie ho popíšeme v podkapitole 3.6.

Výhodou nášho riešenia je, že používateľ frameworku nemusí nič implementovať a framework poskytne základnú funkčnosť. Framework sám zabezpečí transformácia XML elementov na uzly stromu NodeTree. Framework poskytuje základnú implementáciu uzla `INode` a všeobecný `NodeMaker`, ktorý ho vytvára. Tento `NodeMaker` je zaregistrovaný vo všeobecnej triede `ElementToNodeMapping` a tá je zaregistrovaná v `ElementToNodeMappingRegistry`. Implementátor frameworku má k dispozícii strom NodeTree, ktorý však nepodporuje čiastočnú aktualizáciu ani zo strany FATS ani zo zvyšku frameworku, lebo editačná požiadavka vyústi do požiadavky na poskytnutie celého zmeneného dokumentu. Taktiež nepodporuje zahadzovanie a nahrávanie vetiev. Ďalšia fáza spracovania XML sa spúšťa až po tom, čo je celý strom NodeTree vybudovaný. Toto môže byť vhodné pre jednoduché XML jazyky, kde elementov nie je veľa, majú malú funkčnosť a málo atribútov a nie sú medzi nimi komplikované vzťahy. Vtedy môže byť komplexnosť štruktúry NodeTree zbytočná a postačí jej všeobecná implementácia.

3.4 Atribúty a Vlastnosti

V tejto kapitole popíšeme náš návrh tvorby vlastností z atribútov a naopak tvorby atribútov z vlastností, ďalej popíšeme to, ako sme využili techniku tvorby vlastností a atribútov cez triedy typu Builder, ako sme na čítanie komplikovaných atribútov využívajú mikro parsre, vysvetlíme rôzne mapovania, ako sme vyriešili vnáranie rendererov. Táto funkčnosť je zahrnutá v module Property. V tejto podkapitole je vysvetlené, ako sme zapojili modul Property do zvyšku frameworku a popíšeme možnosti a miesta kde sa dá tento modul rozšíriť. Spôsob zapojenia modulu Property je znázornený v komponentovom UML diagrame, ktorý sa nachádza v prílohe 3.

3.4.1 Prínos konverzie atribútov na vlastnosti

Dôvodov prečo konvertovať atribúty na vlastnosti je viacero. Vo všeobecnosti ide o zjednodušenie ich zložitosti. Konkrétne ide o :

1. Vyjadrenie štruktúry pomocou objektov
   Atribúty môžu vo všeobecnosti predstavovať komplexnú štruktúru v zmysle, že sa môžu skladať z množstva podatribútov a tie môžu byť tiež komplexné. Napríklad atribút Font môže byť v elemente definovaný takto: `Font="size-min:10 , size:max:30 , size:opt:  12, color:RGB(0,0,0) , decoration:italic "`. Také-

muto komplexnému atribútu zodpovedá vlastnosť `FontProperty`, ktorá má podvlastnosti `SizeProperty`, `ColorProperty` a `DecorationProperty`. `SizePoperty` má 3 vlastnosti typu `NumberProperty` predstavujúce minimálnu, maximálnu a optimálnu hodnotu veľkosti.

2. Zjednotenie rôznych zápisov
   Zjednotenie zápisov nám veľmi uľahčuje dedenie vlastností, ich menenie počas editácie a uloženie späť do XML dokumentu. Vlastnosť Font z predchádzajúceho bodu sa dá zapísať aj takto: `Font.size="10,30"`, `Font.size.optimum="12"`, `Font.color = "black"` , `Font.decoration = ïtalic"` a mnohé ďalšie kombinácie, ale vlastnosť `FontProperty` bude vždy tá istá.

3. Dedenie
   Vnorený element môže dediť hodnoty niektorých atribútov od svojich rodičov a dokonca nielen od svojich rodičov, ale aj od svojich predkov. Namiesto hľadania tejto hodnoty pomocou mien atribútov, ktoré môžu mať viacero komplikovaných zápisov sa táto hodnota hľadá pomocou identifikátora vlastnosti. Predstavme si aká komplikovaná by mohla byť situácia, keď by sme chceli zdediť atribút font ale zo zmenenou hodnotou `Font.size.optimum`. Museli by sme prehľadať všetkých predkov a v nich všetky atribúty a hľadať v nich jednotlivé podatribúty atribútu Font a tiež podatribúty podatribútov a tak ďalej. Pri objektovej reprezentácii atribútov stačí vyhľadať najbližšieho predka, ktorý môže mať definovaný atribút font a v jeho vlastnosti FontProperty sú už uložené všetky informácie.

4. Pridanie funkcionality
   Keďže vlastnosti sú objekty, tak sa k nim dá pripojiť aj informácia napríklad o tom, ako daná vlastnosť vznikla, napríklad dedením alebo je explicitne vyjadrená pomocou atribútu elementu alebo či atribút z ktorého vlastnosť vznikla nevznikol XSLT transformáciou (čo môže mať vplyv na typ editácie tejto vlastnosti). Hodnota atribútu môže byť výraz, ktorý treba vyrátať. Vlastnosť môže mať kontrolnú funkciu, týkajúcu sa rozsahu zadávaných hodnôt.

3.4.2 Tvorba vlastností z atribútov

Odhliadnuc od technickej realizácie, vlastnosti sa vytvárajú tak, že pri vytváraní uzlov stromu NodeTree, ktoré sa vytvárajú z elementov XML dokumentu pomocou prehľadávania do hĺbky, sa konvertujú atribúty na vlastnosti. Pri vytváraní uzla stromu NodeTree sa zoberú všetky atribúty, ktoré v ňom môžu byť definované a vytvoria sa k nim vlastnosti s predvolenou (default) hodnotou, potom sa tieto hodnoty aktualizujú dedením na základe hodnôt predkov a nakoniec sa zoberú tie atribúty, ktoré sú v tomto elemente explicitne určené a na ich základe sa vlastnostiam pridelia tieto hodnoty. Keďže hodnoty môžu byť veľmi komplikované a ich parsovanie netriviálne, bola na mieste potreba vytvoriť podporu pre parsre. Inšpiráciou boli micro parsre nástroja Batik [1], ale nakoniec sa ukázalo, že

ich návrh na tieto účely vyhovuje. Použili sme preto ich balík org.apache.batik.parser nezmenený. UML diagram tried tohto balíka sa nachádza v prílohe 7.

Framework predpokladá, že neexistujú elementy "brzdiace dedičnosť". Ide o takúto situáciu. Majme element1, v ňom je vnorený element2 a v ňom je vnorený element3. V elemente1 je explicitne uvedená hodnota atributu1, element2 nemá atribút1 a element3 má atribút1, ale jeho hodnota nie je explicitne určená, a teda sa dedí a mala by byť taká, aká je v elemente1. Ak je element2 "brzdiaci", tak sa nič nezdedí a hodnota je predvolená. Situáciu vyjadruje nasledovný obrázok, kde ako atribút uvažujeme farbu pozadia.

Obrázok 11: Brzdiaci element

3.4.3 Tvorba atribútov z vlastností

Konverzia vlastností späť na atribúty elementu je algoritmicky jednoduchá. Keďže uzol stromu NodeTree má všetky vlastnosti a ich hodnoty k dispozícii a navyše vlastnosti sú rozdelené na tie, ktoré sú zdedené a tie, ktoré sú explicitne vyjadrené v tomto uzle. Pri konverzii vlastností na atribúty stačí zobrať len tie vlastnosti, ktoré sú explicitne vyjadrené a skonvertovať ich.

3.4.4 Aktualizácia vlastností

Ak vlastnosti vytvárame doteraz popísaným spôsobom, tak pri zmene hodnoty vlastnosti v uzle U alebo jej zaradením medzi zdedené, je nutné aktualizovať všetky vlastnosti, ktoré od nej dedia v podstrome, ktorý je zakorenený v uzle U, okrem uzlov, ktoré sú zahodené [28] . Tomuto procesu hovoríme aktualizácia vlastností.

K zmene vlastností dochádza v dvoch prípadoch. Prvým je, keď na základe užívateľovho stimulu uzol stromu NodeTree príjme požiadavku na zmenu vlastností a táto

[28]Zahadzovaniu a nahrávaniu uzlov stromu NodeTree za účelom šetrenia pamäte sa venujeme v časti 3.5

požiadavka je typu `dirty`, vtedy sa táto požiadavka prepošle do FATS, ale nerobí sa synchronizácia XML dokumentu s NodeTree.

Druhým prípadom je, keď sa XML dokument zmenil. Túto zmenu zachytí NodeTree. Ak sa zmena týka zmeny textu, pridania stromu elementov, vymazanie stromu elementov alebo nahradenie stromu elementov iným, tak netreba vlastnosti aktualizovať. Ak sa však zmena týka zmeny atribútov, tak je nutné vlastnosti aktualizovať.

Po nahratí zahodených podstromov sa vlastnosti v tomto podstrome nemusia aktualizovať. Keďže pri nahrávaní sa vytvárajú vlastnosti nanovo, a teda nevadí, že sa počas toho, keď bol podstrom zahodený vlastnosti predkov zmenili.

3.4.5 Popis implementácie modulu Property

V predchádzajúcich častiach sme popísali ako má modul Property fungovať a teraz popíšeme návrh jeho implementácie.

Prvou inšpiráciou bola koncepcia projektu Apache FOP, kde sa atribúty konvertovali na vlastnosti. Každá vlastnosť bola identifikovaná svojim menom a svojim identifikačným číslom. Toto meno bolo identické s menom atribútu alebo podatribútu. Bolo vytvorené mapovanie medzi menami vlastností a triedami, ktoré ich vytvárajú (`PropertyMaker`). Táto koncepcia nie je vhodná pre vlastnosti, ktoré sa môžu meniť a nie je dostatočne všeobecná pre rôzne XML jazyky. Dedenie vlastností v nástroji FOP nie je vyriešené spôsobom vhodným pre menenie týchto vlastností, každý uzol FOTree (1.1.1) obsahuje vlastnosti, ktoré v sú v ňom explicitne určené a zoznam odkazov na zdedené vlastnosti. Uvedieme príklad prečo je takýto koncept nevhodný. Majme uzol `x` a ten nech má potomka `y` a ten nech má potomka `z`, nech `y` aj `z` dedia vlastnosť `v` od uzla `x`. Ak sa vlastnosti `v` v uzle `y` explicitne priradí hodnota, tak by uzol `z` mal dediť túto vlastnosť od uzla `y`, takýto mechanizmus ale v nástroji FOP nie je.

Tieto a rôzne ďalšie nedostatky viedli k tomu, aby sme vytvorili vlastný návrh, kde správanie sa pri spracovaní atribútov je oddelené do špeciálnych tried typu `AttributeHandler`. Uzol stromu NodeTree obsahuje všetky vlastnosti, ktoré v ňom môžu byť definované a uchováva ich pomocou triedy `PropertyList`, ktorá ich rozdeľuje na zdedené a explicitne vyjadrené, poskytuje funkčnosť potrebnú pri ich vytváraní, zabezpečuje ich menenie a pridávanie a presúvanie medzi zdedenými a explicitnými. Zložená vlastnosť (`CompoundProperty`) udržuje a spravuje svoje podvlastnosti v rovnakej štruktúre, čo navyše umožňuje skladanie do ľubovoľného množstva úrovní. V našom návrhu nie je prítomné mapovanie z atribútov na vlastnosti ako to bolo v nástroji FOP, ale mapujú sa atribúty na triedy typu `Bridge`, ktoré sú mostom medzi atribútmi a vlastnosťami. Most v sebe zahŕňa triedu na tvorbu vlastností (`PropertyMaker`) a na konverziu vlastností na atribúty (`PropertyChanger`).

Návrh podporuje použitie viacerých menných priestorov tak, že zhŕňa mapovanie medzi kompatibilnými vlastnosťami z rôznych menných priestorov, čo je využiteľné pri vnáraní rôznych XML jazykov, ktoré používajú niektoré atribúty s rovnakým významom, napríklad font.

Framework obsahuje abstraktnú triedu `AttributeHandler`, ktorá je určená pre jeden menný priestor a ktorá sa stará o správne poradie spracovania atribútov v elemente, ktorý je

v tomto mennom priestore a tiež za správne identifikovanie skratiek a rôznych zápisov mien atribútov a rozdelenie atribútov na podatribúty. Rozšírenia triedy `AttributeHandler` sa registrujú na základe menného priestoru do singeltonu [29] `AttributeHandlerRegistry`.

V `AttributeHandlerRegistry` je zaregistrované jedno špeciálne rozšírenie `UniversalAttributeHandler` triedy `AttributeHandler`, ktoré slúži na spracovanie atribútov elementu z neznámeho menného priestoru. Framework podporuje jeho nahradenie iným.

Framework obsahuje abstraktná trieda `BridgeMapping`, ktorá mapuje mená atribútov z jedného menného priestoru na mosty implementujúce rozhranie `IBridge`. V návrhu je zahrnutá aj abstraktná implementácia mosta `Bridge`, ktorá zahŕňa väčšinu potrebnej funkčnosti, pričom využíva funkčnosť tried implementujúcich rozhranie `IPropertyMaker` a `IPropertyChanger`, ktoré sa starajú o vytvorenie vlastnosti z atribútov a zmenu vlastnosti na atribút(alebo atribúty). Rozšírenia triedy `BridgeMapping` sa registrujú v singeltone `BridgeMappingRegistry`, ktorý obsahuje jedno špeciálne rozšírenie `UniversalBridgeMapping` triedy `BridgeMapping`, ktoré mapuje nerozpoznané menné priestory atribútov na rozšírenie `BasicBridge` mosta `Bridge`, ktorý využíva triedy `BasicPropertyMaker` a `BasicPropertyChanger`. Tieto 4 triedy sú navrhnuté na vzájomnú spoluprácu a na spoluprácu základnej implementácie `Property` rozhrania `IProperty`, ktoré je rozhraním pre všetky vlastnosti vo frameworku. Mapovanie medzi kompatibilnými vlastnosťami z rôznych menných priestorov je spravované triedou `PropertyCompatibility`. Pri čítaní komplikovaných hodnôt atribútov pomôže pri implementácii rozhrania `IPropertyMaker` systém mikro parserov z balíka *org.apache.batik.parser* [1].

3.4.6 Použitie modulu Property

Tak ako v module Node ani v module Property nemusí používateľ frameworku urobiť nič a framework poskytne základnú funkčnosť, kde vlastnosti sú iba objektovým zápisom atribútov, bez skladania vlastností z podvlastností, bez konverzie zápisu atribútu na jednotný tvar, bez dedičnosti, vlastnosti sa však dajú meniť a ich zmena sa korektne prevedie do formy SAX alebo DOM atribútov.

Na plnohodnotnú prácu s vlastnosťami musí implementátor vytvoriť sadu vlastností, k nim sadu tried, ktoré ich vytvárajú (`IPropertyMaker`), sadu tried, ktoré ich menia na atribúty (`IPropertyChanger`), sadu mostov (`IBridge`), prepájajúcich atribúty a vlastnosti, vytvoriť mapovanie medzi atribútmi a mostmi a zaregistrovať ho do triedy `BridgeMappingRegistry`, vytvoriť triedu spracovávajúcu SAX alebo DOM atribúty (`AttributeHandler`) a zaregistrovať ju do triedy `AttributeHandlerRegistry`. Používateľovi frameworku pomôžu pri jeho implementovaní UML diagrami balíka Property, ktoré sa nachádzajú v prílohách 4, 5 a 6.

[29] Singelton je návrhový vzor, ktorý zabezpečuje, že je vytvorená najviac jedna inštancia triedy s globálnym prístupovým bodom

3.5 Šetrenie pamäte v module NodeTree

V tejto podkapitole popíšeme ako sme navrhli modul NodeTree nášho frameworku tak, aby bolo možné znížiť pamäťové nároky renderera, ktorý je pomocou tohto frameworku vytvorený.

Pre lepšie pochopenie tejto podkapitoly nám pomôže UML diagram, ktorý sa nachádza v prílohe 8.

3.5.1 Zahadzovanie

Možnosť znížiť pamäťové nároky rendereru sme dosiahli tak, že náš framework poskytuje podporu pre zahadzovanie podstromov internej štruktúry NodeTree. Zahodenie podstromu NodeTree znamená aplikovanie operácie zahodenia na deti koreňa tohto podstromu. Operácia zahodenia uzla zabezpečí, že sa z pamäte [30] odstránia deti tohto uzla, zoznam vlastností, prípadne ďalšie objekty s ním zviazané. Táto operácia sa vykonáva tak, že sa najskôr rekurzívne aplikuje na všetky deti a až potom na samotný uzol. Inštancia koreňa tejto vetvy sa nezahodí a len sa označí ako zahodená a odstránia sa z pamäte jeho deti, zoznam vlastností a ďalšie objekty s ním zviazané, okrem tých, ktoré ho identifikujú (lokácia v XML, meno elementu, ku ktorému patrí, jednoznačný identifikátor a podobne).

3.5.2 Nahrávanie

V predchádzajúcej časti sme popísali ako sa podstrom NodeTree odstráni z pamäte. Náš framework sme navrhli, tak, že pri WYSIWYG editácii môže nastať situácia, keď budeme potrebovať informácie z týchto zahodených uzlov stromu NodeTree, preto framework podporuje proces obnovenia častí tejto internej štruktúry. Tento proces nazývame nahrávanie.

Pri nahrávaní treba brať do úvahy dve situácie, a to keď FATS podporuje nahrávanie a keď nepodporuje nahrávanie. V prvom prípade o nahratie zahodeného podstromu môže požiadať objekt z inej fázy spracovania alebo aj samotný koreň zahodeného podstromu. V tomto prípade strom NodeTree poskytuje transparenciu nahratia, teda objekt z inej fázy spracovania volá metódy koreňa zahodeného podstromu bežným spôsobom, interne sa však zavolá nahratie a až potom je poskytnutá funkcionalita. Pri druhej situácii, keď FATS nepodporuje nahrávanie, je na zodpovednosti objektu z neskoršej inej fázy spracovania, aby pred prístupom k zahodenému podstromu najskôr zavolal metódu `load()` na koreni, čo spôsobí vytvorenie tohto podstromu NodeTree a až potom k nemu pristupoval.

Aby FATS vedel, ktoré XML elementy má čítať, mal by elementy označiť pomocou atribútu id, ktorý je z nejakého špeciálneho menného priestoru a predstavuje jednoznačný identifikátor elementu. Tento identifikátor sa uloží v uzle stromu NodeTree a ostane v ňom uložený aj po zahodení podstromu s koreňom v tomto uzle. Dôležité je, že keď sa budú pomocou FATS opäť vytvárať zahodené uzly, budú im pridelené také isté identifikátory ako mali pred tým, ako boli zahodené. Ak toto FATS nevie zabezpečiť potom nepodporuje nahrávanie.

[30] To či sa dané inštancie naozaj odstránia z pamäte má na starosti Garbage Collector platformy Java

Proces nahrávania zahodeného podstromu sme navrhli nasledovne. Objekt, ktorý iniciuje nahrávanie zavolá metódu `load()` na koreni tohto podstromu. Koreň nastaví v triede `EventHandler` príznak `loading` na `true`, čo má za následok, že sa nespúšťajú ďalšie fázy spracovania stromu NodeTree, a preto sa FATS môže na nahratie zahodených uzlov použiť tie isté metódy ako keby bol XML dokument transformovaný do internej štruktúry NodeTree prvý krát.

- Ak FATS podporuje load, tak koreň zavolá metódu z FATS, ktorá nahrá podstrom NodeTree z podstromu elementov, kde koreňový element má identifikátor, taký ako identifikátor koreňa podstromu v NodeTree. FATS vykoná parsovanie príslušnej časti XML dokumentu pričom nevytvára koreň stromu NodeTree, ale rovno na ňom volá metódu na spracovanie atribútov, poprípade metódu na spracovanie textovej časti alebo pridávanie detí a nakoniec metódu `endNode()` hovoriacu o narazení na uzatváraciu značku pre tento element. Parsovanie podelementov prebieha normálne a pri tom sa vytvárajú a napájajú uzly stromu NodeTree pomocou tých istých metód ako pri vytváraní stromu NodeTree prvý krát. Nakoniec koreň daného podstromu v NodeTree, zmení príznak `loading` v `EventHandleri` na `false`.

- Ak FATS nepodporuje load, tak koreň zavolá metódu z FATS, ktorá nahrá celý strom NodeTree odznovu a na konci `EventHandler` zachytí udalosť `endDocument()` a nastaví príznak `loading` na `false`.

3.5.3 Nepriamy prístup k uzlom stromu NodeTree

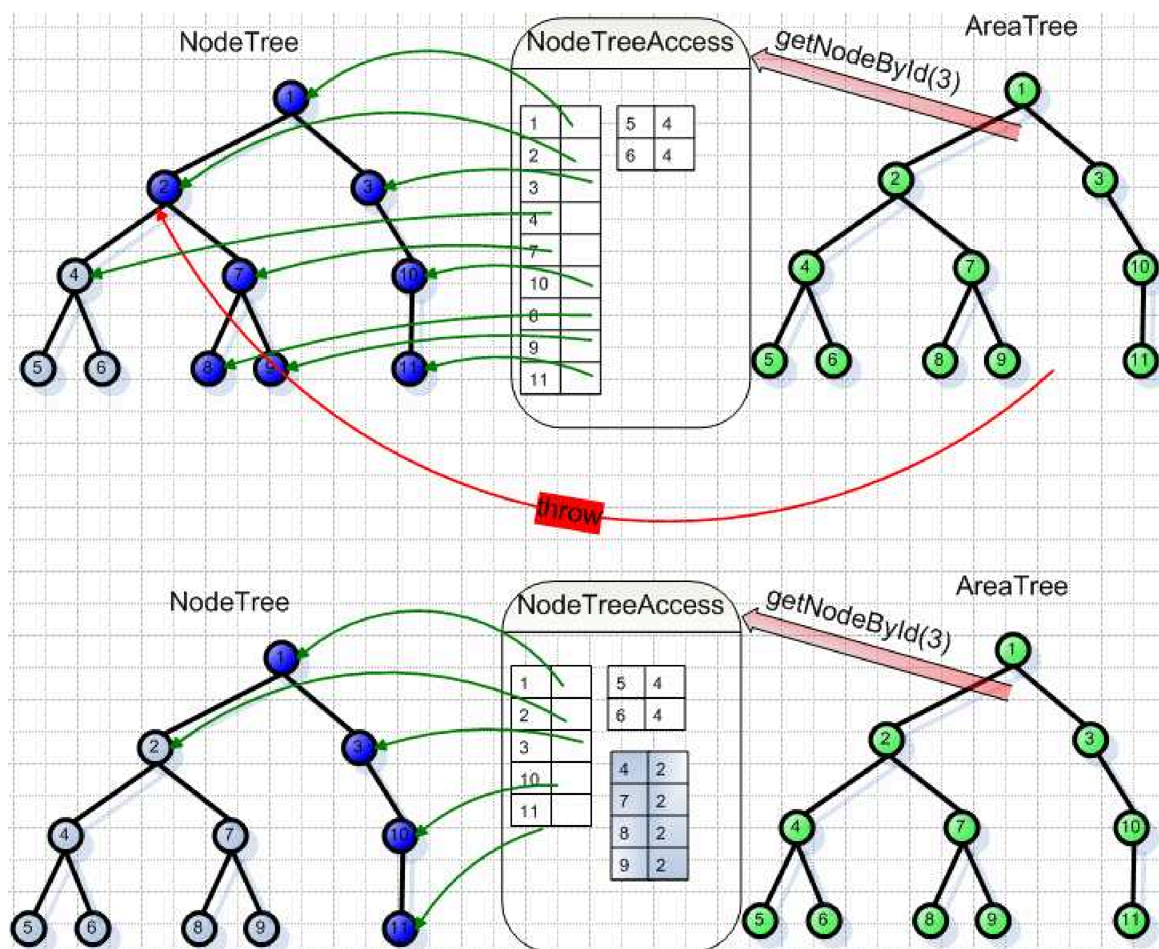
Zahadzovanie podstromov stromu NodeTree šetrí pamäť, ale objekty z neskoršej fázy spracovania stromu NodeTree nemôžu k nemu pristupovať priamo, keďže k inštanciám po zahodení už nie je prístup. Objekt z neskoršej fázy vie identifikátor uzla, ku ktorému chce pristupovať, ale nepozná koreň zahodeného podstromu, v ktorom sa nachádza.

Jednou možnosťou ako danú situáciu vyriešiť je, že si každý uzol bude pamätať množinu identifikátorov uzlov ktoré sú v jeho podstrome. Prístup k inštancii uzla pomocou jeho identifikátora by potom bolo nasledovný. Začalo by sa prehľadávanie stromu NodeTree od koreňa. Tam by sa našiel syn koreňa, v ktorom sa nachádza hľadaný uzol a tak ďalej rekurzívne, až by sa nenašla inštancia daného uzla alebo by sa našiel zahodený uzol, ktorý je koreňom podstromu, v ktorom sa hľadaný uzol nachádza. Tento podstrom by sa potom nahral a prehľadávanie by pokračovalo ďalej, až kým by sa daný uzol nenašiel. Nevýhodou tohto riešenia je, že si musíme zapamätať pri vybalansovanom strome $O(n.\log n)$ identifikátorov a v najhoršom prípade $O(n^2)$ identifikátorov. Navyše aj vyhľadanie pri vybalansovanom strome môže trvať $O(\log n)$ času.

Navrhli a implementovali sme iný prístup, ktorý je efektívnejší. V našom novom prístupe budujeme dve mapovania. V prvom mapovaní je kľúčom jednoznačný identifikátor uzla a hodnotou je inštancia uzla. Pri zahadzovaní podstromu sa z tohto mapovania vyhadzujú príslušné dvojice identifikátora a inštancie, okrem koreňa tohto podstromu a navyše sa tvorí ďalšie mapovanie, kde kľúčmi sú identifikátory zahodených uzlov a hodnota je identifikátor koreňa. Toto mapovanie sa tvorí počas zahadzovania jednotlivých uzlov v

tomto podstrome a upravuje sa pri nahrávaní uzlov. Keď pristupujeme k uzlu, ktorý nie je v zahodenom podstrome, tak inštanciu nájdeme v mapovaní v čase $O(1)$. Ak pristupujeme k zahodenému uzlu, tak v čase $O(1)$ zistíme, že nie je nahratý a identifikátor zahodeného koreňa, v ktorom sa tento uzol nachádza, nájdeme v čase $O(1)$ a teda aj jeho inštanciu tohto koreňa nájdeme v čase $O(1)$. Týmto prístupom sme oproti predchádzajúcemu riešeniu zmenšili aj priestor potrebný na uloženie identifikátorov na $O(n)$.

Pri implementácii druhého riešenia sa využíva trieda `NodeTreeAccess`, ktorá zabezpečuje transparenciu nahratia uzlov stromu NodeTree pomocou už spomenutých dvoch mapovaní.



Obrázok 12: Zahadzovanie podstromu NodeTree

3.6 Zmeny dokumentu vo fáze NodeTree

V tejto podkapitole popíšeme ako sme navrhli a implementovali časť modulu NodeTree, ktorá zachytáva požiadavku na editáciu z neskoršej fázy, spracúva ju a preposiela do FATS.

Tiež popíšeme ako sme navrhli a implementovali rozhranie modulu NodeTree, ktoré slúži na zachytenie aktualizácie zmeneného XML dokumentu.

V častiach 3.6.1, 3.6.2 a 3.6.3 popíšeme všeobecný návrh editačných požiadaviek, rozhraní na ich príjem a ich spracovanie v našom frameworku. V častiach 3.6.4, 3.6.5 a 3.6.6 popíšeme ako sme tento návrh implementovali. V časti 3.6.7 popíšeme k akým zmenám môže dôjsť v XML dokumente počas jeho WYSIWYG editácie a v častiach 3.6.8 a 3.6.9 popíšeme ako sme navrhli a implementovali zachytenie týchto zmien.

K tejto podkapitole sa vzťahujú prílohy 9, 10 a 11, ktoré nám uľahčia jej pochopenie.

3.6.1 Editačná požiadavka

V tejto časti si ujasníme pojem editačná požiadavka, ktorý budeme používať vo všetkých fázach spracovania XML.

V najvrchnejšej fáze (grafická plocha) budeme pod týmto pojmom rozumieť akciu užívateľa za účelom zmeny dokumentu. Tieto zmeny môžu byť jednoduché, napríklad dopísanie písmena, ale aj omnoho komplexnejšie zmeny ako napríklad nahradenie časti dokumentu inou. Tieto akcie sú vyvolané napríklad stláčaním kláves klávesnice, klikaním a pohybom myši alebo prácou s iným vstupným zariadením počítača (tablet).

V AreaTree pod týmto pojmom rozumieme zmenu črty v oblasti, pridanie alebo vymazanie podstromu AreaTree a nahradenie podstromu AreaTree iným.

V NodeTree sú editačné požiadavky nasledovné : dopísanie textu v textovom uzle, zmazanie textu v textovom uzle, nahradenie textu iným v textovom uzle, zmena hodnoty explicitnej vlastnosti, vytvorenie explicitnej vlastnosti s hodnotou, vymazanie hodnoty explicitnej vlastnosti a jej zaradenie medzi zdedené, zmazanie alebo pridanie podstromu NodeTree a nahradenie podstromu NodeTree iným.

XML editačné požiadavky sa delia na jednoduché a zložené. Jednoduché sú tieto: zmena textovej časti XML elementu, nahradenie zoznamu atribútov iným, zmazanie elementu aj s podelementami a vloženie jedného elementu bez podelementov. Zložené požiadavky sú definované rekurzívne a skladajú sa z jednoduchých a zložených požiadaviek. Napríklad vloženie stromu elementov, je zložené z viacerých jednoduchých požiadaviek vloženia elementu alebo napríklad nahradenie podstromu elementov iným sa skladá z jednoduchej požiadavky zmazania stromu elementov a zloženej požiadavky vloženie stromu elementov.

3.6.2 Použitie editačných požiadaviek

V tejto časti vysvetlíme, ako sa v našom návrhu používajú editačné požiadavky.

V každej fáze spracovanie XML sa zachytávajú požiadavky, transformujú do požiadaviek skoršej fázy a preposielajú do skoršej fázy pomocou jej rozhrania (pozri obrázky 8 a 9). Najskôr sa akcia používateľa na pohľade v MVC zachytí a transformuje pomocou kontrolera do požiadavky alebo požiadaviek pre AreaTree alebo pre NodeTree (3.8). Obdobne z AreaTree do NodeTree a z NodeTree do editačných požiadaviek XML. Ak sa jedná o techniku editovania typu dirty rendering alebo dirty layout, tak editačná požiadavka nesie v sebe príznak `dirty`, ktorý hovorí o tom, že neskoršia fáza (označe si ju ako fáza 2)

nečaká na aktualizáciu zo skoršej fázy (označme si ju ako fáza 1), ale ju predvída. Fáza 2 na základe predvídanej aktualizácie potom vykoná aktualizáciu fázy 3, ktorá nasleduje za fázou 2.

3.6.3 Príjem editačnej požiadavky

V tejto časti popíšeme ako sme navrhli proces prijatia a spracovanie editačnej požiadavky v moduloch nášho frameworku.

V našom prvom návrhu sme chápali v každej fáze alebo vrstve [31] editačnú požiadavku ako funkciu s návratovou hodnotou áno alebo nie, podľa toho, či daná požiadavka bola akceptovaná. Neskôr sme usúdili, že tento návrh nie je výhodný, lebo ak požiadavka bola akceptovaná, tak celá vyššia vrstva môže byť vytvorená nanovo, a teda nemá zmysel posielať správu o úspechu (posielať správu o neúspechu má zmysel). Z toho sa odvíjal druhý návrh, v ktorom bola editačnú požiadavka rozdelená na dve časti, na zistenie, či je takáto editácia povolená a editačný príkaz. Ale zistenie, či je takáto editácia povolená v sebe nesie tie isté informácie ako príkaz na túto editáciu a často proces zistenia, či je takáto editácia povolená v sebe obsahuje zmenu nižšej vrstvy, ako keby prišiel príkaz a následne jej kontrolu, čo vedie k duplicite a mrhaniu systémových prostriedkov. Náš finálny návrh chápe editačnú požiadavku ako príkaz. [32] Navyše medzi vrstvou FATS a NodeTree a medzi NodeTree a AreaTree pribudla požiadavka na zistenie povolených zmien a rozhranie pre odpoveď na túto požiadavku. Pričom nemusí ísť o všetky povolené zmeny, ale napríklad pre odporúčané alebo najčastejšie. Je potom na zodpovednosti neskoršej fázy, aby v editačnej požiadavke boli povolené zmeny. Ak v nej nie sú, tak nižšia vrstva vyvolá výnimku a táto musí byť zachytená vo vyššej vrstve.

3.6.4 Príjem NodeTree editačnej požiadavky

Návrh prijímania editačnej požiadavky sme v module NodeTree implementovali prostredníctvom rozhrania `IEditableNode`, ktoré obsahuje metódy `textChangeRequest`, `propertyChangeRequest`, `deleteTreeRequest`, `insertSubtreeRequest` a `replaceTreeRequest`, ktoré okrem iných parametrov obsahujú parameter `isDirty`, ktorý hovorí o tom, či FATS má urobiť aktualizáciu stromu NodeTree. Nehovorí však nič o spôsobe aktualizácie, teda či sa má vykonať čiastočná XSLT transformácia, alebo sa bude čítať celý pozmenený dokument s umelými atribútmi hovoriacimi o zmenách alebo sa celý NodeTree zahodí a vytvorí odznovu, akoby bol čítaný prvý krát alebo iný spôsob. Základná implementácia tohto rozhrania `EditableXMLNode` dedí funkčnosť triedy `XMLNode` a pridáva k nej spracovanie týchto editačných požiadaviek a ich transformáciu na XML editačné požiadavky, ktoré sú popísané v nasledujúcej časti tejto práce.

[31] pojmy fáza a vrstva budeme stotožňovať a tak isto skoršia/neskoršia fáza a nižšia/vyššia vrstva
[32] Od teraz budeme pojmy editačná požiadavka a editačný príkaz stotožňovať

3.6.5 Príjem XML editačnej požiadavky

V tejto časti práce návrhneme časť rozhrania `IFATS`, ktorá slúži na príjem editačnej požiadavky.

Editačnú požiadavku na zmenu XML dokumentu budeme nazývať XML editačná požiadavka. Náš prvý návrh prijímania XML editačných požiadaviek bol tesne vytvorený pre najbežnejšie operácie ako sú zmena textu, zmena atribútov, vymazanie elementu a pridanie elementu a pridanie stromu elementov. O iných zmenách sa neuvažovalo a navyše s poslednou bol problém, pretože implemntátorovi rozhrania `IFATS` bol ako vstup daný len koreň podstromu NodeTree, ktorý musel prechádzať a vytvárať z neho XML kód. Nevýhodou tohto riešenia bolo to, že XML editačná požiadavka bola "v jazyku NodeTree", čo porušuje modulárnosť, a tiež nízka flexibilita.

Tieto nedostatky sme odstránili vo finálnom návrhu časti rozhrania `IFATS` na prijímanie XML editačných požiadaviek. Rozhranie `IFATS` pozostáva z jedinej metódy
`void changeDocument(XMLEditRequest editRequest) throws NotValidException`.
Trieda `XMLEditRequest` obsahuje základné informácie o zmene, ktoré sú dostačujúce pre všetky zmeny týkajúce sa jedného uzla. Tieto informácie sú "v jazyku NodeTree", napríklad ktorého uzla sa zmena týka, nový text, miesto v NodeTree, kde sa zmena odohráva alebo nový uzol, ktorý sa má vložiť, ale aj v jazyku XML, napríklad nové SAX atribúty, ďalej typ požiadavky, napríklad zmena textu alebo zložená požiadavka, o ktorej budeme hovoriť neskôr a typ aktualizácie, napríklad dirty, čiastočná XSLT a iné. Od tejto triedy je odvodená trieda `CompoundXMLEditRequest`,ktorá predstavuje zloženú editačnú požiadavku, ktorá obsahuje zoznam požiadaviek `XMLEditRequest`, a teda aj
`CompoundXMLEditRequest`, ktoré sa vykonávajú v poradí, v akom sú uložené v zozname. Takto sme zabezpečili, že sa pomocou týchto dvoch tried dajú naraz poslať do FATS akékoľvek zmeny NodeTree, a teda sa môžu vykonať a následne validovať aj viaceré nesúvisiace požiadavky. Navyše požiadavky, ako vloženie podstromu NodeTree, sa dajú rozdeliť na viac jednoduchých podpožiadaviek, napríklad vkladanie uzlov postupne, čím implementátor frameworku môže zmenšiť nároky na implementáciu rozhrania `IFATS`. Tieto dve triedy sú dostatočne všeobecné na vyjadrenie XML editačných požiadaviek, ale na vyjadrenie zmeny používajú objekty z NodeTree, preto náš framework poskytuje rozšírenia týchto požiadaviek, ktoré zabezpečujú najčastejšie zmeny NodeTree a tieto zmeny sú vyjadrené čisto v jazyku XML.

Jednoduché (nie zložené) požiadavky, ktoré náš framework poskytuje, sú :

- `XMLTextChangeRequest`, ktorá slúži na zmenu textovej časti XML elementu
- `XMLAttributesChangeRequest`, ktorá slúži na zmenu všetkých atribútov naraz, podporované sú DOM aj SAX atribúty
- `XMLInsertElementRequest`, ktorá slúži na vloženie jedného XML elementu
- `DeleteElementRequest`, ktorá slúži na vymazanie XML elementu aj s podelementami.

Zložené požiadavky, ktoré náš framework poskytuje, sú :

- `XMLInsertTreeRequest`, ktorá slúži na vloženie stromu XML elementov a obsahuje zoznam požiadaviek `XMLInsertElementRequest`, ktorých poradie v tomto zozname predstavuje inorderové prehľadávanie vkladaného stromu elementov
- `XMLReplaceSubtree`, ktorá slúži na nahradenie stromu XML elementov iným a obsahuje dvojprvkový zoznam podpožiadaviek, kde prvá je `XMLDeleteElementRequest` a druhá je tiež zložená a je to `XMLInsertTreeRequest`

3.6.6 Poznámky pre implementátora frameworku

Tieto požiadavky sú referenčné a implementátor frameworku môže použiť ľubovoľné iné, vlastné rozšírenie triedy `XMLEditRequest` na preposlanie zmeny v NodeTree do FATS. Framework sme navrhli tak, aby sa validácia nerobila pre každú podpožiadavku zloženej XML požiadavky, ale len raz pre najvrchnejšiu zloženú XML požiadavku. Jednotlivé medzidokumenty zodpovedajúce postupnosti podpožiadaviek nemusia byť validné, ale výsledný áno a navyše validácia je proces náročný na systémové požiadavky, preto je efektívnejšie robiť ho len raz. Taktiež aktualizácia stromu, o ktorej ešte budeme písať sa nerobí pre každú podpožiadavku, ale až po aplikovaní všetkých zmien v XML dokumente, ale to len v prípade, že najvrchnejšia zložená požiadavka nie je dirty. Ak nejaká podpožiadavka nie je dirty, tak ani zložená požiadavka nie je dirty. Ak sa pri vytváraní zloženej požiadavky nevie, aký má byť typ aktualizácie, tak sa nastaví dirty a keď sa pri vkladaní podpožiadaviek narazí na nejakú, ktorá nie je dirty, tak sa typ zloženej požiadavky nastaví na rovnaký.

3.6.7 Zmena XML dokumentu

V tejto kapitole budeme analyzovať aké typy zmien môžu počas WYSIWYG editácie XML dokumentov nastať. Na základe tejto analýzy budeme môcť urobiť návrh rozhrania modulu NodeTree na zachytenie týchto zmien.

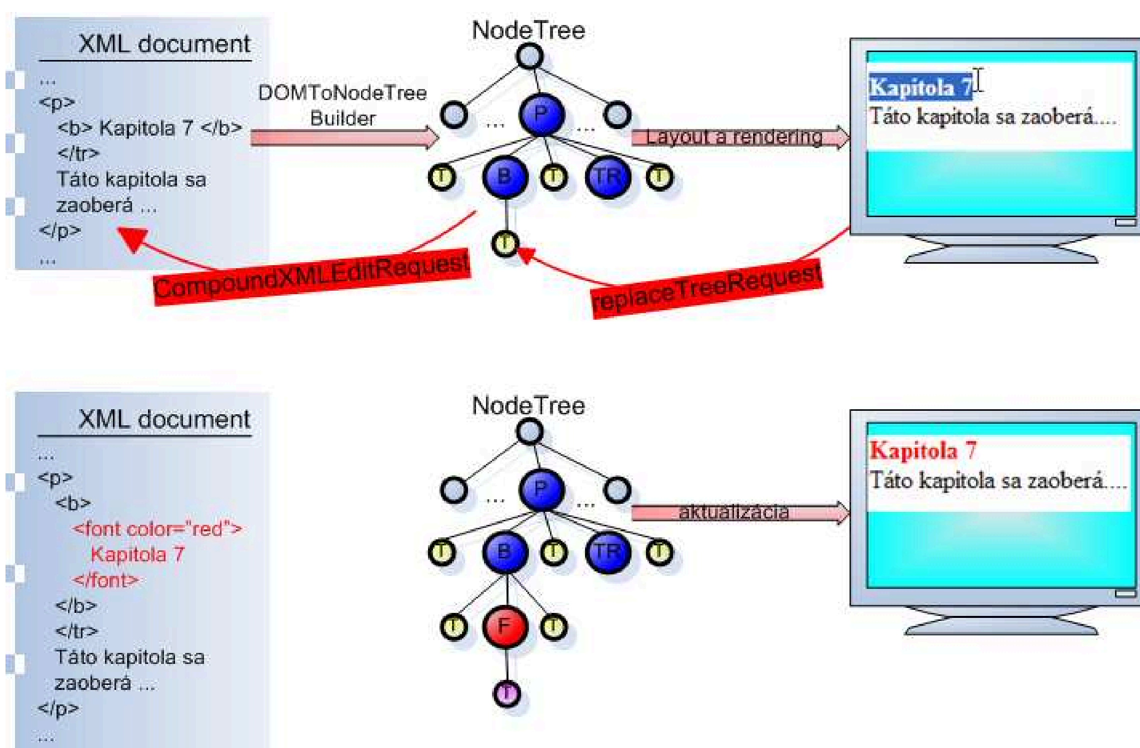
Uvažujme o dvoch prístupoch k WYSIWYG editovaniu XML dokumentov. Prvý prístup je taký, že sa edituje dokument, ktorý je napísaný v nejakom XML jazyku alebo vo viacerých XML jazykoch. Tento dokument obsahuje biznis dáta aj informácie o tom, ako má byť zobrazený a nie je XSLT transformáciou ani iným spôsobom upravený. Tento dokument je v takejto forme poskytnutý rendereru na zobrazenie a editáciu.

Druhý prístup je taký, že XML dokument, ktorý sa edituje, obsahuje čisto len dáta (napríklad z databázy) a nie je špecifikované ako sa majú tieto dáta zobraziť. K tomuto dátovému XML dokumentu môže byť v XML editore pridaný štýl (XSLT transformácia), pomocou ktorého sa dokument transformuje [33] do XML dokumentu, ktorý obsahuje prezentačné dáta a tento je poskytnutý rendereru. WYSIWYG editačnými technikami XML dokumentov sa bližšie zaoberá práca [18].

[33] K transformácii, ktorá pridá dátovému XML dokumentu prezentačné informácie môže dôjsť aj iným spôsobom ako XSLT transformáciou.

Rozoberieme si teraz typické situácie, ktoré z pohľadu nášho frameworku môžu pri WYSIWYG editácii XML nastať :

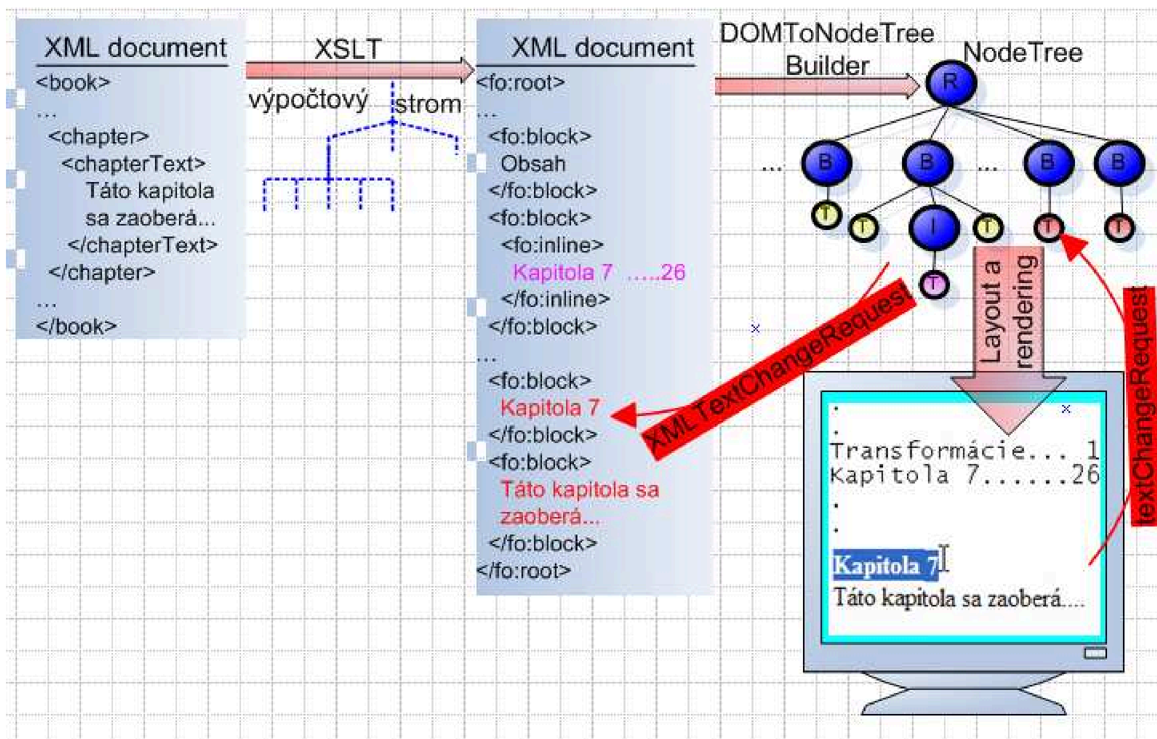
- Dirty
  Pri prvom prístupe WYSIWYG editácie XML dokumentov, ale aj v mnohých iných prípadoch pri druhom prístupe WYSIWYG editácie XML dokumentov, ktoré v sebe neobsahuje skrytú funkcionalitu (napríklad obsah), je ľahké predvídať, ako by mal po aktualizácii XML dokumentu vyzerať nový NodeTree, ktorý je vnútornou reprezentáciu tohto dokumentu v našom frameworku. Preto často nie je nutné, aby FATS lokalizoval, kde v dokumente nastala zmena a aktualizoval NodeTree.



Obrázok 13: Dirty editovanie na úrovni NodeTree

  Na obrázku 13 je požiadavka z neskoršej fázy, ktorá sa snaží zmeniť farbu textu kapitoly, transformovaná do zloženej požiadavky skladajúcej sa z vymazania textu a vloženie elementu. V tejto požiadavke je špecifikovaný typ aktualizácie na dirty. NodeTree odhadne ako sa má zmeniť tak, aby zodpovedal zmenenému XML dokumentu a synchronizácia XML dokumentu s NodeTree sa nerobí.

- Čiastočná XSLT transformácia
  Ak pri druhom prístupe editovania XML dokumentov nie je ľahké odhadnúť, ako bude po aktualizácii vyzerať NodeTree, je nutné ho na základe zmien transformovaného XML dokumentu aktualizovať. Situáciu znázorňuje obrázok 14, kde XSLT pridá XML dokumentu informácie o zobrazení a funkcionalitu navyše akou je obsah.



Obrázok 14: Editovanie XML dokumentu s XSLT transformáciou

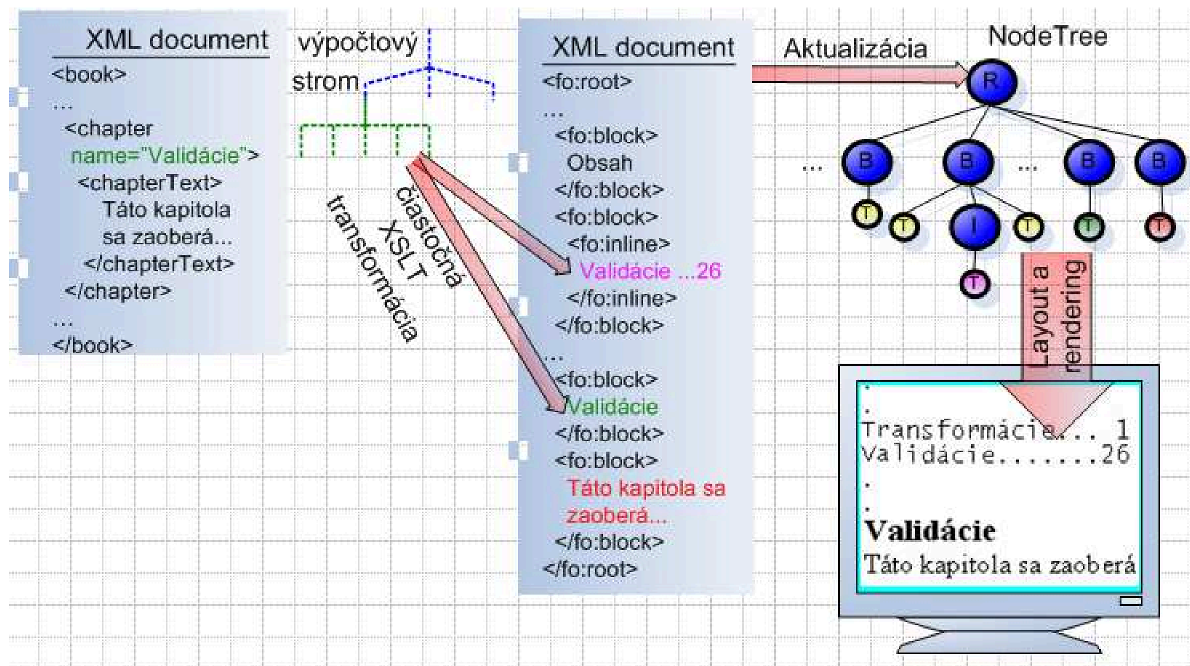
Pri zmene mena kapitoly sa príslušne zmení aj obsah, poprípade aj číslovanie kapitol, ktoré neboli pomenované. FATS môže na základe zmeny svojho výpočtového stromu a ďalších informácii lokalizovať zmeny v transformovanom XML dokumente (obrázok 15) a aktualizovať strom NodeTree.

Jednou možnosťou ako aktualizovať NodeTree je taká, že by sa volali SAX udalosti elementov [34] , ktoré sú nadelementami elementov, kde nastala zmena a sú nezmenené a následne by sa volali SAX udalosti len pre tie detské elementy, ktoré boli zmenené. Na spracovanie týchto SAX udalostí by slúžila už existujúca časť frameworku, len by sa do metód pridal príznak hovoriaci o tom, že ide o aktualizáciu a odlíšil by tak proces, keď je dokument nahrávaný prvý krát. Framework by potom musel na základe porovnávania predchádzajúcej štruktúry NodeTree a prichádzajúcich udalostí vyhodnotiť, o akú zmenu ide. Toto riešenie vyzerá ťažkopádne, pozrime sa ešte na ďalšie situácie.

- Celý dokument s informáciami o zmenách

  Opäť sa pozrime na situáciu v predošlom príklade, keď nevieme odhadnúť, ako bude po aktualizácii XML dokumentu vyzerať NodeTree a navyše FATS nemá funkcionalitu, ktorá by bola schopná volať SAX udalosti pre nezmenený nadelement a potom len pre zmenené podelementy.

[34] Situácia sa dá ľahko predstaviť aj ako prechádzanie stromom DOM.



Obrázok 15: Aktualizácia NodeTree s čiastočnou XSLT transformáciou

Ak však FATS vie pomocou výpočtového stromu a ďalších informácii zmenu lokalizovať, nie je ťažké pridať do elementov umelé atribúty z nejakého špeciálneho menného priestoru hovoriace o tom, či v nejakom prehľadávaní stromu XML elementov, napríklad v preorderi je element prvý zmenený, posledný zmenený, pred prvým zmeneným, za posledným zmeneným, medzi dvoma zmenenými alebo či podstrom elementov obsahuje zmenený element. Aktualizácia NodeTree by potom prebiehala tak, že by sa zmenený a transformovaný XML dokument čítal celý, ale nezmenené elementy by sa ignorovali, ale pri zmenených elementoch by framework na základe predošlého stavu stromu NodeTree a na základe nových SAX udalostí musel analyzovať, aká zmena nastala. Toto riešenie v sebe zahŕňa problémy druhej situácie.

- Celý dokument

  V prípade, že je ťažké odhadnúť, ako bude po zmene aktualizovaný NodeTree vyzerať a FATS nepodporuje ani čiastočnú XSLT transformáciu, ani pridanie dodatočných informácii o zmenách alebo keď sa implementátor frameworku rozhodne, že nie je nutné alebo efektívne robiť aktualizáciu časti NodeTree, nezostáva nič iné ako poskytnúť transformovaný dokument obyčajným spôsobom, tak ako by bol čítaný prvý krát. Takto sa musia zahodiť všetky doteraz vypočítané dáta vrátane stromu NodeTree, prípadne AreaTree a grafických entít a vytvoriť ich nanovo.

3.6.8 Aktualizácia NodeTree

V tejto časti popíšeme ako sme navrhli a implementovali rozhranie modulu NodeTree na aktualizáciu stromu NodeTree na základe zmien XML dokumentu.

Na návrh rozhrania vrstvy NodeTree na zachytenie zmeneného dokumentu nevplýva prvá a štvrtá situácia, popísaná v predchádzajúcej časti. Riešenia druhej a tretej situácie, nútia byť framework príliš komplexný, komplikovaný a ťažkopádny a navyše kladú špecifické požiadavky na FATS. Preto je ponechané na programátorovi FATS, ako zmenu elementu lokalizuje. Na aktualizáciu stromu NodeTree pritom môže použiť existujúce časti frameworku, pomocou ktorých môže z elementov vytvoriť uzly stromu NodeTree a z atribútov zoznam vlastností.

Rozhranie `IEditableNode` na zachytenie zmeneného dokumentu používa metódy : `updateText`, `updatePropertyList`, `insertNodeTree`, `deleteNodeTree` a `replaceNodeTree`. Avšak nemusí ísť iba o jednu zmenu zachytenú jednou z týchto metód, ale môže ísť o blok zmien, tieto metódy preto okrem parametrov popisujúcich príslušnú zmenu obsahujú aj búlovské parametre `isFirst` a `isLast` hovoriace o tom, či je táto zmena prvá, respektíve posledná v danom bloku zmien. Tieto parametre pomáhajú implementátorovi frameworku pri rozhodovaní, kedy zmeny v NodeTree pošle do ďalšej fázy.

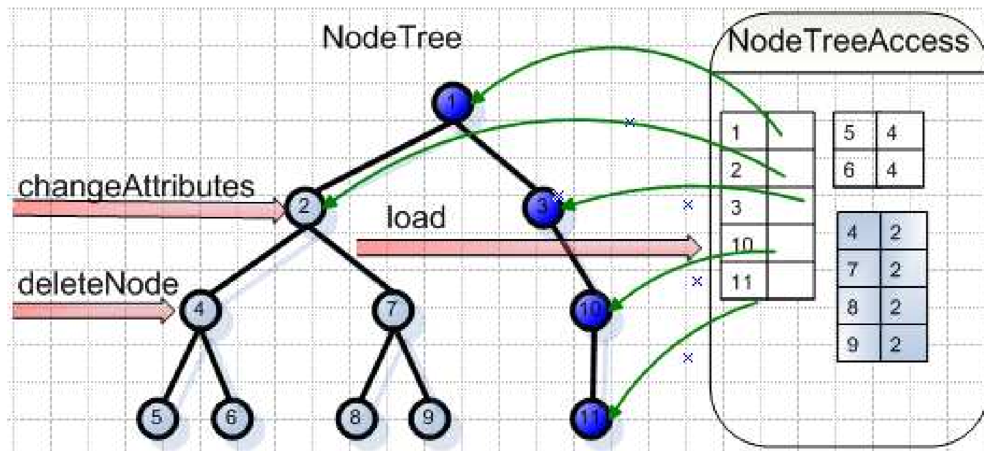
3.6.9 Aktualizácia NodeTree a nahrávanie

V tejto časti popíšeme aké problémy môžu vniknúť pri aktualizácii stromu NodeTree, ak sú niektoré jeho podstromy zahodené (3.5). Ďalej uvedieme ako sme navrhli a implementovali riešenia týchto problémov.

FATS pristupuje k uzlom vrstvy NodeTree nepriamo cez triedu `NodeTreeAccess`, ktorá zabezpečuje transparenciu nahratia. Keďže proces nahrávania pristupuje už k zmenenému XML dokumentu, tak sa nahrá časť NodeTree už aj zo zmenou a túto zmenu už nie je potrebné vykonať, práve naopak, táto zmena by bola chybou. Preto musí FATS kontrolovať či uzol, na ktorom ide zmenu vykonať, nie je zahodený. Ak je, tak FATS pomocou `NodeTreeAccess` zavolá nahratie. FATS si však musí zaznamenať, ktoré zo zmien boli takto zahrnuté, aby ich už nerobil a neskôr o nich informoval modul NodeTree (3.6.10). Takto sa zabráni nekonzistencii medzi XML dokumentom a stromom NodeTree, ktorá by inak vznikla. Jednu z možných situácií znázorňuje obrázok 16.

Mohlo by sa zdať, že ak uzol NodeTree žiada o aktualizáciu, tak musí byť nahratý, to je pravda, ale zmena sa môže týkať aj iných častí NodeTree ako tých, ktoré vytvorili požiadavku na zmenu, napríklad obsah alebo index knihy. XML dokument sa však mohol zmeniť dokonca aj bez akejkoľvek požiadavky pochádzajúcej z nášho frameworku, napríklad pri viacnásobnom prístupe k dokumentu, keď jeho časť bola zmenená iným používateľom alebo keď sa dokument mení v závislosti od zmeny databázy.

Problém môže vzniknúť pri aktualizácii stromu NodeTree vymazaním podstromu alebo nahradením podstromu, ak je táto operácia volaná na koreni stromu NodeTree, ktorý má byť zahodený alebo nahradený. Ak je totiž tento uzol zahodený, tak aby sa na ňom zavolala metóda zahodenia, musí byť najskôr nahratá jeho inštancia, to však ale nie je možné, keďže



Obrázok 16: Chyba pri aktualizácii NodeTree po nahratí

uzol je zahodený. Preto sa vymazávanie podstromu NodeTree a nahrádzanie podstromu NodeTree volá cez rodiča koreňa zahadzovaného alebo nahrávaného stromu, ako operácia typu *zahoď dieťa* alebo *nahraď dieťa*.

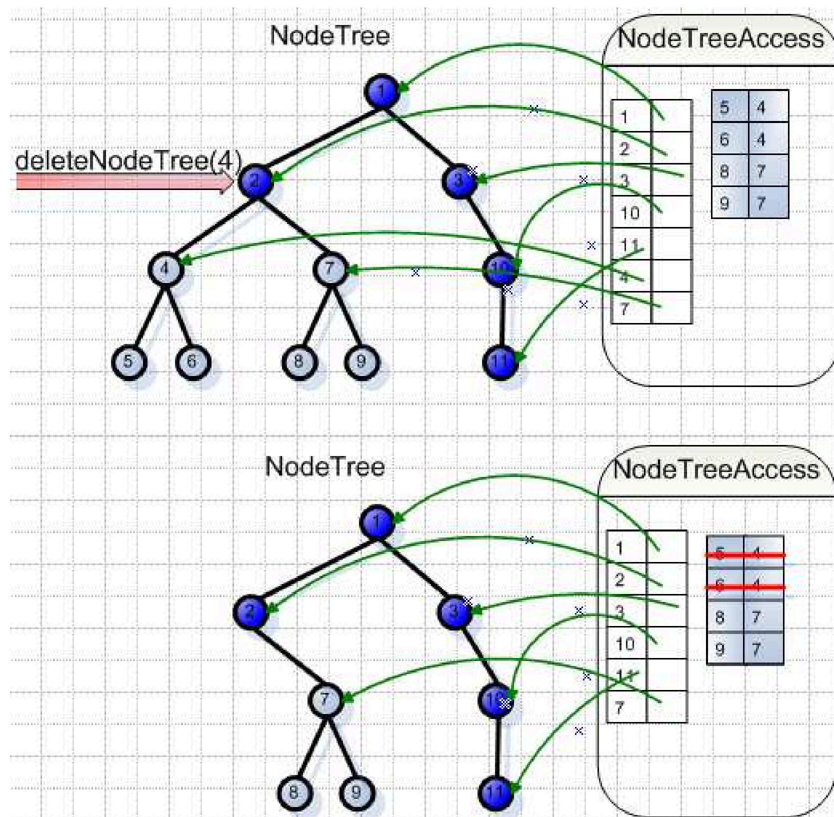
Pri implementácii vymazávania a nahrádzania podstromu si musíme uvedomiť, že keď pri vymazávaní uzlov narazíme na uzol, ktorý je koreňom zahodeného stromu, [35] tak ho nenahrávame, ale len odstránime všetky záznamy o uzloch z tohto stromu z triedy `NodeTreeAccess`. Jednu zo situácií znázorňuje obrázok 17. Tak isto pri implementácii pridávania stromu uzlov a nahrádzania podstromu uzlov iným je nutné aktualizovať mapovanie medzi identifikátormi uzlov a ich inštanciami v triede `NodeTreeAccess`.

3.6.10 Informovanie o nahratej zmene

V tejto časti popíšeme ako sme navrhli a implementovali rozhranie na príjem informácii o zmenách v XML dokumente, ktoré boli nahraté.

Ak pri WYSIWYG editácii vznikla zmena v XML dokumente, ktorá sa týkala zahodenej časti stromu NodeTree, tak sa tieto uzly nahrali už zmenené. NodeTree je teraz síce aktualizovaný, ale nevie ako sa zmenil, a preto nevie túto zmenu poslať, či už do kontrolera z MVC alebo do AreaTree. Preto `EventHandler` obsahuje metódy, pomocou ktorých môže FATS oznámiť ktoré zmeny XML dokumentu boli takto nahraté. Ide o metódy : `changedByText`, `changedByPropertyList`, `changedByInsertion`, `changedByDeletion` a `changedByReplacement`. V týchto metódach sa môžu vytvoriť požiadavky hovoriace o týchto zmenách a preposlať sa do neskorších fáz spracovania. Tieto metódy okrem parametrov popisujúcich zmenu obsahujú aj príznaky `isFirst` a `isLast` hovoriace či je táto zmena prvá, respektíve posledná v danom bloku zmien. Na ich základe sa môže implementátor rozšírenia triedy `EventHandler` rozhodnúť, kedy zaktualizuje ďalšiu fázu.

[35] na zahodený uzol, ktorý nie je koreňom zahodeného stromu, nemôžme naraziť lebo neexistuje objekt, ktorý má na neho referenciu a pri pokuse o získanie jeho referencie cez triedu `NodeTreeAccess` by sa daný uzol nahral.



Obrázok 17: Správne vymazávanie zahodeného uzla

3.7 Rozmiestnenie

V tejto podkapitole popíšeme ako sme navrhli a implementovali časti nášho frameworku, ktoré podporujú oddelený proces rozmiestňovania a vykresľovania uzlov stromu NodeTree. Táto podkapitola popisuje hlavne moduly LayoutManager a AreaTree nášho frameworku, ktorého komponentový UML diagram sa nachádza v prílohe 3. V časti 3.7.1 popíšeme základný koncept rozmiestňovania a vysvetlíme základné pojmy. Statickému a dynamickému rozmiestňovaniu a vykresľovaniu sa venujeme v častiach 3.7.2 a 3.7.3. V časti 3.7.4 popíšeme ako sme navrhli aktualizáciu rozmiestnenia. Vytvorili sme podporu pre urýchlenie rozmiestňovania pomocou dirty editačných techník (3.7.5). V časti 3.7.6 popíšeme ako sme naviazali modulu LayoutManager na modul NodeTree a v časti 3.7.7 ukážeme ako sme navrhli zahadzovania uzlov stromu AreaTree.

3.7.1 Základný koncept rozmiestňovania

V tejto časti popíšeme ako sme navrhli kostru procesu, ktorý zabezpečuje oddelené rozmiestňovanie od vykresľovania a oboznámime sa so základnými pojmami.

Rozmiestňovanie je proces určovania rozmerov a pozície objektov na ploche. V klasick-

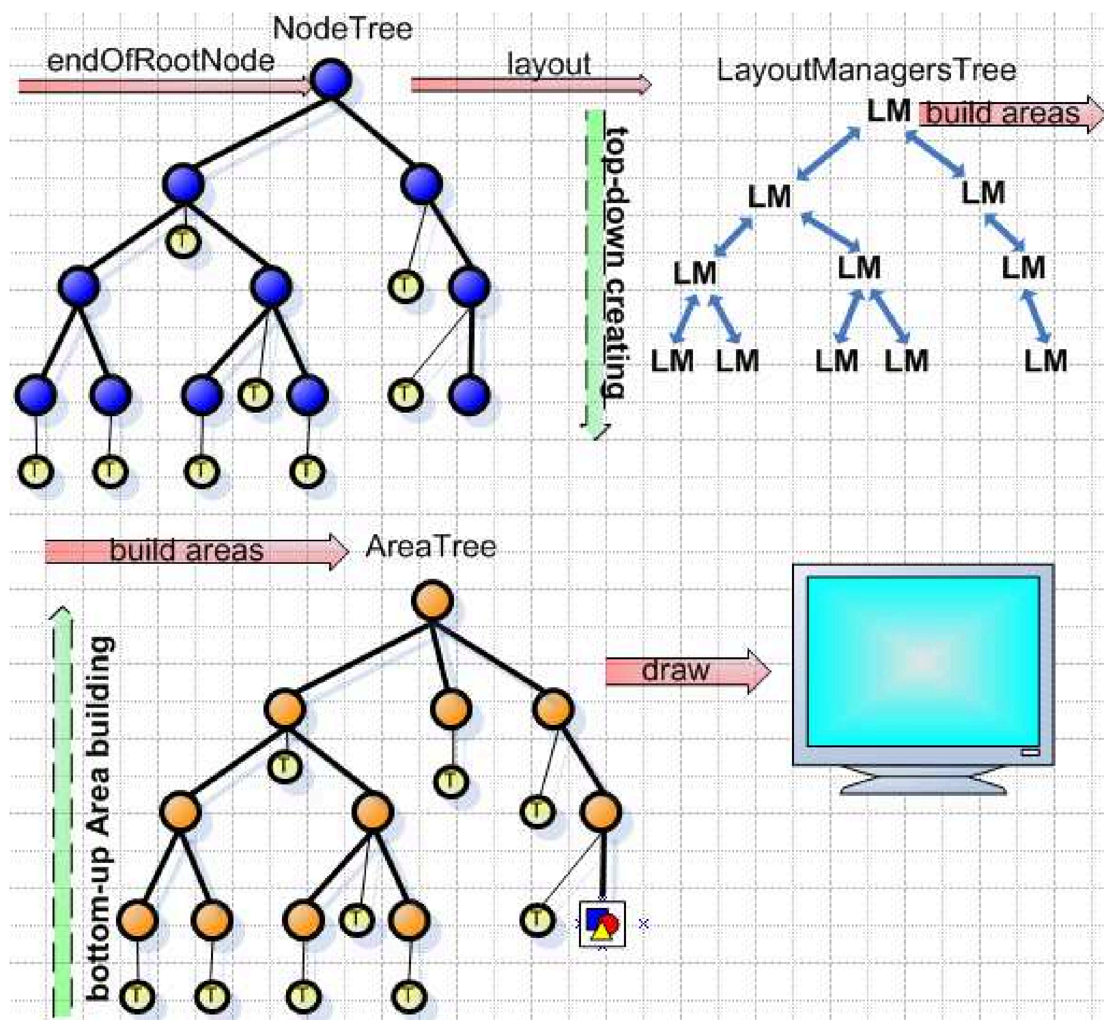
ých rendereroch [36] ide o rozmiestňovanie a vytváranie grafických elementov na obrazovku na základe modelu dát. Keďže nechceme framework zviazať s konkrétnou grafickou knižnicou, budeme pracovať s oblasťami. **Oblasť** (Area) je objekt, ktorý obsahuje všetky informácie o tom, ako má vyzerať a kde má byť zobrazený. Tieto informácie nazývame črty kvôli odlíšeniu od termínu vlastnosť, ktorý používame pre vlastnosti v uzloch stromu NodeTree. Oblasť je abstraktný grafický element, na základe ktorého je pri použití nášho frameworku ľahké vytvoriť grafický element v konkrétnej grafickej knižnici. V našom frameworku pod **rozmiestňovaním** rozumieme proces vytvárania oblastí, určovania ich veľkostí a pozície v 2D priestore na základe modelu dát, ktorým je strom NodeTree. Pri renderovaní mnohých dokumentov je model komplikovaný, a preto je medzi grafickými elementmi veľa závislostí a previazaní. Preto sa na rozmiestňovanie používajú pomocné triedy, **rozmiestňovací manažéri** (layout managers). Rozmiestňovací manažér z objektu modelu vytvára a umiestňuje oblasť alebo aj viac oblastí. Ak sa objekt modelu skladá z viacerých častí, môže rozmiestňovací manažér použiť na ich rozmiestnenie ďalších rozmiestňovacích manažérov. Takto pri procese rozmiestňovania vzniká strom rozmiestňovacích manažérov, ktorý z modelu vstupných dát vytvára strom oblastí. V našom frameworku sa zo stromu NodeTree týmto procesom vytvára strom oblastí, ktorý nazývame **AreaTree**. V ďalších podkapitolách sa budeme zaoberať jednotlivými špecifikami a detailamy tohto procesu.

3.7.2 Statické rozmiestňovanie

V tejto kapitole popíšeme návrh najjednoduchšieho typu rozmiestňovanie kedy je najskôr vytvorený strom NodeTree a potom sa pomocou rozmiestňovacích manažérov vytvorí strom AreaTree. Na konci tejto časti rozoberieme výhody a nevýhody tohto návrhu.

Nami navrhnutý proces statického rozmiestňovania sa začína hneď po tom, čo SAXToNodeTreeBuilder alebo DOMToNodeTreeBuilder zavolá metódu `endOfNode` na koreni stromu NodeTree. Ten zavolá metódu `endOfDocument` triedy `EventHandler`. Ku koreňu stromu NodeTree sa nájde rozmiestňovací manažér, ten sa inicializuje na základe údajov z koreňa a vytvorí oblasť, ktorá bude neskôr došpecifikovaná, potom získa inštancie detí koreňa a na ich základe vytvorí a rozmiestni podoblasti tejto oblasti a nakoniec došpecifikuje všetky jej vlastnosti a môže vykonať ešte optimalizáciu rozmiestnenia. Pri tvorbe podoblastí sa väčšinou tiež využívajú rozmiestňovací manažéri, ale pri jednoduchých uzloch NodeTree to nemusí byť vždy nutné. Vidíme, že strom rozmiestňovacích manažérov sa tvorí od koreňa dole, teda top-down. Mnohé rozmiestňovacie algoritmy postupujú tak, že po vytvorení oblastí špecifikujú veľkosť a umiestnenie najspodnejších, najelementárnejších oblastí (listy stromu AreaTree), ktorých rozmery sú pevne dané, napríklad veľkosť písmena alebo veľkosť obrázku a na ich základe potom budujú oblasti, ktoré sú z nich tvorené. Oblasti sa teda dokončujú zdola hore (bottom-up). V našom návrhu podporujeme rozmiestňovanie za pomoci stromu rozmiestňovacích manažérov, ktorý sa buduje zhora dole (top-down), pri čom sa vytvárajú oblasti, ktoré môžu byť dokončované zdola hore (bottom-up) aj zhora dole.

[36] Pod klasickými redermi máme na mysli tie, ktorých úlohou je len vykresliť model dát pomocou



Obrázok 18: Statický layout

V našom návrhu zodpovedá za vytvorenie rozmiestňovacieho manažéra, ktorý rozmiestni podstrom NodeTree, koreň tohto podstromu. K modularite a rozšíriteľnosti nášho návrhu prispieva aj to, že rozmiestňovací manažéri sú uložení v mapovaní `LayoutManagerMapping`, kde kľúčom je jednoznačný identifikátor manažéra spomedzi všetkých manažérov, založený napríklad na lokálnom mene uzla a hodnotou je inštancia triedy `LayoutManagerMaker`, ktorá je statickou podtriedou triedy `LayoutManager`, ktorú vytvára. Toto mapovanie je určené pre jeden menný priestor a je registrované v singeltone `LayoutManagerMappingRegistry`, ktorý mapuje URI menného priestoru na mapovanie `LayoutManagerMapping`.

Za inicializáciu rozmiestňovacieho manažéra zodpovedá rodičovský rozmiestňovací manažér, ten spúšťa aj rozmiestňovací algoritmus pre príslušný podstrom NodeTree.

grafickej knižnice na obrazovky.

Náš framework je pripravený na zmenu v špecifikácii XML jazyka, ktorý vykresľuje. Napríklad ak v jazyku pribudne nový element, tak v rendereri pribudne k nemu uzol v strome NodeTree a rozmiestňovací manažér v mapovaní. Tak isto zmena rozmiestňovacieho algoritmu je ľahká, stačí nahradiť rozmiestňovacích manažérov v mapovaní inými, čím sme oddelili proces rozmiestňovania od iných procesov v rendereri.

Oblasť v našom návrhu neslúži čisto len na prenos grafických informácii do kontrolera z MVC (3.8), ale má aj funkcionalitu akou je napríklad dirty rendering, zahadzovanie podstromov AreaTree a dynamické renderovanie a táto funkcionalita môže byť rôzna pre rôzne XML jazyky. Rozhodli sa preto, že okrem množiny základných oblastí, ktoré by poskytovali väčšinu potrebnej funkcionality vytvoríme aj mapovanie `AreaMapping`, kde kľúč je jednoznačný identifikátor oblasti založený na mene uzlu stromu NodeTree a hodnota je inštancia triedy `AreaMaker`, ktorá vytvorí oblasť. Toto mapovanie je špecifické pre jeden menný priestor a je zaregistrované do singeltonu `AreaMappingRegistry` na základe URI menného priestoru. Takto je možné pre rôzne XML jazyky vytvárať rôzne oblasti s rôznou funkcionalitou a vytvárať XML dokumenty kombinujúce tieto XML jazyky.

Po skončení práce najvrchnejšieho rozmiestňovacieho manažéra je vytvorený celý strom AreaTree, ktorý dostane príkaz na vykreslenie sa na obrazovku a tento príkaz je preposlaný do kontrolera z MVC (3.8).

Výhodou statického rozmiestňovania je jeho jednoduchá implementácia. Statické rozmiestňovanie je vhodné použiť na renderovanie jazykov, z ktorých zvyčajne nie sú zostrojené veľké dokumenty (MathML) a tiež na renderovanie jazykov, v ktorých jednotlivé časti dokumentu sú príliš prepojené a je ťažké ich rozmiestniť nezávisle. Hlavnou nevýhodou tohto procesu je, že nepodporuje vykresľovanie objektov (napríklad strán) mimo poradia v akom sa nachádzajú v dokumente a že na to, aby bol vykreslený nejaký objekt, sa musí čakať kým je celý NodeTree vytvorený. Ďalšou nevýhodou tohto procesu je, že nie je vhodný pre šetrenie pamäte. Podstromy NodeTree môžu byť počas jeho vytvárania zahadzované, ale po tom čo je strom NodeTree vybudovaný budú musieť byť opäť nahrané kvôli dátam potrebným pre rozmiestnenie.

3.7.3 Dynamické rozmiestňovanie a vykresľovanie

V tejto časti navrhneme tú časť nášho frameworku, ktorá podporuje dynamické rozmiestňovanie a popíšeme jeho výhody a nevýhody.

Pri niekoľko sto stranových dokumentoch môže proces statického rozmiestňovania trvať veľmi dlho. Pri stranovo orientovaných dokumentoch vzniká tak požiadavka, aby bola prvá alebo naposledy upravovaná strana zobrazená ako prvá.

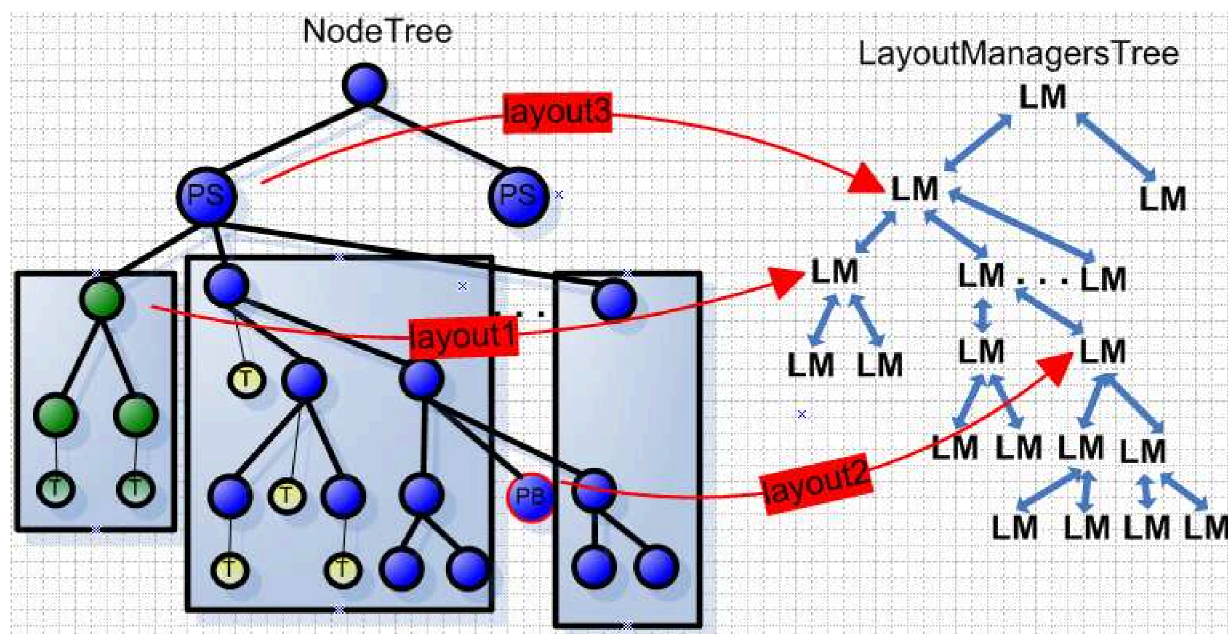
V statickom rozmiestňovaní sa spustí proces rozmiestňovania po tom, čo je zavolaná metóda `endOfDocument` v triede, ktorá dedí od abstraktnej triedy `EventHandler` a je zaregistrovaná v `EventHandlerRegistry`. Toto je predvolené správanie frameworku, framework však podporuje spustenie rozmiestňovania v koreni ľubovoľného podstromu NodeTree a rozmiestni tak oblasti, ktoré k nemu prislúchajú. Toto je možné urobiť pre každý podstrom NodeTree len teoreticky, prakticky je často rozmiestnenie tohto podstromu závislé na rozmiestnení jeho susedných podstromov a algoritmus rodičovského rozmiestňovacieho

manažéra hľadá optimálne rozmiestnenie napríklad aj na základe zoznamu miest, kde môže dôjsť k zalomeniu (break posibility), ktorý poskytne každý podstrom. Sú však aj situácie kedy rozmiestnenie susedných podstromov nevplýva na rozmiestnenie nejakého podstromu. Uvedieme si teraz 3 príklady.

V NodeTree sa môžu nachádzať podstromy v rôznych menných priestoroch predstavujúce použitie rôznych XML jazykov, napríklad použitie MathML na vyjadrenie matematiky. Rozmiestnenie matematického vzorca môže byť proces neovplyvniteľný inými časťami dokumentu.

Druhým príkladom je rozdelenie XSL-FO dokumentu do sekvencii strán, kde každá sekvencia môže mať nastavené iné pravidlá, napríklad okrajov alebo číslovania strán. Podstatné je, že elementy, ktoré sú v druhej sekvencii strán sú určite na inej strane ako elementy z prvej sekvencie strán a teda na seba nevplývajú, a preto môže byť každá sekvencia strán rozmiestňovaná samostatne.

Tretím príkladom je vynútené formátovanie XML jazykom. Napríklad element `</xsl-fo:newpage>` hovorí o zalomení strany (page break), a teda elementy pred týmto elementom nemajú vplyv na elementy za týmto elementom a naopak v rámci jednej sekvencie strán, a preto sa môžu rozmiestniť nezávisle.



Obrázok 19: Dynamický layout

Na obrázku 19 sú zobrazené všetky 3 prípady, pričom zelené uzly sú v inom mennom priestore ako zvyšok stromu NodeTree, uzly označené písmenami PS predstavujú element `PageSequence` a uzol označený písmenami PB predstavuje element `</xsl-fo:newpage>` a spôsobí zalomenie strany (page break). Šípky layout1, layout2 a layout3 pri inorderovom vytváraní stromu NodeTree hovoria o poradí v akom sa budú rozmiestňovať jednotlivé časti stromu NodeTree, tieto časti sú vymedzené obdĺžnikmi. Layout1 zodpovedá prvému

príkladu, layout2 tretiemu a layout3 druhému.

Tomu, že vytváranie a rozmiestňovanie oblastí sa deje počas vytvárania stromu NodeTree budeme hovoriť **dynamické rozmiestňovanie**.

Nezávisle od dynamického rozmiestňovania sa môže diať **dynamické renderovanie**. Ide o rovnaký princíp, len v strome AreaTree, teda ľubovoľná oblasť môže vyslať signál do kontrolera z MVC na vykreslenie časti stromu AreaTree. Vytváranie a zobrazovanie grafických entít prislúchajúcich oblastiam stromu AreaTree sa môže diať už počas jeho vytvárania.

Výhodou nášho návrhu dynamického rozmiestňovania a dynamického renderovania je, že sa nemusia v operačnej pamäti držať niektoré časti interných štruktúr NodeTree a AreaTree a je teda vytvorený priestor pre ich zahadzovanie. Ďalšou výhodou je možnosť vykresľovať prvé grafické objekty omnoho skôr ako pri statickom rozmiestňovaní a to ešte počas vytvárania stromu NodeTree. Výhodou nášho riešenia je aj možnosť vykresľovať grafické objekty mimo poradia, v ako sa nachádzajú v dokumente, čím je umožnené vykresliť ako prvú napríklad naposledy editovanú stranu.

Nevýhodou tohto návrhu oproti statickému rozmiestňovaniu je jeho ťažšia implementácia. Náročný je najmä algoritmus rozmiestňovania.

3.7.4 Aktualizácia rozmiestnenia

V tejto časti navrhnem tri spôsoby aktualizácie rozmiestnenia, ktoré závisia od detailnosti udalosti hovoriacej o tom, že strom NodeTree sa zmenil a od rozmiestňovacieho algoritmu.

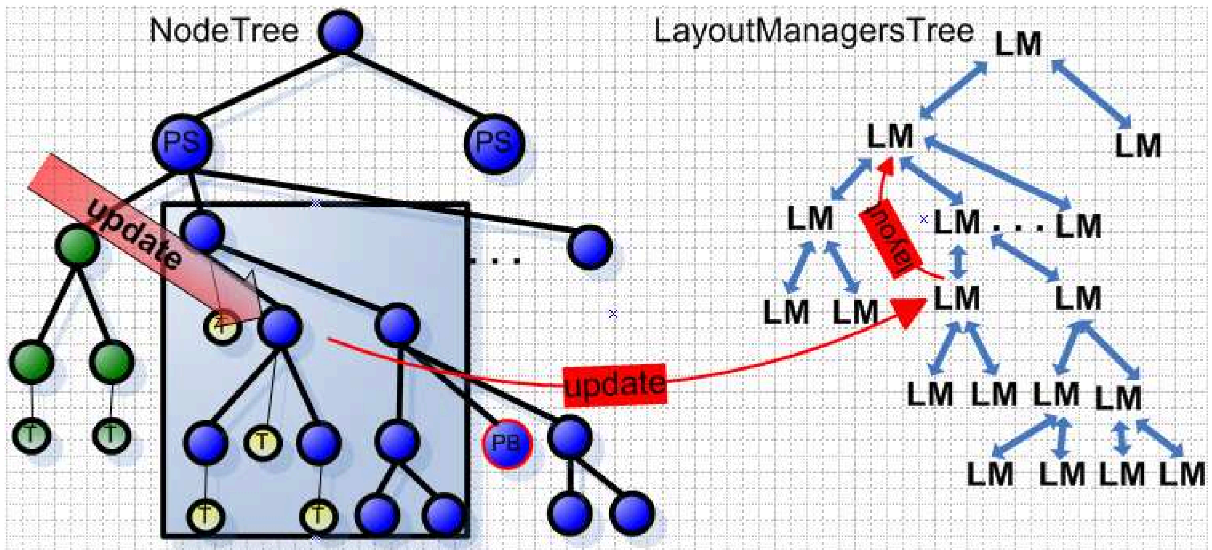
Aktualizácia dokumentu sa vykonáva na základe jeho zmeny. Táto zmena môže pochádzať z dvoch zdrojov. Prvým je užívateľ WYSIWYG XML editora, kde jeho interakcia na pohľade (view) z MVC (3.8) sa zachytí a odovzdá do kotroléra z MVC, ten ju spracuje a prepošle do modelu, ktorým je AreaTree, odtiaľto sa požiadavka na zmenu prenesie do NodeTree a odtiaľ do FATS, ten ju vykoná a aktualizuje NodeTree. Druhý externý činiteľ, napríklad databáza, z ktorej sa generuje dokument obsahujúci jej dáta a na základe zmien týchto dát sa zmení XML dokument, táto zmena je preposlaná do NodeTree ako jeho aktualizácia. Dôležité je, že pri oboch typoch zmien je príslušne aktualizovaný strom NodeTree a na základe tejto aktualizácie navrhneme aktualizáciu rozmiestnenia oblastí.

Keďže rozmiestňovacie algoritmy pracujú rôzne a aj typov aktualizácie je viacero, rozoberme si opäť 3 prípady.

- Všeobecný prípad

  Ak aktualizácia uzla stromu NodeTree nie je bližšie špecifikovaná, iba vieme, že sa buď on sám alebo niektoré z jeho detí zmenili alebo rozmiestňovací algoritmus nie je navrhnutý na aktualizáciu časti modelu, nezostáva nič iné ako urobiť rozmiestnenie najmenšej časti NodeTree, ktorá nie je ovplyvnená ostatnými časťami NodeTree. V najhoršom prípade ide o rozmiestnenie celého stromu NodeTree. V tomto prípade sa vytvárajú všetky oblasti v danom podstrome nanovo.

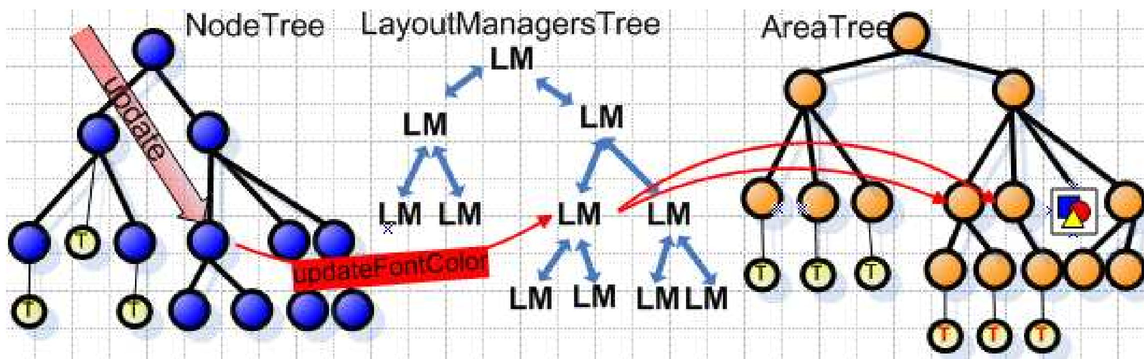
  Na obrázku 20 vidíme ako uzol, ktorý bol aktualizovaný, preposlal túto informáciu jeho rozmiestňovaciemu manažérovi a ten ju preposlal do najbližšieho predka, ktorý



Obrázok 20: Dynamický layout

rozmiestňuje nezávisle od susedných rozmiestňovacích manažérov.

- Lokálne rozmiestňovanie



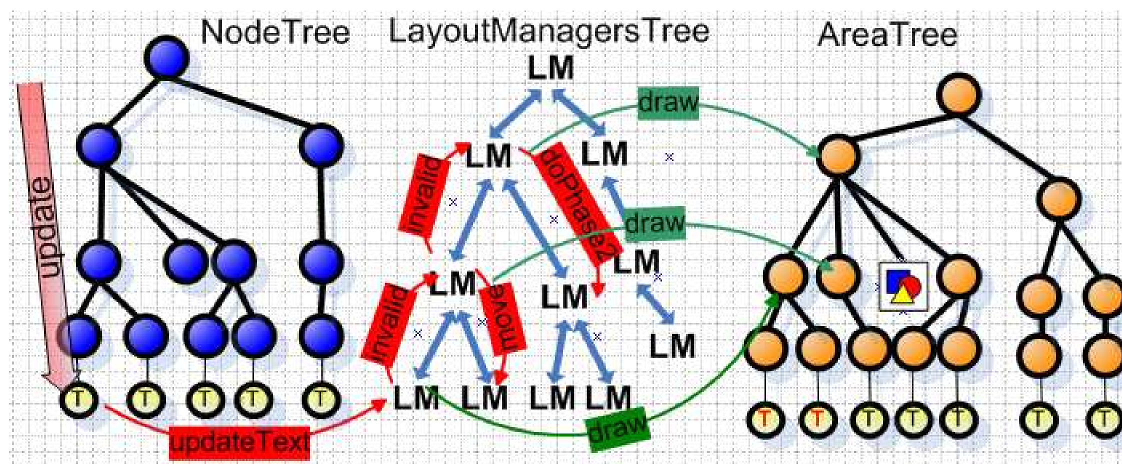
Obrázok 21: Lokálny layout

Ak uzol stromu NodeTree bližšie špecifikuje o akú zmenu ide a túto zmenu rozmiestňovací manažér vyhodnotí ako takú, ktorá nemá vplyv na rozmiestnenie iných oblastí ako tých, ktoré vytvára tento rozmiestňovací manažér a jeho detský rozmiestňovací manažéri, tak ide o lokálnu aktualizáciu. Ak nastala komplikovaná zmena (nahradenie podstromu NodeTree iným, zmena veľkosti písma alebo okrajov), tak sa dá predpokladať, že je nutné vytvoriť príslušný podstrom AreaTree nanovo a tiež sa dá predpokladať, že sa tým zmenia rozmery najvrchnejšej oblasti, ktorú vytvoril rozmiestňovací manažér tohto uzla, a teda je nutné upraviť rozmiestnenie susedných oblastí, a teda sa nedá urobiť lokálna aktualizácia.

Ak je však zmena špecifikovaná uzlom stromu NodeTree jednoduchá (dopísanie písmena, pričom sa neprekročí niektoré ohraničenie oblasti, zmena farby písma, pozadia písma, pozadia strany alebo podčiarknutie písma a podobne), tak sa dá spraviť lokálna aktualizácia.

- Inteligentné rozmiestňovanie

  Tak ako v druhom prípade aj tu predpokladáme, že uzol stromu NodeTree špecifikoval zmenu, ktorá v ňom nastala a tak isto požadujeme isté nároky na rozmiestňovací algoritmus, ktoré si teraz opíšeme.



Obrázok 22: Inteligentný layout

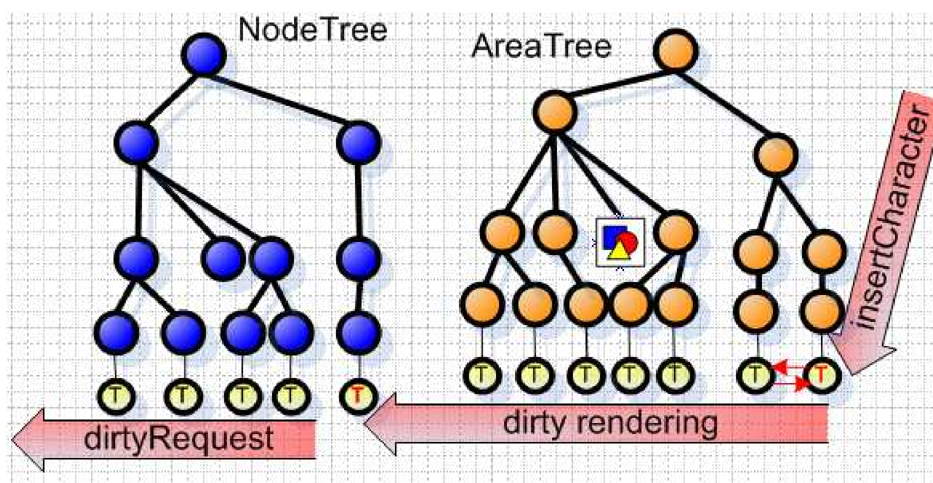
  Schéma fungovania algoritmu je nasledovná. Uzol stromu NodeTree prepošle zmenu do jeho rozmiestňovacieho manažéra a ten sa pokúsi urobiť lokálne rozmiestňovanie. Ak však zistí, že ním vytvorená oblasť sa príliš zväčšila alebo príliš zmenšila, pošle rodičovskému rozmiestňovaciemu manažéru signál `invalid`. Ten na základe už vypočítaných informácii z detských rozmiestňovacích manažérov (napríklad miest zalomenia alebo rozmerov oblastí) urobí už len niektoré fázy rozmiestnenia alebo zaktualizuje existujúce rozmiestnenie (napríklad tak, že poposúva niektoré oblasti) alebo vykoná rozmiestnenie jemu prislúchajúcemu podstromu NodeTree nanovo. Nakoniec ak sa ním vytvorená oblasť príliš zväčší alebo zmenší, tak pošle signál invalid rodičovskému rozmiestňovaciemu manažérovi a takto sa pokračuje smerom hore. Na konci tohto algoritmu sa zmeneným oblastiam pošle signál na prekreslenie. Signál na prekreslenie sa môže posielať aj počas tohto procesu do zmenených častí AreaTree, ktoré sa už nebudú meniť.

Pri aktualizácii rozmiestnenia sa v našom návrhu snažíme použiť čo najviac už vypočítaných informácii. Tieto informácie sú uložené v rozmiestňovacích manažéroch, preto by mal každý uzol stromu NodeTree pamätať referenciu na svojho rozmiestňovacieho manažéra. Ak rozmiestňovací manažér dostane signál na aktualizáciu rozmiestnenia, tak pri

tom môže využívať informácie z detských manažérov, ale ak aktualizácia uzla NodeTree je pridanie podstromu NodeTree alebo vymazanie alebo nahradenie iným, tak ak by rozmiestňovací manažér vychádzal z informácii z detských manažérov, potom by nezistil, že nejaké uzly pribudli, zmizli alebo boli nahradené, preto rozmiestňovací manažér získava v našom návrhu detských manažérov cez deti koreňa stromu NodeTree, ktorý rozmiestňuje. Tu je dobre využiteľná technika `lazy loading` [37] , teda rozmiestňovací manažér ak ešte nebol vytvorený, tak sa vytvorí až keď o jeho inštanciu požiada rodičovský rozmiestňovací manažér. Napríklad, keď je do stromu pridaný koreň iného podstromu NodeTree, tak rozmiestňovací manažér pre pridaný koreň sa vytvorí až keď sa k nemu pokúsi rodičovský manažér pristúpiť.

3.7.5 Dirty layout a dirty rendering

Dirty editácia je editácia XML dokumentu, ktorá nevyžaduje po jeho zmene túto zmenu zosynchronizovať s ďalšími fázami jeho spracovania. Jednotlivé objekty v týchto fázach odhadnú ako bude zmenený dokument vyzerať a samy sa podľa toho zmenia. Presnejšie ide o editáciu na základe dirty editačnej požiadavky. Spôsob používania editačných požiadaviek v našom frameworku sme popísali v časti 3.6.2. V tejto časti navrhneme podporu techník dirty rendering a dirty layout.



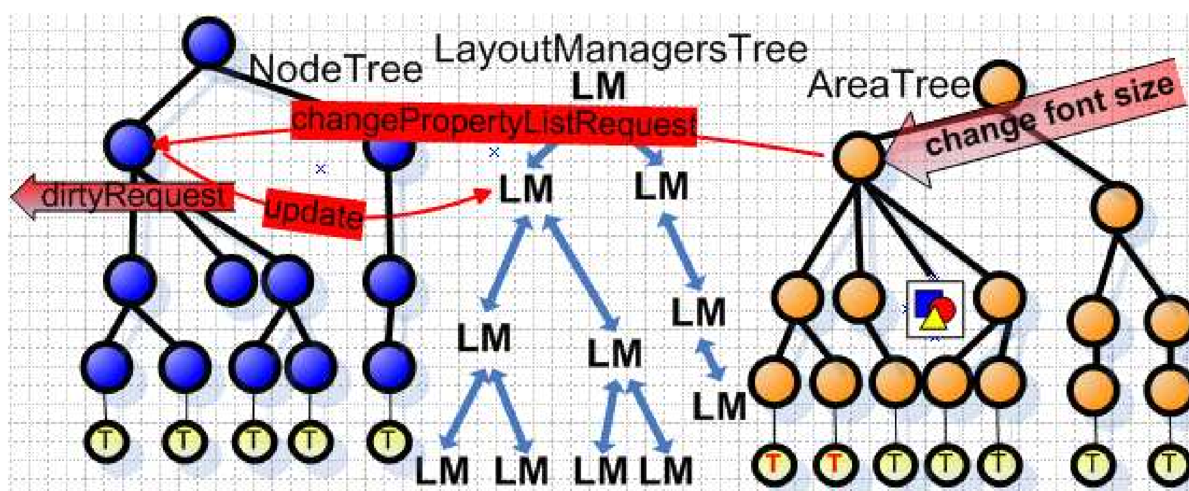
Obrázok 23: Dirty rendering

Pozrime sa najskôr na techniku dirty rendering. Ide o situáciu keď kontroler z MVC (3.8) prepošle editačnú požiadavku vytvorenú na základe užívateľovej interakcie do oblasti v AreaTree, ak je táto požiadavka jednoduchá natoľko, že nie je nutné robiť rozmiestnenie a vieme, že ak sa požiadavka vykoná v XML dokumente, bude tento validný a nevznikne neočakávaná zmena. Takou neočakávanou zmenou môže byť napríklad situácia, keď sa zmení text elementu `<chapter>`, tak sa zmení element predstavujúci položku v obsahu, ale

[37] Lazy loading je častý návrhový vzor, používaný na oddialenie inicializácie objektu až kým nie je potrebný

môže ísť aj o omnoho zákernejšie pravidlá, napríklad ak je hodnota atribútu `font-color` nejakého elementu nastavená na reťazec `"red"`, tak sa zmenia všetky textové dáta v XML dokumente na reťazec `"neočakávaná zmena"`. Vo väčšine používaných XML jazykov však nie je veľa elementov, ktorých zmena spôsobí neočakávanú zmenu XML dokumentu. Informácia o tom, že zmena elementu môže spôsobiť neočakávanú zmenu, môže byť prenesená do objektov z ďalších fáz spracovania XML dokumentu a tak sa ľahko zistí kedy je dirty editácia možná. V niektorých prípadoch sa dá neočakávaná zmena odhaliť, napríklad už spomínaný element obsahu vznikol duplikáciou elementu `<chapter>`, a teda identifikátor uzlov stromu NodeTree je založený na tom istom reťazci.

Dirty rendering sa dá použiť napríklad pri zmene farby písma, pozadia alebo iných grafických elementov alebo pri dopísaní alebo vymazaní písmena, ak takto zmenená oblasť ostane v medziach definovaných pomocou minimálnych, optimálnych a maximálnych rozmerov a podobne.



Obrázok 24: Dirty layout

Pozrime sa teraz na dirty layout. Je to podobná situácia ako pri dirty renderingu. Editačná požiadavka, ktorá pochádza z kontrolera z MVC je preposlaná do AreaTree, môže ale dôjsť k zmene rozmiestnenia, táto požiadavka je preposlaná do príslušného uzla stromu NodeTree. Keď vieme ako bude po zmene XML dokumentu vyzerať strom NodeTree, teda vieme, ktorý uzol NodeTree sa zmení a ako, tak túto požiadavku prepošleme do FATS s príznakom dirty a vykonáme príslušnú predvídanú zmenu v strome NodeTree a na jej základe sa vykoná aktualizácia stromu AreaTree a na jej základe sa vytvoria alebo zaktualizujú grafické elementy. To či uzol stromu NodeTree prepošle požiadavku do FATS ako dirty alebo nie môže byť určené už v požiadavke, ktorú vytvorila oblasť alebo o tom rozhodne samotný uzol. Dirty layout sa dá využiť napríklad pri zmene väčšej časti textu, pri zmene veľkosti písma, pri zmene okrajov strán a podobne.

3.7.6 Napojenie modulu LayoutManager na modul NodeTree

V tejto časti navrhneme spôsob napojenia modulu LayoutManager na modul NodeTree. Výhodou nášho návrhu je, že umožňuje zahadzovanie podstromov NodeTree, pričom sa zachovávajú niektoré informácie o rozmiestnení a zároveň sa z pamäte uvoľňujú inštancie rozmiestňovacích manažérov.

Už sme spomínali, že náš návrh časti frameworku, zodpovedný za rozmiestňovanie, podoruje pamätanie si čo najviac informácii o existujúcom rozmiestnení a že tieto informácie sa uchovávajú v inštanciách rozmiestňovacích manažérov (3.7.4). Tak isto sme spomínali, že uzol stromu NodeTree si pamätá inštanciu rozmiestňovacieho manažéra, ktorý vytvorí a rozmiestni oblasti, ktoré prislúchajú podstromu tohto uzla.

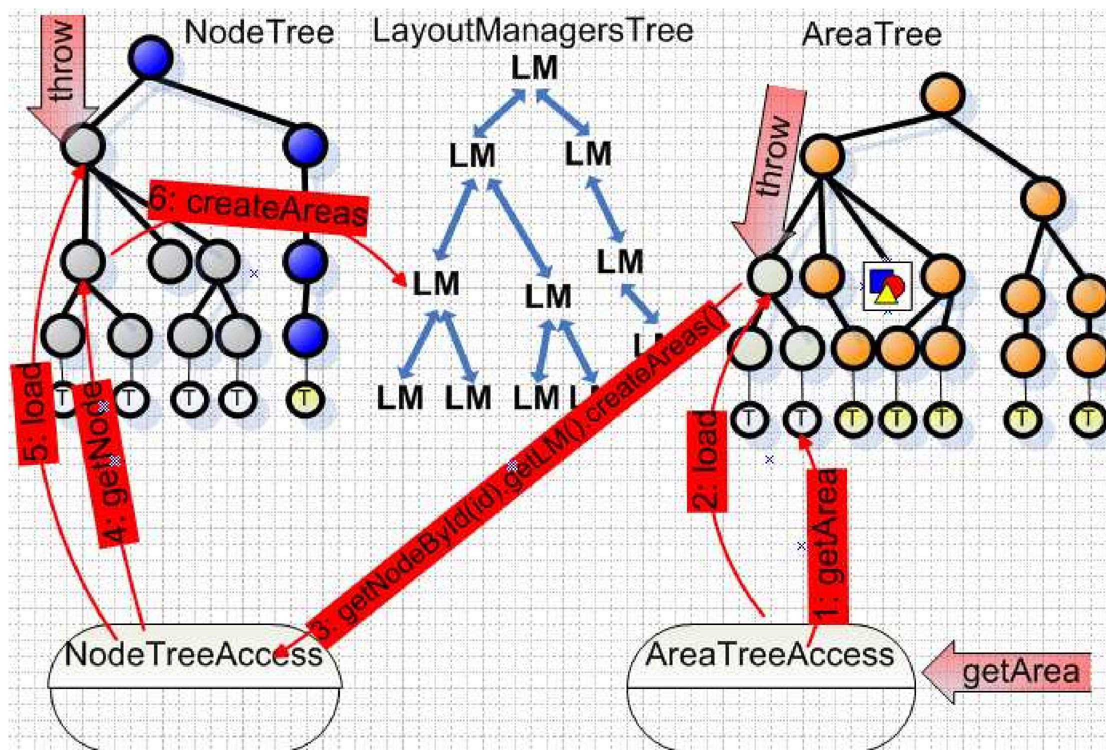
Rozmiestňovací manažér potrebuje informácie o uzloch stromu NodeTree, ktoré rozmiestňuje. Ak však zahodíme podstrom stromu NodeTree a inštancie týchto uzlov by ostali v ich rozmiestňovacích manažéroch, nedosiahli by sme požadovaný efekt, ktorým je uvoľnenie pamäte. Situácia by sa vyriešila, ak by sme odstránili aj týchto rozmiestňovacích manažérov, tým by sme ale stratili informácie o existujúcom rozmiestnení. Preto rozmiestňovací manažéri pristupujú k uzlom stromu NodeTree nepriamo cez singelton `NodeTreeAccess`. Mohlo by sa zdať, že síce uvoľníme pamäť tým, že zahodíme podstrom NodeTree, ale v pamäti ostane ešte veľké množstvo rozmiestňovacích manažérov, to však nie je pravda, lebo ako sme už spomínali v časti 3.7.4 detský rozmiestňovací manažéri sa neukladajú v rodičovskom manažéri, ale sa získavajú cez inštnacie detských uzlov. Pri nahrávaní podstromu NodeTree sa vytvárajú nové inštancie uzlov a ich rozmiestňovací manažéri sú "prázdni". Teraz je na mieste otázka, kde sú teda informácie o rozmiestnení, keď sa podstrom NodeTree zahodí? Odpoveď je - v koreni zahodeného stromu. Jeho inštancia totiž existuje, len sa zahodia jeho dáta, napríklad deti a zoznam vlastností, ale referencia na jeho rozmiestňovacieho manažéra ostáva. Tento rozmiestňovací manažér si pamätá informácie o rozmiestnení aj keď jeho detský manažéri sú prázdni. Ak nastane situácia, že je potrebné prerátať dáta v detských manažéroch kvôli aktualizácii, tak sa vytvoria a inicializujú technikou lazy loading.

3.7.7 Šetrenie pamäte v module AreaTree

V tejto časti popíšem návrh šetrenia pamäte v module AreaTree pomocou zahadzovania a nahrávania jeho podstromov.

Tak ako v prípade NodeTree je aj v prípade AreaTree zbytočné pamätať si informácie o niekoľko sto stranových dokumentoch. Preto framework umožňuje zahadzovanie a nahrávanie podstromov stromu AreaTree a to nezávisle od zahadzovania a nahrávanie postromov NodeTree. Zahadzovanie a nahrávanie oblastí funguje veľmi podobne ako pri uzloch NodeTree. Opäť tu ide o nepriamy prístup k oblastiam cez singelton `AreaTreeAccess`, ktorý zabezpečuje transparenciu nahratia, a teda rozmiestňovací manažéri nemajú referencie na vytvorené oblasti, ale len ich identifikátory. Rovnako pri zahadzovaní podstromu AreaTree sa nezahadzuje inštancia koreňa, len sa zahodia jeho dáta, napríklad deti alebo črty. Pri nahrávaní je postup nasledovný: nájde sa koreň zahodeného podstromu AreaTree, ten získa

inštanciu príslušného uzla z NodeTree z ktorého bol vytvorený, z tohto uzla získa inštanciu rozmiestňovacieho manažéra a zavolá jeho metódu `createAreas`. Ak je príslušný uzol stromu NodeTree zahodený, tak vďaka transparentnosti nahratia, ktorú `NodeTreeAccess` poskytuje nahrá zahodený podstrom, v ktorom sa uzol nachádza, a teda vrstva AreaTree sa tým nemusí zaoberať. Táto situácia je znázornená na obrázku 25.

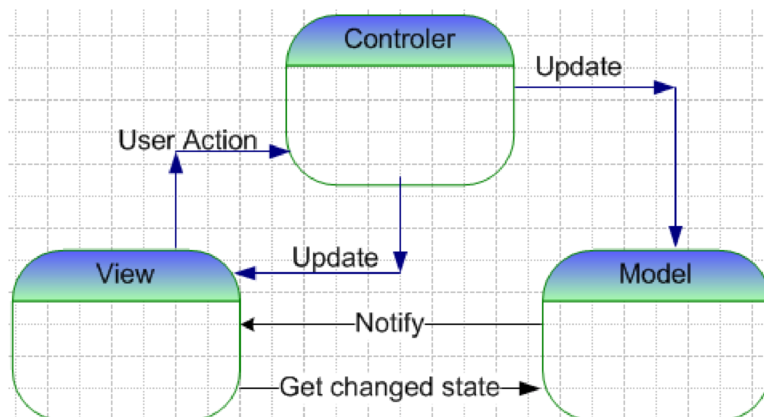


Obrázok 25: Zahadzovanie podstromov AreaTree

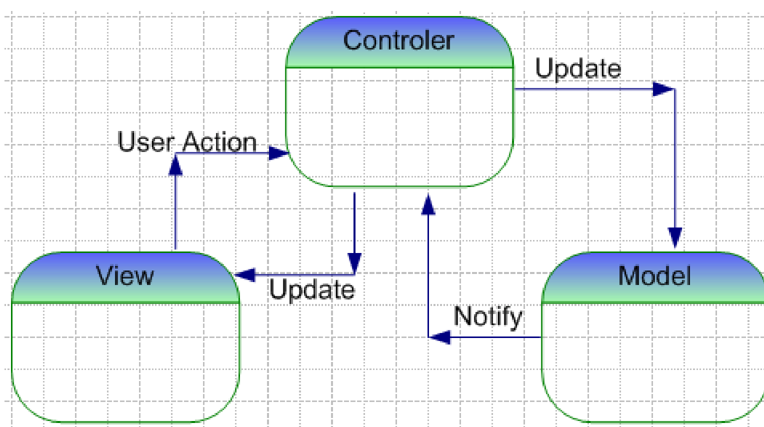
3.8 Renderovanie a začlenenie do MVC

Už sme spomínali, že framework nemá za úlohu zachytávať interakciu užívateľa na vykreslenom dokumente za účelom jeho zmeny, na toto existuje množstvo grafických knižníc a frameworkov, napríklad GEF. Tieto frameworky sú založené na architektonickom návrhovom vzore Model-View-Controler, ktorého schému vidíme na obrázkoch 26 a 27.

View z MVC predstavuje užívateľské rozhranie. Controler zachytáva a spracúva udalosti a požiadavky z Model a View a predstavuje biznis logiku. Model predstavuje dáta, nad ktorými sa pracuje. V našom prípade je modelom AreaTree alebo NodeTree podľa toho, či sa z pohľadu frameworku robí spoločné rozmiestnenie a vykresľovanie alebo oddelené. Náš framework sme navrhli tak, aby vedel spolupracovať s ľubovoľným grafickým frameworkom, ktorý je založený na vzore MVC. V tomto návrhu sa nachádza rozhranie potrebné na príjem požiadaviek na zmenu modelu z kontrolera a rozhranie na posielanie aktualizácie modelu do kontrolera alebo pohľadu. V prípade, že modelom je strom NodeTree, tak tieto



Obrázok 26: MVC s klasickým návrhom komunikácie



Obrázok 27: MVC s upraveným návrhom komunikácie

rozhrania už sú hotové, len pri zmene uzla NodeTree nebude ako listener [38] zaregistrovaný rozmiestňovací manažér, ale kontroler z MVC a požiadavky na zmenu nebude prijímať z AreaTree ale z kontrolera z MVC.

Treba ešte navrhnúť rozhrania pre prijímanie požiadaviek na zmenu AreaTree a požiadaviek na aktualizáciu view z MVC. Tieto rozhrania sme navrhli podobným spôsobom ako rozhrania modulu NodeTree. Náš návrh týchto rozhraní je dostatočne všeobecný a poskytuje všetky základné operácie na aktualizáciu stromu AreaTree a umožňuje aktualizovať pohľad (view) základným spôsobom. Odvodením špecifických tried od tried, ktoré predstavujú požiadaviek na zmenu AreaTree a na aktualizáciu pohľadu, sa dá dosiahnuť lepšia efektivita. Framework obsahuje rozhranie `IEditableArea`, ktoré slúži na príjem požiadaviek na zmenu z MVC a poskytuje metódy `changeTraitRequest`, `deleteSubtreeRequest`, `insertSubtreeRequest` a `replaceSubtreeRequest`. Framework

[38] **Listener** je návrhový vzor známy aj ako observer alebo publish/subscribe. Objekt listener je zaregistrovaný v objekte, ktorý posiela všetkým takto zaregistrovaným objektom udalosti hovoriace o zmene svojho stavu.

obsahuje sadu tried predstavujúcu požiadavky na aktualizáciu pohľadu (view) z MVC. Tieto požiadavky sme navrhli tak, že sa pomocou nich dajú vyjadriť všetky zmeny v strome AreaTree. Ide o nasledujúce triedy : `ViewUpdateRequest`, `ViewCompoundUpdateRequest`, `ViewTraitChangedRequest`, `ViewInsertAreaRequest`, `ViewInsertTreeRequest`, `ViewSubtreeDeletedRequest` a `ViewSubtreeReplacedRequest`.

Oblasť je akýmsi virtuálnym grafickým elementom, ktorý obsahuje všetky informácie o tom, ako má vyzerať a kde má byť umiestnený. Tieto informácie má uložené v objektoch, ktoré nazývame črty (traits). Na základe čŕt oblastí je jednoduché vytvoriť k nim grafické elementy v konkrétnej grafickej knižnici alebo frameworku. O toto vytvorenie sa stará kontroler z MVC.

4 Záver

Navrhli sme framework pre tvorbu rendererov vhodných pre WYSIWYG editovanie XML dokumentov. Tento framework je vďaka jeho všeobecnosti, rozšíriteľnosti a flexibilite vhodný aj na renderovanie akýchkoľvek iných dát. Obsahuje však mnoho funkcionality, ktorá je vhodná práve pre XML dáta a pre prenos požiadaviek a udalostí, ktoré sú potrebné pri WYSIWYG editácii. Framework nie je viazaný na konkrétny XML jazyk a podporuje aj proces WYSIWYG editácie XML dokumentu, ktorý XML jazyk najskôr transformuje pomocou XSLT transformácie a až potom ho poskytne rendereru na vykreslenie. Na trhu existuje málo XML jazykov, pre ktoré existujú kvalitné WYSIWYG editory s otvoreným kódom. Náš framework má otvorený kód, čo prispieva k jeho flexibilite a použiteľnosti pre tvorbu rendererov, ktoré môže byť súčasťou týchto editorov. Tento framework môže byť použiteľný ako časť projektu Euromath 2 [6], ktorého cieľom je vytvoriť platformu pre tvorbu WYSIWYG XML editorov.

Framework je vhodný pre malé dokumenty, pri ktorých sa kladie dôraz na rýchlosť, aj pre veľké dokumenty, kde sa kladie dôraz na šetrenie pamäte. Podporu efektívneho vykresľovania a aktualizácie XML dokumentu sme dosiahli podporou pre čiastočnú XSLT transformáciu, ktorá môže proces aktualizácie XML dokumentu veľmi urýchliť, ďalej tým, že sme vytvorili podporu pre spojené aj oddelené rozmiestňovanie a vykresľovanie. Navrhli sme podporu pre zahadzovanie objektov interných štruktúr, čím je možné výrazne znížiť pamäťové nároky. Zrýchlenie aktualizácie pohľadu na XML dokument sme dosiahli technikou dirty editing, dynamickým rozmiestňovaním a vykresľovaním a podporou pre čiastočnú aktualizáciu rozmiestnenia a vykreslenia. Náš framework sme navrhli dostatočne všeobecne o čom svedčí aj to, že je vhodný pre stranovo orientované dokumenty aj pre nie stranovo orientované dokumenty. Framework je navrhnutý tak, aby podporoval ľubovoľný rozmiestňovací algoritmus, ktorý buduje rozmiestňovacích manažérov zhora dole (top-down) a grafické elementy zdola hore (bottom-up) alebo zhora dole, pričom sme ako referenčný uvažovali Knuthov rozmiestňovací algoritmus [5]. Framework obsahuje podporu pre vytvorenie abstraktných grafických entít, čím je použiteľný s ľubovoľnou grafickou knižnicou a ľubovoľným grafickým frameworkom a zapadá do architektonického návrhového vzoru MVC. Sofistikovaným systémom mapovaní a ich registrovaním do triedy `UserAgent` sme dosiahli modulárnosť, flexibilitu a rozšíriteľnosť nášho frameworku a rendereru, ktorý je pomocou nášho frameworku vytvorený.

Ďalšia práca môže byť venovaná rozširovaniu podporných tried, najmä vlastností, oblastí a ich čŕt, čím sa ešte viac uľahčí práca tvorcu rendereru. Pri dnešnom vývoji procesorov a operačných systémov, ktorý sa uberá smerom paralelizácie procesov a multithreadingu je výhodné, keď aplikácia rozdelí záťaž do viacerých vlákien. Využitie viacerých vlákien vidíme hlavne pri dynamickom rozmiestňovaní, teda keď rozmiestnenie jednej časti dokumentu je nezávislé od ostatných častí a pri dynamickom vykresľovaní, keď jedna časť dokumentu môže byť vykresľovaná, kým ďalšia je ešte len rozmiestňovaná.

Zoznam skratiek

AWT (Abstract Window Toolkit) - Štandardne aplikačné rozhranie pre Java programátorov, ktoré poskytuje grafické užívateľské rozhrania (GUI)

DOM (Document Object Model) - Spôsob prístupu k XML dokumentu, každá entita v XML dokumente je reprezentovaná jedným uzlom v stromovej štruktúre.

DTD (Document Type Definition) - Značkovací jazyk, ktorý sa používa na automatickú kontrolu štruktúry XML alebo SGML dokumentu

FOP (Formating Objects Processor) - Prvý (nekomerčný) projekt určený na vizualizáciu jazyka XSL-FO, zameraný na PDF výstup, podporujúci však aj iné formáty.

GEF (Graphical Editing Framework) - Prostredie na tvorbu grafických editorov [7]

GVT (Graphics Vector Toolkit) - Sada objektov a nástrojov používaných v projekte Batik pre dvojdimenzionálnu grafiku vo formáte XML

ChemML(Chemical Markup Language)- XML jazyk na zápis chemických symbolov, vzorcov a formúl

JAR (Java Archive) - Súborový formát, ktorý je ZIP balíkom, ktorý obsahuje aspoň jeden súbor MANIFEST.MF, ktorý hovorí o tom ako bude tento archív použitý

MathML (Mathematical Markup Language) - XML jazyk na zápis matematických symbolov, vzorcov a formúl

MVC (Model-View-Controler) - Architektonický návrhový vzor oddeľujúci dáta, pohľad a biznis logiku

ODF (OpenDocument Format) - Súborový formát pre kancelárske dokumenty štandardizovaný konzorciom OASIS

PDF (Portable Document Format) - je súborový formát vyvinutý firmou Adobe na ukladanie dokumentov obsahujúcich text aj obrázky nezávisle od softwaru a hardwaru, na ktorom boli vytvorené.

PresentationML (Spreadsheet Markup Language) - Súborový formát pre ukladanie prezentácii

SAX (Simple API for XML) - pôsob prístupu k XML dokumentu, dokument je čítaný sériovo, pričom sú volané udalosti

SGML (Standard Generalized Markup Language) - Značkovací jazyk, na popis informácii. Tak isto je to aj ISO štandard je na ňom založený napríklad jazyk HTML

SpreadsheetML (Spreadsheet Markup Language) - Súborový formát pre ukladanie tabuľkových dát

SVG (Scalable Vector Graphics) - Jazyk pre dvojdimenzionálnu grafiku vo formáte XML

TeX - Sádzací systém vyvinutý Donaldom Knuthom na tvorbu vysoko kvalitných dokumentov

URI (Uniform Resource Identifier) - Reťazec slúžiaci na identifikovanie alebo pomenovanie zdroja. Jeho súčasťou sú URL (Uniform Resource Locator) a URN (Uniform Resource Name)

W3C (World Wide Web Consortium) - Konzorciu, ktoré vyvýja spolupracujúce technológie, špecifikácie, príručky, software a nástroje, ktoré vedú k zlepšovaniu internetu

WordprocessingML (Wordprocessing Markup Language) - Súborový formát pre ukladanie textových dokumentov, napríklad knihy, články alebo poznámky

WYSIWYG (What You See Is What You Get) - Spôsob editácie dokumentov, pri ktorom je verzia zobrazená na obrazovke vzhľadom totožná s výslednou verziou dokumentu

XML (eXtensible Markup Language) - Veľmi jednoduchý a flexibilný značkovací jazyk.

XML Schema - Jazyk vyvíjaný W3C konzorciom na popis štruktúry XML dokumentov, ktorý nemá nedostatky DTD

XPath (XML Path Language) - XML jazyk obsahujúci výrazy pomocou ktorých sa dajú adresovať časti XML dokumentu alebo sa pomocou nich dajú vypočítať hodnoty, ktoré závisia na rôznych častiach dokumentu

XSD (XML Schema Definition) - Inštancia XML Schémy

XSL (eXtensible StyleSheet Language) - Rodina jazykov, ktoré určujú ako majú byť dokumenty v XML formáte formátované alebo transformované. Patria sem jazyky XSLT, XSL-FO, XPath

XSL-FO (XSL Formatting Objects) - XML jazyk na formátovanie dokumentov, je časťou jazyka XSL

XSLT (XSL Transformations) - XML jazyk pre transformacie XML dokumentov. XSLT sa najčastejšie používa na konvertovanie dát medzi XML schémami a na konvertovanie XML dát do PDF alebo webových stránok

Referencie

[1] Apache XML Project; subproject Batik: http://xmlgraphics.apache.org/batik/

[2] Apache XML Project; subproject: Formatting Objects Processor: http://xmlgraphics.apache.org/batik/

[3] W3 Consortium: Extensible Stylesheet Language – Formatting Objects, version 1.0, http://www.w3.org/TR/xsl/

[4] Cooper, J. W.: The Design Patterns Java Companion, Addison-Wesley, 1998

[5] Donald E. Knuth: Digital Typography, Stanford, California: Center for the Study of Language and Information, 1999

[6] FMFI UK: EuroMath2 XML Editor Project, http://euromath2.sourceforge.net/

[7] The Eclipse Foundation: Graphical Editing Framework, http://www.eclipse.org/gef

[8] JEuclid: http://sourceforge.net/projects/jeuclid

[9] W3C Consortium: Mathematical Markup Language, version 2.0 (Second Edition), http://www.w3.org/TR/2003/REC-MathML2-20031021/

[10] OpenXML Developer Group: http://openxmldeveloper.org

[11] Ecma International: Office Open XML Formats, http://www.ecma-international.org/memento/TC45.htm

[12] Ecma International: Ecma Office Open XML File Formats standard

[13] Brian Jones: Office Open XML Formats, http://blogs.msdn.com/brian_jones/default.aspx

[14] International Organization for Standardization: http://www.iso.org/iso/en/commcentre/pressreleases/2006/Ref1004.html

[15] OASIS: OpenDocument v1.1 specification, http://www.oasis-open.org/committees/download.php/12572/OpenDocument-v1.0-os.pdf

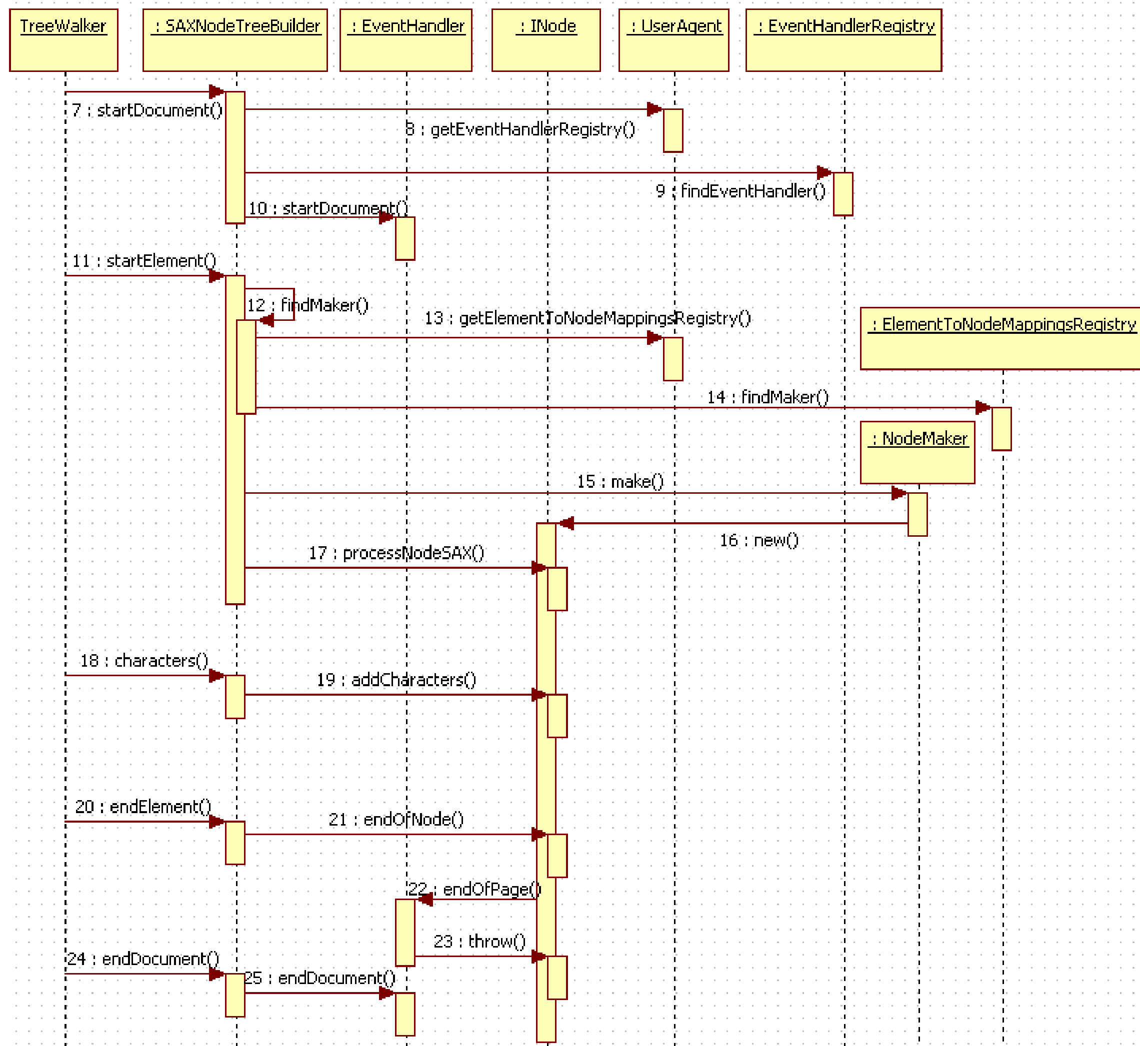
[16] David A. Wheeler: Why OpenDocument Won (and Microsoft Office Open XML Didn't), September 2006, http://www.dwheeler.com/essays/why-opendocument-won.html

[17] Tibor Vyletel: WYSIWYG editácia XML dát, Fakulta matematiky, fyziky a informatiky, Univerzita Komenského, Bratislava, 2004

[18] Tomáš Studva: WYSIWYG editovacie techniky XML dokumentov s použitím XSLT , Fakulta matematiky, fyziky a informatiky, Univerzita Komenského, Bratislava, 2007

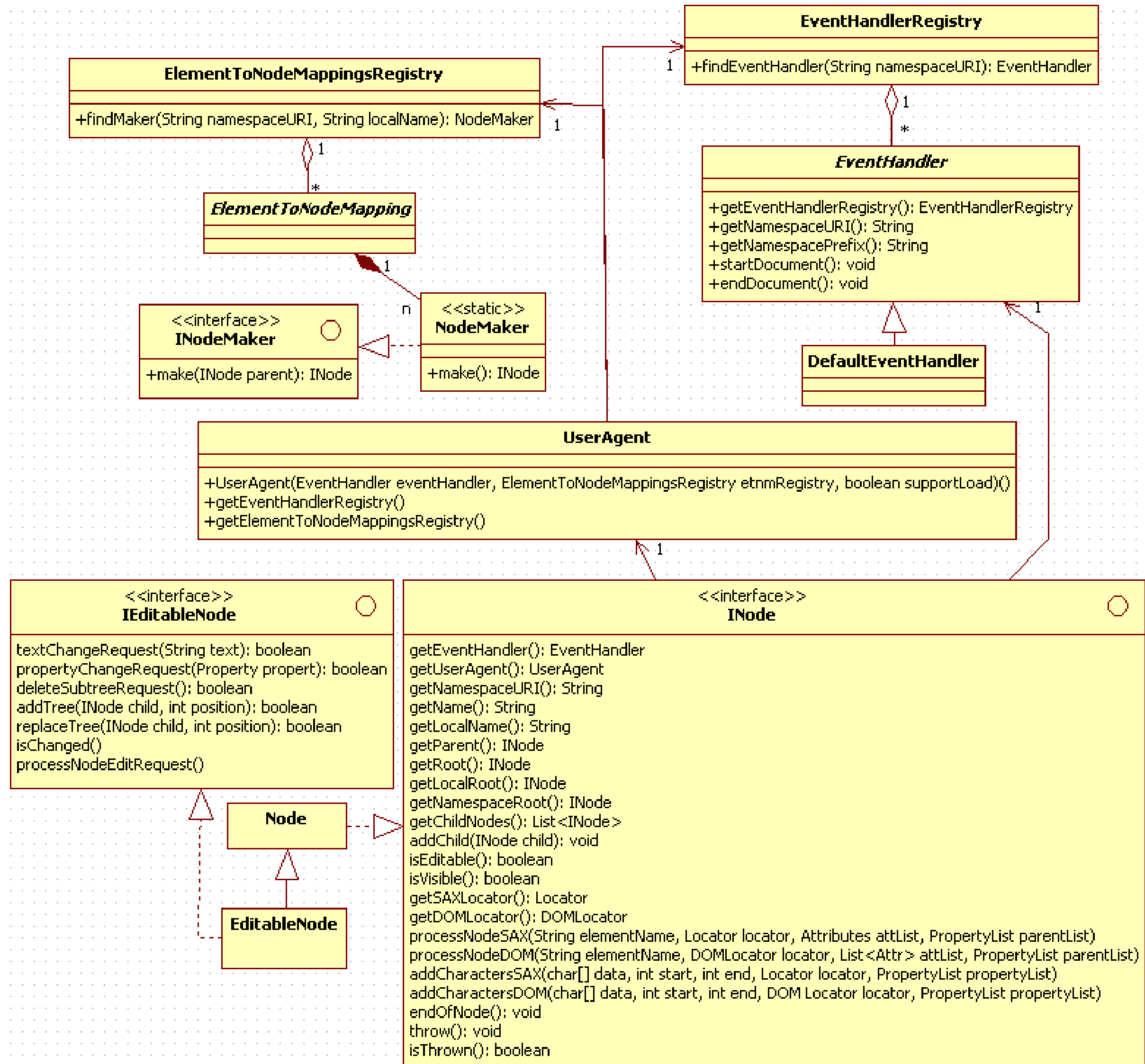
Prílohy

Príloha 1

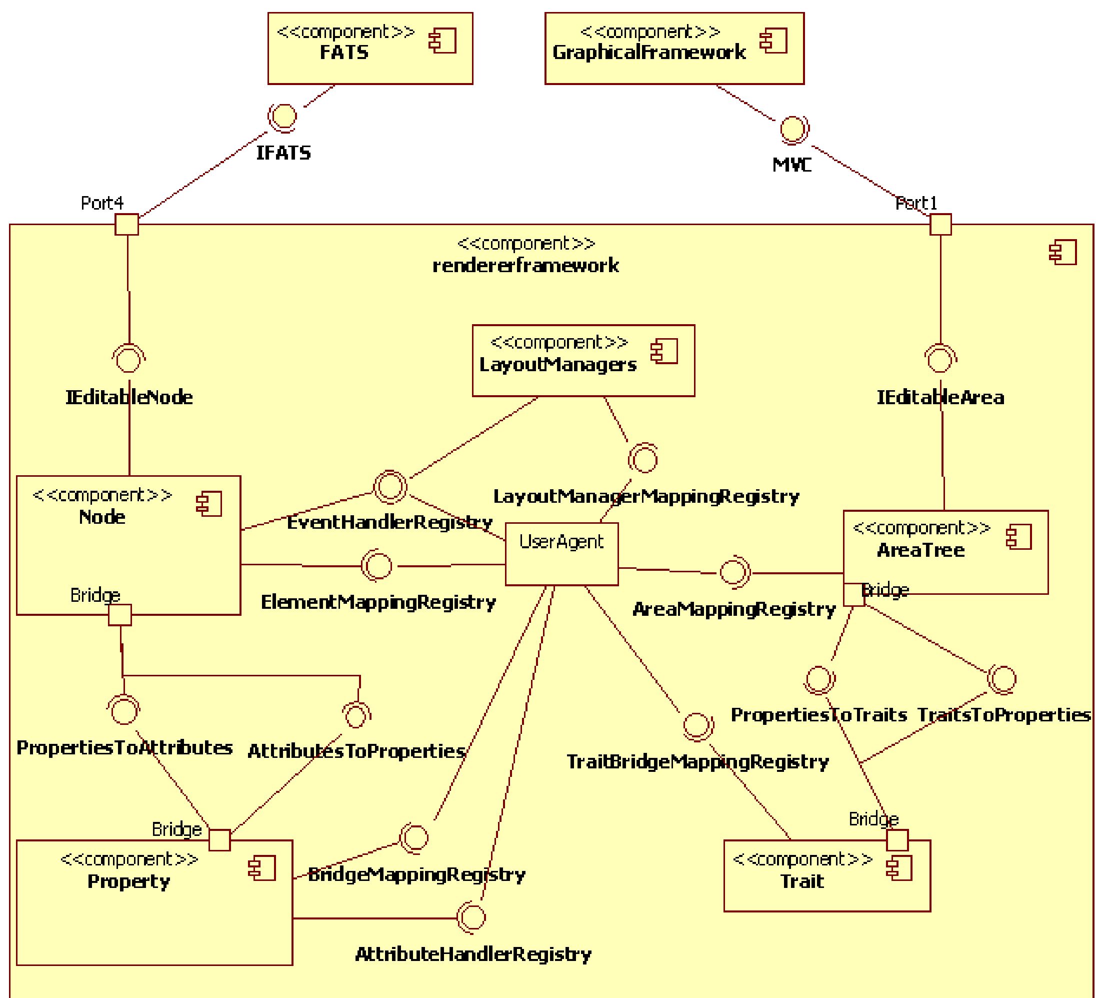


Obrázok 28: Traverzovanie XML dokumentu

Príloha 2

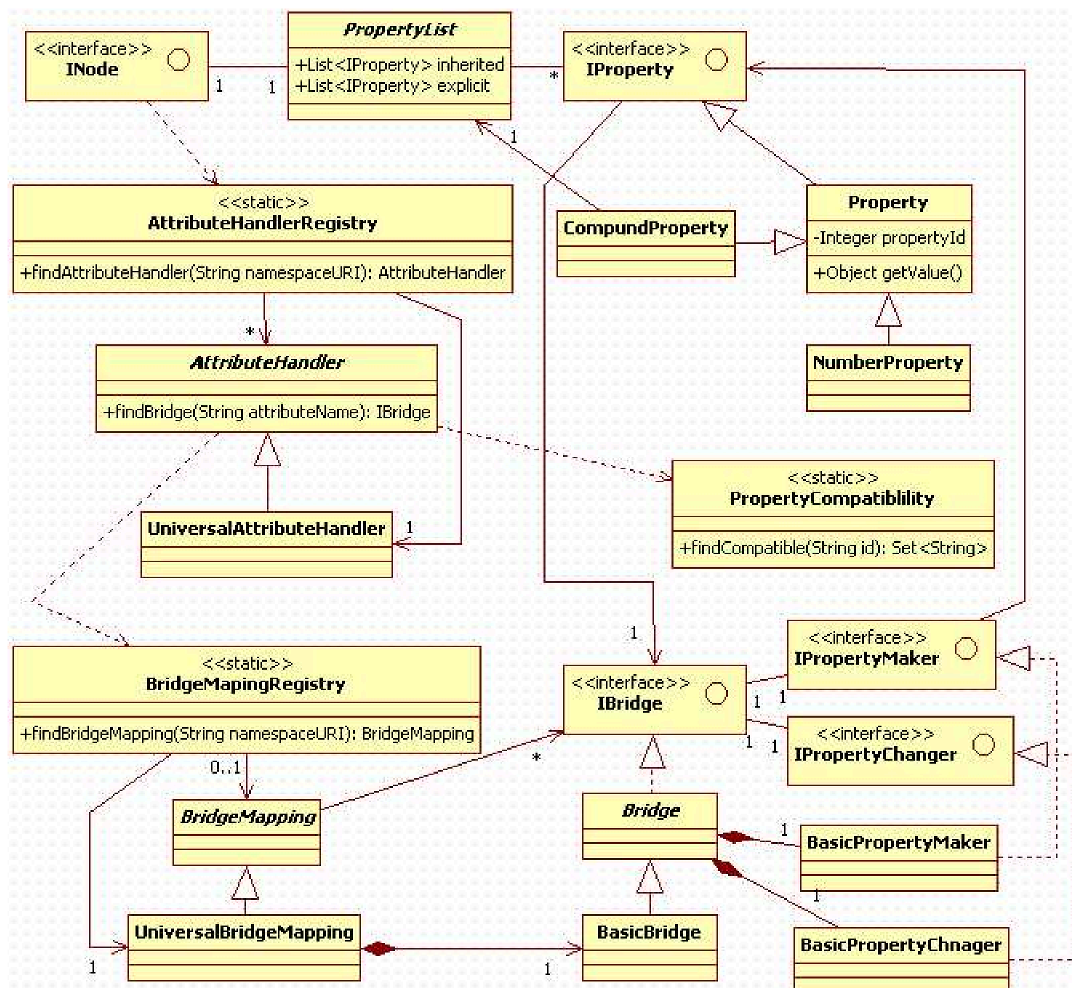


Príloha 3



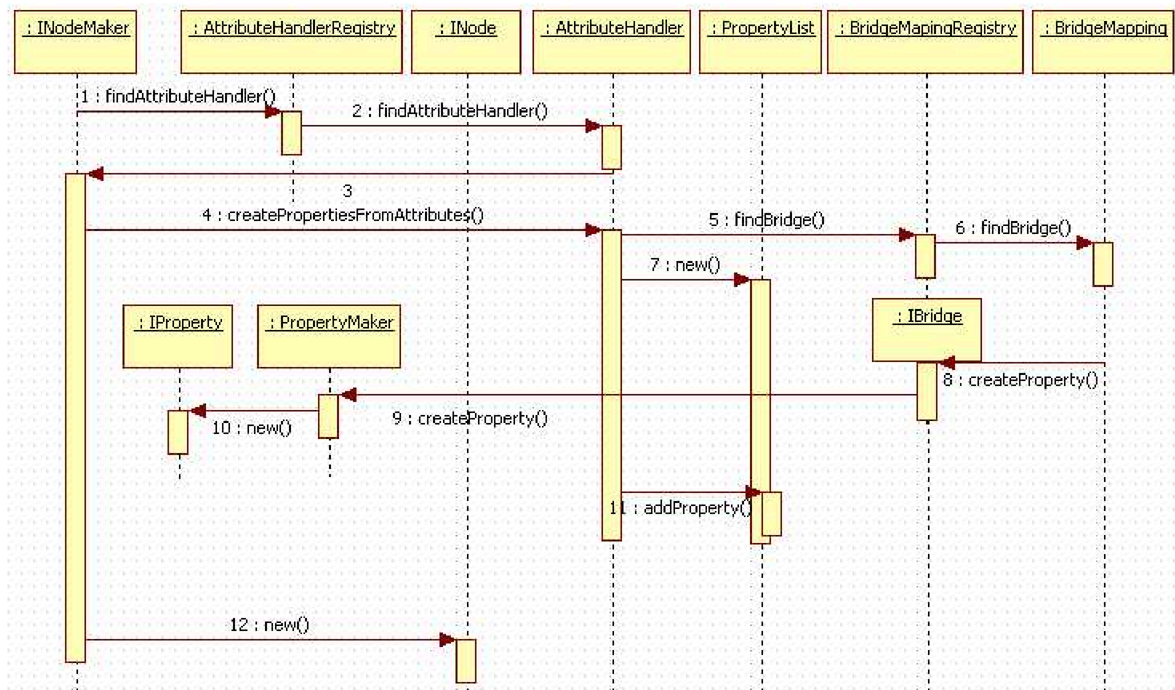
Obrázok 29: Komponentový diagram frameworku

Príloha 4



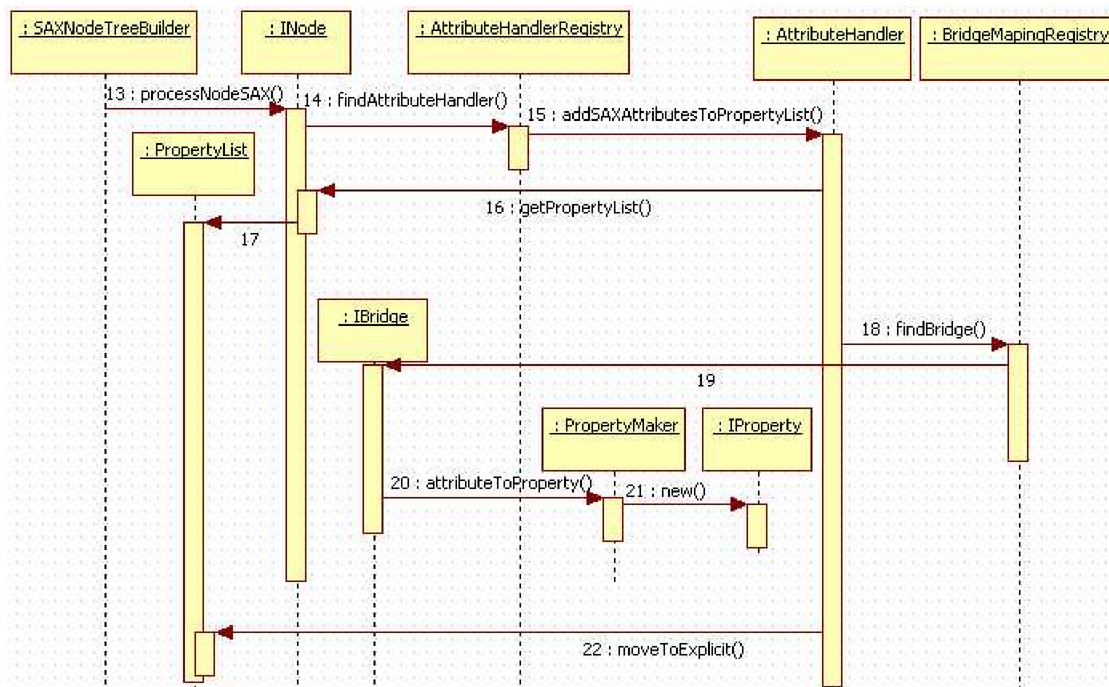
Obrázok 30: Jadro modulu Property

Príloha 5



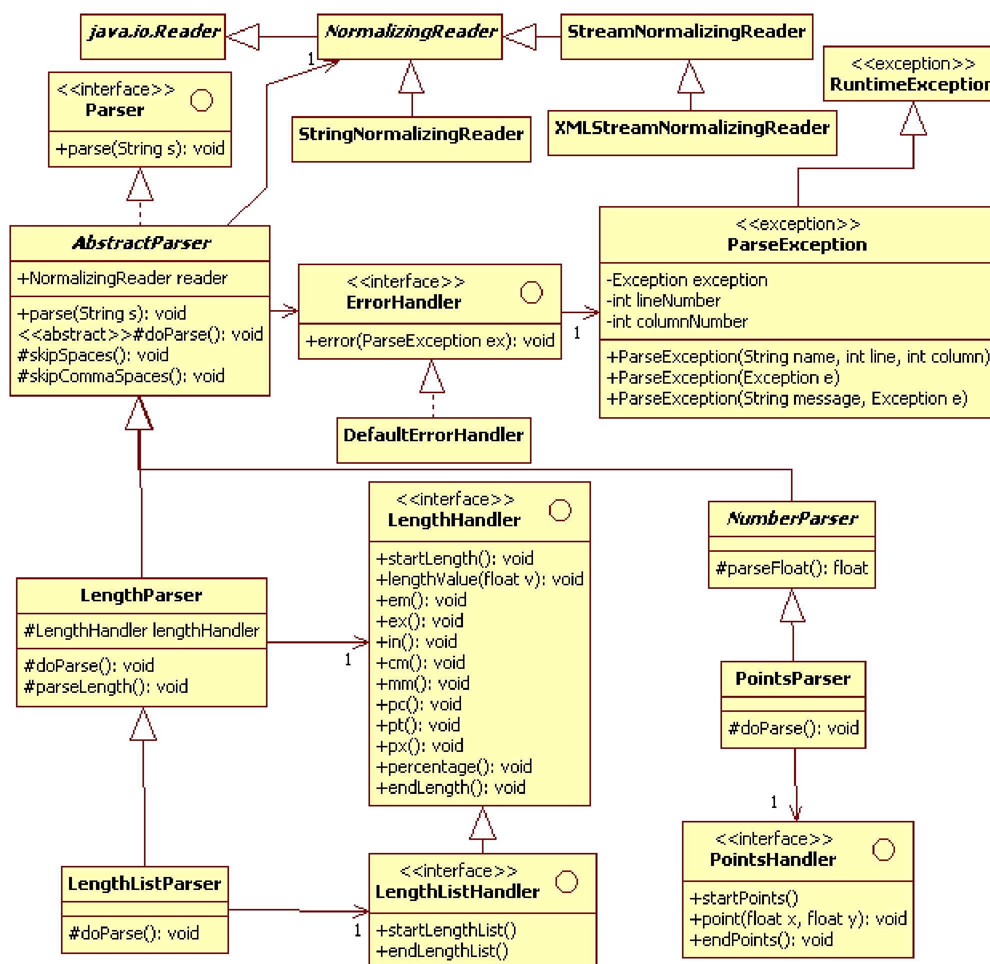
Obrázok 31: Vytváranie zoznamu vlastností s default hodnotami

Príloha 6



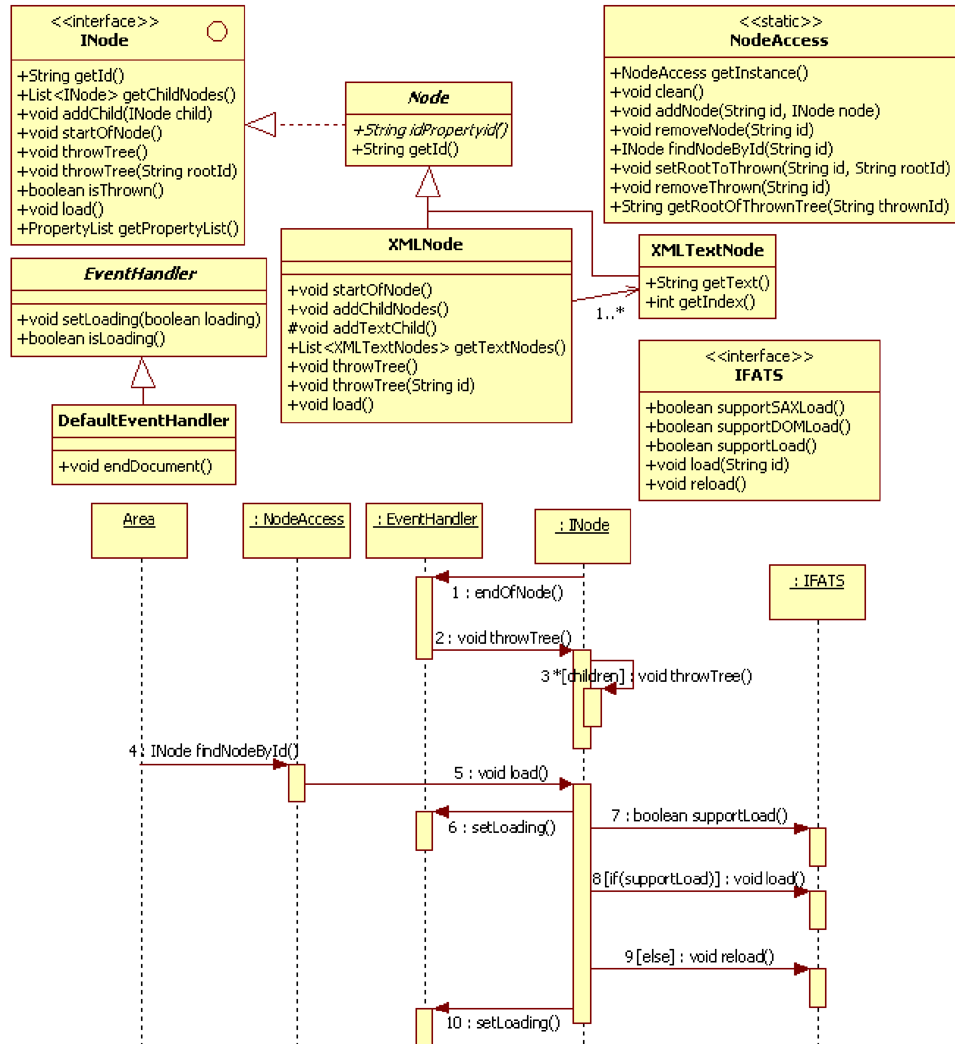
Obrázok 32: Konverzia atribútov na vlastnosti

Príloha 7



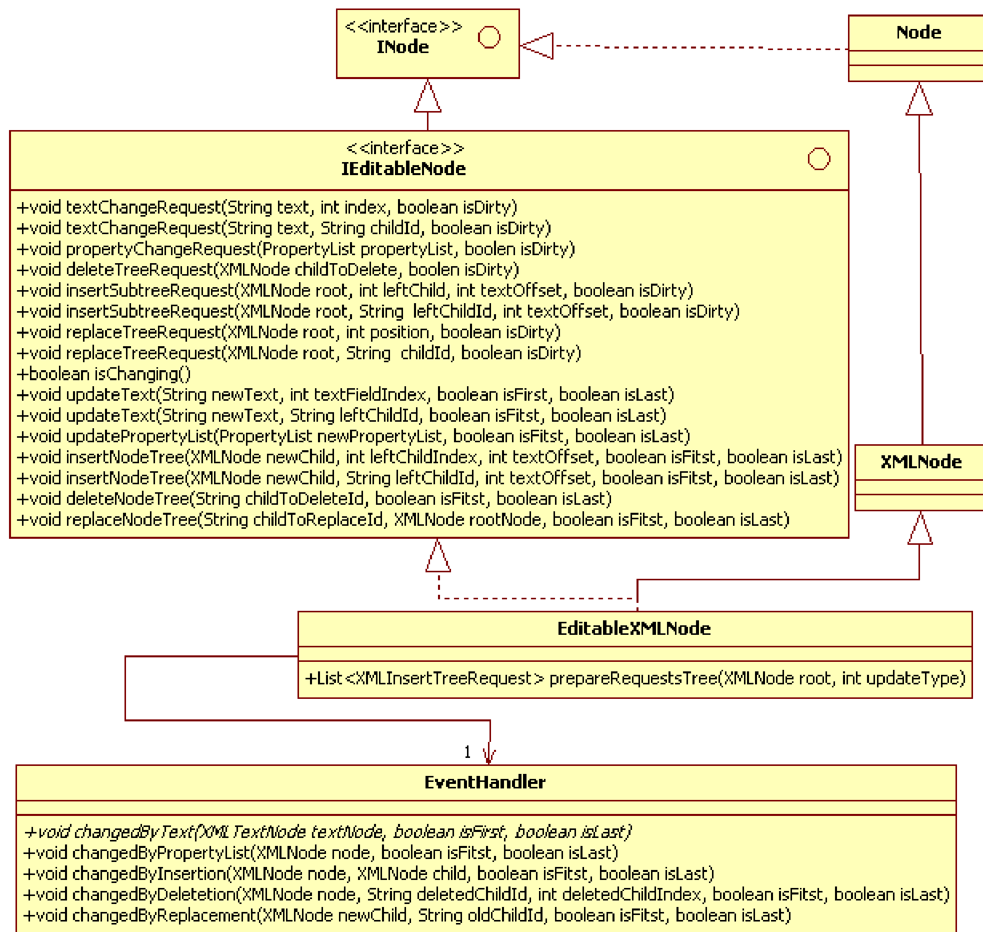
Obrázok 33: Balík org.apache.batik.parser

Príloha 8



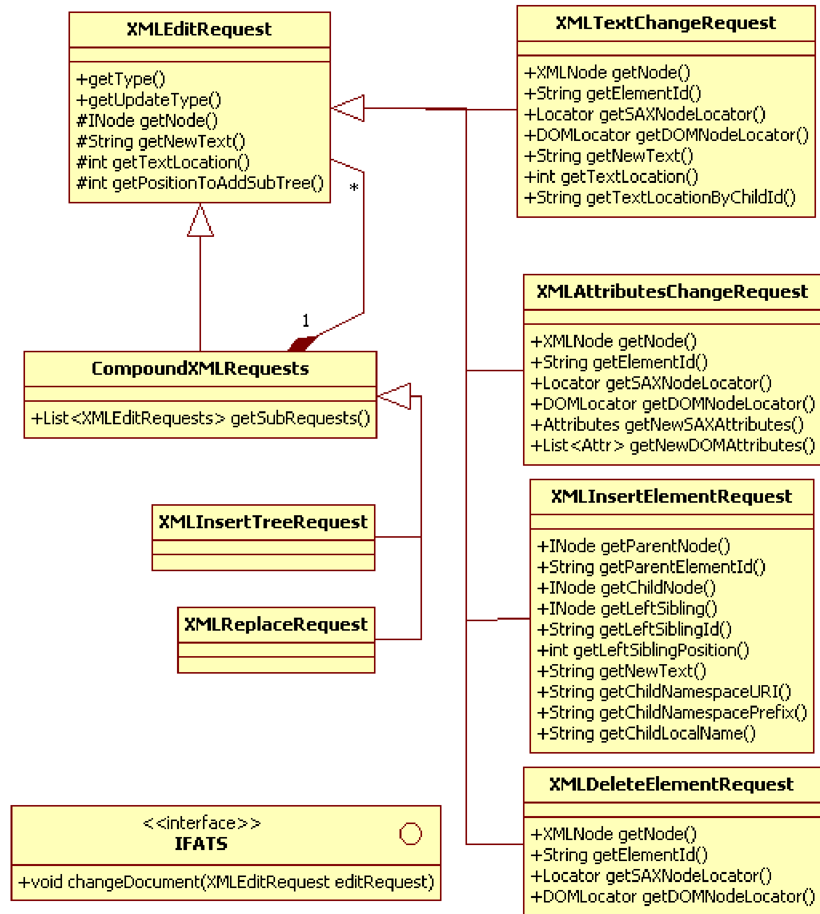
Obrázok 34: Zahadzovanie a nahrávanie uzlov stromu NodeTree

Príloha 9



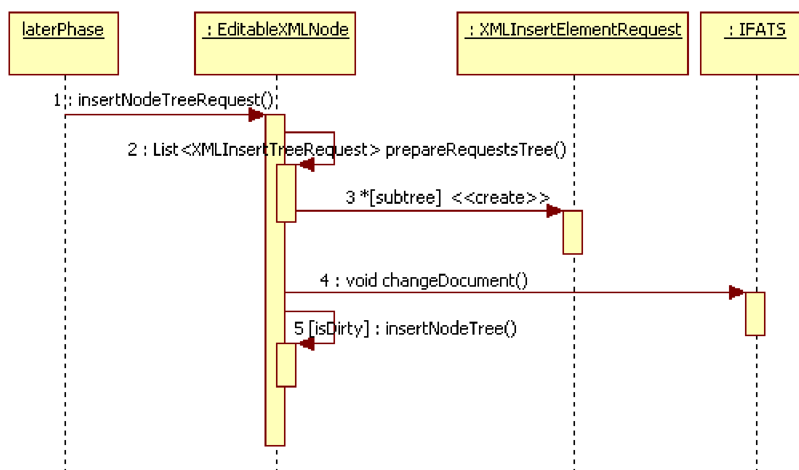
Obrázok 35: Zmeny dokumentu vo vrstve NodeTree

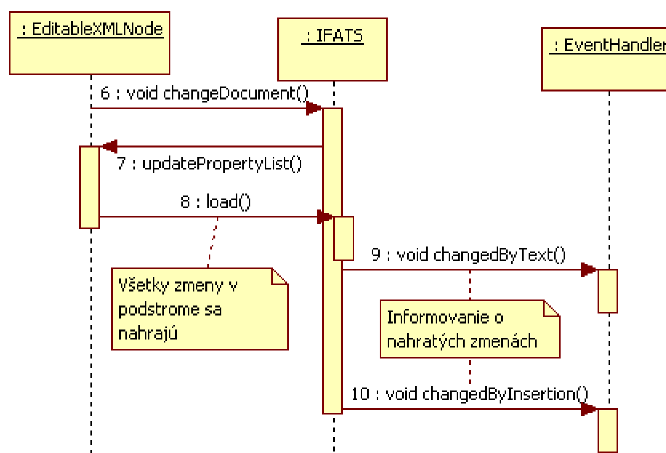
Príloha 10



Obrázok 36: Editačné požiadavky na zmenu XML dokumentu

Príloha 11





Obrázok 37: Zmeny dokumentu vo vrstve NodeTree