VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

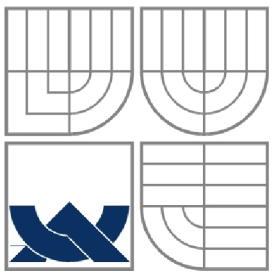
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

WEBOVÉ APLIKACE V JSP ZALOŽENÉ NA AJAX

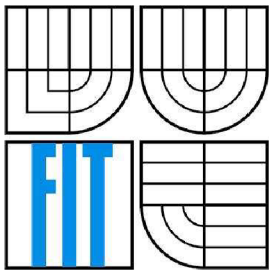
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. Štěpán Moník
AUTHOR

BRNO 2007



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

WEBOVÉ APLIKACE V JSP ZALOŽENÉ NA AJAX

JSP APPLICATIONS BASED ON AJAX

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. Štěpán Moník

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. Radek Burget, Ph.D.

BRNO 2007

Abstrakt

Diplomová práce se zabývá návrhem a tvorbou grafického uživatelského rozhraní webových aplikací. Smyslem je převést klasickou aplikaci napsanou v jazyce Java do podoby webového formuláře pomocí technologií JSP a AJAX.

Klíčová slova

Java, AWT, JSP, Java servlety, grafické komponenty, webové formuláře, HTML, CSS, JavaScript, AJAX

Abstract

This master's thesis is engaged in concept and creation of web application's graphics user interface. The reason is to convert a classic application written in Java language to the web form by force of JSP and AJAX technologies.

Keywords

Java, AWT, JSP, Java servlets, graphics components, web forms, HTML, CSS, JavaScript, AJAX

Citace

Štěpán Moník: Webové aplikace v JSP založené na AJAX, diplomová práce, Brno, FIT VUT v Brně, 2007

Webové aplikace v JSP založené na AJAX

Prohlášení

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

.....
Jméno Příjmení
Datum

Poděkování

Chci poděkovat svému vedoucímu Ing. Radkovi Burgetovi, Ph.D za přátelský přístup a pomoc při rešení obtížných problémů.

Obsah

1 Úvod

V poslední době je zřetelná tendence převádět klasické aplikace, využívající grafické komponenty operačního systému Windows či v Linuxu prostředí Xwindow, do podoby webových formulářů. Java je moderní objektově orientovaný prodramovací jazyk, pomocí něhož se dají vyvíjet jak klasické GUI aplikace pro libovolné operační systémy, tak aplikace webové. Cílem mé diplomové práce je navrhnout a realizovat Javové třídy pro generování webových formulářů, které se budou používat stejně jako třídy pro vytvátváření běžných aplikací.

V kapitole č. 2 popíši, jak v Javě fungují vestavěné třídy pro tvorbu grafického uživatelského rozhraní. V kapitole č. 3 se zabývám technologiemi, které Java poskytuje pro webové programování na straně serveru. Kapitola 4 podrobně pojednává o technologii Ajax, a to včetně její docela zajímavé historie. A konečně v páté kapitole popíši implementaci konkrétních tříd v balíčku *webawt*.

2 GUI v Javě

V současnosti existují dva přístupy k tvorbě grafického uživatelského rozhraní v Javě. Buď ho můžete založit na třídách (komponentách) z balíčku *java.awt* nebo použít balíček *javax.swing*, který tzv. Abstract Window Toolkit (AWT) rozšiřuje. Nicméně například obsluhu událostí či správce rozvržení si *swing* bere z *awt*. Při realizaci své diplomové práce jsem vycházel ze starších AWT komponent (grafické rozhraní se pomocí nich implementuje jednodušeji a intuitivněji).

2.1 Zobrazení jednoduchého okna v Javě

Zobrazení jednoduchého okna pomocí třídy *Frame* z balíčku *java.awt* vypadá následovně:

```
public class SimpleForm extends java.awt.Frame {
    public SimpleForm() {
        add(new java.awt.Button("OK"));
    }
    public static void main(String[] args) {
        SimpleForm sf = new SimpleForm();
        sf.pack();
        sf.setVisible(true);
    }
}
```

A to samé pomocí třídy *JFrame* z *javax.swing*:

```
public class SimpleForm extends javax.swing.JFrame {
    public SimpleForm() {
        add(new javax.swing.JButton("OK"));
    }
    public static void main(String[] args) {
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                SimpleForm sf = new SimpleForm();
                sf.pack();
                sf.setVisible(true);
            }
        });
    }
}
```

Metoda *javax.swing.SwingUtilities.invokeLater()* by se teoreticky volat nemusela, avšak doporučuje se to, aby se zamezilo případnému konfliktu, kdyby chtěl formulář při svém zobrazování přistupovat k prostředkům, které zrovna využívá jiné vlákno (tzv. thread-safety problem). Další problém se swingem spočívá v tom, že v případě tzv. top-level kontejnerů (což je v našem případě *JFrame*), se komponenty nepřidávají přímo do kontejneru, nýbrž do jeho 'content pane', takze až do J2SE v1.5 se muselo psát:

```
this.getContentPane().add(new javax.swing.JButton("OK"));
```

2.2 Správci rozvržení

V prostředí Java lze třídu *java.awt.Container* a její potomky používat k zobrazení skupin komponent. Instanci komponenty *JPanel* (případně *Panel*) můžete použít k zobrazení sady souvisejících tlačítek. Můžete rovněž přidat komponenty do podokna obsahu instance třídy *JFrame* (či *Frame*). **Správci rozvržení (layout managers)** jsou třídy používané k řízení velikosti a umístění jednotlivých komponent přidávaných do kontejneru. Ve většině případů jsou správci rozvržení odpovědni rovněž za zjištění rozměrů kontejneru pomocí jeho metod *getMinimumSize()*, *getPreferredSize()* a *getMaximumSize()*. Správci rozvržení jsou velmi důležitým prvkem programování v Javě, protože zjednodušují nejen rozmístění komponent, ale i určení jejich rozměrů, a umožňují tvorbu flexibilních uživatelských rozhraní.

Java poskytuje mnoho různých správců rozvržení. Každý z nich má své výhody i nevýhody. Někteří se používají snadno, ale jejich možnosti jsou omezené. Jiní se zase používají hůře, zato jsou velmi flexibilní. Pokud žádný ze správců rozvržení integrovaných do prostředí Java nevyhovuje vašim potřebám, můžete si snadno vytvořit správce vlastní.

Chcete-li správce rozvržení přidružit ke kontejneru, musíte vytvořit instanci správce a předat ji metodě *setLayout()* poskytované třídou *Container*. Následující příklad je názornou ukázkou toho, jak vytvořit instanci správce typu *BorderLayout* a přiřadit ji k instanci komponenty *JPanel*:

```
JPanel panel = new JPanel();
panel.setLayout(new BorderLayout());
```

K přidání komponenty do kontejneru se používá přetížená metoda *add()* definovaná ve třídě *Container*. Kontejner se následně stane **rodičovským kontejnerem (parent container)** komponenty. Komponenta přidaná do kontejneru se označuje jako **dceřiná komponenta (child component)**.

Přestože třída *Container* definuje mnoho různých implementací metody *add()*, nejčastěji se používají tyto:

- add(Component comp)
- add(Component comp, Object constraint)

V obou případech je odkaz na dceřinou komponentu odeslán instanci třídy *Container*. Druhá implementace obsahuje rovněž argument **constraint (omezující pravidlo)**. Tento argument poskytuje informace, které umožní správci rozvržení *vymezit* prostor určený pro zobrazení komponenty. To, jaký typ potomka třídy *Object* je v argumentu *constraints* použit, závisí na typu použitého správce rozvržení. Používáte-li instanci správce *GridBagLayout*, musí být argument *constraints* instancí typu *java.awt.GridBagConstraints*. Jiní správci vyžadují textovou hodnotu (typu *String*).

Někteří správci rozvržení omezující pravidla vůbec nepodporují, ale k určení pozice jednotlivých komponent používají pořadí, v němž jsou komponenty do kontejneru přidávány.

2.3 Někteří správci rozvržení integrovaní do prostředí Java

Záměrně se zde nebudu zabývat všemi integrovanými správci rozvržení, protože jsou vyčerpávajícím způsobem popsáni např. na domovských stránkách Javy (http://java.sun.com/). Zmíním se tu pouze o těch, kteří jsou již implementováni v mém projektu.

2.3.1 FlowLayout

Instance správce rozvržení *FlowLayout* uspořádá komponenty v řadách zleva doprava a shora dolů na základě pořadí, v němž byly do kontejneru přidány. Komponentám přitom umožní zabrat tolik prostoru, kolik potřebují. Tento správce rozvržení je užitečný zejména v případech, kdy chcete vytvořit kolekci sousedících komponent, které lze zobrazit v jejich implicitních rozměrech.

Přiklad:

```
public class SimpleForm extends JFrame {

    public SimpleForm() {
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new FlowLayout(FlowLayout.LEFT));
```

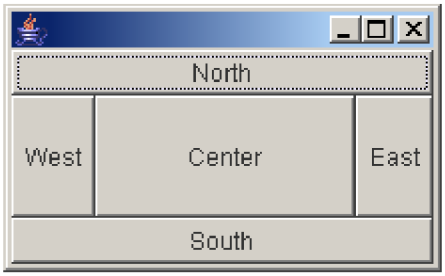
```
        add(new JButton("OK"));
        add(new JButton("Cancel"));
        add(new JButton("Long Button"));
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                SimpleForm sf = new SimpleForm();
                sf.pack();
                sf.setVisible(true);
            }
        });
    }
}
```

Obrázek 1: Uspořádání komponent pomocí správce *FlowLayout*

Při tvorbě nové instance *FlowLayout* můžete použít více konstruktorů, které umožňují, vedle nastavení způsobu zarovnávání komponent v kontejneru (LEFT, CENTER, RIGHT), také nastavení velikostí vertikálních a horizontálních mezer mezi komponentami.

2.3.2 BorderLayout

Instance třídy *BorderLayout* dělí kontejner na pět oblastí, do nichž pak lze komponenty přidávat. Pět oblastí odpovídá hornímu, levému, spodnímu a pravému okraji kontejneru plus oblasti uvnitř kontejneru (viz obrázek).



Obrázek 2: Uspořádání komponent pomocí správce *BorderLayout*

Výše zobrazeného okna docílíme následujícím kódem:

```
//..
setLayout(new BorderLayout());
add(new Button("North"), BorderLayout.NORTH);
add(new Button("South"), BorderLayout.SOUTH);
add(new Button("East"), BorderLayout.EAST);
add(new Button("West"), BorderLayout.WEST);
add(new Button("Center"), BorderLayout.CENTER);
//..
```

2.3.3 GridBagLayout

Správce rozvržení *GridBagLayout* je nejflexibilnějším správcem integrovaným do prostředí Java. Jeho hlavní nevýhodou je ovšem jeho složitost a občas i málo intuitivní použití. Na druhou stranu je to jediný správce, který je dostatečně flexibilní k tomu, aby uspořádal komponenty uvnitř kontejneru přesně tak, jak si přejete. I to je jeden z důvodů, proč je tak často používán.

GridBagLayout rozděluje dostupnou zobrazovací plochu kontejneru do mřížky buňek. Komponenty se potom do jednotlivých buňek umisťují. Zároveň ovšem mohou přetékat i do buňek sousedních. Vedle toho se pro každou vkládanou komponentu dají nastavit další omezující pravidla. To se uskutečňuje pomocí instance třídy *GridBagConstraints*, jejíž použití nyní popíši podrobněji.

Třída *GridBagConstraints* nemá (mimo zděděných) žádné metody, veškerá nastavení se provádějí přímo přiřazením hodnoty k proměnné. Hodnoty jednotlivých datových složek jsou většinou typu *int*. Výjimku tvoří složka *insets*, která je odkazem na třídu *java.awt.Insets*, a složky *weightx* a *weighty*, které jsou typu *double*.

gridx, gridy: Tyto omezující pravidla určují, podle kterého sloupce (řádku) bude komponenta zarovnána. První sloupec (řádek) odpovídá hodnotě 0. Datovým složkám *gridx* a *gridy* můžeme ovšem přiřadit konstantu *GridBagConstraints.RELATIVE*, která určuje, že komponenta může být uvnitř kontejneru umístěna relativně vůči nějaké jiné komponentě. Použijete-li např. konstantu RELATIVE pro datovou složku *gridx* a absolutní hodnotu pro datovou složku *gridy*, bude komponenta umístěna na konec řádku určeného hodnotou datové složky *gridy*.

fill: Velikost komponenty je implicitně nastavena na upřednostňovanou nebo minimální hodnotu, bez ohledu na to, jaká je velikost k ní přiřazené buňky. Omezující pravidlo *fill* určuje, že komponenta by

měla být roztažena na celou dostupnou šířku či výšku nebo šířku a výšku (konstanty HORIZONTAL, VERTICAL, BOTH a NONE).

gridwidth, gridheight: Tyto pravidla vymezí počet sloupců (řádků), na nichž je komponenta zobrazena. Implicitně obsahuje hodnotu 1. Kromě určení explicitního počtu sloupců (řádků), na které má být komponenta roztažena, můžeme použít ještě konstantu REMAINDER. Tato konstanta použitá v datové složce *gridwidth* určuje, že by zobrazovací oblast komponenty měla začínat sloupcem určeným hodnotou *gridx* a končit na posledním dostupném sloupci vpravo. Můžeme použít také konstantu RELATIVE, která v tomto případě způsobí roztažení komponenty na všechny zbývající sloupce s *výjimkou* toho posledního.

anchor: Touto datovou složkou určujeme bod, ke kterému chceme ukotvit komponentu ve vybrané zobrazovací oblasti (buňce), je-li komponenta menší než přidělená zobrazovací oblast. Datová složka *anchor* může obsahovat jednu z následujících devíti hodnot: CENTER, NORTH, NORTHEAST, EAST, SOUTHEAST, SOUTH, SOUTHWEST, WEST nebo NORTHWEST. Implicitní hodnota je CENTER.

insets: Tato vlastnost (omezující pravidlo) je odkazem na instanci třídy *Insets*. Umožňuje definovat určitou výplň kolem komponenty, tj. počet pixelů, které by měly být rezervovány kolem čtyř okrajů (*top*, *left*, *bottom* a *right*) zobrazovací oblasti komponenty. Tato vlastnost se obvykle používá k určení velikosti mezer mezi sousedícími komponentami.

ipadx, ipady: Tyto hodnoty jsou přidávány k upřednostňované nebo minimální velikosti komponenty. Slouží k určení šířky komponenty a předpona „i“ odkazuje na skutečnost, že hodnota výplně je přidána k „interní“ (upřednostňované nebo minimální) šířce (či výšce) komponenty, nikoli ke skutečné (zobrazené) šířce. Pokud má komponenta nastavenu upřednostňovanou šířku například na 40 pixelů a vy prostřednictvím datové složky *ipadx* určíte hodnotu výplně na 10 pixelů, bude se šířka (v případě zobrazení v upřednostňovaných rozměrech) zobrazené komponenty rovnat 50 pixelům.

weightx, weighty: Tyto hodnoty se používají k úpravě šířky sloupců či výšky řádek. Je-li např. šířka kontejneru větší nebo menší než šířka potřebná k zobrazení komponent v jejich upřednostňovaných nebo minimálních rozměrech. Mají-li všechny komponenty v mřížce definovánu vlastnost *weightx* jako 0.0 (implicitní nastavení), je všechen dodatečný prostor rozdělen mezi pravou a levou výplň kontejneru. Teorie ohledně používání datových složek *weightx* a *weighty* je poměrně rozsáhlá a přesahuje rámec této kapitoly, proto případné zájemce odkazuji na http://java.sun.com/.

Popisu *GridBagLayoutu* jsem záměrně věnoval více prostoru, protože jeho implementaci do webových aplikací považuji za jednu ze stěžejních částí své práce. Příklad použití tohoto správce naleznete v kapitole 5.5.

2.4 Obsluha událostí

Zpracování událostí v Javě se provádí pomocí tzv. listenerů (posluchačů). Následující fragment kódu ukazuje implementaci listeneru, který obsluhuje kliknutí myši nad tlačítkem ve swingové GUI aplikaci:

```
public class SwingApplication implements ActionListener {
    ...
        JButton button = new JButton("I'm a Swing button!");
        button.addActionListener(this);
    ....

    public void actionPerformed(ActionEvent e) {
        numClicks++;
        label.setText(labelPrefix + numClicks);
   }
}
```

Obsluha události se skládá ze tří částí:

1. Třída obsluhující událost musí implementovat příslušné rozhraní posluchače nebo musí být odvozena od třídy, která ho již implementuje. Například:

```
public class MyClass implements ActionListener {
```

2. Je třeba registrovat instanci obslužné třídy jako posluchače k jedné nebo k více komponentám:

```
someComponent.addActionListener(instanceOfMyClass);
```

3. Třída obsluhující událost musí obsahovat implementaci metod příslušného rozhraní:

```
public void actionPerformed(ActionEvent e) {
    ...//code that reacts to the action...
}
```

Swingové i Awt komponenty mohou generovat spousty druhů událostí. Následující tabulka ukazuje některé z nich:

Některé události a k nim přiřazení posluchači	
Činnost, která vyvolá událost	**Typ posluchače**
Uživatel klikne na tlačítko, zmáčkne Enter při zadávání textu nebo vybere položku z menu.	ActionListener
Uživatel uzavře frame (hlavní okno).	WindowListener
Uživatel stiskne tlačítko myši, když je kurzor nad komponentou.	MouseListener
Uživatel pohne kurzorem myši nad komponentou.	MouseMotionListener
Komponenta se stane viditelnou.	ComponentListener
Komponenta získá zaměření.	FocusListener
Změní se vybraná položka v tabulce nebo seznamu.	ListSelectionListener
Změní se jakákoli vlastnost komponenty, například text v komponentě label.	PropertyChangeListener

2.4.1 Obsluha událostí pomocí anonymních tříd a adaptérů.

K obsluze událostí se v Javě také využívají tzv. anonymní vnitřní třídy:

```
button.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
            //..
      }
});
```

Tato implementace posluchačů se hojně využívá např. ve vývojových prostředích typu *NetBeans*.

Problém s obsluhou událostí nastává ve chvíli, kdy má rozhraní více metod k implementaci, tedy reaguje na více událostí, ale my chceme překrýt pouze některé z nich. Např. rozhraní *WindowListener* obsahuje 7 metod (otevření okna, zavření okna, minimalizace okna, …), ale my chceme reagovat pouze na případ, kdy je okno minimalizováno. Museli bychom tak překrýt všech sedm metod a šest z nich nechat prázdných. V Javě ale naštěstí existuje třída *WindowAdapter*, která již má všechny prázdné metody implementovány. Můžeme ji použít např. v konstruktoru hlavního okna následujícím způsobem:

```
this.addWindowListener(new WindowAdapter() {
      public void windowIconified(WindowEvent e) {
            //..
      }
});
```

Takovýchto adaptérů samozřejmě existuje více (pro myš, klávesnici, …), nicméně jejich podrobný popis je nad rámec mého projektu.

3 Webové programování na straně serveru

Dnes existuje spousta technologií pro generování dynamických webových stránek. Z počátku se používala pouze technologie CGI. Ta definuje ověřené mechanismy pro integraci **externích bránových programů**, tedy programů, které vytvářejí bránu mezi informacemi jinými než HTML a webovými servery. Webový server a CGI program komunikují na úrovni procesu operačního systému. V operačním systému je proces základní spustitelnou jednotkou programů. Procesy mohou spouštět jiné procesy a během toho předávat informace. Webový server tedy může spustit CGI procesy a předávat informace o požadavku.

CGI je sice mocný nástroj, ale v současné době se již moc nepoužívá. Při zpracování požadavků klientů dochází ke zpouštění programů, které nadměrně zatěžují prostředky serveru včetně CPU a ztěžují generování stránek s odezvou. V tom nejhorším možném případě by každý požadavek klienta vyžadoval spuštění jednoho nebo více procesů. Protože je v moderním internetu CGI program zpravidla implementován jazykem *Perl* nebo jiným skriptovacím jazykem, zvyšuje se navíc neefektivnost také tím, že skriptovací jazyky obvykle ke zpracování systémových požadavků spouštějí další procesy. Uvedené vlastnosti CGI procesů omezují použitelnost webových aplikací, které jsou na této technologii založeny.

Další nevýhodou je to, že CGI programy nepřetrvávají během více požadavků a musejí tedy o klientovi uchovávat stavové informace v systému souborů nebo v databázi. Obecně potřebujeme těsnější vztah mezi našimi doplňky serveru a webovým serverem, aby bylo možné zmenšit dobu odezvy a zlepšit správu stavu klienta. V současné době existuje hodně technologií, které toto zajišťují. Mezi nejznámější a nejoblíbenější patří PHP, ASP, ASP.NET a především Java servlety a s nimi související stránky JSP.

3.1 Java servlety

Java servlety představují rozsáhlou, na platformě nezávislou technologii pro rozšíření funkcí webových serverů. Ryzí webové nebo aplikační Java servery mohou servlet kontejner implementovat jako vlákno uvnitř hlavního procesu. Oblíbenější webové servery jako *Apache* nebo *IIS* vyžadují odlišné technologie v závislosti na jejich vlastních rozhraních. Architektura servletů zahrnuje mimo jiné také zásuvný modul webového serveru, který přesměruje požadavky servletu do odděleného procesu Javy, jež je implementací servlet kontejneru. Takovéto použití zásuvného modulu webového serveru představuje kompromis mezi pevným a volným provázáním, protože modul webového serveru poskytuje pevně provázaný most, který předává informace mezi serverem a volně provázaným servletovým procesorem.

3.1.1 API servletu

API servletu je umístěno v balíčku *javax.servlet* a definuje interakci mezi servlet kontejnerem a samotným servletem. *Servlet* je tedy objekt, který obdrží žádost (*ServletRequest*) a na jejím základě generuje odpověď (*ServletResponse*). Balíček *javax.servlet.http* potom definuje HTTP potomky základních tříd: *HttpServlet*, *HttpServletRequest*, *HttpServletResponse* a *HttpSession*. Servlety mohou být zabaleny do WAR souboru jako webová aplikace.

3.1.2 Specifikace servletu

Původní specifikace servletu byla vytvořena firmou Sun Microsystems (verze 1.0 byla dokončena v červnu 1997). Počínaje verzí 2.3 je specifikace vyvíjena pod Java Community Process. Poslední verze je 2.5. Historii API servletu ukazuje podrobně následující tabulka:

Servlet API history			
Servlet API version	**Released**	**Platform**	**Important Changes**
Servlet 2.5	September 2005	JavaEE 5 , J2SE 5.0	Requires J2SE 5.0, supports annotations
Servlet 2.4	November 2003	J2EE 1.4, J2SE 1.3	web.xml uses XML Schema
Servlet 2.3	August 2001	J2EE 1.3, J2SE 1.2	Addition of `Filters`
Servlet 2.2	August 1999	J2EE 1.2, J2SE 1.2	Becomes part of J2EE, introduced independent web applications in .war files
Servlet 2.1	November 1998	Unspecified	First official specification, added `RequestDispatcher`, `ServletContext`
Servlet 2.0		JDK 1.1	Part of Java Servlet Development Kit 2.0
Servlet 1.0	June 1997		

3.1.3 Životní cyklus servletu

1. Třída servletu je nahrána kontejnerem při inicializaci.
2. Kontejner volá metodu *init()*. Tato metoda inicializuje servlet a musí být volána předtím, než servlet začne obsluhovat žádosti. V celém životním cyklu servletu je tato metoda volána pouze jednou.
3. Po inicializaci může servlet obsluhovat klientské žádosti. Každá žádost je obsloužena v samostatném vlákně (threadu). Kontejner volá metodu servletu *service()* pro každou žádost. Metoda *service()* určuje druh HTTP žádosti (GET, POST, atd…) a v závislosti na tom volá metody *doGet()*, *doPost()*, *doTrace()*, atd… Vývojář servletu musí implementovat tyto metody. Pokud není implementována metoda *doPost()*, znamená to, že servlet neumí zpracovávat žádosti POST. V takové situaci je zavolána metoda rodičovské třídy, která defaultně vyhodí výjimku BAD HTTP Request exception. Vývojář nikdy nesmí přetěžovat metodu *service()*.
4. Nakonec kontejner volá metodu *dostroy()*, která vyřadí servlet z provozu. Stejně jako metoda *init()* se i metoda *destroy()* volána pouze jednou za celý životní cyklus servletu.

3.1.4 ServletConfig a ServletContext

V každé aplikaci je vždy pouze jeden *ServletContext*. Tento objekt mohou použít všechny servlety k získání informací na úrovni aplikace nebo detailů o kontejneru. Naproti tomu obdrží každý servlet svůj vlastní objekt *ServletConfig*. Tento objekt poskytuje servletu inicializační parametry. Vývojář může získat odkaz na *ServletContext* buď přes *ServletConfig* nebo *ServletRequest*.

3.2 JSP

JSP doplňuje architekturu Java servletů, protože poskytuje JSP kontejner, který zajišťuje správu JSP stránek a jejich překládání na servlety.

3.3 Aplikační servery Java 2 Enterprise Edition (J2EE)

Specifikace J2EE zahrnuje Java servlety i JSP stránky. Aplikační servery J2EE zpravidla poskytují platformu Javy pro webové a další služby, například *Enterprise JavaBeans* a *Java Messaging Service*.

3.4 JSP kontejner

J2EE definuje několik kontejnerů včetně JSP kontejneru, servlet kontejneru a kontejneru *Enterprise JavaBeans*. Kontejner je v žargonu objektově orientovaného programování třída nebo komponenta, která uspořádává ostatní třídy nebo komponenty. Specifikace JSP tento původní význam rozšiřuje. Kontejnery J2EE poskytují úplné aplikační prostředí, ve kterém řídí životní cyklus komponent a poskytují jim různé služby. Navíc také ovlivňují vzájemné vazby mezi komponentami a větším aplikačním prostředím.

Kontejnery J2EE nemohou pracovat správně, pokud vývojář nenapíše softwarové komponenty tak, aby se řídily programovacími pravidly, které kontejner definuje. Vývojář musí tato pravidla respektovat, protože kontejner je v různých fázích komponenty předpokládá. Specifikace JSP zvýrazňuje význam těchto pravidel tím, že je nazývá **dohodami**. Kontejnery splnění dohod zajišťují tím, že po aplikacích požadují, aby měly implementována přesná Java rozhraní.
Každý kontejner J2EE poskytuje služby těm komponentám, za které má zodpovědnost. JSP kontejner překládá JSP stránky do kódu Java servletů a výsledek poté překládá a načítá do servlet kontejnerů. Dále také koordinuje vzájemný vztah mezi servlet kontejnerem a přeloženými JSP stránkami. Servlet kontejner poskytuje aplikační prostředí pro Java servlety.

3.4.1 Fáze JSP

Požaduje-li prohlížeč poprvé určitou JSP stránku, stane se následující:

1. Interpretuej se JSP stránka.
2. Vygeneruje se Java servlet
3. Servlet se pomocí standardního překladače, který je dodán s JSP kontejnerem, převede do bajtového kódu Java.
4. Servlet je načten do *virtuálního stroje Java* (JVM) servlet kontejneru.
5. U servletu je vyvolána **služební** metoda.

Požaduje-li prohlížeč následně stejnou JSP stránku a nebyla-li stránka od posledního volání změněna, musí JSP kontejner provést pouze krok 5. Pokud se stránka změnila, je nutné před odesláním odpovědi znovu zopakovat všech pět kroků. Zpracováním kroků 1 až 4 lze vysvětlit, proč mají JSP stránkypomalejší odezvu při prvním zobrazení v prohlížeči.

3.5 Apache Tomcat

Apache Tomcat je webový kontejner vyvinutý neziskovou organizací Apache Software Foundation (ASF). Tomcat implementuje jak servletové tak JSP specifikace od firmy Sun Microsystems. Poskytuje prostředí pro Javové programy, které tak mohou běžet ve spolupráci s webovým serverem. Přidává nástroje pro konfiguraci a správu, ale může být také konfigurován editováním XML souborů. Tomcat obsahuje svůj vlastní HTTP server.

Tomcat tedy podporuje servlety a JSP stránky. JSP stránky překládá na servlety pomocí Tomcat Jasper kompilátoru.

Tomcat je často používán v kombinaci s jiným webovým serverem, nicméně může běžet samostatně. Dříve se předpokládalo, že používání tomcatu bez externího webového serveru je možno pouze v prostředích s minimálními nároky na výkon. Dnes už je ale Tomcat používán jako samostatný webový server i pro velmi náročné aplikace.

Tomcat je na platformě nezávislý, je schopen běžet na jakémkoli operačním systému, který má nainstalováno Java Runtime Environment (JRE).

3.6 WAR soubory

WAR soubor (zkratka z Web ARchive) je v podstatě JAR (Java ARchive) soubor používaný k distribuci a seskupování JSP stránek, servletů, Javových tříd, XML souborů, knihoven značek a statických webových stránek, které dohromady představují webovou aplikaci.

WAR soubor může být, stejně jako JAR, digitálně podepsán. Ve WAR archivu existují některé speciální soubory a adresáře: Adresář /WEB-INF obsahuje soubor web.xml, který definuje strukturu webové aplikace. Pokud aplikace obsahuje pouze JSP soubory, nemusí web.xml nutně obsahovat. Pokud aplikace používá servlety, potom servletový kontejner používá web.xml, aby zjistil na jaký servlet je namapovaná URL žádost. Web.xml je také používán k definici kontextových proměnných, na které může být odkazováno uvnitř servletů, a k definici různých závislostí při deploymentu aplikace.

Příklad web.xml souboru:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

<web-app>
    <servlet>
        <servlet-name>HelloServlet</servlet-name>
        <servlet-class>mypackage.HelloServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>HelloServlet</servlet-name>
        <url-pattern>/HelloServlet</url-pattern>
    </servlet-mapping>

    <resource-ref>
        <description>
            Resource reference to a factory for javax.mail.Session
            instances that may be used for sending electronic mail
messages,
            preconfigured to connect to the appropriate SMTP server.
        </description>
        <res-ref-name>mail/Session</res-ref-name>
```

```
        <res-type>javax.mail.Session</res-type>
        <res-auth>Container</res-auth>
    </resource-ref>
</web-app>
```

Jediná nevýhoda distribuce webových aplikací za použití WAR souborů spočívá v tom, že kvůli jakékoli změně, byť je sebemenší, se musí WAR soubor vygenerovat znovu.

Adresář /WEB-INF/classes je implicitně nastaven v proměnné *classpath*. Je to místo, kam se nahrávají .class soubory.

4 Technologie Ajax

Základní otázkou při implementaci událostí do webové aplikace bylo, jak zařídit, aby se po každé vyvolané události nemusel na server odesílat celý formulář a následně ze serveru načítat celá stránka. Řešení tohoto problému spočívá v použití technologie AJAX.

4.1 Java applety

Snaha o aktualizování části webové stránky bez nutnosti načítání celého jejího obsahu a s tím spojená větší flexibilita celého webového rozhraní aplikace zaměstnávala programátory snad od samých počátků existence protokolu HTTP. Proto v průběhu let vznikly spousty technologií, které toto (alespoň částečně) umožňují. Jsou to například Java applety, kterým se kdysi prorokovala slibná budoucnost, ale především díky obstrukcím ze strany Microsoftu s implementací Javy do operačních systémů Windows, se nikdy pořádně nerozšířily. Nicméně applety na dlouhou dobu dotáhly výše uvedenou myšlenku nejdále: Jedná se o samostratnou (a plnohodnotnou) aplikaci napsanou v Javě, která má vyhrazenou část webové stránky pro své výstupy. Může tedy navázat s webovým serverem nové socketové spojení, skrz něj přenášet jakákoli data a na jejich základě měnit obsah vyhrazené části stránky. Applety dokonce mohou, stejně jako běžné Java aplikace, otevírat nová okna s klasickým uživatelským rozhraním. A díky tomu, že applety mají oproti běžným Javovým aplikacím určitá omezení, především co se týče přístupu na lokální disky, nejsou pro uživatele nijak nebezpečné.

4.2 Historie Ajaxu

Termín Ajax poprvé použil informatik Jesse James Garrett v únoru roku 2005. Přestože byl termín Ajax vymyšlen v roce 2005, většina technologií, které umožňují jeho funkci, vznikla o desetiletí dříve. A to především díky iniciativě Microsoftu ve vývoji tzv. Remote Scripting (vzdáleného scriptování). Techniky pro asynchronní nahrávání obsahu do existující webové stránky bez jejího opětovného načítání. Z tohoto úsilí vzešel element IFRAME (představen v Internet Exploreru 3 roku 1996) a u Netscapeu potom element LAYER (představen v Netscape 4, ovšem později zavržen s vývojem prohlížeče Mozilla). Oba elementy měly *src* atribut, který mohl obsahovat externí URL, a pokud rodičovská stránka obsahovala JavaScript měnící tento parametr, mohlo být dosaženo podobného efektu jako dnes s Ajaxem. Tyto technologie byly obvykle sdružovány pod obecným název DHTML (Dynamické HTML). Také technologie Flash umožňuje od verze 4 načítat XML a CSV soubory ze vzdáleného počítače bez nutnosti znovunačítání celé stránky.

Microsoft's Remote Scripting (MSRS), představený v roce 1998, působil jako elegantnější náhrada za výše popsané techniky: Natahoval data pomocí Java appletu, se kterým se komunikovalo pomocí JavaScriptu. Tato technika fungovala jak v Internet Exploreru (od verze 4) tak v Netscape Navigatoru (též od verze 4). V Internet Exploreru 5 potom Microsoft konečně představil objekt XMLHttpRequest a jeho výhod poprvé využil ve webové aplikaci Outlook Web Access, která byla dodávána společně s Microsoft Exchange Server 2000.

Komunita webových vývojářů, nejprve spolupracujících skrze newsgroup *microsoft.public.scripting.remote* a později na úrovni blogové komunity, postupně vyvíjela řadu technik pro vzdálené scriptování aby dosáhla schodného výsledku na všech prohlížečích. Výsledek jejich úsilí byl představen v roce 2002 v podobě modifikovaného MSRS, který měl konečně nahradit Java applety technologií založené na XMLHttpRequest.

World Wide Web Consorcium má několik doporučení týkajících se dynamické komunikace mezi serverem a prohlížečem. Pouze některá jsou však plně podporována. Mezi ně patří především specifikace *Document Object Model (DOM) Level 3 Load and Save*.

4.3 Nativní vs. webové aplikace

Hlavním smyslem používání Ajaxu je překonání neustálého nahrávání webových stránek. Ajax vytvořil nezbytné počáteční podmínky pro vývoj komplexních, intuitivních a dynamických uživatelských rozhraní na webových stránkách – ovšem na realizaci tohoto cíle se stále pracuje.

Na rozdíl od nativních aplikací jsou webové stránky tzv. volně vázané (loosely coupled), což znamená, že data, která zobrazují, nejsou pevně provázána s datovým zdrojem a předtím, než jsou odeslána prohlížeči (nutno upozornit, že tzv. *user agent* používaný v Angličtině jako odborný termín pro klienta přistupujícího ke službám WWW, nemusí být nutně prohlížeč), musí být „seřazena" do HTML formátu. Z tohoto důvodu musí být stránky znovu načteny pokaždé, když je potřeba zobrazit jiná data. Použitím objektu XMLHttpRequest k vyžádání a vrácení dat bez znovunahrávání celé stránky programátor obchází tento požadavek a jeho stránky se potom chovají více jako pevně vázané (tightly coupled) aplikace. Ovšem s rozdílnými časovými prodlevami mezi odesláním požadavku a zobrazením odpovědi.

Například v klasické desktopové aplikaci má programátor na výběr, zda komponentu při inicializaci naplní všemi odpovídajícími daty, nebo pouze těmi, která jsou zrovna na vrcholu a tedy mají být zobrazena - tím celé nahrávání urychlí, obzvláště, když je datová sada velká. V jiném případě by

mohla aplikace načítat data v závislosti na tom, jakou položku uživatel vybere. Této funkcionality je těžké docílit na webové stránce bez použití Ajaxu. K aktualizaci komponenty na základě uživatelova výběru by bylo potřeba celou stránku znovu nahrát.

4.4 Výhody Ajaxu

4.4.1 Uživatelské rozhraní

Nejzřejmější důvod pro používání Ajaxu je vylepšení uživatelského prostředí. Stránky využívající Ajax se chovají více jako klasické aplikace. S Ajaxem mohou být stránky dynamicky aktualizovány a s rychlejší odezvou na uživatelovy požadavky.

4.4.2 Menší objemy dat

Generováním HTML přímo v prohlížeči a přenášením pouze JavaScriptových funkcí a aktuálních dat se Ajaxové stránky jeví rychlejšími. Výhodu představuje i pro lidi, kteří u svého připojení platí podle objemu stažených dat. Příkladem této techniky může být velká datová sada, vrácená jako výsledek SQL dotazu, zabírající více stránek. S Ajaxem může být HTML stránky (např. tabulky s příslušnými TD a TR tagy) vytvořeno lokálně v prohlížeči, místo toho, aby se stahovalo s první stránkou dokumentu.

Toto „nahrávání obsahu na vyžádání“ (load on demand) jde u některých webovým aplikací tak daleko, že se u funkcí obsluhujících události stáhnou pouze hlavičky a zbytek se nahrává tzv. *on the fly* až když je potřeba. Tato technika pochopitelně výrazně snižuje zatížení přenosových linek, zvláště u aplikací s komplexní logikou a funkčností.

4.4.3 Oddělení dat, formátu, stylu a funkce

Méně zřetelný přínos z použití Ajaxu spočívá v tom, že podněcuje programátory čistě oddělovat metody a formáty pro různé druhy informací přenášené skrz web. Přestože Ajax působí jako směsice jazyků, kterou si můžou programátoři používat jak se jim zlíbí, jsou v jádru vedeni k následujícímu:

1. Oddělit hrubá data, která jsou vložená v XML a většinou pocházejí z databáze na straně serveru.
2. Oddělit formátovací struktury stránky, které jsou skoro vždy součástí HTML (nebo lépe XHTML) a je tedy možné je dynamicky měnit pomocí DOM.
3. Oddělit styly stránky: Všechno od druhů písem až po umisťování obrázků je přenášeno jako reference na CSS.
4. Oddělit funkčnost stránky, která se skládá z následujícího:
    1. JavaScriptu v prohlížeči (neboli DHTML),
    2. Standardního HTTP a XMLHttp pro přenášení informací od klienta na server a

3. Scriptování na straně serveru a/nebo programů napsanými v jakémkoli jazyce, pomocí kterých se klintovi doručí informace na základě jeho specifické žádosti.

4.5 Nevýhody Ajaxu

4.5.1 Integrace do prohlížeče

Dynamicky vytvořená stránka se neregistruje do historie, takže zmáčknutím klávesy „Back“ nemusíme obdržet požadovaný výsledek.

Vývojáři implementují různá řešení tohoto problému. Tato řešení mohou zahrnovat použití tagů IFRAME k vyvolání změn, které rozšíří historii používanou tlačítkem „Back“. Například Google Maps provedou vyhledávání v neviditelném IFRAME a potom výsledek vloží do elementu na viditelné stránce. Nutno poznamenat, že W3C nezařadilo *iframe* do svého XHTML 1.1 Recomendation; doporučuje místo toho používat element *object*.

Jiným problémem jsou aktualizace dynamických stránek, které uživatelům komplikují záložkování specifického stavu aplikace. Řešení tohoto problému existují a většina jich používá tzv. URL fragment identifier (do Češtiny se překládá jako kotva a je to ta část URL za znakem #). Je to možné díky tomu, že většina prohlížečů povoluje JavaScriptu měnit kotvu dynamicky, takže Ajaxové aplikace mohou parametry měnit podle stavu aplikace. Toto řešení také vylepšuje podporu tlačítka „Back“. Přesto to není kompletní řešení.

4.5.2 Starosti s dobou odezvy

Se síťovým zpožděním, nebo-li intervalem mezi žádostí uživatele a odpovědí ze serveru, se musí při vývoji Ajaxových aplikací počítat. Bez rychlé odezvy, chytrého cachování a správné manipulace s XMLHttpRequest objektem budou uživatelé vystaveni častým prodlevám. Navíc, když se generuje celá stránka najednou, je načítaný obsah postupně zobrazován. To bohužel nejde u načítání menších částí obrazovky, takže výsledek potom působí ještě pomaleji. Uživatel by proto měl být upozorňován na to, že na pozadí se dějí nějaké datové přenosy.

4.5.3 Optimalizace vyhledávačů

Webové stránky, které používají Ajax k nahrávání dat, která mají být indexována vyhledávacími stroji, musejí dávat pozor aby poskytovaly stejná data dostupná přes URL a ve formátu, který čte vyhledávač. Vyhledávací stroje totiž zpravidla nespouštějí JavaScript potřebný pro Ajax. Toto ovšem není specifický problém Ajaxu, stejná situace nastává se stránkami, které poskytují dynamická data jako odpověď na formulář (tento problém je občas nazýván *skrytý web (hidden web)*).

4.5.4 Závislost na JavaScriptu

Ajax je závislý na JavaScriptu, který nejenže bývá implementován rozdílně v různých prohlížečích, ale často se liší i mezi verzemi stejného prohlížeče. Kvůli tomu je potřeba stránky s JavaScriptem testovat v různých prohlížečích. Není neobvyklé vidět JavaScriptový kód napsaný dvakrát, jedenou pro Internet Explorer a podruhé pro Mozillu a kompatibilní prohlížeče. Úroveň IDE podpory pro JavaScript je také překvapivě slabá.

Další nepříjemností je to, že uživatel může podporu JavaScriptu úplně vypnout, čímž v případě Ajaxu zruší funkčnost celé stránky.

4.5.5 Webové analýzy

Spousta webových analýz je založena na předpokladu, že kdykoli je měněn nebo aktualizován obsah, je načítána nová stránka. Od té doby co Ajax změnil tento proces, musí se dávat pozor jak na celý dokument, tak na jeho jednotlivé části. Analytické systémy, které umožňují sledovat i jiné události než jen jednoduché načtení stránky (např. kliknutí na tlačítko nebo odkaz) jsou potom schopny obsloužit i stránky s rozsáhlým využitím Ajaxu.

4.6 Document Object Model (DOM)

DOM (akronym anglického **Document Object Model** – *objektový model dokumentu*) je objektově orientovaná reprezentace XML nebo HTML dokumentu. DOM je API umožňující přístup či modifikaci obsahu, struktury, nebo stylu dokumentu, či jeho částí.

Původně měl každý webový prohlížeč své vlastní specifické rozhraní k manipulaci s HTML elementy pomocí JavaScriptu. Vzájemná nekompatibilita těchto rozhraní však přivedla konsorcium W3C k myšlence standardizace, a tak vznikl *W3C Document Object Model* (zkráceně *W3C DOM*). Tato specifikace je platformně a jazykově nezávislá. Předchozí specifická rozhraní byla nazvána *Intermediate DOM* (anglicky *přechodný DOM*).

DOM umožňuje přístup k dokumentu jako ke stromu, což je zároveň datová struktura používaná ve většině XML parserů (Xerces, MSXML) a XSL procesorů (Xalan). Tato technologie, nazývaná *grove (Graph Representation Of property ValuEs)*, vyžaduje nahrání celého parsovaného dokumentu do paměti, z čehož plyne, že její optimální použití je tam, kde je k jednotlivým elementům dokumentu přistupováno v náhodném pořadí nebo opakovaně. Existuje i alternativní technologie pro případ, že je potřeba postupná, nebo jednorázová úprava – sekvenční model SAX, který má v těchto případech výhodu rychlejšího zpracování a nižší paměťové náročnosti.

5 Transformace GUI na webové rozhraní

Vzhledem k povaze zadání jsem se rozhodl, že bude nejlepší, použít technologii Java servletů. Základní třída *webawt.Frame* (ekvivalent třídy z balíčku *java.awt*) je tedy potomkem třídy *javax.servlet.http.HttpServlet*. Vývojové prostředí *NetBeans* implementuje (předgeneruje) HttpServlet následujícím způsobem:

```
import java.io.*;
import java.net.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Frame extends HttpServlet {
    /** Processes requests for both HTTP <code>GET</code> and <code>POST</code> methods.
     * @param request servlet request
     * @param response servlet response
     */
    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        /* TODO output your page here
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Servlet Frame</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>Servlet Frame at " + request.getContextPath () + "</h1>");
        out.println("</body>");
        out.println("</html>");
         */
        out.close();
    }
    /** Handles the HTTP <code>GET</code> method.
     * @param request servlet request
     * @param response servlet response
     */
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
        processRequest(request, response);
    }
    /** Handles the HTTP <code>POST</code> method.
     * @param request servlet request
     * @param response servlet response
     */
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
        processRequest(request, response);
    }
    /** Returns a short description of the servlet.
     */
    public String getServletInfo() {
        return "Short description";
```

```
    }
}
```

Obsluha žádostí GET a POST je zde sloučena do jediné metody *processRequest()*. V té tedy bývá umístěna stěžejní část kódu servletu. Zapoznámkovaný kód demonstruje její jednoduché použití.

Rozšířením třídy *javax.servlet.http.HttpServlet* o metody a vlastnosti, které zabezpečují přidávání a následné rozmístění komponent (resp. vygenerování patřičného HTML kódu), jsem tedy vytvořil ekvivalent třídy *java.awt.Frame*. Vedle toho bylo potřeba vytvořit třídy správců rozvržení a samotných komponent.

5.1 Správci rozvržení

V balíčku *java.awt* existují dvě rozhraní, která mohou správci rozvržení implementovat. Jsou to *LayoutManager* a *LayoutManager2*. Já mám rozhraní pouze jedno, jmenuje se jednoduše *Layout*. Dvě klíčové metody tohoto rozhraní jsou *add()* a *paintComponents()*. Pomocí *add()* přidává třída *Frame* do správce jednotlivé komponenty, pokud použije překrytou verzi metody, může určit i tzv. omezení (constraints), kterými se má správce při zobrazování komponenty řídit, metoda *paintComponents()* potom musí zajistit vygenerování patřičného HTML kódu a správně do něj komponenty umístit. Správci rozvržení, které jsem zatím implementoval, jsou: *FlowLayout*, *BorderLayout* a *GridBagLayout*.

5.2 Komponenty

Rodičovská třída všech komponent se jmenuje *Component*, stejně jako v balíčku *java.awt*. Následující schéma porovnává předky tlačítek z *java.awt* a *webawt*.

```
java.lang.Object
  └ java.awt.Component
      └ java.awt.Button
```

```
java.lang.Object
  └ webawt.Component
      └ webawt.Button
```

Třída *Component* má spoustu metod, kterými se nastavují vlastnosti komponenty. Také obsahuje jednu abstraktní metodu *paint()*, kterou musejí potomci implementovat a vygenerovat v ní patřičný HTML kód.

Veřejné metody třídy *webawt.Component*:

```
public boolean equals(Object o)
public void setXhttpOutput(StringBuffer xo)
public void addActionListener(ActionListener al)
public java.util.ArrayList<ActionListener> getActionListeners()
public void addKeyListener(KeyListener kl)
public java.util.ArrayList<KeyListener> getKeyListeners()
public void addFocusListener(FocusListener fl)
public java.util.ArrayList<FocusListener> getFocusListeners()
public GridBagConstraints getConstraints()
public void setConstraints(GridBagConstraints gbc)
public void setName(String name)
public String getName()
public void setText(String text)
public String getText()
public void setSize(int width, int height)
public int getWidth()
public int getHeight()
public void setFont(java.awt.Font newfont)
public java.awt.Font getFont()
```

```
public void setForeground(java.awt.Color color)
public java.awt.Color getForeground()
public void setBackground(java.awt.Color color)
public java.awt.Color getBackground()
public abstract String paint(String style, boolean usepreferredwidth,
boolean usepreferredheight)
```

Z výše uvedeného vyplývá, že rozšiřování balíčku *webawt* o další komponenty je velice snadné. Spočívá v podstatě pouze z implementace patřičného HTML kódu do metody *paint()*. Ta má tři parametry. V prvním správce rozvržení předává metodě řetězec obsahující libovolný počet CSS stylů, které si přeje, aby komponenta implementovala, další dva parametry určují, zda má komponenta použít své preferované rozměry (pokud ne, jsou v parametru style obsaženy patřičné CSS styly, které rozměr/rozměry nastaví).

5.3 Implementace událostí do webawt

Implementace událostí do projektu se skládá ze dvou částí – z klientské a serverové.

5.3.1 Implementace událostí na straně klienta

Klientská část je napsaná v *JavaScriptu* a její stěžejní částí je použití třídy *XMLHttpRequest*. Ta po vyvolání události (např. stisknutí tlařítka) zašle serveru žádost o informaci, co všechno se má na stránce změnit (jedná se vlastně o aktualizaci DOM). V praxi je tato žádost realizována pomocí parametrů URL servletu aplikace. Výňatek z kódu to demonstruje:

```
//..
var url = window.location.href + "?component=" + component + "&action=" +
action + "&param=" + param;
if (typeof XMLHttpRequest != "undefined") {
    req = new XMLHttpRequest();
} else if (window.ActiveXObject) {
    req = new ActiveXObject("Microsoft.XMLHTTP");
}
req.open("GET", url, true);
//..
```

Ve výpisu je vidět, že je potřeba rozlišit, jaký prohlížeč je používán, a podle toho se zařídit při vytváření instance třídy XMLHttpRequest. Internet Explorer totiž XMLHttpRequest vytváří jako objekt ActiveX.

Jako odpověď se objektu *XMLHttpRequest* vrátí XML soubor, který může vypadat např. takto:

```
<actions>
      <TextArea0 property="text">Thu Dec 21 14:34:52 CET 2006</TextArea0>
      <TextArea0 property="color">#ff0000</TextArea0>
      <TextArea0 property="bgcolor">#ffafaf</TextArea0>
      <Button0 property="bgcolor">#ffafaf</Button0>
</actions>
```

Tato odpověď je posléze zpracována a na základě získaných informací je změněn DOM (Document Object Model). Kompletní JavaScriptový kód je obsažen v souboru *window.js*, který je součástí projektu.

5.3.2 Implementace událostí na straně serveru

Zpracování žádostí na straně serveru je poněkud komplikovanější. Samotný servlet (třída *Frame*) musí nejprve rozhodnout, zda od klienta přichází žádost o celý formulář nebo pouze o aktualizaci na základě proběhnuté události. Toto rozhodnutí provádí na základě parametru component v URL dotazu: Pokud parametr v URL chybí nebo obsahuje prázdný řetězec, pošle servlet prohlížeči znovu celou stránku. V opačném případě podnikne kroky k obsluze proběhnuté události:

1. Přiřadí všem komponentám ve formuláři *StringBuffer*, do kterého budou odteď zapisovat změny, které se na nich provedou (nejčastěji voláním metod s prefixem *set*). Celý řetězec se změnami se potom odešle v podobě XML souboru klientovi. Formát XML souboru bude podrobně popsán později.
2. Najde komponentu, jejíž název (vlastnost *name*) odpovídá parametru *component* z URL. Dále z URL zjistí hodnotu parametru *action* (seznam možných hodnot bude popsán dále) a podle ní zavolá u komponenty příslušné posluchače.
3. V případě, že událost generuje i nějaké parametry (např. v případě stisku klávesy je potřeba předat informaci o tom, která konkrétní klávesa byla stisknuta), jsou uloženy v proměnné *param*. Ta je posléze zpracována a její obsah předán instanci jedné ze tříd dědících od bázové třídy *ComponentEvent*. Potom jsou zavolány metody konkrétního posluchače, kterým je instance *ComponentEvent* předána jako parametr.

Důležité je chování metod s prefixem *set*. V závislosti na tom, zda se teprve generuje celá stránka nebo zda se jedná o reakci na událost, metoda buď pouze změní vlastnost instance, nebo zároveň s tím přidá řádek do *StringBufferu*, který bude následně poslán klientovi jako odpověď na jeho žádost.

Příklad jednoduché metody s prefixem *set*:

```
public void setText(String text) {
      if (xhttpOutput != null) {
            xhttpOutput.append(String.format("<%1$s
property=\"%2$s\">%3$s</%1$s>\n", this.name, "text", text));
      }
      this.text = text;
}
```

5.3.3 Rozdílné hierarchie tříd obsluhujících události

V balíčku *java.awt.event* existuje rozhraní *EventListener*, které jsem ve *webawt* vynechal a implementuji rovnou třídy jako *ActionListener*, *KeyListener*… Co se týče tříd typu *ActionEvent*, *KeyEvent*…, které jsou předávány jako parametry metodám posluchačů, tak ty mají v *awt* celkem komplikovanou hierarchii. Například pro již zmíněný *KeyEvent* vypadá takto:

```
java.lang.Object
  └ java.util.EventObject
      └ java.awt.AWTEvent
          └ java.awt.event.ComponentEvent
              └ java.awt.event.InputEvent
                  └ java.awt.event.KeyEvent
```

Pro zjednodušení jsem do *webawt* implementoval pouze třídu *ComponentEvent*, od které dědí všechny další třídy předávané jako parametr metodám obsluhujícím události.

5.3.4 Parametry URL při žádosti o reakci na událost

```
http://<adresa servletu>?
component=<komponenta>&action=<akce>&param=<parametr události>
```

komponenta – komponenta, nad kterou byla vyvolána událost
akce – konkrétní typ události (možné hodnoty níže)
parametr události – pokud je potřeba ze strany klienta předat nějaké informace o provedené události, jsou uloženy v této proměnné

Podporované akce nad komponentami:
click – zasílá se v případě kliknutí myši na komponentu
keydown – zasílá se v případě stisknutí klávesy
keyup – zasílá se v případě puštění klávesy
focus – zasílá se, když komponenta získá zaměření
blur – zasílá se v případě, když komponenta ztratí zaměření
itemselect – zasílají komponenty implementující ItemListener v případě, že se změní výběr
textchange – zasílá se poté, co uživatel změní text u komponenty *TextField* (u *TextArea* se tak neděje z důvodu rizika přenosu neúměrně velkých objemů dat)

Ještě je nutno dodat, že dotazy na aktualizaci DOM se posílají pouze v případě, že je na událost navázaný příslušný posluchač. Tím se zabraňuje zahlcení serveru zbytečnými dotazy.

5.3.5 XML soubor vrácený ze serveru jako reakce na událost

Obecný formát:

```
<actions>
      <komponenta property=vlastnost>nová hodnota</komponenta>
      ...
</actions>
```

komponenta – název komponenty

vlastnost – vlastnost komponenty, kterou je potřeba změnit

nová hodnota – nová hodnota vlastnosti

Podporované vlastnosti:

text – mění text komponenty (ve většině případů se jedná o TextElement mezi otevíracím a uzavíracím tagem HTML)

color – mění barvu popředí (nejčastěji písma) komponenty

bgcolor – mění barvu pozadí komponenty

font – mění vlastnosti použitého písma, *nová hodnota* má potom následující formát:

```
<Button0 property="font">Arial,bold,normal,16pt</Button0>
```

Jedná se o čtyři parametry oddělené čárkou: První určuje druh písma, druhý jeho tučnost, třetí styl (kurzíva) a čtvrtý velikost. JavaScript potom jednotlivé hodnoty přiřadí příslušným vlastnostem v DOM.

Následující vlastnosti se týkají kontejnerových komponent (List):

additem – přidává položku do kontejneru. V závislosti na tom, jestli je uveden parametr *index*, buď na místo odpovídající indexu nebo za poslední položku:

```
<List0 property="additem" index="2">Item</List0>
```

nebo

```
<List0 property="additem">Item</List0>
```

removeitem – odstraní položku/položky z kontejneru. A to buď na základě indexu, nebo podle její textové hodnoty.

```
<List0 property="removeitem" index="2"/>
```

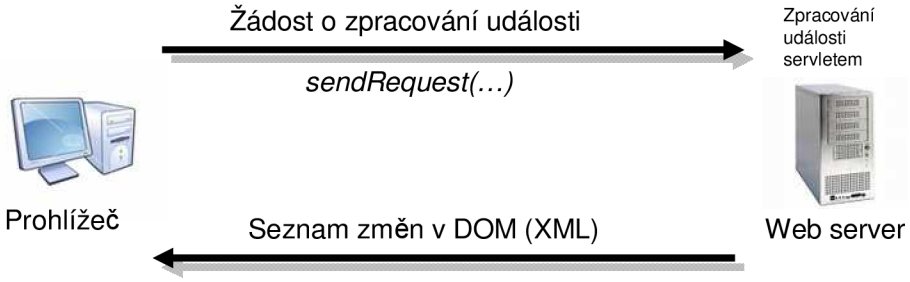
nebo

```
<List0 property="removeitem">Item</List0>
```

removeall – odstraní vševhny položky z kontejneru.

selectitem – označí položku v kontejneru na základě parametru *index*.

5.3.6 Schéma zpracování události



Obrázek 3: Schéma zpracování události

5.4 Použití balíčku *webawt*

K tomu, aby bylo možno používat třídy z *webawt* je potřeba mít v počítači nainstalován nějaký webový kontejner. Nejrozšířenější je v současné době Apache Tomcat, který je podrobně popsán ve třetí kapitole. Nicméně vývojové prostředí NetBeans od firmy Sun Microsystems již Tomcat obsahuje. Takže pokud vývojář použije pro rozšíření balíčku právě NetBeans, nemusí se o webový kontejner starat. Na přiloženém CD-ROM disku je celý projekt pro NetBeans IDE 5.0.

5.5 Stejný kód Javy zobrazí v prohlížeči to samé, co v okně

5.5.1 GridBagLayout

Následujícím fragmentem kódu a screenshotem chci ukázat, jak je implementován komplikovaný správce rozvržení *GridBagLayout*. Kód je zkopírovaný z internetu a bez jakýchkoli úprav použit nejprve v klasické Javové aplikaci využívající formulář z balíčku *java.awt* a posléze ve webové aplikaci založené na formuláři z *webawt*.

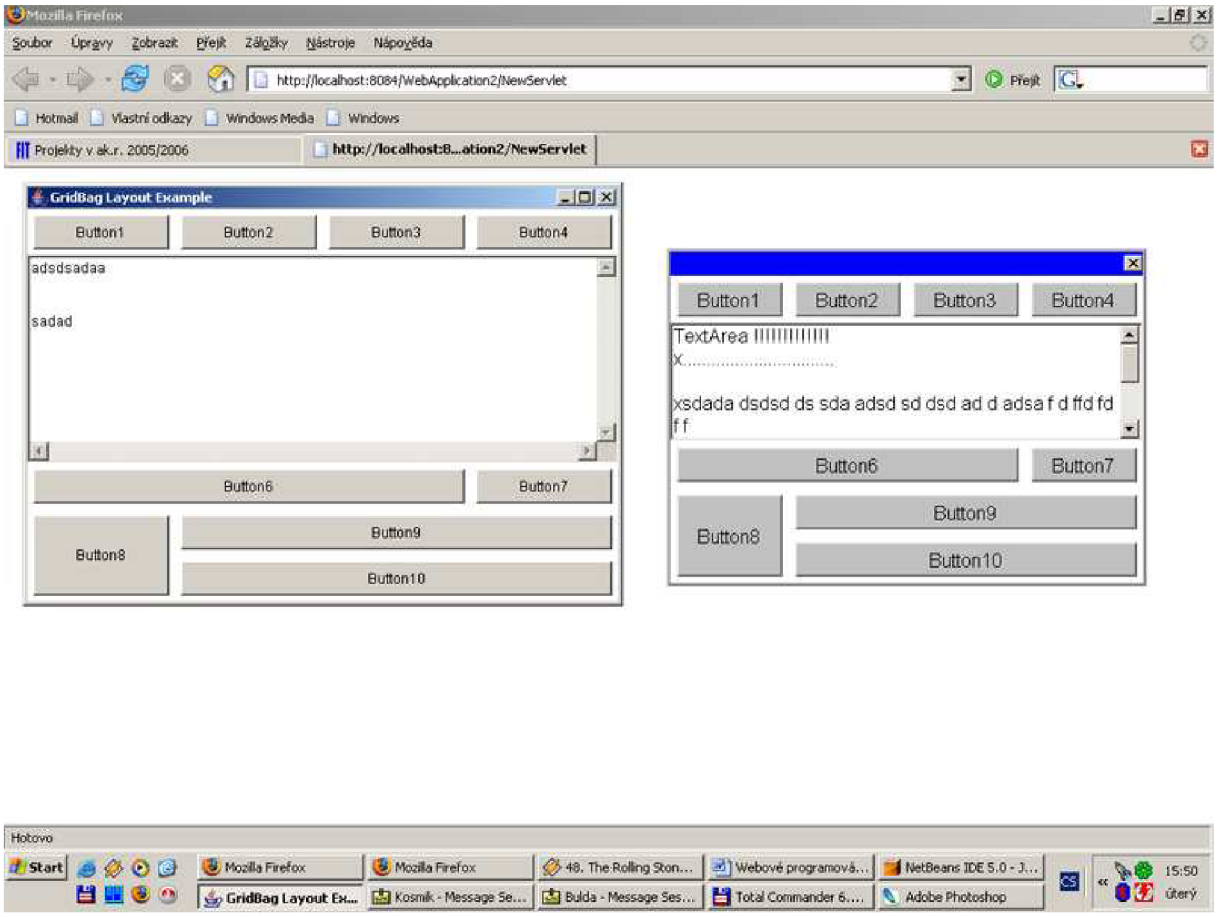
```
protected void makebutton(String name,
                          GridBagLayout gridbag,
                          GridBagConstraints c) {
        Component button;
        if (name==”TextArea”)
           button = new TextArea(name);
        else
           button = new Button(name);
        gridbag.setConstraints(button, c);
        add(button);
}

public NewServlet() {
        GridBagLayout gridbag = new GridBagLayout();
        GridBagConstraints c = new GridBagConstraints();
        setPosition(300,100); //Umistuje okno v prohlizeci, neni v awt.
        setLayout(gridbag);
        c.fill = GridBagConstraints.BOTH;
        c.insets = new Insets(5,5,5,5);
```

```
        makebutton("Button1", gridbag, c);
        makebutton("Button2", gridbag, c);
        makebutton("Button3", gridbag, c);
        c.gridwidth = GridBagConstraints.REMAINDER; //end row
        makebutton("Button4", gridbag, c);

        c.insets = new Insets(0,0,0,0);
        makebutton("TextArea", gridbag, c); //another row

        c.insets = new Insets(5,5,5,5);
        c.gridwidth = GridBagConstraints.RELATIVE; //next-to-last in row
        makebutton("Button6", gridbag, c);

        c.gridwidth = GridBagConstraints.REMAINDER; //end row
        makebutton("Button7", gridbag, c);

        c.gridwidth = 1;                //reset to the default
        c.gridheight = 2;
        makebutton("Button8", gridbag, c);

        c.gridwidth = GridBagConstraints.REMAINDER; //end row
        c.gridheight = 1;               //reset to the default
        makebutton("Button9", gridbag, c);
        makebutton("Button10", gridbag, c);
}
```

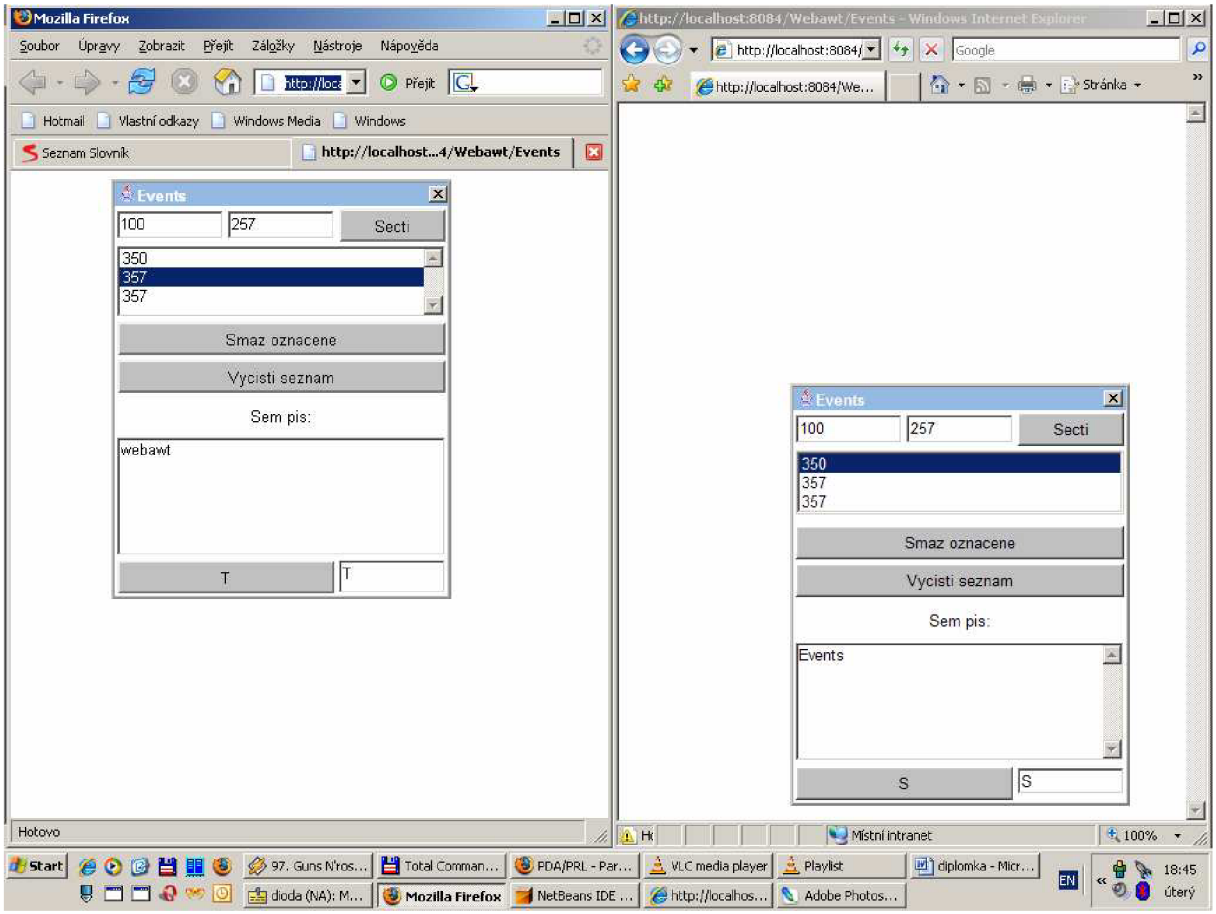
Výše uvedený kód zobrazí, v závislosti na tom, zda ho použijete v konstruktoru formuláře *java.awt.Form* nebo *webawt.Form*, buď klasické okno s komponentami nebo webový formulář, jak je vidět na obrázku č.4.

5.5.2 Výstup aplikace v prohlížečích Mozilla Firefox a Internet Explorer

Obrázek č.5 ukazuje výstup aplikace v prohlížečích Mozilla Firefox a Internet Explorer. Jak je patrno, výsledek je téměř totožný.



Obrázek 4: Porovnání skutečného okna s komponentami rozloženými pomocí správce rozvržení *GridBagLayout* s formulářem z *webawt*, ve kterém je použit stejný správce.



Obrázek 5: Porovnání výstupu u prohlížečů Mozilla Firefox a Internet Explorer

7 Závěr

Na základě požadavků jsem navrhl způsob transformace klasického GUI na webové rozhraní pro technologii JSP a v podobě balíčku *webawt* jsem implementoval ekvivalenty některých tříd z *java.awt*. Projekt je použitelný především při převodu klasických GUI aplikací do webové podoby. Téma jsem si vybral, protože mám rád moderní objektově orientované programování, jehož symbolem se pro mě stal jazyk Java. Má diplomová práce zajímavým způsobem spojuje webové programování na straně serveru s klasickými GUI aplikacemi. Ke tvorbě uživatelského rozhraní na straně klienta (v prohlížeči) potom využívá v současnosti populární technologii Ajax. Celkově tak vlastně kopíruje moderní trendy ve vývoji webových aplikací.

Třídy z balíčku *java.awt* nejsou pochopitelně implementovány všechny. Smyslem diplomové práce bylo spíše navrhnout systém, který bude možno dále jednoduše rozšiřovat. Co se týče omezení zvoleného přístupu, myslím, že vyplývá z textu jednotlivých kapitol technické zprávy, ale ve stručnosti vypíchnu nejdůležitější body: Aby se u webového formuláře dosáhlo stejného vzhledu a funkčnosti jako u klasické aplikace, je potřeba použít to nejmodernější, co programování na straně webového klienta nabízí (absolutní pozicování u CSS, JavaScript, Ajax, modifikace DOM…) a to samozřejmě nepodporují všechny prohlížeče stejně. Často se tak stává, že se stejná věc píše několikrát pro různé prohlížeče. Dalším problémem je, že ne všichni uživatelé jsou k internetu připojeni dostatečně rychle, což může mít pro Ajaxové aplikace nepříjemné následky v podobě dlouhých prodlev při reakcích na události. Při vývoji Ajaxových aplikací se také, jak jsem zjistil, musí dávat velký pozor na to, aby se server nezahlcoval zbytečnými žádostmi. A v neposlední řadě si vůbec nejsem jistý, jestli ono “doslovné” převádění nativních aplikací na webové je správnou cestou. Myslím si totiž, že na webové aplikace by měly být už pro jejich podstatu kladeny trochu jiné požadavky. Od samého počátku jsou koncipovány jako dokument (a na tom nic nezmění ani to, když se do nich umístí deset Flashových oken) a uživatelé je tak vnímají. Berou je, spíše než jako samostatnou aplikaci, jako dokument ve Wordu, u kterého si klikáním na odkazy mohou měnit obsah.

Na druhou stranu je pravda, že současným webovým aplikacím by prospělo, kdyby existovala nějaká obecná shoda nad způsobem, jak se mají programovat. Technologií (především na straně serveru) je dnes tolik, že je problémem už jenom samotný výběr platformy pro projekt.

Z čistě teoretickéo hlediska jsem dospěl k závěru, že by se do webové podoby dal implementovat celý balíček *awt*. Vyžadovalo by to sice spoustu času, ale vzhledem k roztříštěnosti technologií v oblasti webového programování, která v současnosti panuje, mají podobné projekty šanci na úspěch.

8 Literatura

[1] Brett Spell: Java Programujeme profesionálně
Computer Press 2002
ISBN 80-7226-667-5

[2] Gary Bollinger, Bharathi Natarajan:
JSP – Java Server Pages
Podrobný průvodce začínajícího tvůrce
Grada Publishing a.s. 2003
ISBN 80-247-0340-8

[3] Pavel Herout: Java – grafické uživatelské prostředí a čeština
Kopp 2001
ISBN 80-7232-150-1

[4] Pavel Herout: Java - bohatství knihoven
Kopp 2003
ISBN 80-7232-209-5

[5] Ryan Asleson, Nathaniel T. Schutta:
Ajax – Vytváříme vysoce interaktivní webové aplikace
Computer Press, a. s. 2006
ISBN 80-251-1285-3

[6] Domovské stránky jazyka Java (The Source for Java Developers)
http://java.sun.com

[7] Wikipedia – The Free Encyclopedia
http://www.wikipedia.org

9 Přílohy

CD-ROM se zdrojovými kódy balíčku *webawt*, projektem do NetBeans IDE, ukázkovými servlety a AWT aplikacemi.