UNIVERZITA PARDUBICE
Fakulta elektrotechniky a informatiky

Designér formulářů pomocí Java Swing a XML

Jan Patočka

Bakalářská práce
2013

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: **2012/2013**

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Jan Patočka**
Osobní číslo:	**I10166**
Studijní program:	**B2646 Informační technologie**
Studijní obor:	**Informační technologie**
Název tématu:	**Designér formulářů pomocí Java Swing a XML**
Zadávající katedra:	**Katedra informačních technologií**

Zásady pro vypracování:

Cílem práce je vytvoření tzv. modulu Designér, pomocí kterého bude možné měnit za běhu aplikace vzhled a funkčnost vybraných formulářů.
Teoretická část:
V teoretické části práce budou popsány technologie vhodné pro návrh a implementaci modulu aplikace (XML, Java Reflection, RMI).
Implementační část:
Formuláře budou popsány pomocí XML dokumentů uložených na serveru. Modul umožní generaci XML souboru z datového modelu a z již existujícího formuláře. Dále pak editaci tohoto souboru pomocí vlastního textového i grafického editoru (designéru). Designér bude zobrazovat návrh formuláře metodou WYSIWYG a bude podporovat Drag&Drop úpravy.

Rozsah grafických prací:

Rozsah pracovní zprávy:

Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam odborné literatury:

ECKEL, Bruce. Myslíme v jazyku Java: knihovna zkušeného programátora. 1. vyd. Praha: Grada, 2001, 470 s. ISBN 80-247-0027-1
HEROUT, Pavel. Java a XML. 1. vyd. České Budějovice: Kopp, 2007, 313 s. ISBN 978-80-7232-307-4.

Vedoucí bakalářské práce: **Ing. Zdeněk Šilar**
Katedra informačních technologií

Datum zadání bakalářské práce: **21. prosince 2012**
Termín odevzdání bakalářské práce: **10. května 2013**

L.S.

prof. Ing. Simeon Karamazov, Dr.
děkan

Ing. Lukáš Čegan, Ph.D.
vedoucí katedry

V Pardubicích dne 29. března 2013

Prohlášení autora

Prohlašuji, že jsem tuto práci vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Souhlasím s prezenčním zpřístupněním své práce v Univerzitní knihovně.

V Pardubicích dne 30. 4. 2013 Jan Patočka

Poděkování

Chtěl bych mnohokrát poděkovat vedoucímu své práce, panu Ing. Zdeňku Šilarovi za jeho odborné rady a za veškerý čas, který mi s ochotou věnoval. Také bych rád poděkoval softwarové firmě Ortex spol. s r.o., pro kterou jsem tuto bakalářskou práci vytvářel.

Anotace

Cílem práce je vytvoření modulu Designér, umožňující měnit za běhu aplikace vzhled a funkčnost formulářů. V teoretické části práce budou popsány technologie využité při implementaci (XML, Java Reflection, Swing, RMI). Formuláře budou popsány pomocí XML dokumentů uložených na serveru. Modul umožní generaci XML souboru z datového modelu a z již existujícího formuláře. Dále pak editaci tohoto souboru pomocí vlastního textového i grafického editoru (designéru).

Klíčová slova

Java, RMI, Swing, XML, formulář, designér

Title

Form designer using Java Swing and XML

Annotation

The aim of this thesis is to create a module called Designér, which allows to change visual appearance as well as functionality of forms. In the theoretical part of the thesis the technologies used for implementation are described (XML, Java Reflection, Swing, RMI). The forms are characterized using XML documents stored on a server. The module provides a possibility to generate XML files from the database or from an already existing form. Further it allows you to edit said files using an included text and/or graphics editor (designer).

Keywords

Java, RMI, Swing, XML, Form, Designer

Obsah

Seznam zkratek

WISIWIG	What I see is what I get
XML	Extendable Markup Language
UML	Uniform Markup Language
RMI	Remote Method Invocation
HTML	Hypertext Markup Language
SAX	Simple API for XML
DOM	Domain Object Model
D&D	Drag and Drop

Seznam obrázků

Seznam zdrojových kódů

Úvod

V dnešních aplikacích je vzhled velice důležitý. Pro mnoho uživatelů dokonce důležitější, než samotná funkčnost a rychlost. Firmám se tedy vyplatí investovat do grafické stránky jejich projektů. I přes veškerou snahu firem však není možné vyhovět požadavkům všech zákazníků. Jednou z možností je vytvoření aparátu, který by dokázal upravovat vzhled aplikace přesně podle požadavků klienta, bez nutnosti změn v jejím kódu. Tento aparát bude vytvořen jako modul aplikace Orsoft Open, vyvíjené firmou Ortex spol. s r.o.

Cílem mé práce je návrh a vhodná implementace výše zmíněného modulu, který by měl poskytovat následující možnosti:

- generace XML souboru z datového modelu
- generace XML souboru z již existujícího formuláře
- vytvoření formuláře podle XML souboru
- aplikování XML souboru na existující formulář
- editace XML souboru
    - v textovém režimu
    - v grafickém režimu (WISIWIG editor)

Větší část práce je věnována popisu těchto funkcí. Pozornost je zaměřena především na možnosti jejich implementace. Důraz je kladen na znuvupoužitelnost a rozšiřitelnost kódu. Každá kapitola práce má jasně daný výstup, přičemž kapitoly jsou mezi sebou částečně provázány.

Práce je určena především znalým v oblasti programování GUI aplikací. Je psána tak, aby i neznalý čtenář pochopil podstatu řešených problémů, ovšem pro plné pochopení většiny tezí je potřeba se v oblasti programování pohybovat. Po přečtení práce získá programátor informace potřebné pro zpracování rozsáhlejšího projektu a je seznámen s některými pokročilejšími technikami programování. Díky těmto poznatkům se může vyvarovat chybám ve vývoji vlastních aplikací.

1 Použité technologie

Tato kapitola se zaměřuje na popis použitých technologií a na důvody jejich výběru.

1.1 Programovací jazyk Java

Jak již bylo řečeno v úvodu této práce, mým úkolem je vytvořit modul k existující aplikaci. Celá aplikace Orsoft Open je napsána v programovacím jazyce Java a je tedy více než vhodné použít stejný jazyk i pro tvorbu modulů této aplikace.

Základy a koncepci jazyka definovali v roce 1991 zaměstnanci firmy Sun Microsystems James Golsing, Patrick Naughton, Chris Warth, Ed Frank a Mike Sheridan. Jazyk byl původně označován jako „Oak“ a v roce 1995 byl přejmenován na „Java“. Většina syntaktických pravidel byla převzata z jazyků C a C++, proto je pro programátory v jazyku C kód jazyka Java snadno srozumitelný. Podobnost s jazyky C a C++ však u syntaxe končí. Java nepodporuje řadu vlastností těchto jazyků, zejména ty, jejichž využívání vede k problémům, častým chybám či nestabilitě aplikace. Příkladem těchto vlastností mohou být:

- mnohonásobná dědičnost
- práce s ukazateli (pointery)

Vzniká tedy jazyk se zjednodušenou syntaxí, celkově snadněji zvládnutelný, což vede k větší produktivitě a menší frekvenci chyb. [3]

Překladač jazyka Java provádí podobně jako v případě jazyků C a C++ přísnou kontrolu statických typů, programátor je nucen dodržovat příslušná pravidla pro manipulaci s těmito typy. Daná vlastnost také vede k vytváření spolehlivějších programů. Další důsledné kontroly provádí Java při běhu programů. Systém run-time jazyka Java uchovává informace o všech objektech použitých v aplikaci, což umožňuje určit typ i při běhu programu. Při něm se například ověřuje možnost převedení jednoho typu na typ odlišný. Kontrola typů při běhu programu také vytváří prostor pro používání zcela nových dynamicky zaváděných typů při zachování jisté míry bezpečnosti. [4]

Další výraznou výhodou jsou prostředky pro automatickou správu paměti. Tím se liší od jazyků C či C++, kde je za správu paměti odpovědný programátor. Při alokaci a dealokaci paměti se využívá systém garbage collection. Periodicky jsou hledány objekty, na které v programu neexistuje vazba a jim přidělenou paměť uvolňuje, nehrozí tedy vyčerpání paměti procesu v důsledku chyb programátora jako v jazycích C a C++. Ve větších aplikacích je hledání chyb typu zapomenutá dealokace velmi náročné, nemluvě o tom, že výskyt takové chyby se často neprojeví při testování ale až po delší době používání programu klientem. Garbage collection tedy elegantně vyřeší všechny problémy správy paměti a ušetřený čas můžeme efektivně využít. [4]

Obliba Javy spočívá hlavně v její přenositelnosti. Jednou zkompilovanou aplikaci můžeme tedy spouštět na různých procesorech i operačních systémech. Této vlastnosti je docíleno tím, že Java negeneruje spustitelný kód jako překladače ostatních programovacích jazyků. Místo toho generuje tzv. bajtový kód (byte-code). Je to vysoce optimalizovaná množina instrukcí navržená tak, aby mohla být spuštěna v run-time systému, jenž se nazývá virtuální stroj Javy (Java Virtual Machine, JVM). JVM provádí bajtový kód a je platformě závislý. Pokud tedy pro daný systém existuje JVM, můžeme zde spouštět naše programy. [3]

Programy napsané v Javě jsou spouštěné virtuálním strojem, jsou tedy interpretované. Obecně platí, že interpretovaný program běží podstatně pomaleji, než program přeložený do spustitelného kódu. V případě Javy však tento rozdíl není tak výrazný. Jak již bylo řečeno, byte-code je důmyslně optimalizován. Kromě toho spolu s virtuálním strojem poskytuje firma Sun také překladač JIT (Just In Time), jenž překládá bajtový kód do nativního kódu část po části podle požadavků, čímž výrazně zrychlí hlavně cykly. [3]

Jak vyplývá z uvedeného, Java představuje nástroj pro programování aplikací pro obrovskou spoustu zařízení. Je ovšem jasné, že každé zařízení nemá stejný výkon. Například mobilní telefon nedokáže konkurovat výkonu osobního počítače. Z tohoto důvodu Java podporuje následující platformy – edice pro vývoj konkrétních aplikací:

- Java ME – mobilní telefony a různá zabudovaná zařízení
- Java SE – konzolové a GUI desktopové aplikace
- Java EE – aplikace pro rozsáhlé distribuované systémy

1.2 Knihovna Swing

Již od verze Javy 1.0 je k dispozici sadu nástrojů Abstract Window Toolkit (AWT). Model AWT je však velmi restriktivní (například povoluje pouze 4 typy písma) a není objektově orientován. Tuto koncepci, počínaje verzí Javy 2, plně nahrazuje knihovna Swing. [1]

Knihovna nabízí širokou škálu komponent od jednodušších (tlačítka, popisky) po složitější (tabulky, grafy, stromy). Ne vždy je funkčnost těchto komponent dostačující pro aktuálně řešený problém. Vytváření vlastních komponent je však snadné. Model knihovny Swing je objektově orientován, může tedy být vytvořena třída, která je následníkem již existující třídy a překrýt metody, které nám u té původní nevyhovují. Ostatní metody si zachovají svou funkcionalitu a programátor dokáže s minimálním úsilím vytvořit komponentu odpovídající jeho požadavkům. Další výhodou je implementace událostního modelu. [1]

Jakákoli komponenta může vyvolat událost. Každý typ události je vyjádřen samostatnou třídou. Ta typicky nese informace o původci události a jeho hodnotách. Pro zpracování událostí používáme tzv. listenery (handlery). Jsou to třídy, které musí implementovat příslušné rozhraní. Listener může být zaregistrován vybrané komponentě. Od okamžiku registrace komponenta reaguje na příslušný typ události.

Kostrou této knihovny jsou objekty JavaBeans. Jedná se o obyčejné třídy, které by měly implementovat rozhraní Serializable a dodržovat následující konvence:

- Pro vlastnost xxx se vytvářejí dvě metody `setXxx()` a `getXxx()`. Datový typ návratové hodnoty metody „get“ je totožný s datovým typem argumentu metody „set“.
- V případě booleovských vlastností (pouze hodnoty true, false) můžeme místo metody „get“ nahradit metodou s příponou „is“. V našem případě tedy metodou `isXxx()`.

Tyto objekty jsou využívány při tzv. vizuálním programování, které výrazně urychluje návrh grafických aplikací. Komponenty lze pomocí myši přetáhnout z palety na určené místo a poté pomocí správce vlastností nastaví požadované hodnoty. Správce vlastností pracuje se „set“ a „get“ metodami objektu JavaBean. Pro přehlednost a rychlost jsou vytvářeny třídy BeanInfo. [1]

Třídu BeanInfo lze získat s pomocí nástroje Introspector a jeho statické metody `getBeanInfo()`. Jak název třídy napovídá, obsahuje informace o JavaBean, jako jsou typ a název vlastnosti, metoda k jejímu čtení a zápisu, seznam veřejných metod a metod pro zpracování a obsluhu událostí. [1]

Nástroj pro vizuální programování, designér, pomocí BeanInfo zjistí jaké vlastnosti komponenta má (vybrané vlastnosti mohou být označeny jako důležité a jsou zobrazeny jako první), a zavolá na každou vlastnost metodu čtení (get). Při změně hodnoty atributu je volána metoda zápisu (set). Tyto metody jsou volány pomocí reflexe, která je vysvětlena v kapitole 1.4.

1.2.1 ClientProperties – atribut třídy JComponent

JComponent je abstraktní třída, rozšiřující abstraktní třídu Container z balíčku java.awt. Je předkem všech „Swingových“ komponent. Bylo o ní napsáno mnoho a obsahem mé práce není její popis. Rád bych však zdůraznil atribut clientProperties. Daný atribut je implementován jako HashTable, obsahuje dvojice klíč, hodnota. Při vytváření designéru jsem narazil na potřebu uchovávat na komponentě data, která běžně nenese. Příkladem může být název sloupce tabulky, jehož hodnotu má komponenta zobrazovat a jiné. S výhodou jsem využil možnost uchovávat data právě ve vlastnosti clientProperties.

1.3 Technologie XML

Jazyk XML je fenoménem současné doby. Jedná se o rozšiřitelný značkovací jazyk (což je doslovný překlad eXtensible Markup Language). Je standardem konsorcia W3C a vychází z jazyka SGML (stejně jako například HTML). XML není programovacím jazykem, je to tzv. metajazyk, který jednoznačně určuje význam dat. Je nezávislý na platformě, ve spojení s Javou se tedy jedná o velice silný nástroj. Často je uváděna symbióza programovacího jazyka Java a značkovacího jazyka XML frází: „*Java poskytuje*

přenositelný kód, XML přenositelná data.“[1]. Nezávislosti je dosaženo zejména tím, že obsahem XML dokumentu jsou principiálně textové informace, které jsou doplněny o informaci o použitém kódování (tuto informaci obsahuje hlavička XML dokumentu). [2]

Textový soubor také nemá následující problémy typické pro binární soubory:

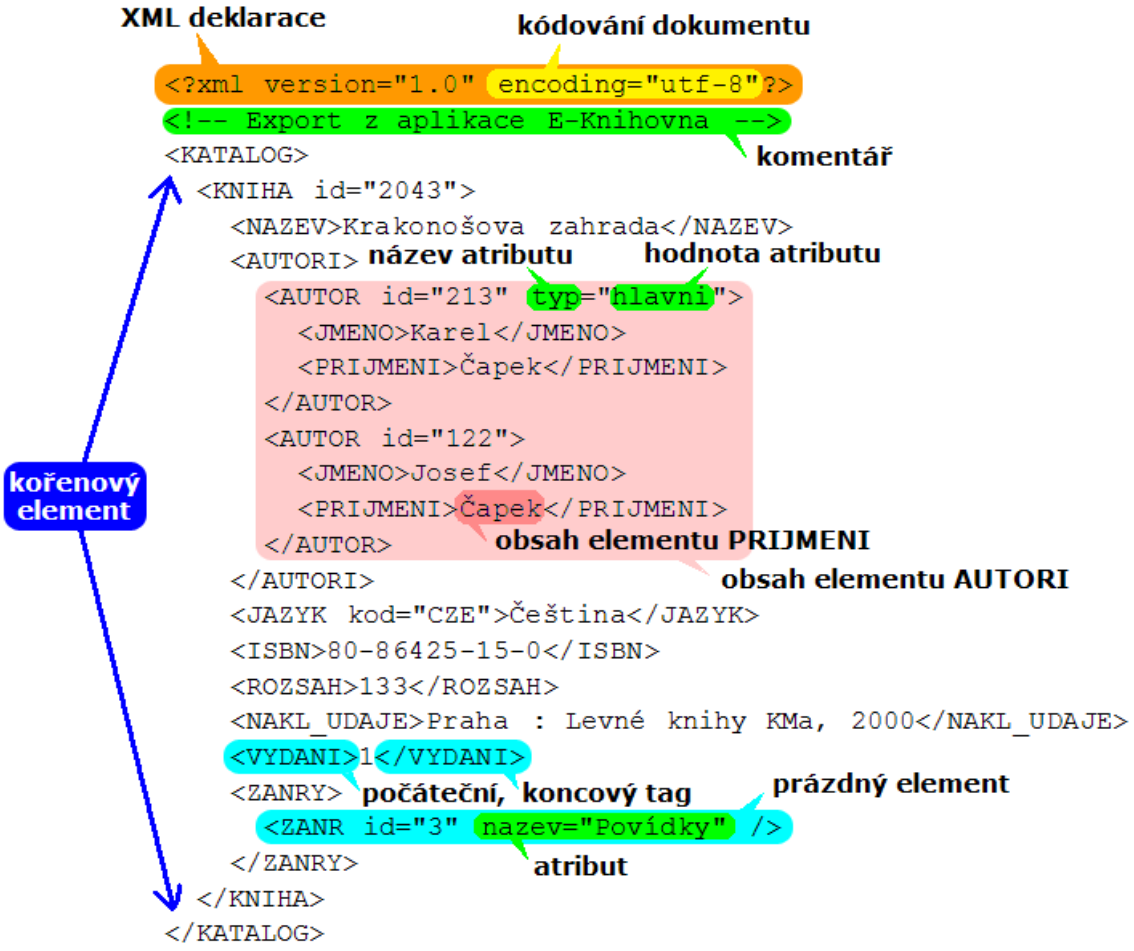
- omezený rozsah čísel (rozsah datového typu je omezený)
- typ záporných čísel (binární doplněk či první znaménkový bit)
- způsob uložení čísel (big-endian, little-endian) [2]

Rozsah využití technologie XML se neustále rozšiřuje, hlavní využití je uváděno v následujících oblastech:

- B2B (bussines-to-bussines) aplikace – XML slouží jako formát v elektronické podobě pro bezproblémovou výměnu dat mezi aplikacemi (tzv. datově orientované dokumenty)
- Elektronické publikování – knihy a dokumentace pomocí takzvaných XML dokumentů orientovaných na sdělení, popř. XHTML webové stránky
- Konfigurační, nebo i zdrojové soubory – např. konfigurační a mapovací soubory ORM Hibernate (framework pro propojení Javy s databází) [2]

XML dokumenty používané mým modulem Designer se dají zařadit do první ze skupin, v této práci se tedy zaměřím na datově orientované dokumenty. Následuje popis struktury XML dokumentů a pravidel pro jejich tvorbu. K popisu je využit obrázek 1.

[1] **HEROUT, Pavel.** *Java a XML*. 1. vyd. České Budějovice: Kopp, str. 17, 2007.



Obrázek 1 - Struktura XML dokumentu

Jedním z nejdůležitějších použitých termínů je element. Představuje počáteční a koncovou značku (tag) a všechny informace uložené mezi nimi. Každý XML dokument musí obsahovat právě jeden element, který je kořenový, obaluje všechny ostatní elementy, které jsou vnořené. Elementy do sebe mohou být teoreticky vnořeny neomezeně, příliš veliké zanoření však není doporučené, neboť se takový dokument stává nepřehledným. Elementy se dále dělí na:

- prázdné (`<a />` nebo `<a></a>`)
- párové (`<jmeno>Jan</jmeno>`)

Může se zdát, že prázdný atribut nenese žádné informace, že je zbytečný. Není to ovšem pravda. Samotná existence prázdného elementu může mít význam, nebo může být doplněn atributy. Jsou to prvky jazyka XML, které vždy představují dvojici `název="hodnota"`. Každý element může obsahovat více atributů oddělených mezerou. Atributy vždy umísťujeme do počáteční značky. Příklad využití atributů je patrný z obrázku 1. [2]

Při návrhu struktury XML dokumentu si programátor vždy musí rozmyslet, pro jaké informace je vhodné použít atribut a pro jaké vnořený element. Atributy mají kompaktnější

zápis a často je jejich zpracování podpůrnými technologiemi snazší. Jejich nevýhodou může být fakt, že u atributů nezáleží na pořadí a že se nemohou v jedné značce opakovat. Platí pravidlo, že atribut může být vždy nahrazen vnořeným elementem. Následující zápisy jsou ekvivalentní (nesou stejná data).

Kód č. 1 - Datově ekvivalentní elementy

```
<cena měna="CZK">5635.20</cena>

<cena měna="CZK" hodnota="5635.20"/>

<cena>
    <měna>CZK</měna>
    <hodnota>5635.20</hodnota>
</cena>
```

Na těchto příkladech je názorně předvedena důležitost vhodného návrhu struktury. Přílišné používání vnořených elementů vede nejen k horší čitelnosti ale i k výraznému zvýšení počtu znaků, tedy k faktu, že dva soubory se stejnými daty se výrazně liší svou velikostí na disku či jiném datovém úložišti.

Pro psaní validních XML dokumentů je nutné dodržovat následující pravidla:

- v celém dokumentu musí být právě jeden kořenový element
- je nezbytné, aby každá počáteční značka měla odpovídající koncovou značku (výjimkou je prázdný element, ten však musí končit znaky />)
- elementy se nesmějí křížit (vnořený element musí končit dříve, než jemu nadřazený element)
- elementy nemohou mít stejně pojmenované atributy
- hodnoty atributů musejí být uzavřeny v uvozovkách (případně apostrofech)
- komentáře nesmějí být vnořené ani uvnitř značek
- v datech se nesmí vyskytovat nepovolené znaky, například „<“ nebo „&“

Při nalezení chyby v HTML prohlížeč automaticky chybu opraví (doplní chybějící koncový tag aj.). Bohužel každý prohlížeč může opravy chyb implementovat jinak, což vede k nejednoznačnostem, které je možné u HTML tolerovat. V případě XML, které je spíše datově orientované nesrovnalosti nepřicházejí v úvahu. Platí zde zásada jednoznačnosti. Pokud se tedy při kontrole ukáže dokument jako nevalidní, chyba se automaticky neopravuje, pouze se vypíše místo a typ chyby. Dokud není dokument validní, není jej možné zpracovávat podpůrnými prostředky. [2]

Jak již bylo řečeno XML dokument je ve své podstatě textový soubor, lze ho zpracovávat po znacích či řádcích. To je ale značně neefektivní. Vzhledem k rozšířenosti XML jazyka

máme možnost využívat sofistikovanější technologie zpracování. Jsou jimi takzvané parsery nebo také analyzátory. Dnes jsou implementovány téměř ve všech programovacích jazycích, ani Java není výjimkou. Dvě základní skupiny parserů jsou:

- SAX (Simple API for XML)
- DOM (Document Object Model) [5]

Parser SAX je zástupcem takzvaného proudového čtení XML dokumentu. Tento způsob je také někdy nazýván událostmi řízené zpracování. SAX postupně čte XML dokument a pro každou ucelenou část vyvolá událost. Prací programátora je odchycení a zpracování těchto událostí. Výhody tohoto parseru jsou:

- velká rychlost načítání
- malá paměťová náročnost
- možnost zpracovávat dokumenty větší, než je množství operační paměti
- možnost zpracovávat dokumenty, v proudu (stream) dat

Parser SAX je mocným nástrojem, pro aplikace, které zobrazují informace z XML dokumentu a nemají důvod je měnit. Nedovoluje totiž úpravy ani přidávání nových elementů. Další nevýhodou je skutečnost, že je dokument čten sekvenčně, není možné se vracet zpět a informace potřebné později je nutné ukládat v námi zvolené organizaci paměti. [5]

Dá se říci, že parser DOM je přesným opakem předchozího. Vytvoří v paměti stromovou strukturu popisující daný XML dokument. Touto strukturou je možné volně pohybovat, cokoli měnit a přidávat další uzly. Je zřejmé, že jednou z hlavních nevýhod je paměťová náročnost. Uvádí se, že při zpracování XML dokumentu parserem DOM je požadováno až desetkrát více paměti, než je skutečná velikost dokumentu. Další nevýhodou je vyšší časová náročnost prvotního zpracování. Zmíněnou nevýhodu však vyváží skutečnost, že po vytvoření struktury jsou již operace velmi rychlé. [5]

Pro svou práci jsem si zvolil parser DOM. Pracuji se soubory, které mají řádově desítky kB, jediná nevýhoda tohoto parseru, paměťová náročnost, je zde bezpředmětná (vzhledem k velikosti dnešních pamětí). S výhodou jsem využil rekurzivního procházení stromem. V různých úrovních stromu potřebuji různé informace, pro které by v průběhu sekvenčního čtení musely být vytvářeny složité datové struktury. Při rekurzivním volání metody všechny její proměnné zůstávají v zásobníku (stack) a při návratu zpět jsou znovu přístupné.

1.4 Balík Java Reflection

V dnešních aplikacích máme často potřebu dynamické identifikace typů. Tu můžeme rozdělit na dvě kategorie:

- Tradiční, která vychází z předpokladu, že všechny typy jsou přístupné už při překladu.
- Reflexe (Reflection), který umožňuje vyhledávat informace až za běhu.

Java Reflection je silným nástrojem, který umožňuje mimo jiné zjišťovat či nastavovat hodnoty atributů a volat metody neznámých tříd. Je jasné, že takové operace se nemusí vždy podařit, většina metod poskytovaných balíkem reflection je tedy výjimečných (vyhazují výjimky, které je nutné ošetřit). Využití a síla nástrojů Java Reflection budou demonstrovány v kapitole 3.1.2. [1]

1.5 Knihovna RMI

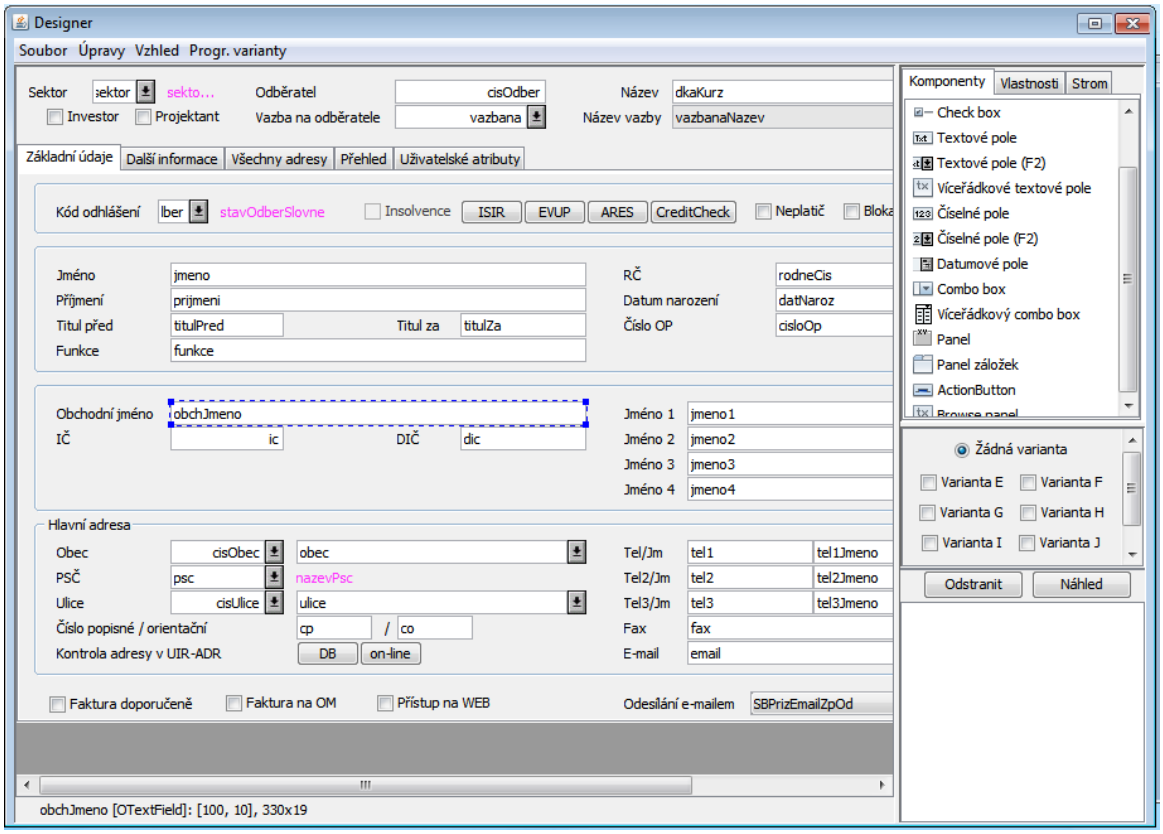
Obliba aplikací klient-server stoupá, i jazyk Java tedy musí poskytovat nástroj, pomocí kterého mohou být volány metody na vzdáleném serveru. Tím je Remote Method Invocation (RMI). Jedná se o distribuovaný objektový systém. Pomocí tříd a metod z balíčku *java.rmi* je možné realizovat klient – server aplikace bez nutnosti znalosti nízkoúrovňových síťových prostředků (např. sockety) a psaní relativně složitého kódu. Opět je to technologie, která ušetří programátorovi čas, který může využít efektivněji. RMI je dostupné již od verze JDK 1.1 a významně zvýšilo popularitu Javy.

Využívá se zde takzvaných vzdálených rozhraní, která

- jsou vždy veřejná (modifikátor public)
- jsou potomky rozhraní *Remote*
- každá metoda musí vyhazovat výjimku RemoteException

Z toho vyplývá, že programátor klienta musí dodržovat zásadu programování proti rozhraní. Na serveru tedy existuje třída (či více) implementující vzdálené rozhraní, s jehož pomocí klient oznamuje serveru, která metoda s jakými parametry má být vykonána. Po vykonání metody je klientovi vrácena hodnota. [1]

2 Modul Designér



Obrázek 2 - Vzhled Designéru

Modul Designér je komplexním nástrojem pro práci s vizuální i funkční podobou formulářů používaných aplikací Orsoft Open. Všechny zdrojové kódy jsou umístěny v balíčku s názvem *designer*. Ten obsahuje přibližně 25 tříd. Při implementaci jsem se snažil dodržovat zásady objektově orientovaného programování, každá třída řeší svůj problém a v případě potřeby volá metody ostatních tříd. Kód je relativně přehledný a k rozšíření modulu obvykle stačí upravit pouze třídy, které se týkají nové funkcionality.

2.1 Popis funkcionality

Funkce Designéru je zcela intuitivní. Uživateli je zobrazeno okno, ve kterém je schopný upravovat polohu, velikost i vlastnosti jednotlivých komponent formuláře. Samozřejmostí je také mazání či vytváření zcela nových komponent. Vše je ovladatelné myší či klávesnicí. Vzhled formuláře upravovaného v Designéru přesně odpovídá vzhledu skutečného (Designér je WISIWIG editorem). Na pozadí (skrytě před uživatelem) Designér generuje XML dokument popisující aktuální podobu formuláře. Podle tohoto dokumentu je po uložení sestavován skutečný formulář.

Designér si v paměti udržuje historii změn. Náhodné smazání komponenty neznamená vytváření nové a nastavování původních hodnot. Stačí se vrátit v historii o krok zpět

pomocí položky menu či kombinace kláves CTRL + Z. Pohyb v historii je možný v obou směrech, klávesami CTRL + Y se posuneme dopředu.

Při vytváření Designéru byla důležitým faktorem intuitivnost ovládání. Všechny klávesové zkratky jsou totožné s již zažitými (např. CTRL + C pro kopírování), uživatel se nemusí učit nové kombinace kláves. Dalším faktorem bylo především zrychlení práce programátorů. Designér je totiž nejen nástrojem pro koncového uživatele, ale i pro programátory. Inspiroval jsem se zkušenostmi ostatních kolegů v práci a implementoval možnosti, které v ostatních designérech chybí.

2.2 Srovnání s designérem v IDE NetBeans

Mnou vytvořený Designér je velmi odlišný, přesto se jej pokusím v některých aspektech porovnat. Zatímco NetBeans nabízí designér použitelný pro jakoukoli aplikaci, Designér je modulem jedné aplikace a je jí přizpůsoben. NetBeans přímo generuje Java kód a nabízí nám všechny vlastnosti, které komponenty mají. Naproti tomu Designér generuje XML dokument, pracuje pouze s komponentami, které jsou používány aplikací, a nabízí omezený počet vlastností. Jeho výhodou je však skutečnost, že pro změnu formuláře nemusíme překládat celý projekt, ale pouze použít upravený XML dokument. Umožňuje tedy úpravy za běhu aplikace. Nyní se zaměřím na funkce, které Designér má, ale v NetBeans chybí.

První z nich je možnost měnit typ komponenty. Příkladem může být situace, kdy vytvoříme JTextField (jednořádkové textové pole), umístíme jej do formuláře, nastavíme mu požadované vlastnosti. Problém nastane ve chvíli, kdy se ukáže, že vhodnější komponentou by byla JTextArea (víceřádkové textové pole). V NetBeans musíme starou komponentu smazat a vytvořit novou. Té nastavit stejné vlastnosti, jako měla původní. Tyto kroky v Designéru dělat nemusíme, protože umožňuje změnit typ komponenty. Vnitřně se samozřejmě také vytvoří nová komponenta, ale shodné vlastnosti jsou jí nastaveny automaticky podle původní.

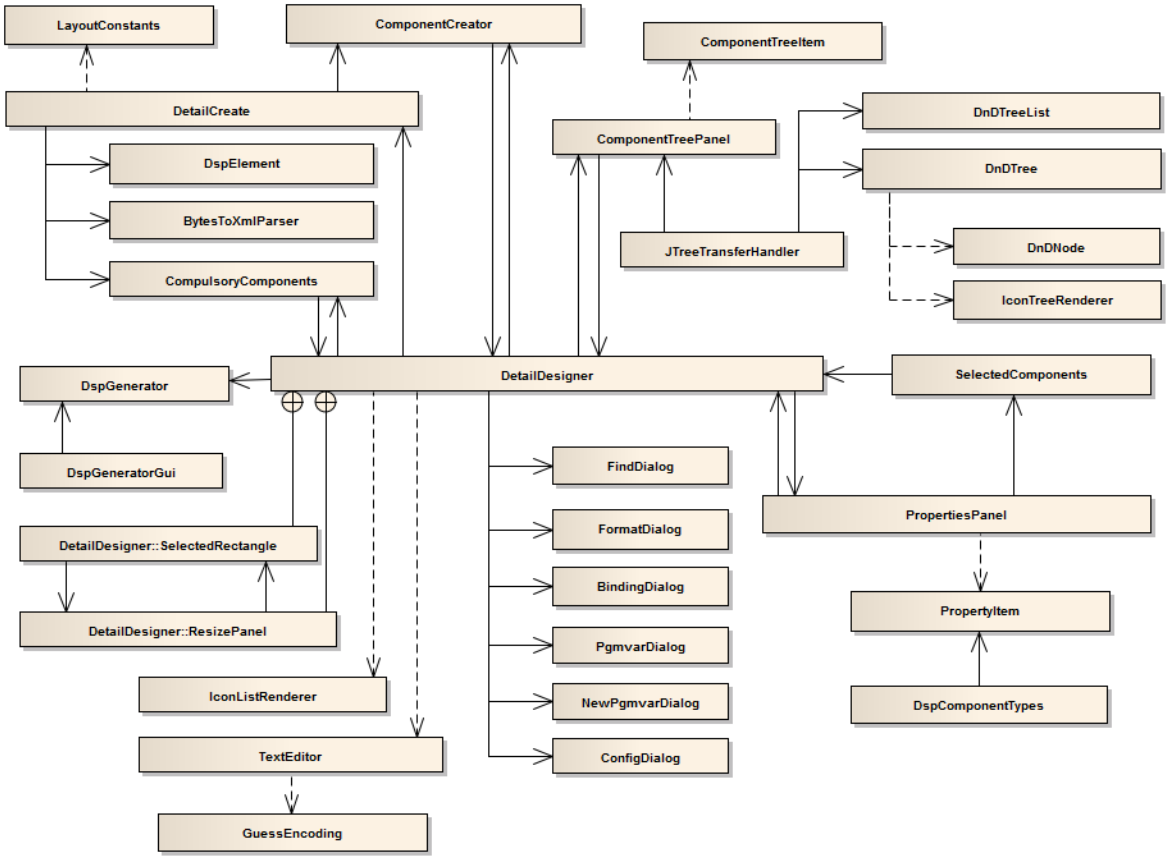
Další funkcí navíc je možnost měnit vlastnosti více komponentám najednou. Při stisknutém CTRL může být myší označeno více komponent a u všech nastavit například stejnou šířku nebo souřadnici X. Tvorba formuláře, ve kterém mají být komponenty stejně veliké a zarovnané pod sebou je tedy záležitostí okamžiku.

Poslední funkce, kterou bych rád zdůraznil, je možnost hledání komponenty podle jména. Ve větších formulářích (např. formuláře s velikým počtem záložek) se často stává, že komponentu nějaký čas hledáme, přestože známe její jméno. Z tohoto důvodu jsem implementoval metodu, která po stisku kláves CRTL + F umožní zapsání jména hledané komponenty. Po stisku klávesy ENTER je komponenta označena, pakliže existuje. Pokud je komponenta na jiné záložce, vybere se správná záložka. Nalezení komponenty je tedy také snadné.

2.3 Struktura a implementace modulu

Následuje popis struktury modulu, tedy tříd z balíčku *designer* a vazeb mezi nimi. Hlavní pozornost bude zaměřena na důležité třídy a jejich vybrané metody.

2.3.1 UML diagram tříd



Obrázek 3 - Kompletní diagram tříd

Diagram z obrázku 3 zobrazuje všechny třídy z balíčku *designer* a sleduje především vazby mezi třídami. Atributy a metody jednotlivých tříd jsou vynechány záměrně, vzhledem k omezené velikosti obrázku. Z diagramu je patrné, že hlavní třídou je DetailDesigner. Ta je jakýmsi centrálním prvkem, který zpracuje akci klienta, zavolá příslušnou metodu jiné třídy a zobrazí výsledek akce.

Název DetailDesigner je volen záměrně. Formuláře, které mohou být upravovány Designérem, zobrazují jeden řádek databázového dotazu. Například adresář je zobrazen jako tabulka. Pokud potřebuji více informací o jedné osobě, stiskem klávesy ENTER se tabulka změní na formulář, jehož pole jsou naplněna hodnotami o vybrané osobě. Tento formulář nazýváme detailem osoby. Jedná se o firemní konvenci, kterou jsem musel dodržet. Proto jsou třídy nazvány DetailDesigner či DetailCreate.

Další konvencí, kterou používám je „DSP“. Je to zkratka anglického slova „display“ a původně tak byly nazývány masky (znakové soubory), popisující formuláře ve staré

aplikaci napsané v jazyce Cobol. XML dokumenty, které generuji, jsou také maskami, používám pro ně stejné označení.

2.3.2 Třídy pro generaci XML dokumentu

První důležitou vlastností modulu je vytváření XML dokumentů. Tuto funkčnost zajišťují třídy DspGenerator a DspGeneratorGui. Druhá z nich je pouhou grafickou nástavbou první, pozornost zaměřím na první třídu.

Požadovány jsou dvě možnosti generace a to:

- generace z již existujícího formuláře
- generace z datového modelu (tabulky v databázi)

V aplikaci se nachází obrovské množství formulářů. Je nemyslitelné, aby se všechny museli navrhnout znovu. S vytvořením modulu Designér se výrazně změnila metodika vytváření nových detailů, mou povinností však byla podpora původní metodiky. Toho je docíleno metodou `createXmlFromDetail`, která postupně prochází všechny prvky formuláře a zapisuje jejich vlastnosti do obecné struktury používané Designérem. Tato struktura může být upravena a před zobrazením formuláře jsou na stávající formulář aplikovány změny.

Generování z datového modelu výrazně urychluje vytváření nových formulářů. Dříve si programátor musel navrhnout formulář v NetBeans IDE a každému prvku nastavit odpovídající vlastnosti. Prvků jsou často desítky a pro každý bylo nutné provádět stejný postup:

- přetáhnout z palety správnou komponentu na své místo
- otevřít správce vlastností
- nastavit vlastnosti
- zavřít správce vlastností

Tato činnost byla časově náročná, objevila se potřeba urychlit ji. Stačí otevřít Designér, vybrat si tabulku a označit sloupce, které je žádané zobrazovat v detailu. Je zavolána metoda `createXmlFromBrowseColumns`. Ta z datového modelu zjistí informace o vybraných sloupcích a automaticky vykoná následující činnosti:

- zjistí typ komponenty použitelný pro datový typ sloupce
- prováže pole s hodnotami sloupce (nastaví, jakou hodnotu má pole zobrazovat)
- vytvoří popisek (JLabel), který umístí nalevo od komponenty
- komponenty řadí dvousloupcově pod sebe

- místo vytváření jedenáctého řádku vytvoří novou záložku a na ní začíná znovu na prvním řádku

Zmíněná činnost netrvá déle než desítky milisekund a programátor poté pouze upraví vzhled formuláře, případně nastaví vlastnosti, které se z datového modelu vyčíst nedají.

2.3.3 Třídy pro vytváření, úpravy detailu

Funkce tříd BytesToXmlParser, DetailCreate a ComponentCreator je přesným opakem předchozích. Jejich úkolem je vytvoření či úprava formuláře přesně podle příslušného XML souboru.

BytesToXmlParser využívá parser DOM k načtení XML souboru do stromové struktury. Kontroluje správnost podaného XML a v případě úspěšných kontrol vrací referenci na kořen stromu (kořenový element XML dokumentu).

DetailCreate rekurzivně prochází podaný strom a postupně vytváří instance třídy DspElement. Jedná se o objekt, který obsahuje všechny možné atributy XML dokumentu. Je to obecný předpis, podle kterého se vytváří či edituje komponenta. Po vytvoření DspElementu jsou volány funkce třídy ComponentCreator. Existují dvě možnosti tvorby detailu:

- náhled
- skutečný detail

Takto vytvořené formuláře se od sebe nijak neliší vzhledem (podmínka WISIWIG editoru), ale výrazně se liší funkčně.

Náhled je využíván Designérem a je svým způsobem nefunkčním formulářem. Všem komponentám jsou odebrány listenery. Výsledkem je komponenta, která nepřijímá události vyvolané uživatelem, jeví se spíše jako prostý obrázek komponenty. Všechny komponenty náhledu jsou vytvářeny (tzn. žádné nejsou editovány). Dalším rozdílem je skutečnost, že některé vlastnosti nejsou komponentě nastaveny přímo, ale jsou umístěny do atributu clientProperties, o kterém je zmínka výše. Příkladem může být viditelnost. I neviditelná komponenta musí být Designérem zobrazena, aby bylo možné její nastavení měnit. Informace o viditelnosti je v tomto případě uchována nestandardně právě v HashTable, která je poskytována každým potomkem třídy JComponent.

Při vytváření skutečného formuláře je komponentám ponechána veškerá jejich funkčnost. Pokud upravuje stávající detail, předpokládáme, že všechny jeho komponenty jsou pojmenované a toto jméno je unikátní. Při úpravě tohoto detailu je pro každý DspElement zjištěno, zda je upravována existující komponenta či vytvářena nová (hledá se komponenta se shodným názvem). Pokud komponenta neexistuje, vytvoří se nová a nastaví se jí všechny vlastnosti. Jedná-li se o existující komponentu, žádná nová se nevytváří, pouze se změní vlastnosti stávající.

2.3.4 Třída TextEditor

Nyní dokážeme generovat DSP a z něj zpět vytvářet detail, stále nám však něco chybí. Musíme vytvořit aparát, který dokáže editovat XML dokument, tedy měnit vlastnosti komponent a vytvářet nové. První možností je vytvoření textového editoru. Nyní je to spíše historická záležitost, ovšem v první fázi vývoje se textový editor jevil jako správný krok. Jeho naprogramování je výrazně snazší než v případě grafického designéru, rychleji tedy bylo umožněno otestování tříd pro vytváření detailů. TextEditor vypadá jako každý jiný textový editor (např. notepad), nabízí ovšem funkce pro zjištění validity dokumentu a zobrazení náhledu.

Nevýhodou je samozřejmě skutečnost, že uživatel musí být dobře obeznámen se strukturou a možnostmi XML dokumentu. Hrozí také riziko chybných zápisů. Takto upravovat detaily by mohl pouze zaškolený člověk a to se nezdálo efektivní.

2.3.5 Třída DetailDesigner

Dalším krokem tedy bylo vytvoření grafického designéru. Uživateli je umožněno upravovat formulář myší a klávesnicí, zatímco se veškeré změny ve XML souboru dějí skrytě. Zcela odpadá nebezpečí vytváření nevalidního XML, pokud tedy není chyba na straně programu. Tyto chyby se však snadno hledají a opravují.

Designér může být otevřen ve dvou režimech:

- v programátorském
- v uživatelském

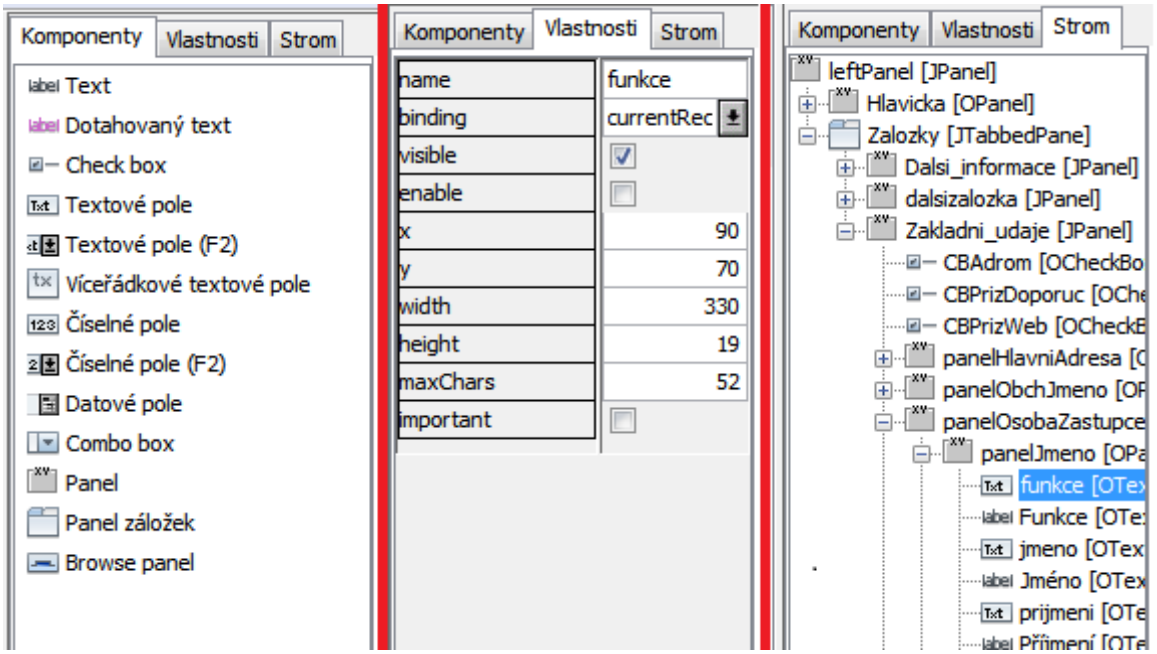
V prvním případě jsou všechny funkce přístupné. Programátor může vytvářet, upravovat i mazat cokoli. Dokonce je mu zpřístupněna možnost otevřít XML v textovém editoru a upravovat jej přímo. Důležitou vlastností, kterou poskytuje tento režim je možnost označit povinné komponenty. Ty jsou považovány za nezbytné pro daný formulář a nelze je ve druhém režimu smazat ani zneviditelnit.

Snahou druhého režimu je minimalizace chyb uživatelů. Upravovat je možné pouze vlastnosti, které neznamenají velké změny funkcí formuláře. Povinné komponenty označené programátorem nelze smazat, ani jinak poškodit (změna jména, změna sloupce, na který je pole navázáno). Komponenty však může uživatel libovolně přesouvat a měnit jejich velikost. Vzhled formuláře tedy může měnit takřka neomezeně.

Designér tedy poskytuje možnost:

- „drag and drop“ úpravy pozice a velikosti
- zobrazení palety pro vytváření nových komponent (obrázek 4)

- zobrazení a změnu vlastností (obrázek 4)
- zobrazení stromu komponent pro snazší orientaci (obrázek 4)
- vyhledání komponenty podle jména
- změnu typu komponenty
- a další

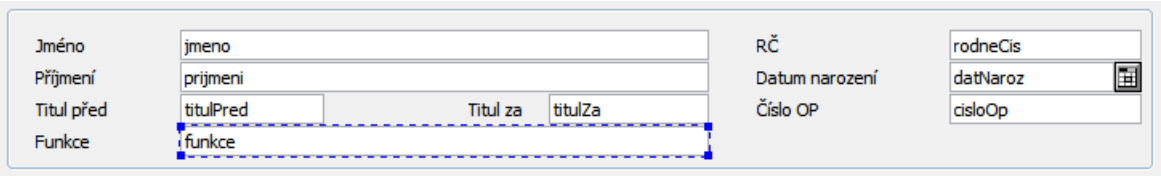


Obrázek 4 - Paleta komponent, správce vlastností, strom komponent

Strom komponent (obrázek 4) zobrazuje obrázek komponenty, její jméno a typ. Neslouží ale jen k zobrazení prvků formuláře. Cokoli je označeno ve stromě je automaticky vybráno i v náhledu formuláře (obrázek 5), dokonce můžeme přímo ve stromu komponenty přesouvat na jiné místo. Strom se dá s výhodou použít, pokud potřebujeme označit všechny komponenty jednoho panelu. V náhledu bychom je museli označovat po jednom, ale na stromě lze při stisknuté klávese SHIFT označit interval.

2.3.6 Třída SelectedComponents

Jak již název této třídy napovídá, jejím úkolem je práce s vybranými komponentami. Přesněji řečeno s vlastnostmi vybraných komponent. V Designéru pomocí levého tlačítka myši můžeme komponentu označit. Pokud je komponenta označena, zvýrazňuje ji modrý rámeček (obrázek 5). Jak již bylo řečeno, vlastnosti označené komponenty zobrazuje správce vlastností (obrázek 4). Třída SelectedComponents zajišťuje správné vyplnění vlastností. V případě změny vlastnosti (její přepsání ve správci) je též zavolána tato třída.

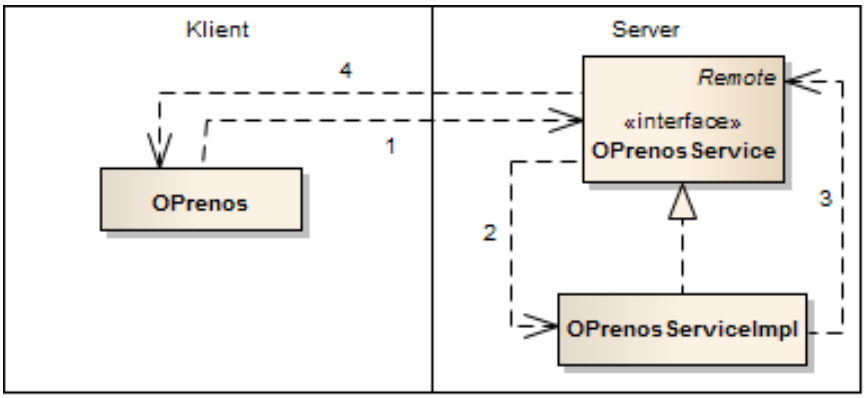


Obrázek 5 - Označení vybrané komponenty

Návrh této třídy byl velmi problematický. Už při návrhu bylo jasné, že s postupem času budou přibývat požadavky na další vlastnosti a komponenty. Snažil jsem se tedy třídu vytvořit tak, aby k jejímu rozšiřování bylo potřeba psaní co nejméně kódu. S výhodou zde využívám Java Reflection, které bylo popsáno výše. Práci s vlastnostmi komponent je věnována kapitola 3.1.2.

2.3.7 Třídy pro přesun souborů mezi klientem a serverem

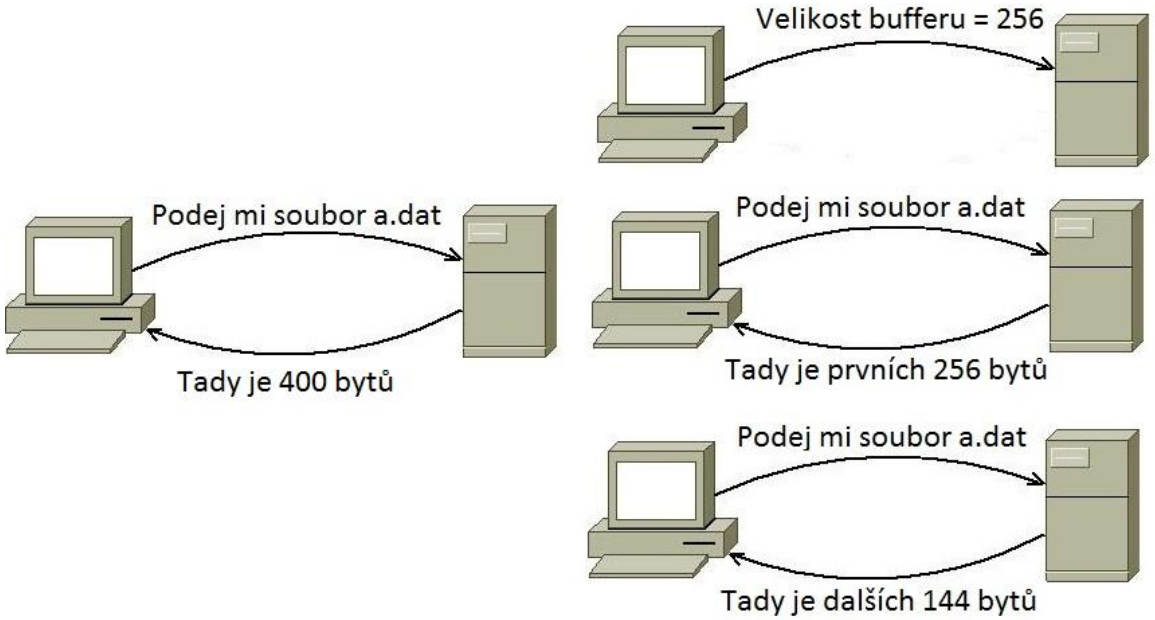
Nyní dokážeme XML soubor vygenerovat, upravovat a vytvářet podle něj formuláře. Posledním problémem je právě přesun souborů. XML soubory by měli být uloženy na serveru a odsud být stahovány, pokud jsou potřeba. Potřebuji volat metody na serveru, využiji tedy RMI.



Obrázek 6 - Třídy pro přenos souborů

Na obrázku 6 je schematicky znázorněno, jak probíhá volání metod, které podávají či očekávají soubor. Klient obdrží od serveru vzdálené rozhraní, s jehož pomocí volá metody třídy implementující toto rozhraní. Po vykonání metody na serveru je vrácen klientovi výsledek metody.

Při vytváření těchto tří tříd byla snaha o vytvoření universálního nástroje pro přesun souborů. Musel jsem tedy připravit metody pro přesun malých i velikých souborů. Malé soubory je snadné přenést v jednom poli bytů, u velikých souborů je nutné použít takzvaný buffer a posílat soubor po částech.



Obrázek 7 - Přenos souboru bez a s bufferem

Obrázek 7 popisuje, jak by mohl vypadat přesun souboru s názvem a.dat o velikosti 400 bytů. V prvním případě nám server pošle celý obsah souboru v jednom poli, ve druhém voláme metodu na serveru tak dlouho, dokud nám nepošle pole, které má menší velikost, než námi nastavený buffer. Výhodou bufferovaného přenosu je možnost přesunu jakkoli velikých souborů, nevýhodou je fakt, že soubor musí být nejprve uložen na disk klienta, teprve poté s ním můžeme pracovat.

XML dokumenty používané Designérem jsou dostatečně malé, aby bylo možné použít jednorázový přenos. Server tedy pošle na klienta pole bytů a pomocí třídy BytesToXmlParser z něj přímo v paměti vytvoříme DOM dokument (stromovou strukturu). Při ukládání vytvoříme z dokumentu pole bytů a pošleme jej na server.

3 Řešené problémy

V této kapitole se zaměříme na vybrané algoritmy a doplníme informace potřebné k plnému pochopení problému. Také si otestujeme rychlost operací a provedeme jisté optimalizace.

3.1 Popis implementovaných algoritmů

Až dosud byl popis metod jednotlivých tříd spíše laický. Vybral jsem tři zásadní algoritmy, jež nejsou zcela intuitivní, případně obsahují nestandardní postupy.

3.1.1 Drag and Drop

Drag and drop (D&D) je systém, díky kterému můžeme přesouvat objekty s pomocí myši. Obecně po nás vyžaduje tyto úkony:

- stisk tlačítka myši nad objektem, který chceme přesouvat
- posunutí myši na cílové místo
- uvolnění tlačítka

Při přesunu (drag) je zpravidla objekt neustále překreslován vzhledem k pozici myši. Každý moderní systém tuto funkcionalitu umožňuje, neboť výrazně usnadňuje práci. Příkladem může být přesun či kopírování souborů v operačních systémech. Zvlášť pro obyčejné uživatele je tento systém přístupnější, než psaní příkazů do konzole.

Java swing neposkytuje možnost přesouvání komponent, implementace je tedy na nás. V kapitole 2.3.3 jsem se zmínil o odebírání listenerů komponentám. Tento krok nám poslouží nejen k vytvoření "nefunkční" komponenty, ale i k přesouvání komponent myší. MouseEvent, který je vyvolán při jakékoli události myši nad komponentou, má totiž následující vlastnost. Pokud není odchycen přímo na komponentě, která jej vyvolala, putuje nahoru stromem komponent, až je odchycen nebo se nachází v kořeni. Naše komponenty listenery nemají, pošlou tedy vyslaný MouseEvent dále. Ten dojde až ke spodnímu panelu Designéru, který obsahuje listener pro obsluhu D&D operací:

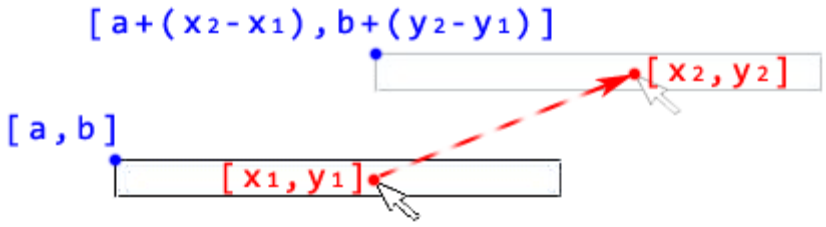
- Klik tlačítka (stisk a uvolnění)
- Stisk tlačítka
- Posun myši se stisknutým tlačítkem
- Uvolnění tlačítka

Při kliku na komponentu je komponenta označena. Jak již bylo řečeno, se stisknutou klávesou CTRL můžeme označit libovolné množství komponent. Tyto komponenty

budeme přesouvat najednou D&D musí být schopen pracovat obecně s neznámým počtem komponent.

Pokud nad komponentou stiskneme levé tlačítko myši, dojde k přípravě všech proměnných využívaných D&D systémem. Jsou jimi pozice myši a pozice všech označených komponent. Jedná se o výchozí (vztažný) stav, s jehož pomocí dopočítáme všechny pozice při následující události.

Tou je přesun. Při pohybu myši se stisknutým tlačítkem je periodicky vytvářena událost. Ta nese mnoho informací, nás zajímá pouze aktuální poloha myši. Výpočet polohy komponenty zobrazuje obrázek 8. Z matematického hlediska přičítáme k bodu vektor, posouváme tedy tento bod ve směru tohoto vektoru. Předešlý výpočet je v cyklu aplikován na každou komponentu. Vždy používáme vztažné body, vzniklé při stisknutí myši. Využíváme zde globálních souřadnic (souřadnice přímo na obrazovce).



Obrázek 8 - Výpočet nové pozice při přesunu komponenty

Pokud jsou komponenty na správném místě, uvolníme stisknuté tlačítko. Pro každou komponentu je zjištěno nad jakým panelem se nachází (kam ji chceme pustit) a jsou přepočítány její souřadnice tak, aby pozice přesně odpovídala lokálním souřadnicím panelu.

3.1.2 Práce s vlastnostmi komponent

Jak již bylo uvedeno, pro práci s vlastnostmi komponent slouží třída SelectedComponents (kapitola 2.3.6). Rád bych na její funkci demonstroval sílu nástroje Java Reflection. V případě kvalitního návrhu nám tato technologie nám umožňuje výrazné zkrácení kódu a zároveň jeho snadné doplňování.

Nejprve si ukážeme, jak by bylo možné vyřešit problém bez pomoci reflexe. Pro každý typ komponenty bychom připravili blok programu. Zdrojový kód č. 2 zobrazuje strukturu metody, která nastavuje jakoukoli vlastnost libovolné komponentě. Tento návrh je zcela intuitivní, ale má také velké množství nevýhod. Taková metoda bude obsahovat obrovské množství kódu, ve kterém bude mnoho velmi podobných bloků. Ty se budou lišit pouze v drobnostech, programátor tedy bude kopírovat bloky a přepisovat právě tyto odlišnosti. Vzniká tak obrovský prostor pro chyby z přehlédnutí. Další nevýhodou je skutečnost, že jsou vzhledem k požadavkům firmy vlastnosti postupně doplňovány. Je pravděpodobné, že nová vlastnost nebude platná pouze pro jednu komponentu, ale pro větší množství komponent. Následkem této skutečnosti je nutnost přidání dalších bloků kódu do různých částí metody, což znovu zvyšuje riziko výskytu chyby.

Kód č. 2 - Práce s vlastnostmi bez využití reflexe

```
public void setProperty(Component comp, String propName, Object propValue) {
  if (comp instanceof JTextField) {
    //upravuji vlastnost textového pole
    if (propName.equals("name")){
      ((JTextField)comp).setName((String)propValue);
    }
    if (propName.equals("text")){
      ((JTextField)comp).setText((String)propValue);
    }

    // další vlastnosti...
  }
  // další komponenty
}
```

Je zřejmé, že musí existovat snazší způsob. Tím je právě využití reflexe. Zdrojový kód č. 3 zobrazuje metodu, funkčně zcela totožnou s předchozí. Zatímco v předchozím případě byla zobrazena pouze malá část metody, zde je celá. Zdrojový kód je podstatně kratší a přehlednější. Využíváme konvence názvů setterů, víme tedy, jak se jmenuje metoda, která nastavuje daný atribut. Bohužel ani tento návrh není dokonalý. Metoda `getMethod()` potřebuje přesnou třídu. Pokud používáme rozhraní, nebo potomka očekávané třídy, námi hledaný setter nebude existovat. Problémy nastanou také v případě primitivních typů.

Kód č. 3 - Práce s vlastnostmi s pomocí reflexe

```
public void setProperty(Component comp, String propName, Object propValue) {
  if (comp != null && propName != null) {
    try {
      //příprava názvu metody podle konvence
      String methodName = "set" + propName;

      //nastavení vlastnosti komponentě
      Class clazz = comp.getClass();
      Method method = clazz.getMethod(methodName, propValue.getClass());
      method.invoke(comp, propValue);

    } catch (NoSuchMethodException ex) {
      System.out.println("Komponenta nemá vlastnost: " + propName);
    } catch (IllegalArgumentException ex) {
      System.out.println("Špatná hodnota argumentu");
    } catch (Exception ex){
      System.out.println("Jiná výjimka");
    }
  }
}
```

Řešení předchozích problémů a další zkrácení zdrojového kódu nám nabízí třída PropertyUtils. Není součástí distribuce Javy, ale není problém si stáhnout .jar soubor, který ji obsahuje. Ten následně může být přidán do našeho projektu. Zdrojový kód č. 4

zobrazuje, jak využít tuto třídu pro náš problém. Takto navržená metoda obslouží všechny komponenty a všechny jejich vlastnosti. Pokud se rozhodneme přidat vybraným komponentám vlastnost, nemusíme metodu nijak měnit.

Kód č. 4 - Práce s vlastnostmi pomocí PropertyUtils

```
public void setProperty(Component comp, String propName, Object propValue) {
  if (comp != null && propName != null) {
    try {
      //nastavení vlastnosti komponentě
      PropertyUtils.setProperty(comp, propName, propValue);
    } catch (NoSuchMethodException ex) {
      System.out.println("Komponenta nemá vlastnost: " + propName);
    } catch (IllegalArgumentException ex) {
      System.out.println("Špatná hodnota argumentu");
    } catch (Exception ex) {
      System.out.println("Jiná výjimka");
    }
  }
}
```

Metoda getProperty bude vypadat obdobně, pouze využijeme metody `PropertyUtils.getProperty()`. Její návratový typ bude Object.

Poslední, co zbývá připravit je třída, která bude obsahovat seznam všech komponent a k nim všechny povolené vlastnosti. Tou je třída DspComponentTypes. Ke každé vlastnosti ještě doplňuje informaci o datovém typu vlastnosti. Podle této třídy sestaví Designér správce vlastností. Při označení komponenty se volá metoda `getProperty()`. Při změně hodnoty jakékoli položky ve správci vlastností je zavolána metoda `setProperty()`, které se v parametru předá nová hodnota.

3.1.3 Práce s programovými variantami

Dalším požadavkem firmy je možnost úprav takzvaných programových variant. Jedná se o rozdílné zobrazování formulářů pro různá data. Příkladem může být například tabulka odběratelů. Ty dělíme na velkoodběratele a maloodběratele. U každé skupiny nás zajímají jiné položky. Pokud posledním vybraným byl maloodběratel a nově vybraným je velkoodběratel, programátor zavolá metodu, která překreslí formulář podle nově nastavené programové varianty.

Až doposud tyto překreslení musel každý programátor psát přímo do kódu. Díky existence aparátu pro změnu vzhledu formuláře za běhu stačí zavolat mnou připravenou metodu a předat ji v parametru novou programovou variantu (programová varianta je reprezentována znakem). Jak ale Designér přizpůsobit?

Nejdříve připravíme strukturu XML dokumentu, která dokáže popsat jedné komponentě více možností. Původní struktura už je otestována a bude-li to možné, měli bychom se vyhnout zásadním změnám. Řešením, které se ukázalo dokonale funkční, je duplikovat

pod sebe více elementů se stejným názvem komponenty, ale různou hodnotou atributu `pgmvar`. Nyní naše struktura dokáže popsat programové varianty.

Kdybychom nic nepřidali do tříd pro úpravu či generaci detailu, aplikovala by se postupně každá programová varianta komponenty a vždy by aktivní zůstala ta poslední. Musíme tedy aplikovat vždy jen první z variant (první varianta je vždy označena mezerou a je výchozí) a ostatní pro úpravy prozatím vynechat. Abychom nemuseli při každé změně varianty znovu procházet celý strom XML dokumentu, uložíme si všechny položky a jejich varianty do struktury ArrayList. Při úpravě varianty budeme procházet položky tohoto seznamu, což výrazně urychlí samotné překreslení detailu. Tyto komponenty nazýváme variantní.

Pokud se změní programová varianta, projdeme všechny variantní komponenty. Pokud má komponenta nově nastavenou variantu, aplikuje se. Pokud se tak nestalo, použije se varianta výchozí (tzv. mezerová). Programátor se nestará o překreslování, to je plně v režii našeho nástroje.

Zbývá poslední krok pro plnou podporu programových variant. Tím je příprava Designéru. Musíme se vypořádat s tím, že jedna komponenta v náhledu představuje více komponent s různými vlastnostmi. Opět s výhodou využijeme atributu clientProperties (kapitola 1.2.1). Do tohoto atributu vložíme HashMapu (kolekce dvojic klíč - hodnota). Jejím klíčem bude programová varianta a hodnotou bude kopie této komponenty. Tato kopie ovšem bude mít vlastnosti, které odpovídají programové variantě v klíči. Každá komponenta tedy "ponese" informace o všech jejích programových variantách.

Na obrázku 3 je zobrazen vzhled Designéru. Pod paletou komponent si můžeme všimnout panelu pro změnu aktuální programové varianty. Po zaškrtnutí některé z variant Designér najde komponenty, které mají tuto variantu, a aplikuje změny. Pro snazší orientaci mají komponenty s touto variantou zelený rámeček, jsou tedy na první pohled rozeznatelné od ostatních. Jakákoli vlastnost upravená těmto komponentám se nyní zapisuje do naší HashMapy, přesně na komponentu, která náleží námi vybrané programové variantě.

Upravovat programové varianty může pouze programátor, uživatel k této vlastnosti Designéru nemá přístup.

3.2 Ukázka XML souboru popisujícího formulář

Nyní si představíme jednoduchý XML dokument, vysvětlíme si význam jednotlivých tagů a ukážeme si, jak vypadá detail, který tento XML dokument popisuje. Následující výklad bude popisovat zdrojový kód č. 5.

Kód č. 5 - Ukázka XML popisujícího formulář z obrázku 9

```
<?xml version="1.0" encoding="UTF-8"?>
<DSP type="0" length="712" height="503" entityClass="cz.ortex.model.O6a">
 <HEADER name="hlavicka" length="712" height="60" direction="north" >
  <dkaIco1 row="20" col="70" length="60" height="19" name="dkaIco1" type="9" format="##
  <nema_binding row="20" col="10" length="22" height="20" name="jLabel1" type="l" parer
  <dkaIco2 row="20" col="140" length="70" height="19" name="dkaIco2" type="9" format="#
  <nema_binding row="20" col="220" length="60" height="20" name="jLabel11" type="l" par
  <dkaKurz row="20" col="290" length="170" height="19" name="dkaKurz" type="x" parent="
 </HEADER>
 <TABS name="jTabbedPane1" row="60" col="0" length="712" height="443" direction="center
  <TAB title="Základní údaje" index="1" nextTab="2" name="jPanel3" >
   <GBOX row="20" col="10" length="320" height="110" title="" name="jPanel8" visible="t
    <nema_binding row="20" col="10" length="36" height="20" name="jLabel14" type="l" pa
    <dkaName1 row="20" col="70" length="240" height="19" name="dkaName1" type="x" parer
    <nema_binding row="40" col="10" length="27" height="20" name="jLabel18" type="l" pa
    <dkaStr row="40" col="70" length="240" height="19" name="dkaStr" type="x" parent="j
    <dkaPlzNase row="60" col="70" length="80" height="19" name="dkaPlzNase" type="X" pa
    <nema_binding row="60" col="160" length="45" height="20" name="jLabel20" type="l" p
    <dkaPostf row="60" col="210" length="100" height="19" name="dkaPostf" type="x" pare
    <dkaOrt row="80" col="70" length="240" height="19" name="dkaOrt" type="x" parent="j
    <nema_binding row="80" col="10" length="34" height="20" name="jLabel19" type="l" pa
    <nema_binding row="60" col="10" length="24" height="20" name="jLabel22" type="l" pa
   </GBOX>
  </TAB>
  <TAB title="Parametry" index="2" nextTab="3" name="jPanel5" >
  </TAB>
  <TAB title="Bankovní spojení" index="3" nextTab="4" name="OPanel0" >
  </TAB>
  <TAB title="Faktury" index="4" name="jPanel1" >
  </TAB>
 </TABS>
</DSP>
```

Kořenovým elementem každého XML souboru vygenerovaného Designérem je element DSP. Ten nese informaci o výchozí velikosti celého formuláře a o tzv. entitní třídě. Je to třída, jejíž struktura odpovídá tabulce, kterou výsledný formulář zobrazuje. V tomto případě je to adresář (tabulka adresáře má název O6a). DSP obsahuje dva vnořené elementy:

- HEADER - hlavička detailu
- TABS - panel záložek

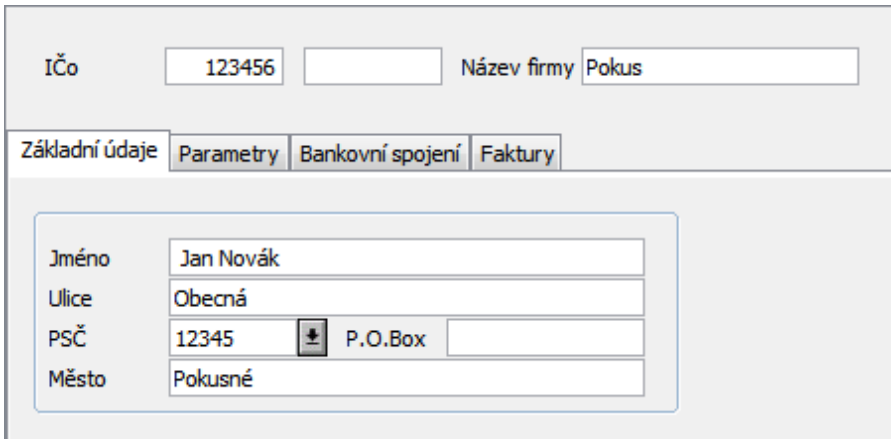
Do hlavičky se typicky dávají důležité informace, které chceme zobrazovat neustále. Je charakterizována výškou, šířkou a samozřejmě názvem. Naše hlavička obsahuje pět položek. První dvě jsou popisky. To poznáme podle jména elementu (nema_binding) a také podle atributu `type="l"`. Další tři komponenty jsou textová pole. Dvě z nich jsou numerická (`type="9"`), třetí je textové (`type="x"`). Jména elementů označují název sloupce v tabulce O6a, jehož hodnotu chceme zobrazovat ve formulářovém poli.

Panel záložek obsahuje tři záložky. To poznáme podle počtu vnořených tagů se jménem TAB. Každá záložka má své pořadí a popisek. Atribut name zde odpovídá jménu

komponenty, se kterou záložka propojena (zpravidla JPanel). Pouze první záložka obsahuje nějaké položky, ostatní jsou prázdné.

Dalším značkou je GBOX. Je to zkratka z anglického group box. Jak již název napovídá, jedná se o panel, který obsahuje skupinu komponent, které spolu nějakým způsobem souvisí. Důležitými atributy jsou umístění, velikost a popisek. Pokud GBOX neobsahuje atribut `border="false"`, bude mít rámeček (jako v našem případě).

Uvnitř GBOXu se nachází deset položek. Pět z nich jsou obyčejné popisky a dalších pět jsou opět textová pole pro zobrazení hodnot z databáze. Formulář, který jsme si právě popsali, vypadá přesně jako na obrázku 9.



Obrázek 9 - Formulář popsaný XML souborem z kódu č. 5

Překvapivý může být vzhled pole pro zobrazení PSČ. Na začátku této práce jsem se zmínil možnosti vytváření vlastních komponent odvozených od těch z balíčku Swing. Tato komponenta je odvozena od JPanelu, který obsahuje dvě položky – textové pole a tlačítko. Navenek však vystupuje jako jediná komponenta. Tuto komplexní komponentu označuje atribut `type="X"`. Další nestandardní komponentou je pole zobrazující IČo. Jedná se o textové pole, které dokáže zobrazovat pouze číselné hodnoty, typicky zarovnává text doprava.

Obecně pracujeme s rozsáhlejšími formuláři. Tento byl vytvořen pro účely výkladu, což je důvodem absence variantních položek. Také zanoření není veliké. Tato struktura nám však umožňuje prakticky libovolné zanoření (záložka může obsahovat panel záložek, popř. group box může být uvnitř tagu GBOX), díky čemuž dokážeme popsat většinu běžných formulářů.

3.3 Rychlost zásadních operací a celková odezva

Všechny požadavky na Designér jsou nyní splněny. Dalším důležitým krokem je optimalizace kritických součástí, tedy generace XML souboru a úprava či vytvoření formuláře. Prvním krokem je zjištění doby potřebné k těmto operacím.

Princip měření času je jednoduchý. Do proměnné si uložíme čas před zavoláním metody a po skončení metody jej odečteme od aktuálního času. Pro zjištění aktuálního času nám Java nabízí tyto statické metody:

- `System.currentTimeMillis()` - čas v milisekundách
- `System.nanoTime()` - čas v nanosekundách

První z nich je pro naše potřeby nepoužitelný. Čas totiž měří přibližně po 16 milisekundách. 15 milisekund tedy naměří jako 0 milisekund. Měření v nanosekundách je podstatně přesnější.

Nejprve se zaměříme na generování XML dokumentu. Po provedení 100 měření se ukazuje, že průměrná doba generace je přibližně 9 milisekund. Maximální doba nikdy nepřekročila 20 ms. Tyto časy jsou prakticky zanedbatelné, nevzniká tedy potřeba optimalizace.

Jiná situace nastala při testování úpravy detailu. Časy se zde pohybovali v rozmezí 80 a 3500 milisekund. Je nepřístupné, aby uživatel čekal tři sekundy na zobrazení formuláře. Náš úkol bude nalezení bloku kódu, který zabere takto dlouhou dobu. Ukazuje se, že hledanou metodou je ta, která upravuje pořadí záložek. Designér totiž umožňuje změny indexů záložek a při úpravě detailu je správně seřadí. Metoda odebere stávající záložku z panelu záložek a pomocí metody `insertTab()` ji vloží na správné místo. Právě metoda `insertTab()` je kamenem úrazu. Musíme se pokusit o její vynechání.

Řešením je vytvoření instance třídy ArrayList, která bude nahrazovat panel záložek. Jejími prvky budou dvojice popisek, panel. Z této struktury odebereme dvojice, jejichž pořadí se změnilo a umístíme je na správné místo. Z panelu záložek odstraníme všechny záložky a pomocí metody `addTab()` je znovu přidáme ve správném pořadí z naší instance třídy ArrayList. Použití metody `insertTab()`, která umí přidávat záložky na libovolné místo, se vyhneme použitím metody `addTab()`, která vkládá záložku na konec.

Takto upravená metoda se ukázala podstatně rychlejší. Časy úpravy detailu se nyní pohybují mezi 40 a 250 milisekundami v závislosti na množství komponent formuláře. Toto časové rozmezí je již tolerovatelné, další optimalizace již nejsou potřebné.

Závěr

Výsledkem této bakalářské práce je kompletně funkční aparát pro editaci vzhledu formulářů používaných v aplikaci Orsoft Open. Tato práce významně mění metodiku vytváření formulářů. Není již používán designér dodávaný v NetBeans IDE, ale využívá se možnost generace XML souboru z datového modelu a následné rozmístění jednotlivých prvků formuláře pomocí grafického editoru.

Modul umožnil zobecnění velkého množství postupů. Důsledkem této skutečnosti je výrazné zrychlení tvorby formulářů a snížení počtu opakujících se kódů. Programátoři tedy mají více prostoru pro vývoj samotné aplikace.

Pro ověření podstatných částí modulu byla provedena měření časové náročnosti. Při testování metody pro úpravu textových formulářů se ukázalo, že čas potřebný pro aplikování změn na existující formulář překračuje v některých případech 3 sekundy. Chyba byla lokalizována a pomocí vhodné optimalizace odstraněna.

V blízké době bude tento modul přesunut z vývojové do distribuční verze aplikace. Takto se modul dostane ke koncovým zákazníkům. Je velmi pravděpodobné, že tito uživatelé budou mít množství požadavků na rozšíření funkcí Designéru. Při vyřizování většiny požadavků by se neměly objevit žádné problémy, vzhledem k použití kvalitního návrhu struktury XML souboru.

Do budoucna je očekáván další vývoj, především v oblasti rozšiřování vlastností komponent a automatizace dalších funkcí. Velká část práce bude využita také pro tvorbu plánované webové verze aplikace. Vzhled formulářů v obou aplikacích by měl být totožný. Vzhledem k této skutečnosti se přímo nabízí využití již existujících XML souborů, popisujících rozložení formulářových prvků. Odpadá tak práce při navrhování vzhledů formulářů. Další výhodou je skutečnost, že i uživatelem upravený formulář bude v obou verzích aplikace zobrazován totožně. Obě aplikace budou synchronizovány díky společné databázi XML souborů.

Zadání práce bylo bez výhrad splněno, v mnoha případech byly řešené problémy nad rámec zadání.

Literatura

[1] **ECKEL, Bruce**. *Myslíme v jazyku Java: knihovna zkušeného programátora*. 1. vyd. Praha: Grada, 2001, 470 s. ISBN 80-247-0027-1.

[2] **HEROUT, Pavel**. *Java a XML*. 1. vyd. České Budějovice: Kopp, 2007, 313 s. ISBN 978-80-7232-307-4.

[3] **SCHILDT, Herbert**. *JavaTM2: příručka programátora*. Brno: SoftPress, 2001, 552 s. ISBN 80-86497-04-6.

[4] **GRAND, Mark**. *Java: referenční příručka jazyka*. 2. vyd. Praha: Computer Press, c1998, 475 s. ISBN 80-7226-071-5.

[5] **HAROLD, Eliotte Rusty a W MEANS**. *XML v kostce: pohotová referenční příručka.* 1. vyd. Praha: Computer Press, 2002, 439 s. ISBN 80-7226-7 12-4.