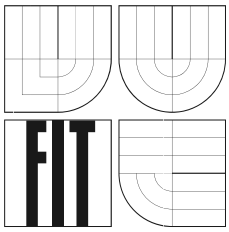
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



Tvorba informačních systémů v jazyku Python

Bakalářská práce

2006 **Marek Schmidt**

Tvorba informačních systémů v jazyku Python

Odevzdáno na Fakultě informačních technologií Vysokého učení technického v Brně
dne 27. dubna 2006

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Aleše Smrčky. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

. . . . . .
Marek Schmidt
27. dubna 2006

Abstrakt

Práce se zabývá zmenšením propastí mezi tvorbou informačních systémů jakožto desktopových nebo webových aplikací a vytvořením prostředí pro tvorbu tzv. nezávislých informačních systémů. Obsahem práce je návrh a implementace knihovny v jazyku Python, která umožňuje vytvářet programy nezávisle na formě uživatelského rozhraní. Definuje programové rozhraní pro "nezávislé uživatelské rozhraní" a implementuje jej v třech různých "front-endech" – Webovém, desktopovém na knihovně GTK a na platformě Mozilla v technologii XUL.

Klíčová slova

nezávislé informační systémy, python, webové aplikace, Apache, mod_python, GTK, XUL

Abstract

This work tries to close a gap between development of information systems as desktop or as web applications and to create a framework for development of ‘independent information systems’. It consists of a design and its implementation in a Python library, which allows development of programs independent of the form of user interface. It defines a programming interface for ‘independent user iterface’ and implements it using three distinct ‘front-ends’ – Web-, GTK- and XUL- front-ends.

Keywords

independent information systems, python, web applications, Apache, mod_python, GTK, XUL

Obsah

Kapitola 1

Úvod

Cílem této práce je vytvořit knihovnu pro tvorbu informačních systémů v jazyku Python. Existuje mnoho nástrojů pro tvorbu informačních systémů, ať už jako desktopových aplikací, nebo jako webových aplikací. Tato práce se zabývá možností spojení tvorby informačních systémů jakožto desktopových nebo jako webových aplikací – tedy tvorbou *nezávislých informačních systémů*. Systém vytvořený za pomoci této knihovny se tak bude moci použít jako desktopová i jako webová aplikace beze změny zdrojového kódu.

V první části této zprávy se pokusím představit jazyk Python, popsat základní pojmy objektově orientovaného programování a pokusím se nalézt základní rozdíly mezi desktopovými a webovými aplikacemi.

V druhé části pak navrhnu knihovnu pro tvorbu nezávislých informačních systémů. Specifikuji požadavky, navrhnu její architekturu a základní principy.

Třetí závěrečná část popisuje implementaci knihovny a její tři ‘front-endy’.

Kapitola 2

Současný stav

2.1 Jazyk Python

Jazyk Python je dynamický objektově orientovaný jazyk. Mezi jeho vlastnosti patří:

- Snadná a čistá syntaxe – Není zbytečně "ukecaný". Programátor tak dosahuje vyšší produktivity práce. Syntaxe využívá i bílé znaky, čímž se snaží vnutit programátorovi přehledný styl zápisu programu. Programy napsané v jazyku Python jsou tak snadno čitelné.
- Dynamicky typovaný – Typová kontrola probíhá až za běhu programu.
- Objektově orientovaný – Umožňuje i vícenásobnou dědičnost.
- Modulární – Podporuje i hiearchické jmenné prostory, vhodný i pro rozsáhlé projekty.
- Mocná standardní knihovna – Obsahuje prostředky pro tvorbu webových i desktopových aplikací, přístupu k databázím, podporu různých síťových protokolů, práci s XML a jiné.
- Svobodný – Existuje svobodná implementace portovaná na většinu důležitých operačních systémů. Programy napsané v Pythonu jsou tak jednoduše přenositelné.

Pro své vlastnosti se využívá především jako prototypovací nástroj, nástroj pro rychlý vývoj aplikací (RAD) nebo pro skriptování aplikací.

2.2 Objektově orientovaný přístup při tvorbě aplikací

Aplikace viděná objektově orientovaným pohledem není tvořena ničím jiným, než množinou vzájemně se ovlivňujících *objektů*.

Objekty mají tři vlastnosti:

1. Identita – To, co odlišuje objekty od sebe.
2. Stav – Určen atributy. Atributy mívají jméno (barva, velikost), a hodnotu (červená, 7). Hodnotou atributu může být také odkaz (reference) na jiný objekt.
3. Chování – Určeno metodami. Metody bývají obvykle totéž co procedury v strukturovaně orientovaném programování s tím, že jako jejich první argument je samotný objekt. Metoda tak může upravovat stav objektu nebo volat metody jiných objektů (také nazýváno jako "posílání zpráv jiným objektům").

Objektově orientovaný přístup je vhodný zejména v případě, že smyslem aplikace je modelovat reálné nebo konkrétní objekty. Knihovny pro tvorbu desktopových aplikací bývají většinou objektově orientované, protože tak mohou přirozeně pracovat s objekty uživatelského rozhraní, jako jsou okna, tlačítka, textová pole a podobně.

2.3 Tvorba dynamických webových stránek

Systém WWW (World Wide Web – zkráceně webu) je založen na architektuře klient-server. Server poskytuje hypertextové dokumenty, obvykle ve formátu HTML (HyperText Markup Language). Dokumenty jsou na serveru identifikovány pomocí URL (Uniform Resurce Locator) a vzájemně jsou propojeny pomocí hypertextových odkazů. Klienti se serverem komunikují pomocí protokolu HTTP (HyperText Tranfer Protocol).

Klient (obvykle webový prohlížeč) obdrží od uživatele URL a na základě té vznese HTTP požadavek na daný dokument konkrétnímu HTTP serveru (součástí URL je také adresa serveru). Server pak odpoví HTTP odpovědí, která obsahuje buď to požadovaný dokument nebo chybu.

Dynamické webové stránky tento systém rozšiřují o dva další principy:

1. Na dané URL se nenachází konkrétní dokument, na místo toho se tam nachází skript, který je spuštěn a teprve jeho výstup je poslán klientovi jako výsledný dokument. Toto probíhá pro klienta naprosto průhledně.

2. Dokumenty mohou obsahovat formuláře, kterými může klient odeslat data na server.

Skript je tedy program, který je na serveru spuštěn při každém HTTP požadavku, jeho činnost se dá shrnout do tří kroků:

1. Zpracuje vstupy z formulářů, v závislosti na použité metodě HTTP požadavku.

2. Provede užitečnou práci, například ve spoluprácí s databází.

3. Vytvoří výsledný dokument, který se pošle klientovi.

Protokol HTTP byl navržen jako bezestavový. Každý HTTP požadavek je tedy nezávislý na ostatních. Skripty však často potřebují navázat na předchozí požadavky a uchovat data v průběhu *relace* – po sobě následujících požadavků stejného klienta.

Webový server má prakticky dvě možnosti, jak uchovat data v průběhu relace:

- Uchovávat data u klienta – V takovémto případě musí zakódovat data do řetězce a tento řetězec uložit do formuláře jako skrytý HTML prvek *input*. Webový prohlížeč na straně klienta pak při dalším odeslání formuláře odešle formulář i se zakódovanými daty.

  Tato možnost není vhodná z několika důvodů: Data jsou přenášena zbytečně ze serveru na klienta a pak zase zpět. Server si musí dále dávat velký pozor na to, jaká data mu přišla, protože klient měl možnost data přečíst a případně pozměnit.

- Uchovávat data na serveru – Pak stačí klientovi odeslat klíč relace, který identifikuje danou relaci. Klient pak posílá relační klíč při každém dalším požadavku. Podle tohoto klíče pak server najde v úložišti data patřící této relaci.

2.3.1 Existující nástroje pro tvorbu webových aplikací

Nástrojů pro tvorbu dynamických webových stránek, či vznešeněji řečeno *webových aplikací*, existuje celá řada. Zde se budu zabývat především těmi založenými na jazyku Python.

CGI

Common Gateway Interface (CGI) je standardní rozhraní pro použití externích programů z webových serverů. Programy zpracovávají standardní vstup a zapisují na standardní výstup ve speciálním formátu, který je předepsán CGI. Výstup musí obsahovat hlavičku, ve které popisuje svůj výstup nebo posílá příkazy serveru.

Výhoda použití CGI je v tom, že je to standard – lze použít jakýkoliv program napsán v libovolném jazyce. Psaní programů s CGI rozhraním je také snadné, protože se nemusí zabývat síťovými protokoly, stačí jen zapisovat na standardní výstup a o vše ostatní se postará HTTP server.

Nevýhoda použití CGI je v pomalosti. Pro každý nový požadavek je totiž potřeba vytvořit nový proces. Problém pomalosti řeší například rozšíření FastCGI.

Standardní knihovna Pythonu obsahuje modul cgi, který ulehčuje tvorbu CGI skriptů.

Tvorba webových aplikací s Apache a mod_python

Mod_python je modul, který integruje interpret Pythonu do HTTP serveru Apache. Protože tento interpret je spuštěn přímo v procesu obsluhujícící HTTP požadavky, může dosahovat větší rychlosti zpracování požadavků než skripty spuštěné přes rozhraní CGI. Umožňuje také uchovávat v paměti procesu data mezi požadavky, což se dá využít například pro uchování připojení k databázi po celou dobu běhu serveru.

Mod_python obsahuje moduly pro zpracování dat z webových formulářů, zachování stavové informace a další.

Rozličné webové frameworky

Webové frameworky jsou ucelené knihovny pro tvorbu webových aplikací. Umožňují vytvářet webové aplikace na vyšší úrovni. Programátor se může zabývat především logikou aplikace a nemusí řešit problémy nízké úrovně, jako je HTTP protokol, zpracování HTML formulářů a podobně.

Příkladem jsou TurboGears [3] nebo Django [1]. Oba jsou založeny na modelu MVC (Model-View-Controller).

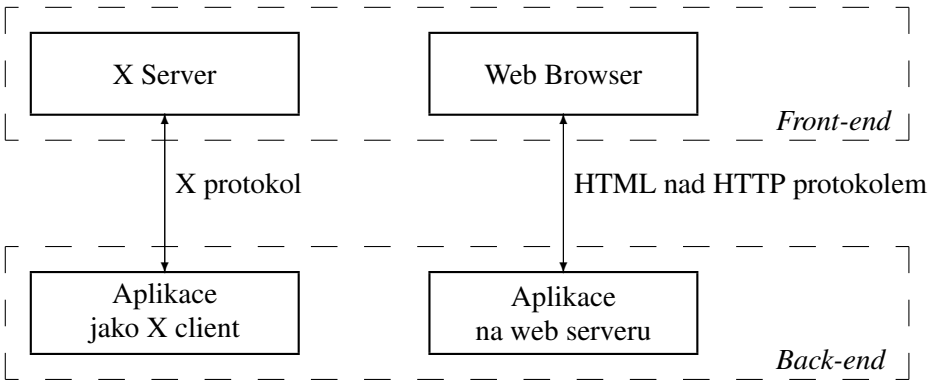
MVC model tvorby aplikací rozděluje aplikaci na tři moduly:

1. Model – Reprezentuje data, se kterými aplikace pracuje. Obvykle jako přístup k SQL databázi pomocí objektově-relačního mapování.
2. View – Pohled. Zobrazuje data uživateli. Tvoří se obvykle pomocí HTML šablon.
3. Controller – Řadič. Implementuje logiku aplikace. Zpracovává vstupy od uživatele a provádí změny modelu.

Webvé frameworky postavené na jazyku Python umožňují rychlý vývoj aplikací, kde stačí napsat minimum kódu pro funkční aplikaci.

2.3.2 Rozdíly v tvorbě desktopových a webových aplikací

Architektura webových i desktopových aplikací je v principu stejná (viz Obrázek 2.1) V obou případech si lze představit dva prvky, front-end, který komunikuje s uživatelem a back-end, který provádí samotnou logiku aplikace.



Obrázek 2.1: Srovnání architektury webových a desktopových aplikací.

V moderních operačních systémech, které mají své desktopové prostředí postavené na systému X Window, jsou aplikace X-klienty a komunikují s X-serverem, který se stará o zobrazení na obrazovku a vstupy ze vstupních zařízení. X protokol je síťově transparentní (stejně jako HTTP protokol), nezáleží tedy na tom, zda aplikace (X-klient) běží na stejném stroji, u kterého sedí uživatel.

Jaké jsou tedy rozdíly?

X protokol je stavový a přenáší data nízké úrovně – popisuje přesně, jak se má co zobrazit. Naopak protokol HTTP je bezestavový a jazyk HTML popisuje jen obsah dokumentu, konkrétní podobu dokumentu vytvoří až webový prohlížeč.

Náročnost zpracování u webových aplikací tedy nese klient. Server pracuje jen s vysokoúrovňovým jazykem HTML. To umožňuje obsluhu mnohem více klientů najednou.

Další výhodou webových aplikací je jejich dostupnost. Dobře napsanou webovou aplikaci může okamžitě používat každý, kdo disponuje univerzálním klientem – webovým prohlížečem.

Na druhou stranu tvorba webových aplikací je obtížnější. Při tvorbě desktopových aplikací jsou data uchována v paměti procesu, který trvá po celou dobu práce uživatele. Webové skripty však obvykle trvají jen po dobu zpracování HTTP požadavku. Programátor webových aplikací se tak musí zabývat uchováváním dat v rámci relací.

Webové aplikace také obvykle neposkytují takovou rozmanitost uživatelského rozhraní jako desktopové aplikace, protože se snaží nepředpokládat nic o klientské technologii. Dobře napsaná webová aplikace musí být použitelná na osobních i kapesních počítačích nebo mobilních telefonech.

Existuje mnoho snah o vytvoření "lepšího" univerzálního klienta. Mnohé z nich využívají proprietární technologie (například jazyk Flash, Java Applety nebo rozličná rozšíření jazyka HTML) a nejsou tedy přístupné každému. Jiné jdou standardní cestou, například skriptování v jazyce ECMAScript, asynchonní přenos XML dokumentů přes HTTP protokol (AJAX), nebo snaha organizace w3c o specifikaci tzv. Rich Web Clients[9].

Kapitola 3

Návrh knihovny pro tvorbu nezávislých informačních systémů

3.1 Specifikace požadavků

Cílem projektu je ucelený rámec (framework) v jazyku Python, který bude poskytovat základní prostředky pro tvorbu nezávislých informačních systémů.

Základní výčet požadavků je následující:

1. Aplikaci napsanou pouze s využitím této knihovny půjde spustit na lokálním počítači jako desktopovou aplikaci nebo s použitím HTTP serveru jako webovou aplikaci, a to beze změny zdrojového kódu aplikace.

2. Použití aplikace jako webové aplikace by nemělo předpokládat žádné speciální požadavky týkající se použitého webového prohlížeče. Aplikace nesmí vyžadovat podporu skriptovacího jazyka ECMAScript nebo jiných nestandardních technologií.

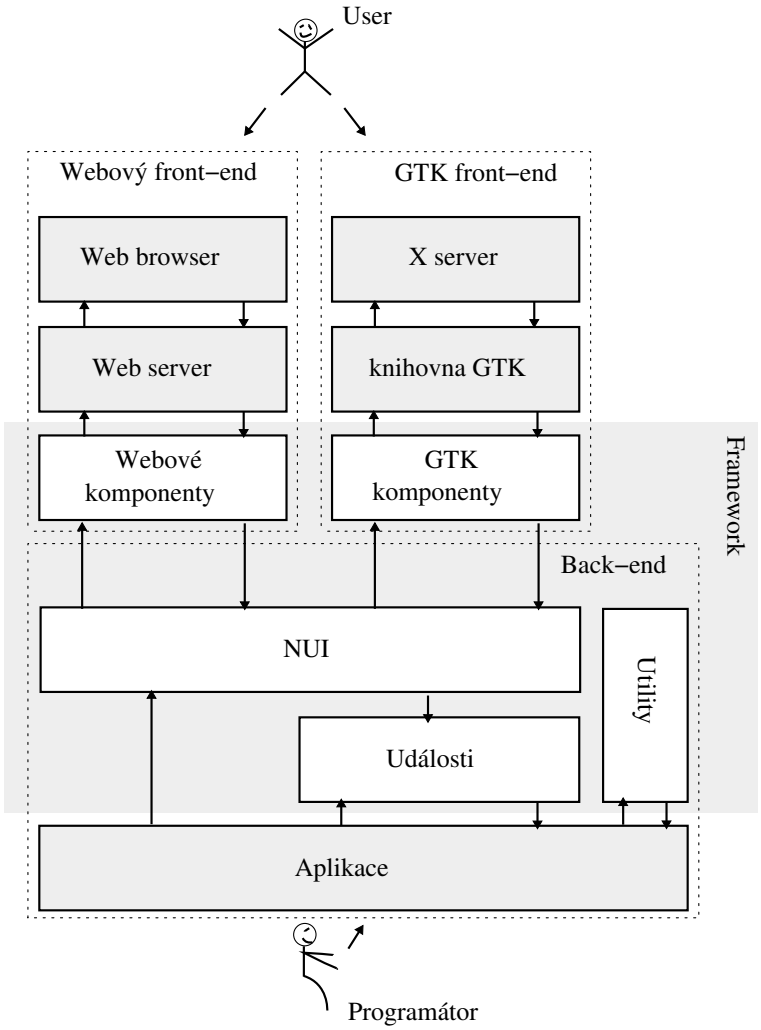
   Knihovna musí také zajistit uchování stavu aplikace v průběhu relace, a to pokud možno naprosto transparentně vzhledem k programátorovi aplikace. Programátor by se neměl zabývat tím, že mezi zobrazením stránky a odesláním formuláře zpět na server byl proces aplikace přerušen.

3. Knihovna by měla poskytovat základní prvky uživatelského rozhraní, jako jsou tlačítka, editační políčka, zaškrtávací tlačítka a jiná udělátka.

3.2 Architektura

Systém, který jako celek bude nezávislý na tom, zda se použije jako webová nebo jako desktopová aplikace bude muset mít části, které závislé budou. Tyto části budou přicházet do styku s uživatelem, a budu jim tedy říkat front-endy. Druhá část je naopak část, se kterou přichází do styku programátor aplikace. Pro něj se systém jeví jako nezávislý, protože abstrahuje konkrétní podoby uživatelských rozhraní. Tuto část nazvu back-end.

Programátor tvoří aplikaci za pomocí objektů, které nejsou závislé na konkrétním front-endu. To jsou jednak objekty, které vůbec nijak nesouvisejí s uživatelským rozhraním, jako je například spojení s databází. Dále to jsou objekty nezávislého uživatelského rozhraní (NUI) – tedy objekty, ze kterých byla abstrahována konkrétní podoba uživatelského rozhraní.



Obrázek 3.1: Architektura nezávislého informačního systému

Aplikace komunikuje s uživatelským rozhraním buď to voláním metod příslušných komponent, nebo se také může zapsat k odebírání událostí na příslušné komponentě. Událostí může být například stisknutí tlačítka, nebo vybrání položky z nabídky.

Abstraktní nezávislé rozhraní je pak konkrétně implementováno v jednotlivých front-endech. Implementace front-endů se bude výrazně lišit podle toho, zda se jedná o webový nebo o desktopový front-end.

Webový front-end umožňuje použití aplikace jako webové aplikace prostřednictvím webového prohlížeče. Webové komponenty budou zobrazovat svůj výstup jako úseky kódu jazyce HTML a budou umět zpracovat vstupy z webových formulářů. K tomu využijí HTTP server Apache s modulem mod_python.

Desktopový front-end bude implementován pomocí knihovny GTK. GTK komponenty tak budou jen obalovat objekty knihovny GTK.

Atributy třídy PObject	
_name	Jméno objektu. Unikátní v rámci rodiče tohoto objektu.
_id	Identifikátor objektu. Složenina jmen všech svých předků oddělena tečkou.
_parent	Odkaz na vlastního rodiče
_root	Odkaz na kořen stromu
_children	Tabulka odkazů na děti

Tabulka 3.1: Rozhraní PObject

3.3 Návrh programového rozhraní (API)

Největší omezení plynou ze snahy použít knihovnu jako webovou aplikaci. Proto jsem při návrhu přihlížel především na to, zda daná funkce bude vůbec implementovatelná jen s použitím jazyka HTML.

Bylo potřeba mít na zřeteli to, že knihovna se bude používat ve vícevláknovém prostředí. Z toho plyne omezení použití globálních a statických proměnných. K takovým proměnným je potřeba přistupovat pouze se správným zamykáním prostřednictvím semaforů nebo jiných prostředků výlučného přístupu. Pokud možno by se statické a globální proměnné neměly používat vůbec.

3.3.1 Reprezentace stavu aplikace

Uchováním stavu aplikace je potřeba se zabývat kvůli použití webového front-endu. Jak již bylo naznačeno v úvodu, jedno uživatelské sezení znamená restartování procesu aplikace při každém HTTP požadavku. Je tedy potřeba vhodně reprezentovat stav aplikace a ten pak umět uchovat a opět rekonstruovat do původní podoby.

Všechny objekty knihovny musí umět zachovat svůj stav v průběhu relace. Programátor by se neměl zabývat explicitním uchováváním stavu objektů. Veškeré objekty knihovny, které programátor použije by měly být schopny zachovat svůj stav v průběhu celé relace automaticky.

Aby toto bylo snadno proveditelné, budou veškeré takové objekty uloženy ve stromové struktuře. Když bude chtít programátor zachovat objekt po celou dobu sezení, vloží tento nový objekt jako potomka některého objektu, který se již v tomto stromu nachází. Kořenem stromu pak bude objekt reprezentující samotnou relaci.

Třídy objektů, které budou mít tuto vlastnost budou odvozeny ze základní třídy `PObject` (viz tabulka 3.1).

Zvláštní třídou pak bude třída `PRoot`, která bude implementovat kořen tohoto objektového stromu. Kořen by měl být schopen provádět ono zachování stavu celého stromu. Dále by měl umět najít objekt podle zadaného identifikátoru.

Může nastat situace, kdy stav objektu uložit nelze, nebo to není žádoucí. Takovým příkladem mohou být externí zdroje, jako je například spojení s databází. Takové případy bude řešit třída `SessionWrapper`. Jejím argumentem bude konstruktor, který se bude volat při každém novém rekonstruování stavu aplikace. Objekt třídy `SessionWrapper` pak bude fungovat jako obálka nad zkonstruovaným objektem.

Metody třídy Event	
addHandler (handler)	Zaregistruje *handler* jako obsluhu události.
call (. . .)	Způsobí vyvolání události u všech zaregistrovaných obsluh událostí. Volitelně se může této metodě předat proměnný počet dalších parametrů popisující událost.

Tabulka 3.2: Rozhraní Event

3.3.2 Události

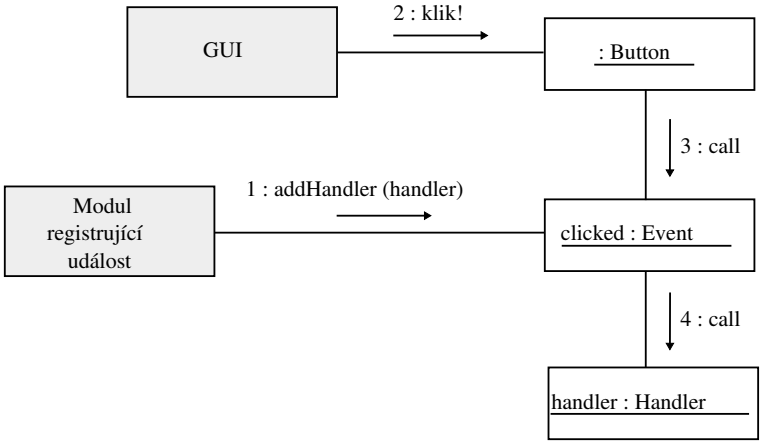
Události jsou způsob, jak zajistit, aby se různé objekty mohly dozvědět o změně stavu jiného objektu. Samotná událost je reprezentována objektem třídy `Event`. Třída `Event` má dvě metody (viz tabulka 3.2).

Použití událostí je uvedeno na diagramu 3.2. Jedná se o příklad události *clicked* reprezentující stisknutí tlačítka (objektu třídy `Button`). Programátor musí nejdříve vytvořit obsluhu události *handler* a tu pak zaregistrovat u události pomocí metody *addHandler*. Když uživatel prostřednictvím grafického uživatelského rozhraní (GUI) stiskne tlačítko, tak o této skutečnosti uvědomí tlačítko událost *clicked*. Událost pak zavolá všechny své zaregistrované obsluhy událostí.

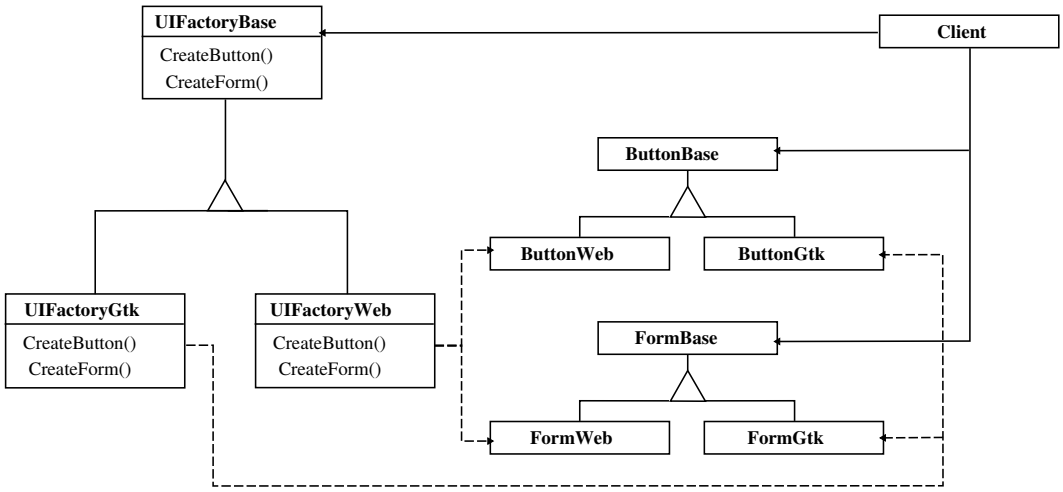
3.3.3 Komponenty nezávislého uživatelského rozhraní

Každý front-end bude implementovat každou komponentu ve zvláštní třídě. Aby byla zaručena nezávislost, musí být pro danou komponentu všechny její implementace odvozené z jediné bázové třídy, která bude popisovat její rozhraní. Při vytváření instancí konkrétní třídy však kód aplikace nesmí přímo vědět, který front-end je právě používán, protože jinak by aplikace nebyla nezávislá. Použijeme tedy návrhový vzor *Abstraktní továrna*, jak je zobrazeno na obrázku 3.3 (převzato z [8]).

Na obrázku jsou znázorněny dvě komponenty (formulář `Form` a tlačítko `Button`) a dvě jejich implementace ve front-endech Web a GTK. Klient (programátor aplikace) pracuje jen s abstraktními rozhraními. Virtuální metodou továrny `CreateButton` může vytvořit tlačítko, aniž by věděl, která konkrétní třída bude výsledkem této konstrukce.



Obrázek 3.2: Diagram kolaborace událostí



Obrázek 3.3: Návrhový vzor *Abstraktní továrna*

Ovládací prvky

Ovládací prvky by měly pokrývat alespoň prvky známé z formulářů v jazyce HTML, především tedy tlačítka, popisné textové a editační pole (viz tabulka 3.3).

Tabulky

Účelem tabulky je především zobrazení výsledků SQL dotazů z databáze. Obecně by však měla být použitelná pro zobrazování tabulkových dat z libovolného zdroje. Z tohoto důvodu se vytvoření tabulky sestává ze tří částí

1. Vytvoření samotné tabulky, specifikace vlastností pro tabulku, jako je požadovaný počet zobrazených řádků a podobně.
2. Vytvoření sloupců. Sloupce by se měly zabývat konkrétní formou zobrazení dat. (Například může řídit zarovnání, či různé transformace formátu dat.)
3. Vytvoření *kurzoru*. Kurzorem může být jakýkoli objekt implementující rozhraní třídy `Cursor` (viz tabulka 3.4). Úkolem kurzoru je na požádání vracet data pro zobrazení v tabulce.

Objekt, který chce být použit jako kurzor musí implementovat toto rozhraní. Název je odvozen od databázových kurzorů, protože plní podobnou funkci – procházet všemi řádky tabulkových dat.

Třída	Popis
Button	Tlačítko. Mělo by poskytovat nastavení popisu tlačítka a přístup k události *stisknutí tlačítka*.
LineEdit	Editační pole, které může editovat uživatel. Musí poskytovat metodu k nastavení a čtení vyplněného textu.
StaticText	Statický text, který nelze editovat uživatelem.
CheckBox	Zaškrtávací tlačítko.
RadioBox	Výběr jedné z více možností

Tabulka 3.3: Základní množina ovládacích prvků

Metody třídy Cursor	
begin ()	Inicializace. Veškerá práce s kurzorem bude prováděna mezi zavoláním begin a end. Například odeslání SQL dotazu databázi by mohla být vhodná činnost prováděna v metodě begin.
end ()	Ukončení práce s kurzorem. Pro jakoukoliv další práci s kurzorem je potřeba znovu zavolat metodu begin. Chování kurzoru v jiném případě není definováno.
getIterator ()	Tato metoda by měla vrátit iterátor nad všemy řádky dat. Toto umožňuje snadné procházení všech dat.
getItem (i)	Metoda getItem by měla vrátit řádek podle daného indexu *i*.
getItems (i, j)	Obdobně jako metoda getItem vrací řádky podle zadaných indexů. Vrátí však seznam řádků mezi indexy *i* a *j*.
getLength ()	Metoda getLength vrátí celkový počet všech řádků.

Tabulka 3.4: Rozhraní třídy Cursor

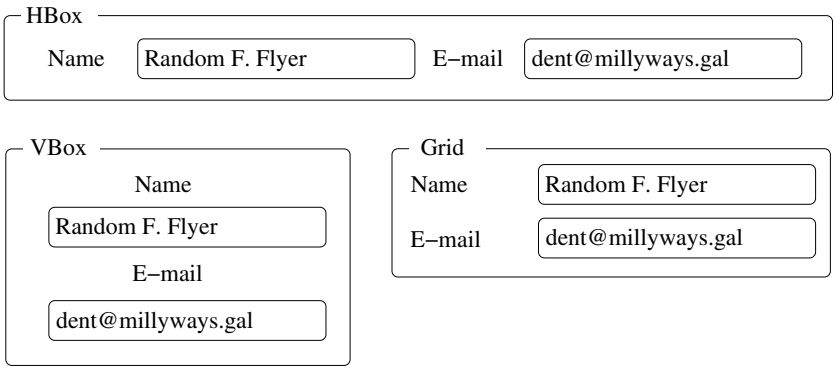
Model rozvržení komponent

Model rozvržení komponent popisuje, kam se jednotlivé komponenty usadí v rámci okna aplikace.

V existujících knihovnách pro tvorbu uživatelských rozhraní se uplatňují dva principy:

- Absolutní umístění – Komponenty jsou umisťovány podle zadaných souřadnic.
- Hiearchické umístění – Komponenty se umisťují do kontejnerů. Kontejnery mohou umisťovat komponenty horizontálně nebo vertikálně a mohou obsahovat další kontejnery.

Kombinování tří různých kontejnerů (vertikální a horizontální zarovnání a zarovnání do mřížky, viz obrázek 3.4) umožňuje vytvářet bohaté formuláře, aniž by se musel programátor zabývat přesným umístěním komponent. Výsledné uživatelské rozhraní tak může být nezávislé na použitém zobrazovacím zařízení nebo velikosti písma.



Obrázek 3.4: Ukázka použití tří různých kontejnerů (vertikální box, horizontální box a grid).

Vytváření uživatelských rozhraní pomocí externího XML dokumentu

Programátor může tvořit uživatelské rozhraní dvěma způsoby. Buď to voláním konstruktorů příslušných komponent nebo načtením externího XML dokumentu. Vytváření uživatelských rozhraní v externím dokumentu v deklarativním jazyce má především tyto výhody.

- Oddělení prezentace od logiky aplikace – Programátor se nemusí potápět hluboko v kódu aplikace, aby změnil polohu dvou tlačítek.
- Změny uživatelského rozhraní může snadno provádět i neprogramátor.
- Snadná lokalizace – Stačí vytvořit pro každý jazyk jiný dokument.
- Deklarativní programování se zdá přirozenější pro popis uživatelských rozhraní.
- Nabízí se možnost použití externích klikacích nástrojů, které zjednoduší návrh a implementaci uživatelských rozhraní.

Formát XML dokumentu pro popis uživatelského rozhraní bude následucící:

1. Kořenový element má název `ui`.
2. Elementy odpovídají názvem konstruktoru konkrétního prvku nezávislého uživatelského rozhraní.
3. Atributy elementů odpovídají parametrům konstruktoru.
4. Speciální atribut `id` umožní jednoznačné pojmenování objektu tak, aby se na něj mohlo z programu odkazovat.

Zde je ukázka popisu výše uvedeného formuláře.

```
<ui>
    <Grid rows='2' cols='2'>
        <StaticText text='Name'/>
        <LineEdit id='edit_name' text='Random F. Flyer'/>
        <StaticText text='Email'/>
        <LineEdit id='edit_email' text='dent@milliways.gal'/>
    </Grid>
</ui>
```

3.3.4 Práce s databází

Knihovna by neměla mít žádné zvláštní požadavky na práci s databází. Spolupráce s databází je potřeba jen při zobrazování tabulek a pro tyto účely poslouží univerzální třída `Cursor`.

Přímé propojení databáze s ovládacími prvky je možné implementovat jako nadstavbu nad touto knihovnou a není účelem této práce.

Kapitola 4

Implementace

Implementace byla rozdělena do několika modulů. Především jde o rozdělení knihovny na back-end a jednotlivé front-endy. Moduly jsou spojeny v jmenném prostoru `disp` (odvozeno ze zkratky anglického názvu tohoto projektu Development of Information Systems in Python).

Každý front-end se skládá ze dvou částí. Kromě vlastních tříd implementujících jednotlivé komponenty uživatelského rozhraní to je ještě *spouštěč*. Jeho úkolem je připravit prostředí pro to, aby volání kontruktorů prvků nezávislého uživatelského rozhraní (NUI) se převádělo na konkrétní konstruktory front-endu.

Podle návrhu by se pro tento účel měla vytvořit abstraktní továrna, ze které se budou vytvářet instance tříd NUI. Taková konstrukce je však v dynamických jazycích jako je Python zbytečná. Abstraktní továrnou může být samotný modul, do kterého se dynamicky přesunou konstruktory jednotlivých komponent.

Typická činnost spouštěče se tak dá popsat ve dvou krocích:

1. Importuje modul front-endu – Do jmenného prostoru `disp` importuje moduly konkrétního front-endu.

   Například webový front-end importuje konstruktor ButtonWeb jako disp.form.Button, což nahrazuje vytvoření abstraktní továrny a její hypotetickou metodu CreateButton.

2. Vytvoří instanci hlavní třídy aplikace. Ta je definovaná programátorem a obvykle je odvozena od třídy disp.application.Application. V této chvíli už je tedy aplikace navázána na konkrétní front-end. Pokud front-end umožnuje uložení stavu aplikace, tak v této chvíli načte předchozí stav aplikace a stav obnoví.

Vytvoření spouštěče je závislé na konkrétním front-endu a na aplikaci. Přinejmenším se musí spouštěč dozvědět o tom, jak se jmenuje a ve kterém modulu se nachází třída, která bude hlavní třídou aplikace.

4.1 Webový front-end

Webový front-end umožňuje přistupovat k aplikaci pomocí libovolného webového prohlížeče. Je postaven nad HTTP serverem Apache a jeho modulem mod_python.

Při použití webového front-endu je potřeba vytvořit spouštěč a zkopírovat jej do adresáře, ze kterého jej bude spouštět webový server. Pak už jen zbývá nakonfigurovat Apache a modul mod_python, aby používal vytvořený spouštěč.

4.1.1 Serializace a uložení stavu aplikace

Při použití webového front-endu je potřeba uložit stav aplikace mezi dvěma HTTP požadavky.

Stav aplikace je reprezentován stromem objektů odvozených od třídy `PObject`. Tento stav je potřeba převést do podoby, kterou je možno uložit, tedy nejlépe do řetězce. Procesu převedení objektů do řetězce se říká *serializace*.

Možnosti uložení stavu aplikace

Nabízí se více možností jak a kam uložit stav aplikace.

- Uložení v procesu na serveru – Stav aplikace je uchován přímo v paměti procesu na serveru.

  Výhoda této možnosti spočívá v tom, že není třeba nic serializovat. Objekty mohou existovat v paměti tak, jak byly vytvořeny. Tato možnost však lze použít pouze v případě, že můžeme zaručit, že stejný požadavek klienta bude obsloužen vždy tímtéž procesem. Prakticky to má cenu jen v případě, že nám stačí jediný proces s více vlákny. Takové uspořádání se však nedá rozšiřovat na shluk serverů.

- Uložení u klienta – Serializovaný stav aplikace se přenese klientovi. Ten pak tuto informaci opět odešle serveru při každém svém dalším požadavku.

  Při této možnosti může server pracovat bezestavově. Nemusí se zatěžovat ukládáním stavových informací a může tak obsluhovat více požadavků. Na druhou stranu se však zbytečně zvýší množství přenášených dat. Další nevýhodou je bezpečnost – data od klienta jsou z principu nedůvěryhodná a musela by být pečlivě kontrolována.

- Uložení v databázi na serveru – Serializovaný stav aplikace se uloží do úložiště na straně serveru, pod určitým klíčem. Tento klíč se pošle klientovi. Klient pak odesílá klíč s každým svým dalším požadavkem v relaci.

Pro implementaci jsem se rozhodl využít možnost uložení stavu v úložišti na serveru, které poskytuje modul mod_python.Session. To je obvykle databáze v souboru na serveru. Do tohoto úložiště vkládám serializovaný stav aplikace. Identifikátor relace je pak klientovi poslán prostřednictvím *cookie*.

Pro samotnou serializaci lze použít knihovnu `Pickle`, která je součástí standardní knihovny jazyka Python.

Optimalizace neměnných objektů

Serializovaný řetězec, který vyprodukuje knihovna Pickle je zbytečně velký. Mnoho dat o objektech se ukládá zbytečně, protože to jsou data, která se dají získat jinou cestou. Mezi zbytečně serializovaná data patří zejména

1. Atributy třídy PObject, jako jsou odkazy na rodiče, tabulka odkazů na děti a název. Tyto informace lze rekonstruovat ze samotných identifikátorů.

2. Informace o objektech, které svůj stav nijak nezměnily od vytvoření aplikace. Za předpokladu, že inicializační procedura bude pracovat vždy deterministicky, lze rekonstruovat objekt jen tím, že se znovu zavolá inicializační procedura.

3. Atributy objektů, které nemá cenu uchovávat. Toto je závislé na konkrétních objektech, protože jen ony sami vědí, které atributy potřebují a které ne.

Proto samotnou serializaci prostřednictvím knihovny Pickle nejprve upravuji.

1. Atributy třídy PObject, které se dají zrekonstruovat jen ze známých identifikátorů, budou před serializací odstraněny. Při deserializaci jsou pak zrekonstruovány na základě identifikátorů.

2. Každý objekt třídy PObject získává navíc atribut `dirty`. Serializovány jsou pak jen objekty, které mají atribut `dirty` nastaveny na pravdivou hodnotu.

   Nevýhoda tohoto přístupu spočívá v tom, že objekty si musí samy hlídat svůj stav, což může vést k těžce vyhledávatelným chybám. Pokud se objekt nechce účastnit této optimalizace, může nastavit svůj atribut `dirty` na pravdivou hodnotu již v konstruktoru. Takový objekt se pak bude serializovat vždy.

3. Implicitně se serializují všechny atributy objektu. Pokud však objekt implementuje metodu `_getstate`, serializuje se návratová hodnota této metody. Objekt tak má možnost uložit jen to, co považuje za nutné.

4.1.2 Vykreslení komponent

Každá třída reprezentující komponentu webového front-endu musí obsahovat metodu `web_out`. Ta má jediný argument a tím je kontext. Metoda vrací řetězec obsahující kus HTML kódu odpovídajícímu zobrazené komponentě.

V kontextu se předávají informace, které komponenta potřebuje vědět, aby se mohla správně zobrazit. Prakticky jedinou takovou informací je URL adresa aplikace na serveru, která je potřeba pro vytváření hypertextových odkazů.

Kontejnery (komponenty obsahující jiné komponenty) pak rekurzivně volají metodu `web_out` u všech svých potomků. Konečným výsledkem je pak HTML stránka, která se odešle klientovi.

Programátor má možnost ovlivnit výsledný vzhled takovéto webové aplikace úpravou kaskádových stylů. Implicitně je použit soubor style.css. Tento může programátor přepsat jakýmkoli jiným souborem v jazyce CSS. Každá komponenta je zobrazována s vlastní CSS třídou (atribut *class* v elementech jazyka HTML), což umožňuje velkou míru kontroly nad stylem aplikace.

Při tvorbě nezávislých informačních systémů by se však programátor neměl zabývat vzhledem aplikace vůbec.

4.1.3 Zpracování vstupů

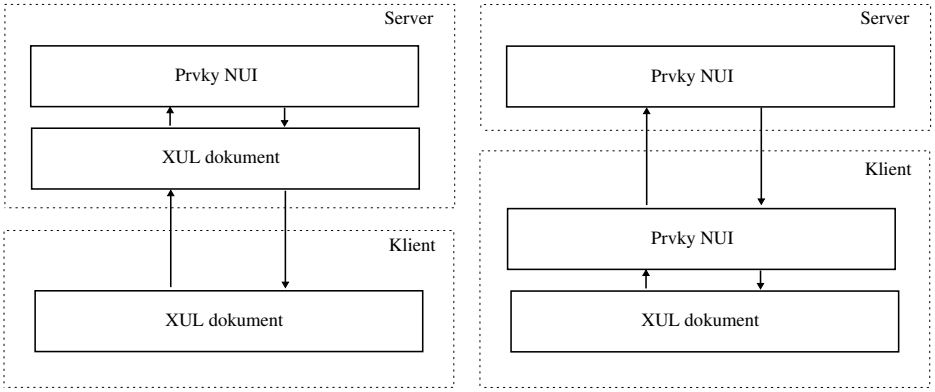
Vstup může nastat ve dvou případech.

1. Odesláním webového formuláře (aktivováním formulářového prvku *submit* v HTML dokumentu)

2. Kliknutím na hypertextový odkaz.

Z omezení daným jazykem HTML plyne, že při přechodu na jinou stránku přes hypertextový odkaz webový prohlížeč nemá jak odeslat nově vyplněná data z formulářů. Uživatel tedy musí vyplněný formulář odeslat kliknutím na nějaké tlačítko (formulářový prvek *submit*). Toto omezení by šlo obejít pomocí skriptování v jazyce ECMAScript, což by ale bylo v rozporu s požadavky.

Každá komponenta má svůj jednoznačný identifikátor (viz 3.3.1), který použije i pro jednoznačnou identifikaci v zobrazeném HTML dokumentu. Vstupy z HTML formulářů se na server dostanou jako dvojice *identifikátor* — *hodnota*, kde hodnota může být například vyplněný text v editačním poli. Při zpracování vstupu se musí vyhledat k identifikátoru odpovídající komponenta.



(a) Úroveň XUL (b) Úroveň NUI

Obrázek 4.1: Dvě možné úrovně protokolu.

K tomu stačí projít strom objektů od kořene podle jednotlivých jmen z identifikátoru. Vyhledanému objektu se pak zavolá metoda `web_in`, které se předá hodnota.

Obecně může jedna komponenta vytvořit více HTML prvků, které se mohou použít jako vstup. Takové prvky tedy v dokumentu musí označit identifikátorem složeným ze svého identifikátoru a nějakého řetězce, který se jistě nevyskytne jako název nějakého potomka.

4.2 XUL front-end

4.2.1 Co je to XUL

XML User Interface Language (XUL) je jazyk pro popis uživatelského rozhraní. Je používán především programy pocházejících z balíku internetových programů Mozilla (viz [6]).

Jazyk XUL není otevřeným standardem, ale je podporován rozšířeným webovým prohlížečem Firefox. Pokud takový prohlížeč narazí na dokument v jazyku XUL, tak zobrazí okno s prvky uživatelského rozhraní popsaném v takovém dokumentu. Chování aplikace se pak dá programovat v jazyce ECMAScript.

XUL front-end je tedy implementován ve dvou částech. Klientská část je implementována v jazyku ECMAScript a je spuštěna u klienta. Skript pak dynamicky vytváří prvky jazyka XUL a komunikuje se serverem posíláním HTTP požadavků za použití objektu `XMLHTTPRequest`. Komponenty na serverové část pak pracují jako proxy, komunikují s klientskými komponentami pomocí zpráv.

Serverová část front-endu je obdobná jako u webového front-endu a využívá se zde stejný princip serializace.

4.2.2 Komunikační protokol mezi klientem a serverem

Nejprve bylo třeba rozhodnout, na jaké úrovni bude klientská strana pracovat. V úvahu přicházely dvě možnosti, jak jsou naznačeny na obrázku 4.1.

1. Úroveň XUL. XUL klient by neměl ponětí o komponentách nezávislého uživatelského rozhraní, pouze by zobrazovat daný XUL dokument. Komunikační protokol by se pak týkal změn v tomto dokumentu.

   Klientská strana tak bude jednodušší, ovšem bude se muset přenášet větší množství dat.

Zprávy od serveru klientovi	
create	Vyvtoření objektu. Argumenty této zprávy jsou třída objektu, který se má vytvořit a identifikátor objektu.
call	Zavolání metody klientského objektu. Má tři argumenty. Identifikátor objektu, název metody a argumenty metody.
alert	Zobrazení chybové hlášky. Má jediný argument a to text hlášky.
Zprávy od klienta serveru	
init	První zpráva. Oznamuje spuštění aplikace u klienta. Server by měl odpovědět zprávami create, kterými informuje klienta o tom, co má zobrazit.
call	Volání metody serverového objektu. Má tři argumenty. Identifikátor objektu, název metody a argumenty metody.
state	Oznámení o stavu objektu na klientské straně. Například může obsahovat nový text editačních polí nebo stav zaškrtávacích tlačítek. Má dva argumenty. Identifikátor objektu a stav.

Tabulka 4.1: Typy zpráv mexi XUL klientem a serverem

2. Úroveň nezávislého uživatelského rozhraní. XUL klient bude vědět o objektech NUI a sám si bude na základě nich vytvářet XUL dokument. Komunikační protokol se pak bude týkat vytváření objektů NUI, předávání atributů a volání metod.

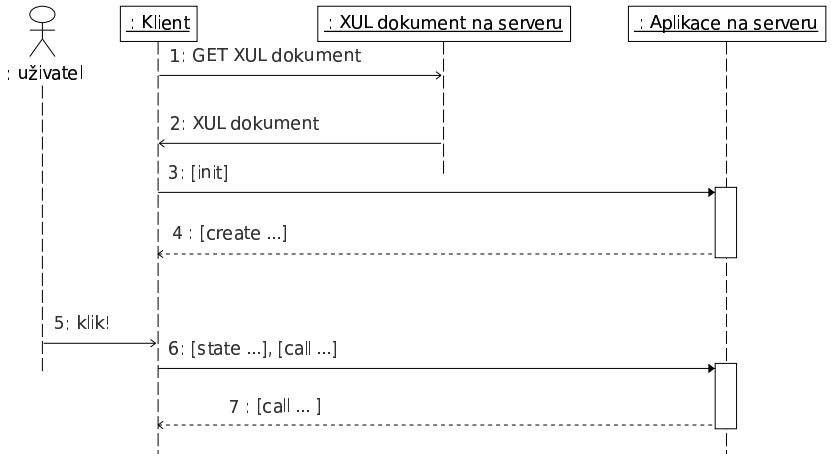
   Při tomto řešení bude spočívat velké množství práce na klientské straně. Po síti se přenáší jen minimum dat, protože se jedná o nejabstraktnější úroveň. Server nemusí mít o jazyku XUL ani ponětí.

Rozhodl jsem se implementovat druhou možnost, což znamenalo implementaci komponent na klientské straně v jazyce ECMAScript.

Komunikační protokol mezi klientskými a serverovými komponentami jsem založil na formátu JSON [2] (Java Script Object Notation). Jeho výhoda je jeho jednoduchost a dostupné a rychlé knihovny pro práci s ním v Pythonu i v jazyce ECMAScript. Zprávy v tomto formátu (viz tabulka 4.1) se pak přenášejí protokolem HTTP. V současné implementaci může server posílat zprávy klientovi pouze jako součást HTTP odpovědi. Komunikaci tedy vždy musí iniciovat klient.

Na obrázku 4.2 je ukázán příklad takové komunikace. Nejprve server načte XUL dokument. Ten je pro každou aplikaci stejný a slouží pro načtení klientské části front-endu vytvořené v jazyce ECMAScript. Po načtení této stránky pak klientská část pošle serveru zprávu `init`. Server inicializuje aplikaci a ke každé komponentě uživatelského rozhraní, kterou aplikace vytvoří, pošle klientovi zprávu `create`.

Při nějaké uživatelské akci, například při stisku tlačítka, je potřeba předat řízení aplikaci na serveru. Veškeré změny v komponentách provedené uživatelem se pošlou na server pomocí zpráv `state`, případně `call`. Pokud obsluha události v aplikaci na serveru upraví stav nějaké komponenty uživatelského rozhraní, pošle tuto informaci opět klientovi v odpovědi jako zprávy typu `call`.



Obrázek 4.2: Sekvence posílání příkazů mezi klientem a serverem.

4.3 GTK front-end

Knihovna GTK je knihovna pro tvorbu uživatelských rozhraní, původně vytvořena pro program GIMP. Od toho tedy její název GIMP ToolKit. V současnosti je to jedna ze dvou nejpoužívanějších knihoven pro tvorbu grafických uživatelských rozhraní pracujících na X Window System. V tomto projektu využívám knihovnu PyGTK 2.0, což je obálka nad knihovnou GTK pro jazyk Python (viz [7]).

GTK front-end se skládá z tříd implmentujících jednotlivé prvky nezávislého uživatelského rozhraní. Každá taková třída pak funguje jako obálka nad objekty knihovny GTK, jak ukazuje tabulka 4.2.

Komponenta	Prvek GTK
Table	gtk.TreeView
Button	gtk.Button
LineEdit	gtk.Entry
StaticText	gtk.Label
TabBox	gtk.Notebook
CheckBox	gtk.CheckButton

Tabulka 4.2: Mapování komponent nezávislého uživatelského rozhraní na prvky knihovny GTK.

Protože knihovna GTK byla jedním ze vzorů nezávislého uživatelského rozhraní, je implementace GTK front-endu relativně jednoduchá.

4.4 Integrace s knihovnou SQLObject

SQLObject [5] je knihovna pro objektově relační mapování pro jazyk Python. Umožňuje pohodlnou práci s SQL databází. Tabulky v SQL databázi jsou reprezentovány třídami, řádky jejich instancemi a sloupce jejich atributy. S databází se tak pracuje jako s objektovou databází. Knihovna SQLObject navíc abstrahuje od konkrétních detailů jednotlivých SQL databází, takže umožňuje vytvářet systémy nezávislé na konkrétní použité databázi. Podporuje většinu rozšířených databázových systémů, jako jsou MySQL, PostgreSQL, SQLite nebo Firebird.

Současné frameworky, jako jsou TurboGears nebo Ruby on Rails používají objektově relační mapování především proto, že umožňují rychlý vývoj aplikací.

Pro integraci SQLObject s knihovnou disp bylo potřeba zejména implementovat rozhraní Cursor pro pohodlné zobrazování dotazů v tabulkách. Toto implementuje třída SOCursor. Je to adaptér nad existujícím kurzorem knihovny SQLObject, který má veškerou potřebnou funkčnost.

V některých případech se může vyskytnout potřeba vytvářet spojení s databází dynamicky. Takové spojení by se mělo umět zachovávat v průběhu celé uživatelské relace. Tuto funkci implementuje třída SOConnection. Je to SessionWrapper nad objektem spojení knihovny SQLObject, takže se spojení vytvoří znovu při každé deserializaci automaticky. Třída SOClass pak může reprezentovat přímo jednu tabulku.

Programátor tedy může v průbehu běhu aplikace vytvořit objekty reprezentující spojení s databází nebo přímo s tabulkou databáze. Objekty samy zajistí připojování k databázi v průběhu celé relace.

Kapitola 5

Závěr

Výsledkem projektu je použitelná knihovna, se kterou se dají vytvářet jednoduché nezávislé informační systémy. Knihovna obsahuje prozatím poměrně chudou množinu komponent, ovšem architektura umožňuje snadno vytvářet další. Z funkcí, které by knihovna měla dále poskytovat bych uvedl ještě autorizaci uživatelů, která není řešena a případný programátor aplikace by ji musel řešit na aplikační úrovni.

Použití tří různých front-endů si vyžádalo velkou míru abstrakce uživatelského rozhraní, což na jednu stranu je výhodné, protože to zjednodušuje tvorbu aplikace, ovšem na druhou stranu to znemožňuje kontrolu nad detaily. Z toho důvodu by pro pokračování projektu mohlo být vhodné zaměřit se na jednu technologii a tu maximálně využít. Nabízí se jazyk XUL, neboť ten sám o sobě se dá považovat za nezávislý – dá se použít jako webová aplikace v prohlížečích Mozilla nebo jako desktopová aplikace pomocí knihovny XULRunner [4].

Dá se očekávat, že budoucí technologie bude nadále smazávat rozdíly mezi desktopovými a webovými aplikacemi.

Dodatek A

Ukázková aplikace

Ukázková aplikace bude jednoduchý adresář. Pro uložení dat použijeme MySQL databázi prostřednictvím knihovny SQLObject.

A.1 Kód aplikace

Nejprve vytvoříme modul s databází. Bude pro začátek tvořen jedinou tabulkou, ve které bude obsaženo jméno, příjmení a e-mailová adresa. Využijeme k tomu pohodlí poskytované knihovnou SQLObject.

```
1  # Soubor dbschema.py:
2
3  from sqlobject import *
4
5  sqlhub.processConnection = connectionForURI(
6    'mysql://user:password@127.0.0.1/db')
```

Budeme udržovat jediné spojení pro celou aplikaci. Předpokládá se, že na lokálním stroji běží MySQL a na něm máme databázi ‘db’ a k ní účet ‘user’ s heslem ‘password’.

Definujeme tabulku označující osobu:

```
8  class Person(SQLObject):
9    firstname = StringCol()
10   lastname = StringCol()
11   email = StringCol()
12
13 if __name__ == '__main__':
14   Person.createTable()
```

Tento modul slouží zároveň k vytvoření databáze a zároveň k přístupu k ní. Pro vytvoření tabulek stačí spustit tento skript interpretem Pythonu.

```
$ python dbschema.py
```

Pokud se podařilo připojit k databázi, tak v této chvíli je v ní vytvořena tabulka *person*.

Nyní navrhneme podobu uživatelského rozhraní. Bude obsahovat tabulku s kontakty, formulář pro zadání údajů a filtr pro vyhledání kontaktů podle jména.

Soubor form.xml:

```
<ui>
  <HPane>
    <VBox>
      <Grid rows="1" cols="3">
        <StaticText text="Filter: "/>
        <LineEdit id="edit_filter" size="40"/>
        <Button text="Filter">
          <handler event="clicked" handler="onFilter"/>
        </Button>
      </Grid>
      <Table id="table" rows="20">
        <TableColumn column="firstname" label="First Name" />
        <TableColumn column="lastname" label="Last Name" />
        <TableColumn column="email" label="E-mail" />
        <handler event="lineSelected" handler="onPersonSelected"/>
      </Table>
    </VBox>
    <VBox>
      <FrameBox label="Details">
        <Grid rows="4" cols="2">
          <StaticText text="First name"/>
          <LineEdit id="edit_firstname" />
          <StaticText text="Last name"/>
          <LineEdit id="edit_lastname" />
          <StaticText text="E-mail"/>
          <LineEdit id="edit_email" size="40" />
        </Grid>
      </FrameBox>
      <HButtonBox>
        <Button text="Update">
          <handler event="clicked" handler="onUpdate"/>
        </Button>
        <Button text="New">
          <handler event="clicked" handler="onNew"/>
        </Button>
        <Button text="Delete">
          <handler event="clicked" handler="onDelete"/>
        </Button>
        <Button text="Export">
          <handler event="clicked" handler="onExport"/>
        </Button>
      </HButtonBox>
    </VBox>
  </HPane>
</ui>
```

Výsledkem by mělo být okno rozdělené na dvě části (HPane). V levé části bude tabulka a filtr, v pravé pak formulář pro zadání údajů a rámeček s tlačítky.

Elementy *handler* označují obsluhy událostí, které bude potřeba naprogramovat ve zdrojovém kódu aplikace.

Následuje zdrojový kód aplikace:

```
1  # Soubor main.py:
2
3  from disp.application import *
4  from disp.form import *
5  from disp.db import *
6  from disp.cursor import *
7  from disp.xmlui import *
8  from disp.file import *
9  from dbschema import *
10
11 class Main (Application):
12   def initialize (self):
13     form = Form (self)
14
15     uiloader = XMLUIBuilder ()
16     uiloader.loadFile ('form.xml', self, form)
```

Vytvoříme formulář a do něj načteme uživatelské rozhraní z XML souboru. Metoda loadFile má tři argumenty.

1. Jméno souboru, ze kterého se má načíst popis uživatelského rozhraní
2. Objekt, do kterého se uloží odkazy na objekty uvedené pomocí atributu *id* a který bude obsahovat metody obsluh událostí uvedné v elementech *handler*.
3. Kořen, do kterého se mají vkládat komponenty načtené z XML souboru. Obvykle to bývá nějaký formulář.

```
18     self.table.cursor = SOCursor (Person, orderBy = 'lastname')
```

Nastavíme kurzor na kurzor pro čtení z SQLObject-ových tabulek. První argument udává tabulku a další argumenty parametry SQL příkazu SELECT. V tomto případě pouze specifikujeme řazení.

```
20     # Aktuálně zvolená osoba v tabulce.
21     self.person_id = None
22
23     self.file_output = FileOutput (self,
24       func = self.export,
25       content_type = 'text/xml',
26       filename = 'output.xml')
```

Třída FileOutput slouží k zápisu dat do souboru uloženého u klienta. Musí se specifikovat funkce, která provede zápis a typ souboru (ten je využíván jen webovým front-endem a měl by být nastaven na MIME typ výsledného souboru). Argument *filename* je volitelný a specifikuje implicitní jméno souboru.

```
28      form.open ()
29
30    def onUpdate (self):
31      if self.person_id:
32        person = Person.get (self.person_id)
33        person.set (
34          firstname = self.edit_firstname.text,
35          lastname = self.edit_lastname.text,
36          email = self.edit_email.text)
37        self.table.update ()
```

Metoda onUpdate je obsuha události na stisknutí tlačítka Update. Jeho funkcí je aktualizovat údaje o vybrané osobě podle nových dat z formuláře. Zde využijeme schopnosti knihovny SQLObject pracovat s daty v SQL databázi jako s objekty.

Atribut *text* je atribut tříd LineEdit a vrací obsah editačního boxu. Nakonec je potřeba sdělit tabulce, že má znovu načíst data, protože byla změněna. To se provede metodou *update*.

Obsluhy událostí New a Delete jsou obdobné.

```
40    def onNew (self):
41      person = Person (
42        firstname = self.edit_firstname.text,
43        lastname = self.edit_lastname.text,
44        email = self.edit_email.text)
45
46      self.person_id = person.id
47      self.table.update ()
48
49    def onDelete (self):
50      if self.person_id:
51        Person.delete (self.person_id)
52        self.table.update ()
53
54    def onPersonSelected (self, person):
55      self.edit_firstname.text = person.firstname
56      self.edit_lastname.text = person.lastname
57      self.edit_email.text = person.email
58
59      self.person_id = person.id
60
61    def onExport (self):
62      self.file_output.open ()
63
64    def export (self, stream):
65      stream.write('<people>')
66      self.table.cursor.begin ()
67      for person in self.table.cursor:
68        stream.write('<person>')
69        stream.write('<firstname>%s</firstname>' % person.firstname)
70        stream.write('<lastname>%s</lastname>' % person.lastname)
```

```
71          stream.write('<email>%s</email>' % person.email)
72          stream.write('</person>')
73        self.table.cursor.end()
74
75        stream.write('</people>')
```

Metoda export implementuje zápis do souboru. Zapíše informace o všech osobách, které zrovna tabulka zobrazuje prostřednictvím tabulkového kurzoru. Kurzor implementuje i metodu __iter__, takže umožňuje i jednoduchý průchod všech prvků obyčejnou konstrukcí *for*.

V tomto jednoduchém příkladě neřešíme speciální znaky a kódování znaků.

```
78    def onFilter(self):
79      filtr = self.edit_filter.text
80      if filtr == '':
81        self.table.cursor = SOCursor(Person, orderBy = 'lastname')
82      else:
83        self.table.cursor = SOCursor(Person, orderBy = 'lastname',
84          clause = OR(
85            LIKE(Person.q.lastname, filtr),
86            LIKE(Person.q.firstname, filtr)))
87      self.table.update()
```

A.2 Nasazení aplikace

Pokud máme funkční databázi a nastavený server Apache s modulem mod_python, můžeme vystavit naši aplikaci jako webovou aplikaci.

Vytvoříme adresář `/var/www/aplikace/web/` a zkopírujeme tam zdrojové soubory. Dále tam vložíme obsah adresáře `deploy/web/` z adresáře projektu.

Tyto soubory upravíme následnovně:

Soubor .htaccess:

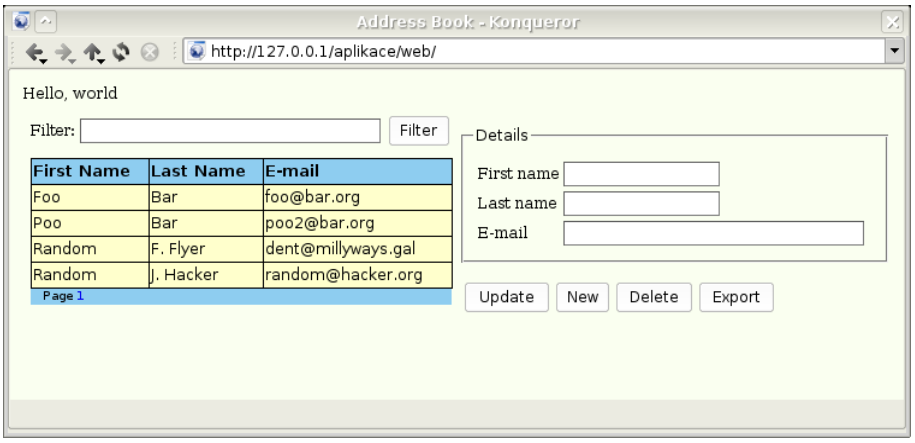
```
1 AddHandler mod_python .py
2 PythonDebug On
3 PythonHandler index
4 PythonInterpPerDirectory On
```

Udává, že soubory s příponou py bude zpracovávat modul `index.py`. Ten bude vypadat následovně:

```
1  import disp.web
2  import disp.web.launcher
3
4  import main
5
6  def handler(req):
7    return disp.web.launcher.launch(
8      req,
9      main.Main,
10     "./index.py",
11     "/var/www/aplikace/web/")
```

Inicializuje spouštěč a předá mu řízení. Musí mu zde předat hlavní třídu aplikace (to je námi vytvořen modul main a jeho třída Main. Dále pak je to URL adresa dokumentu, jak jej má popisovat ve vygenerovaných HTML stránkách. Posledním argumentem je cesta k lokálním souborům (jako je například náš XML soubor s popisem uživatelského rozhraní).

Pokud šlo vše dobře, tak bychom měli dostat funkční webovou aplikaci.



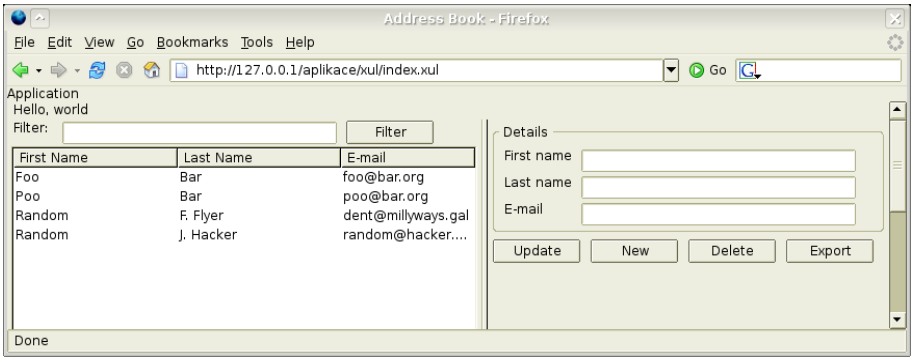
Obrázek A.1: Výsledná webová aplikace

Front-end XUL se instaluje obdobně. Do adresáře `/var/www/aplikace/xul/` stačí zkopírovat soubory aplikace a obsah adresáře `deploy/xul/` z adresáře projektu.

Pak se musí upravit soubor `handler.py` tak, aby načetl správný modul a nastavil cestu k aplikaci:

```
1  import disp.xul
2  import disp.xul.launcher
3
4  import main
5
6  def handler(req):
7    return disp.xul.launcher.launch (
8      main.Main,
9      req,
10     "/var/www/aplikace/xul/")
```

K spuštění XUL front-endu je potřeba webový prohlížeč Firefox, alespoň ve verzi 1.5.



Obrázek A.2: Výsledná aplikace v XUL

Poslední front-end je GTK front-end. Ten je na použití nejjednodušší, protože jeho spouštěč je jediný soubor `run.py` v adresáři `deploy/gtk/`.

Stačí upravit tak, aby načetl hlavní modul aplikace. Cestu k datovým souborům nepotřebuje, předpokládá totiž, že všechny cesty k datovým souborům začínají z aktuálního pracovního adresáře.

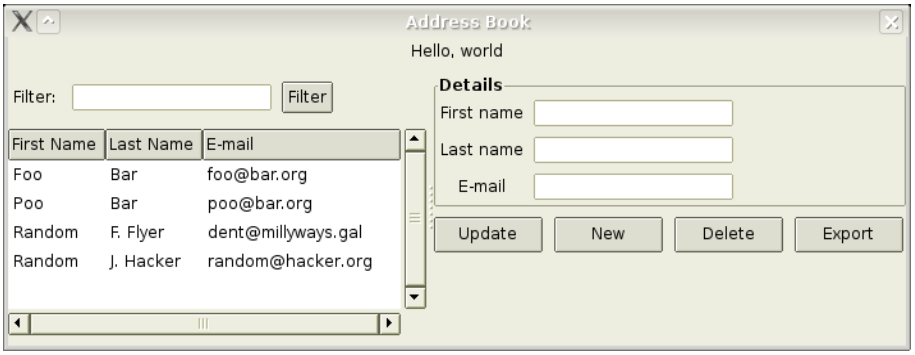
Soubor run.py:

```
1 #!/usr/bin/env python
2
3 import disp.gtk
4 from main import Main
5
6 app = Main ()
7 app.run ()
```

Aplikaci pak můžeme spustit přímo

```
$ python run.py
```

Jak bylo ukázáno, aplikace je přenositelná beze změny zdrojového kódu a lze ji použít jako desktopovou i jako webovou aplikaci.



Obrázek A.3: Výsledná desktopová aplikace v GTK

Literatura

[1] Django – The Web framework for perfectionist with deadlines. [online], [cit. 2006-04-20].
URL http://www.djangoproject.com/

[2] Introducing JSON. [online], [cit. 2006-04-20].
URL http://www.json.org/

[3] TurboGears – Front-to-Back Web Development. [online], [cit. 2006-04-20].
URL http://www.turbogears.org/

[4] XULRunner. [online], 20 April 2006, [cit. 2006-04-20].
URL http://developer.mozilla.org/en/docs/XULRunner

[5] BICKING, I.: Main SQLObject documentation. [online], [cit. 2006-04-20].
URL http://www.sqlobject.org/SQLObject.html

[6] BOSWELL, D.; KING, B.; OESCHGER, I.; aj.: *Creating Applications with Mozilla*. Sebastopol, California, USA: O'Reilly & Associates, Inc, první vydání, 2002, ISBN 0-596-00052-9, 474 s.

[7] FINLAY, J.: PyGTK 2.0 Reference Manual. [online], Version 2.8.1 September 17, 2005 [cit. 2006-04-20].
URL http://www.pygtk.org/pygtk2reference/index.html

[8] GAMMA, E.; HELM, R.; JOHNSON, R.; aj.: *Návrh programů pomocí vzorů – Stavební kameny objektově orientovaných programů*. Praha: Grada, první vydání, 2003, ISBN 80-247-0302-5, 388 s.

[9] JACKSON, D.: Rich Web Clients Activity Statement. [online], v 1.11 2006/04/05, [cit. 2006-04-20].
URL http://www.w3.org/2006/rwc/Activity.html