Mobilní sběr a server-side zpracování dat pomocí Java technologií

Cell phone data collection and server-side processing with help of Java technology

Martin Kolařík

Bakalářská práce
2007

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav aplikované informatiky
akademický rok: 2006/2007

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Martin KOLAŘÍK**
Studijní program:	**B 3902 Inženýrská informatika**
Studijní obor:	**Informační technologie**
Téma práce:	**Mobilní sběr a server-side zpracování dat pomocí JAVA technologií.**

Zásady pro vypracování:

- Vytvoření aplikace pomocí J2MEE, která bude odesílat zadané data na server.
- Serverová část bude zpracována v J2EE a bude schopna data zpracovat a přehledně zobrazit podle požadavků v WWW rozhraní.
- Součástí aplikace bude tříúrovňové zabezpečení přístupu k serverové části.
- Předpokládané využití vytvořené aplikace bude v medicíně pro sběr a vyhodnocení glykemických dat.

Rozsah práce:

Rozsah příloh:

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

Seznam odborné literatury:

Pecinovský, R.: Myslíme objektově v jazyku Java 5.0. Grada.
Brůha, L.: Java-Hotová řešení. ComputerPress, 2003.
Kiszka, B.: 1001 tipů a triků pro programování v jazyce Java. ComputerPress, 2003.
Momjian, B.: PostgreSQL – Praktický průvodce, ComputerPress, 2003.

Vedoucí bakalářské práce: **Ing. Martin Sysel, Ph.D.**
Ústav aplikované informatiky

Datum zadání bakalářské práce: **13. února 2007**

Termín odevzdání bakalářské práce: **24. května 2007**

Ve Zlíně dne 13. února 2007

prof. Ing. Vladimír Vašek, CSc.
děkan

doc. Ing. Ivan Zelinka, Ph.D.
ředitel ústavu

Abstrakt

Cílem práce bylo vytvořit aplikaci v jazyce Java pro sběr dat pomocí mobilního telefonu. Data jsou po odeslání uchována na serveru, který je zpřístupní uživatelům podle jejich práv. Práva jsou přidělována na základě rolí, přičemž v základní podobě je rozdělení rolí třístupňové. K vytvoření serverové části je použito pravidel J2EE, což přehledně celou aplikaci strukturuje. Aplikace sbírá data o obsahu cukru v krvi pacientů trpících diabetem a nabízí jejich přehled ošetřujícímu lékaři. Lékař tím získá průběžný přehled o stavu pacienta, což je v případě diabetu důležité.

Klíčová slova

Java, J2ME, J2EE, Spring, třívrstvá architektura, tenký klient, sběr dat

Abstract

The intention of this bachelor thesis was to create application in Java language, which should serve for acquiring data using cell phone. Data are stored in server, which offers them to users according to their permissions. Permissions are taken from roles, which are by default three. The sever part of the application was created with help of J2EE patterns. Thanks to them the application is well structured. Mine implementation collects gykemic data of diabetics and allows the physician to view them. It gives the physician continuous view of diabetic health, what is in the case of diabetes important.

Key words

Java, J2ME, J2EE, Spring, three-tier architecture, thin client, data collecting

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a použil jsem pouze uvedené prameny.

Ve Zlíně 10. května 2007

Obsah

Úvod 8

TEORETICKÁ ČÁST 9

1 Co je to Java? 10

1.1 Jazyky první generace 10

1.2 Jazyky druhé generace 11

1.3 Vsuvka o komponentových modelech 12

1.4 Jazyky třetí generace 13

1.5 Java je… 16

2 Třívrstvá architektura 17

2.1 Technická motivace 17

2.2 Jiné motivace 18

2.3 Více vrstev 19

3 Java a podniková řešení 23

3.1 Java Enterprise Edition 23

3.2 Kde J2EE rozběhnout 26

4 Spring 28

4.1 Co je to Spring? 28

4.2 Jak na Spring? 30

5 Java a mobilní zařízení 32

PRAKTICKÁ ČÁST 34

1 O řešení obecně 35

2 Organizace řešení a vývojové nástroje 37

3 Východiska 39

4 Objektový návrh 40

4.1 Doménové objekty 40

5 Implementace serveru 42

5.1 Informační vrstva, objektově-relační mapování 42

5.2 Vrstva aplikační logiky 43

5.3 Rozhraní mobilního klienta 43

5.4 Rozhraní webového klienta 45

6 Běhové prostředí serveru 47

6.1 Apache Tomcat . . . 47
6.2 Spring . . . 48
6.3 Databáze . . . 51
7 Mobilní klient . . . 52
7.1 Uživatelské rozhraní . . . 52
7.2 Spojení se serverem . . . 54
7.3 Konfigurace . . . 54
8 Webový klient a jeho obsluha . . . 56
8.1 Implementace řadičů . . . 56
8.2 Implementace pohledů . . . 58
8.3 Chování a ovládání . . . 59
9 Možnosti rozvoje . . . 62

Závěr . . . 63

Conclusion . . . 64

Literatura . . . 65

Úvod

Diabetes, cukrovka, je nemoc, která provází lidstvo od nepaměti [10]. Nemoc nepříjemná; především proto, že je při současných znalostech nevyléčitelná. Dva existující rozdílné druhy cukrovky mají rozdílné projevy a vyžadují rozdílné způsoby léčení. Oba druhy cukrovky však sdílejí základní způsob ověřování postupu léčby a stavu pacienta, jímž je měření hladiny cukru v krvi (glykemie).

Diabetes je teoreticky dobře prozkoumaná nemoc a dobře je proto také známo, že kromě momentální koncentrace cukru je nutné sledovat i její vývoj v čase. Lékař z pravidelných a četných dat dokáže zjistit mnohem více a dokáže podle toho vybírat léčbu lépe. Těmto teoretickým skutečnostem však praxe neodpovídá. Typický diabetik navštěvuje lékaře jednou měsíčně, přičemž jeho glykemie se změří den před touto návštěvou. Lékař více dat nemá.

Kdyby bylo možné získávat data častěji, bylo by možné lépe a efektivněji léčit; možná, že by diabetici mohli redukovat i své návštěvy u diabetologů. Myšlenka na častější měření glykemie jistě není nová. Dříve však nebyly dostupné technologie, pomocí kterých by bylo možné (bez návštevy lékaře) data sbírat. Mobilní telefony jsou samozřejmé dnes, ale ještě před deseti lety o nich prakticky nikdo nic nevěděl. Myšlenka využít mobilní telefon pro sběr glykemických dat se proto mohla objevit až při rozšíření mobilních telefonů.

Přirozeným tokem úvah o možnostech řešení s mobilními telefony lze dospět k jazyku *Java*, neboť prakticky všechny mobilní telefony jsou prostředím tohoto jazyka vybaveny. Současně, vlastnosti Javy dovolují vytvořit program jednou a rozběhnout jej vícekrát. Obě skutečnosti podporují názor, že aplikaci pro mobilní telefony je nejlepší vyvíjet právě v Javě.

Použití mobilního telefonu jako vstupního terminálu omezuje programátorův rozlet. Mobilní telefon je zařízení, které má telefonovat a vydržet dlouho bez nabíjení, není určen k velkým a rychlým výpočtům. Pořízení dat je pro telefon únosné, jejich zpracování už ne. Data musejí být zpracována někde jinde, v serveru, v místě, které je trvale dostupné a které klientům v telefonu poskytne službu, když si ji vyžádají. Provedení serveru je z pohledu sběru dat libovolné. Protože však bude Java použita v telefonu, je logické použít Javu i v serveru, aby zůstala použitá prostředí i prostředky podobné.

Aplikace pro sběr glykemických dat, dílem běžící v telefonu a dílem v serveru, může být především v ambulantním sektoru velmi užitečná.

TEORETICKÁ ČÁST

1 Co je to Java?

Pro většinu čtenářů termín *Java* patrně není nový, neboť první oficiální vydání tohoto *programovacího jazyka* (Java 1.0 SDK) se datuje již do roku 1996. Počátky je ale možné vysledovat až k roku 1990. Tehdy skupina vývojářů společnosti Sun začala vyvíjet nový jazyk s cílem získat nástroj pro spotřební elektroniku; nástroj, který by umožnil vyvíjet programy rychleji a s méně chybami [43].

Chybovost kódu je problematická věc a Java byla první jazyk, který systematicky uvedl některé z cest k jejímu zmenšení. Jak však bude postupně zřejmé, není to jediný technologický přínos Javy.

Od roku 1996 Java prošla vývojem, zpočátku mírným, později velice rychlým. V současnosti je již těžké Javu přesně definovat, protože z původní ideje programovacího jazyka postupně vznikly celé mohutné architektury, úřední i neúřední standardizační postupy či technologie od Javy odvozené (pochopitelně na Javě závislé). Vše se prolíná, doplňuje a hovoří-li dnes někdo o Javě, tak bez bližšího popisu nelze uhodnout, co přesně má na mysli.

Společným základem nicméně stále zůstává Java jako *programovací jazyk*.

Aby Java mohla být lépe pochopena, je třeba stručně projít historií programovacích jazyků (ovšem s vědomou filtrací informací s ohledem na Javu). Programovací jazyky mají navzájem mnoho společného, takže může být těžké stanovit nějaké dělení. Pro účely této práce budou jazyky rozděleny do třech generací podle míry jejich abstrakce a logiky tak, aby se jejich podstatné rysy daly reflektovat v Javě.

* *Generace jazyků je již ustálený a používaný pojem, kdy strojový kód přestavuje generaci prvou, assembler generaci druhou, všechny vyšší jazyky (C, Modula, Java, C#) generaci třetí a databázové jazyky patří do generace čtvrté. Dělení pro účely tohoto textu je sestaveno z didaktických důvodů jinak [35].*

1.1 Jazyky první generace

Typický zástupce: Fortran [14], Basic [4]. Charakteristickým jazykem jazyků první generace je nekoncepční přístup k algoritmizaci a k práci s datům a absence jakékoli vyšší jazykové podpory pro operace složitější, než je samotný zápis algoritmu.

V jazycích zprvu neexistoval pojem dynamicky přidělované paměti [11]. Jazyky jsou omezené syntakticky (např. délka identifikátorů ve Fortranu), sémanticky (Fortran ani Basic

neobsahují např. strukturované výjimky nebo vyšší cykly) i funkčně (jazyky neměly od svého počátku rekurzi). Jazykům zcela chybí podpora OOP [31] a prakticky v nich chybí jakákoli snaha omezovat chyby programátorů.

Psát s chybami v těchto jazycích je opravdu snadné. V roce 1962 mohl překlep ve Fortranu stát NASA ztrátu nosné rakety Mercury, vnitřní kontrolní mechanizmy však chybu odhalily ještě před startem [24].

1.2 Jazyky druhé generace

Typický zástupce: C, C++, Modula2, Ada [35]. Charakteristickým jazykem jazyků druhé generace je univerzální abstrakce dat a zdrojů a precizní podpora pro algoritmické a strukturované programování [42]. Všechny jazyky druhé generace byly v průběhu let rozšířeny o podporu objektového programování, což v nich způsobilo někdy větší, někdy menší změny. V jazycích je běžná práce s dynamickou pamětí, je běžná silná podpora datových typů. Většina z těchto jazyků definuje jako svou součást standardní knihovny, které mají zaručit přítomnost stejných nástrojů bez ohledu na cílovou platformu a které poskytují programátorům množství stále se opakujících úloh již v hotové podobě (práce s textovými řetězci, soubory, základní abstraktní datové typy apod.).

Pochopitelně, jazyky se od sebe liší. I když se pominou syntaktické rozdíly (zde lze hrubě rozlišit větve jazyků se složenými závorkami a bez nich), zůstane jich docela dost.

Jazyky odvozené od C mají blízko k práci s pamětí, takže např. návratové parametry jsou předávány adresou, textové řetězce jsou typicky bloky paměti ukončené nulou apod. Mimochodem, tato konstrukce řetězců je patrně jedinou příčinou gigantického množství chyb typu *buffer overflow*. Může se jen spekulovat, zdali by těchto chyb bylo méně, kdyby C jazyky pracovaly s řetězci bezpečněji od počátku . . . [5]

C jazyky nepodporují také vnořené definice (procedury v procedurách). Jedna z největších nevýhod těchto jazyků je způsob použití hlavičkových souborů: každý překládaný modul shrnuje všechny dovezené hlavičky a překládá se jako monolit. To způsobuje řadu chyb a potíží při dovozech či při změnách definic během překladu složitějších projektů.

Jazyky ideově odvozené od Algolu (Pascal, Modula2, Ada) se snaží některé skutečnosti programátorovi více usnadnit. Například definují abstrakci parametrů předávaných odkazem (VAR), některé z jazyků zavádějí pojem otevřeného pole (open array). Naopak, přímá práce

s pamětí je v těchto jazycích někdy složitější [35].

Tyto jazyky zavádějí od počátku přísnou typovou kontrolu, některé z nich dokonce uvozují koncept (Ada), který byl plně rozvinut až v W3C XSD (definování typů omezováním a rozšiřováním, [53]). Možná že i z tohoto důvodu je Ada standardním programovacím jazykem v NASA (řídicí programy pro sondy apod., [2]). Vůči C jazykům mají tyto jazyky omezené výrazové prostředky; v těchto jazycích například lze sčítat pouze jediným binárním operátorem, některé operátory neexistují a přiřazovací příkaz je skutečný příkaz, takže jej nelze použít jako součást výrazu.

Většina těchto jazyků používá nějaký způsob oddělení části veřejné (exportní či definiční soubor, co mohou použít jiní) a části neveřejné. Prakticky se na úrovni jednotek překladu dosahuje podobného chování, jako dnes běžně mají třídy (se soukromými a veřejnými členy).

Všechny jazyky druhé generace mají potíže s dynamickou pamětí. Starost o její správu má v rukou plně programátor, takže při použití těchto jazyků vždy musí být jednou z testovacích fází prověřování, jestli se všechna přidělená dynamická paměť vrací zpět.

Objektová rozšíření jazyků druhé generace (v případě C++ přímo přepis do zcela jiné sémantické formy) přinesla jistou nekonzistenci. Zatímco strukturované programování se ustálilo a všechny jazyky je mají prakticky shodné, objektové přístupy se jazyk od jazyka liší. To komplikuje přechody mezi technologiemi a rovněž zvyšuje množství chyb.

Nejvíce propracovaný, ale také současně nejvíce komplikovaný, je objektový model C++. Přinesl přetěžování metod, operátorů, vícenásobnou dědičnost, čistě virtuální funkce apod. Pro všechny tyto komplexní vlastnosti však C++ nemá dostatečně abstraktní podporu, takže chce-li programátor C++ pracovat s pojmy OOP dnešní doby (interface, property, implementace rozhraní), musí používat atomických náhražek.

1.3 Vsuvka o komponentových modelech

Vedle programovacích jazyků určených primárně ke kódování algoritmů existují velké oblasti návrhu informačních systémů, která s jazyky příliš nesouvisí. S příchodem objektového programování musela být zavedena objektově orientovaná analýza a s jejím používáním začalo být snadné uvažovat o znovupoužitelnosti tříd či větších celků tříd — komponent [30].

Aby bylo možné komponenty spojovat a aby komponenty uměly bez své přímé znalosti spolupracovat, začaly vznikat formalizmy pro jejich uspořádání. Brzy se tento myšlenkový

proud rozvinul do *komponentových objektových modelů* [40], které na vyšší úrovni abstrakce, než jakou používají programovací jazyky, popisovaly rozhraní komponent včetně způsobu poznávání komponent, jejich vzájemného volání nebo jejich vzájemného sdílení svých dat. Typickými zástupci jsou například modely CORBA, nebo COM [40].

Každý komponentový objektový model musí řešit, mimo jiné, přežívání instancí komponent, jsou-li tyto instance poskytovány do cizích komponent. Jak zjistit, že mou instanci používá pět jiných komponent a že já, její původce, ji nemohu zničit? Typické objektové modely řeší tenhle problém *počítáním* odkazů na komponenty. Například v COMu jde situace ještě dále a stanovuje se pravidlo, že instance *nesmí* zanikat jinak, než sama uvnitř po zániku posledního odkazu.

Myšlenka počítání odkazů je elegantní a jednoduchá. Bohužel, přes veškerou snahu tvůrců komponentových modelů stále zůstává při jejich použití prostor pro programátory, kteří mohou porušovat pravidla, a tím způsobovat chyby. Samotný komponentový model, bez programátorovy kázně a bez podpory programovacího jazyka nedokáže chybám v počítání odkazů zabránit.

Přesto však, počítání referencí je myšlenka důležitá, neboť jazyky třetí generace při správě paměti počítání referencí používají buď přímo, anebo v odvozených postupech.

1.4 Jazyky třetí generace

Nejprve se objevila Java, později jazyky platformy .Net [26]. Je zde rozdíl v kategoriích, neboť Java je jazyk; zatímco .Net je vývojová platforma, která v sobě definuje i abstraktní rozhraní jazyka. Pokud nějaký jazyk vyhoví .Net požadavkům, stává se stejně mocným (a vzájemně převoditelným), jako každý jiný .Net jazyk. V současné době existují především jazyky C#, Visual Basic a Managed C++, jsou však již k dispozici i jazyky další (Python.Net apod.). Pro jednoduchost bude .Net dále používán jako *jméno jazyka*. Není to přesné, ale protože .Net přesně určuje vlastnosti jazyků, je to trefné.

Jazyky třetí generace z předešlých generací převzaly a mohutně zobecnily dva principy:

Totální objektová orientace

Jazyky třetí generace jsou od nejvnitřnějších struktur objektové, včetně základních jazykových struktur. To přináší ohromnou bezpečnost (v každém okamžiku lze přesně zjistit, s čím se pracuje) a vysokou čitelnost (funkční operace při programování náležejí vždy

tomu prvku, který ovlivňují) [8].

Totální používání ukazatelů

Jazyky třetí generace neznají nic jako globální nebo lokální proměnnou, která by neměla primitivní typ. Nelze vytvořit proměnou s typem záznam. Všechny struktury (tj. data ne-primitivních typů) jsou *vždy* dynamicky alokované. To dovoluje jazykům zmenšit paměťovou náročnost, protože při komunikaci objektů nikdy nedochází k přesunům bloků dat (přenášejí se jen ukazatel). Samozřejmě, plné kopie objektů mohou vznikat a také dle potřeby programátora vznikají [29].

U těchto dvou popsaných principů to však neskončilo. Myšlenky byly rozvinuty a především propojeny. Důsledkem byl vznik dalších důležitých vlastností jazyků třetí generace:

Zrušení vracení dynamické paměti

Tato vlastnost jazyků třetí generace přijde klasickým programátorům jako nejvíce podivná. Ovšem jen na první pohled. Jazyky si vzaly příklad z počítání referencí v komponentových modelech a podpořily tuto funkčnost jako svou *vnitřní součást*. Od toho okamžiku začalo být možné, aby jazyk kontroloval, jak se s instancemi objektů pracuje a kdy tyto instance zanikají. V jazycích třetí generace je všechno objekt; sleduje-li proto jazyk instance, sleduje všechno [8].

Jazyk, resp. jeho běhová část, se o odstraňování již nepotřebných instancí stará sám (mechanismu se říká *garbage collection*, setřásání). Momentálně je technologicky mírně vpředu .Net, který většinu případů použití instancí dokáže odhalit již během překladu a současně s generováním kódu generuje i kód odstraňující instance. Java používá více „rozpojený" přístup, kdy setřásání provádí z druhé strany, pomocí jiného vlákna [16].

Jak je taková automatická práce s pamětí efektivní? Zprvu to nebylo nic moc, ale nyní jsou výsledky impozantní. Java je asi o 10 % pomalejší než shodný program v C++, vezme-li se průměr rozličných testů (matematické, algoritmické, vědecké). V běžných jednoduchých algoritmech je Java zpravidla rychlejší [33]. .Net, pouze při měření rychlosti operací s dynamickou pamětí, dosahuje o cca 30 % až 40 % lepších výkonů, než programátorem optimalizované přidělování paměti s využitím Win32 API.

Reflexe typů

Vlastnost, která není při běžném programování příliš vidět. Bez ní by však nebylo možné vytvořit žádné strojové komunikace, nástroje pro metaprogramování či výkonné vývojové

balíky (frameworks). O co jde?

Jazyky třetí generace poskytují programátorovi *plnou* informaci o třídách. Za *běhu programu* lze s touto informací vytvářet třídy a jejich instance, *aniž* by tyto třídy byly zapsány v kódu; lze přistupovat na data, lze volat metody objektů, které *nebyly v době překladu známy*.

Typické použití této vlastnosti lze najít v předávání objektů mezi programy (*serializace* či *remoting*) nebo při budování komplexních systémů zvenku (např. pomocí konfigurace) [36].

Jazyky třetí generace mají i další společné vlastnosti: generují mezikód (který může ale nemusí být na cílové platformě nativně překládán), ve srovnání s jazyky druhé generace jsou syntakticky vyčištěné, sémanticky omezení, apod. Z pohledu programátora jsou to však ve srovnání se čtyřmi hlavními popsanými principy drobnosti. Nejsou to však drobnosti pro analytiky nebo zadavatele informačních systémů.

Další novinka, kterou jazyky třetí generace přinesly, jsou extenzivní knihovny objektů (funkcí). Rozvíjí se tak myšlenka jazyků druhé generace, které zavedly standardní knihovny. Knihovny jazyků třetí generace jsou natolik bohaté, že nahrazují API operačního systému. Nejen to, kromě API operačního systému jsou v knihovnách prakticky všechny myslitelné opakovatelné úlohy, jako jsou abstraktní datové typy, regulární výrazy, datové proudy, síťová spojení, podpora hlavních komunikačních protokolů, tvorba a obsluha GUI apod. Programátor *neprogramuje nic navíc*, věnuje se jen svému algoritmu [29].

Z existence *mezikódu* a bohatých knihoven plyne poslední princip:

Nezávislost na platformě

Program v jazyku třetí generace běží všude, kde běží jeho jazyková podpora (*runtime*). Program se napíše jednou, přeloží do mezikódu a o mezikód a běh na různých operačních systémech se stará jazyk. Program mimo jazykovou podporu nic víc nepotřebuje, protože její součástí jsou knihovny nahrazující API operačního systému [8].

Například, tato práce je psána v editoru, který je vytvořen v Javě a bez změny kódu automaticky běží v Unixech, Win32 systémech a v MacOS.

1.5 Java je . . .

- Jazyk třetí generace se všemi popsanými principy.
- Jazyk s výchozí syntaxí jazyka C.
- Jazyk s objektovým modelem bez vícenásobné dědičnosti, bez vlastností (properties) a bez přetěžování operátorů.
- Jazyk s podporou rozhraní (interface).
- Jazyk s jazykovou podporou pro prakticky všechny současné platformy.

Celkově vzato, Java je jazyk spíše minimalistický, puritánský. To není na škodu, vlastnost, kterou jazyk nemá, nemůže programátor špatně použít.

Kdyby měl být stručně shrnut technologický přínos Javy (Java vznikla dříve než .Net), jednalo by se především o tyto skutečnosti:

Skrytí práce s dynamickou pamětí

Obsluha dynamické paměti a její používání v jazycích druhé generace vede k mnoha chybám a často k neefektivitě. Java začala paměť spravovat sama.

Absolutní abstrakce cílové platformy

Výsledkem překladu Javy je mezikód, který se až při běhu buď interpretuje, nebo nativně překládá. Napíše-li se v Javě program, skutečně jej lze spustit všude. Příjemným důsledkem této abstrakce je také, že prakticky všechny opakovatelné operace již programátor dostane v jazyce (v jeho knihovnách).

Totální objektovost a typová reflexe

Obě vlastnosti spolu jdou ruku v ruce, pro vyšší programování je důležitá zejména reflexe. Možnost zjistit za běhu o každé třídě úplně všechno, možnost použít objekt, aniž by bylo třeba uvést jeho definici v programu, to jsou věci, které programátorům otevřely zcela novou úroveň přístupu. Pomocí reflexe je možné programovat a pracovat s již hotovými, cizími, binárními objekty. Konkrétní příklad použití reflexe bude popsán dále, v kapitole o frameworku „Spring“.

str. 28
4

2 Třívrstvá architektura

2.1 Technická motivace

V následujícím textu bude dobré uvést do hry *modelový* příklad, nějaký libovolný program, který byl někdy vytvořen. Program patrně k něčemu slouží a tudíž asi něco i dělá. Jeho funkce řeší nějaký konkrétní problém, požadavek. Složitější program bude řešit požadavků či problémů více; je přitom pravděpodobné, že některé z funkcí tohoto složitějšího programu při různých režimech nebo povelech vůbec nebudou použity (zůstávají však k dispozici).

Většinou se rovněž dá očekávat, že program nebude samoúčelný a jednorázový; je efektivnější implementovat *algoritmus* místo konkrétního řešení. Takový přístup ovšem zjevně vyžaduje, aby se algoritmu před jeho během připravila *data*, která mají být zpracována. To se v programu projevuje tak, že program má nějaký *datový vstup*. Třeba jednoduché čtení z příkazové řádky, nebo vstupní soubor. Má-li program grafické rozhraní, bude patrně většina vstupů pocházet od akcí uživatele.

Výsledky běhu programu (tedy důvod jeho existence) obyčejně program někam zaznamená, opět například velice jednoduše do výstupního souboru či příkazové řádky.

Jaký mají popsané děje význam, jak se program použije? Sám od sebe? Kdy se provede jeho funkce? Plánovaně? Když se program rozhodne? Co to vůbec znamená, použít program? Nebude překvapivé, když se v odpovědích na tyto otázky objeví *já spustím* nebo *uživatel spustí* nebo *když je třeba*. Shrnuto, pro využití funkcí programu je třeba někoho, kdo rozhodne o spuštění. Pro použití *výsledků* programu je třeba navíc někoho, kdo ví co s nimi. Program vykonává pokyny **uživatele**, program **slouží**. Uživatel přitom nemusí být osoba, může to být například automatické spuštění (které ale uživatel naplánoval); prostě každý, kdo má zájem na spuštění funkce a kdo to zařídí. Stalo se zvykem nazývat tohoto šikovného iniciátora **klientem**. A program (část programu), který vykonává funkci, který klientovi *slouží*, ten se pak logicky nazývá **server** [7].

Klient připravuje programu vstupní data a také odebírá jeho výsledky. V jednoduchých programech bývají tyto části primitivní a dost často integrované přímo do funkčního kódu. Ve složitějších programech (například má-li být výstupem programu sada HTML stránek) výstupní část s vysokou pravděpodobností autor uceleně *oddělí* od výkoného kódu. Totéž bude platit i o vstupní části; bude-li zpracovávat složitější vstup, bude oddělena od zpracování.

Čím přehledněji a lépe bude program napsán, tím snadněji bude možné uvedené dělení v programu nalézt. Obě části programu, tj. **výkonná část** s funkcemi a **prezentační** část se vstupy a výstup, však zatím zůstávají uzavřeny do jedné monolitické aplikace.

Co se stane, když program pro svou práci bude potřebovat nějakou bázi dat. Přímočaré řešení je bázi dat vzít a instalovat ji spolu s programem. Je zjevné, že tím se naprosto zamezí *sdílení* dat. Někdy to nevadí, někdy to vadí. Vadí to například když je báze dat velká, když je licenčně chráněná nebo když je sdílení dat cílem. Například, pokud bude program vyhledávat supernovy, bude muset pracovat s bázemi dat o velikosti terabajtů a podobně.

Dobrá, program potřebuje pracovat s databází. Kód, který s daty pracuje, se proto stane jeho součástí. Databázový *server* bude běžet na cizím počítači, program se stane databázovým *klientem* a patrně pomocí síťového spojení bude se serverem spolupracovat. Uchopí-li se nyní logika uspořádání, je zřejmé, že jeden úkol využívá pro svůj běh dva počítače. Databázový server je specializovaný, stará se pouze **o úschovu dat**, klientský počítač se svým programem obstarává **zbytek**: výkonnou část, prezentační část, možná i nějaké místní ukládání dat a podobně.

Uvedené uspořádání je celkem časté, například je typické pro levné účetní a podnikové systémy.

2.2 Jiné motivace

Doposud byly při rozvíjení myšlenek dominantní potřeby programu, bylo třeba dosáhnout nějakých funkcí v rámci daného zadání. Funkce programu však nejsou vše. Přesněji, funkce programu odpovídají důvodům vzniku (přání zadavatele), avšak z důvodů vzniku nijak neplyne, jak by měly tyto funkce být naprogramovány. Prostor pro možnosti je zde velký a přirozeně se musí vždy omezovat. Některé další specifikace přicházejí rovněž od zadavatele (zadavatel například požaduje ovládání přes Internet), jiné specifikace si stanovuje programátor (dodavatel programu) sám. Tyto *netechnické* specifikace nakonec bývají silnější, než technické. Proč? Protože netechnické specifikace jsou zpravidla *ekonomické*. Není-li možné program zaplatit, nevznikne. Není možné programovat dva roky, když za půl roku musí být problém vyřešen. A především, programy musejí být dobře udržovatelné a spravovatelné. Čím je programový systém větší, tím silnější požadavek na údržbu a správu je. Dokonce u většiny programových systémů náklady na jejich správu a údržbu přesáhnou po určité době jejich vstupní cenu. Technicky se hovoří o rozdílu mezi TCO (*Total Cost of Ownership*) and

TCA (*Total Cost of Acquisiton*) [47].

Netechnické specifikace posouvají uvažování o skládání funkcí a bloků programu agresivně dále. Co dělat s popsaným modelovým programem, když správce databázového serveru vymění databázi? A co dělat, když se do databáze dostanou data z jiné přehlídky oblohy, která jsou jinak organizovaná? Jednoduše, upraví se klientský program. Je to ale opravdu jednoduché? **Není**. Klientů mohou být stovky. Je možné donutit je všechny program inovovat a vyměnit, je také možné nechat starou i upravenou databázi pracovat současně. Obojí nicméně stojí **peníze navíc**, jak porušení kompatibility (výměna programů), tak zachování kompatibility (dvě databáze). Zjevně by bylo lepší mít celý systém lépe promyšlen a rozdělen tak, aby rozšíření databáze nezpůsobilo takové problémy.

Klientský program je monolitický a **oddělení logiky** získávání dat znamená jej **přepsat**. Možná i docela hodně přepsat, pokud bylo získávání dat integrováno až do částí spolupracujících s uživatelem. Ale, podaří se to. A co vznikne? Vznikne nový klientský program, který bude prezentovat data uživateli a vykonávat jeho příkazy. Dále vznikne *nová mezivrstva*, která na jedné straně bude spolupracovat s databází a na druhé straně s klientským programem. Směrem ke klientovi bude mít tato mezivrstva rozhraní, které výměna databáze již nikdy **neovlivní**.

Kam s touto novou mezivrstvou? Uváží-li se, že potíže s výměnou databáze vznikly s klientským programem u uživatele, bylo by hloupé dávat mezivrstvu opět na stejné místo, k uživateli. Tudíž, nová mezivrstva musí pracovat na **serveru**.

2.3 Více vrstev

Aplikováním technických i netechnických hledisek vznikla v programu *vrstvená architektura* [28]. Vznikla postupně, takže vyvíjený modelový případ obsahuje všechny mezipodoby. Vrstvený program se stal lépe *udržovatelným* — jeho jednotlivé části na sobě nejsou závislé a, což je důležité i pro programátory, i lépe *programovatelným*. To proto, že rozhraní vrstev přehledně říká, co a jak se má použít. Programátor nemusí tápat, kde kterou funkci najít, nebo hůře, kde kterou funkci implementovat.

Modelový program je třívrstevný, nic však zatím není známo o povaze spojení klientského programu s mezivrstvou a nic ani o vnitřku této mezivrstvy. A více se bez znalosti konkrétních požadavků na řešení ani dozvědět nelze. V principu záleží na pouze tom, kolik práce musí být vykonáno v klientském programu. Určitě v něm vždy bude obsluha uživatelského rozhraní,

může v něm být nějaká podpora uživatelských nastavení či akcí a podobně. Může v něm být i nějaká logika, ale vždy když se to dopustí, mohou vzniknout stejné potíže jako s rozšířením databáze. Samozřejmě, povaha této logiky může být taková, že je to vyloučeno a pak je takové řešení zcela bezpečné. Množství dovolené klientské práce může rovněž být ovlivněno možnostmi klientského počítače. Například mobilní telefon toho zvládne zjevně méně než stolní počítač; oba dva přitom zvládnou podstatně více, než webový prohlížeč. Lze tak říci, že použití jednoduchých klientů **vynucuje** přesun logiky do serveru a nepřímo tak podporuje rozvržení programu do vícevrstevné architektury.

Tak či tak propojení klienta se serverem může být různé podle povahy řešeného problému. Trend, který se dá v současnosti vysledovat, nahrazuje proprietární způsoby komunikace standardními a vytváří na straně serveru služby, které klient spotřebovává když potřebuje. Na bázi vícevrstvé architektury tak vznikají *SOA*, Service Oriented Architectures [38]. Programová logika je důsledně skryta do služeb serveru a klient může být velmi jednoduchý.

Již bylo zapsáno dost, aby mohly být poznatky utřízeny a formalizovány. Byla popsána aplikace jednovrstvá, dvouvrstvá a třívrstvá. Co je pro jednotlivé typy charakteristické:

Jednovrstvá architektura, single-tier architecture

Program je monolitický, běží na jednom počítači, je prakticky nemožné sdílet program více uživateli, je prakticky nemožné sdílet více těmito programy nějaká společná data. Program obsahuje podporu databáze, aplikační logiku i prezentační vrstvu. Změny v architektuře systému znamenají úpravy klientského programu.

Dvouvrstvá architektura, two-tier architecture

Program je rozdělen na databázovou část a klientskou část. Obě části programu běží (mohou běžet) na různých počítačích. Pro více klientských programů je snadné sdílet společná data. Pro více klientských programů nemusí být snadné sdílet společné operace (příklad: co dělat, když dvě účetní najednou začnou upravovat tutéž fakturu). Klientský program obsahuje celou aplikační logiku a celou prezentační vrstvu. Změny v architektuře databáze znamenají změnu klientského programu, změny v aplikační logice rovněž znamenají změnu klientského programu.

Třívrstvá architektura, three-tier architecture

Program je rozdělen na klientskou (presentation tier), aplikační (business tier) a databázovou (information tier) část [28]. Všechny tři části mohou běžet na rozdílných počítačích,

rozlišuje se klientský počítač, aplikační server a databázový server. Klientský program obsahuje prezentační vrstvu, aplikační server logiku aplikace a databázový server uchovává data. Má-li dojít ke změně chování aplikace, stačí upravit jen aplikační server. Má-li dojít ke změně vzhledu aplikace, stačí upravit jen klientskou aplikaci.

Nutno podotknout, že kromě ekonomického a programátorského přínosu lze v třívrstvé architektuře vysledovat také jaksi více pořádku.

Aplikační server ve třívrstvé architektuře nese logiku aplikace, současně je však mezivrstvou mezi prezentační a informační vrstvou. Proto typicky bývají i aplikační servery rozděleny podle podobného schematu na vnitřní vrstvy:

Vrstva rozhraní

Slouží v aplikačním serveru k překladu výsledků aplikační logiky do podoby srozumitelné klientům a naopak, k překladům klientských volání do volání aplikační logiky.

Tato vrstva se obyčejně liší podle toho, jakého klienta má obsloužit. Je-li klientem stroj (program), bývá vrstva programová, realizovaná například pomocí protokolu SOAP [39] a navázané technologie WebServices [51]. Je-li klientem přímo uživatel (třeba webový klient), je součástí vrstvy modelová část webového rozhraní.

Vrstva aplikační logiky

Skutečná aplikační logika, neboli procedury či algoritmy, které vytvářejí funkci programu. Navenek je vrstva připojená do vrstvy rozhraní, dovnitř vrstva využívá vrstvu databázové abstrakce.

V této vrstvě není prakticky nic zajímavého, obsahuje pouze algoritmy.

Vrstva databázové abstrakce

Vrstva, která zabaluje databázová volání, obyčejně s využitím nějakého objektově-relačního mapování, do funkcí. Navenek, pro vrstvu aplikační logiky, vrstva poskytuje binární data, která jsou výsledkem předem daných databázových dotazů.

Neboli, obsahem vrstvy jsou databázové dotazy a povely, které vyjadřují funkce rozhraní vrstvy.

Důvody k vnitřní struktuře aplikačního serveru jsou podobné jako důvody vedoucí ke vzniku samotné třívrstvé aplikace. Například, je-li aplikační server správně strukturován a dojde-li k výměně databázového serveru nebo objektově-relačního modelu, stačí upravit pouze

jedinou vrstvu: vrstvu databázové abstrakce. Omezení zásahů do programu při změnách okolí jen na některé jeho části je důležitý prvek tvorby bezpečných programů.

3 Java a podniková řešení

Rozdělení činností programu do několika vrstev je univerzální a není spojeno s žádným upřednostněným programovacím jazykem. Protože je však takové dělení do značné míry věcí disciplíny programátora, má smysl vyhledávat správné vzory implementací a snažit se je opakovaně používat. Je přirozené, že použití vzoru usnadňuje programátorovi jeho práci: místo řešení pomocných a infrastrukturních otázek se může plně soustředit na svůj úkol.

V roce 1998 komunita vývojářů v Javě dala vzniknout uskupení JCP — Java Communition Process [44]. Toto uskupení má v současnosti přes 700 členů (včetně společností Google, IBM nebo Intel) a má jediný cíl: společně vyvíjet a revidovat specifikace, referenční implementace a testy Java technologií. Pochopitelně, v zájmu všech členů JCP je, aby vyvinuté specifikace a vzory byly co nejlepší, protože si je navrhují pro sebe. Výsledkem práce JCP jsou JSR, *Java Specification Requests* [44]. Prostředí JCP je velmi vhodné pro tvorbu obecných a architektonických návrhů, a často se tak také děje. Prakticky všechny současné Java standardy (včetně uspořádání a obsahu samotných vývojových platforem) pocházejí z dílny JCP.

Java měla od svého vzniku problém prosadit se, protože v majoritním segmentu uživatelů počítačů dominovaly (a dominují) operační systémy společnosti Microsoft. Java prosazovala nezávislost na operačním systému, na svou dobu podivné praktiky (o paměť se stará jazyk) a mimo to byla docela pomalá. Je jasné, že Java musela v informačních technologiích hledat své místo. Našla ho právě s přispěním JCP ve dvou oblastech (je užitečné sledovat data a čísla jednotlivých JSR specifikací): **v serverových nasazeních** všech druhů (zde vedla cesta přes webové servery) a naopak ve spartánských podmínkách **mobilních zařízení** (v úvodu bylo uvedeno, že Java původně vznikla jako programovací nástroj pro spotřební elektroniku, [43]).

3.1 Java Enterprise Edition

V předešlém textu již bylo řečeno, co by to asi serverová nasazení mohla být; **ideálně** sem spadají všechny střední (aplikační, business) vrstvy, viz kapitolu „Více vrstev“. Java, JSR a *n*-vrstvé aplikační modely se ideálně setkaly a v dnešní době nabízejí prakticky nedostižně bohaté, otevřené a důsledně otestované nástroje a vzory pro vícevrstvé aplikace. Souhrnný název pro toto řešení je J2EE, *Java Enterprise Edition* [22]. Je vcelku přirozené, že J2EE není nic pevného ani svazujícího. Není to programovací jazyk, není to postup. J2EE je souhrn mnoha technologií, které ve svém celku podporují rychlý vývoj podnikových aplikací.

str. 19
2.3

Většina z těchto technologií prošla JCP a je implementována v několika vydáních různými společnostmi (stačí si vybrat). Skládací princip J2EE (či spíše zastřešující povaha) také dovoluje, aby každý tvůrce aplikace použil jen to, co se mu zrovna hodí, a případně, aby to, co se mu z J2EE zrovna nehodí, nahradil nějakým lepším řešením. Je pravda, že J2EE se málokdy použije jako celek, spíše se z dobře definovaného standardu vybírají bloky řešící některou konkrétní část aplikace.

Co se však z J2EE bere prakticky **bezvýhradně** jako celek, jsou návrhové vzory a doporučené dělení do vrstev. Postupně tak vznikla situace, kdy J2EE představuje uznávané a dodržované myšlenky a pravidla, které jsou podpořeny tím, že J2EE nabízí rovnou i některá z možných řešení.

J2EE se nasazuje v serverech, takže mnoho jejích částí odkazuje k serverům, k serverovému zpracování či webovému rozhraní. Další velká skupina J2EE řeší interoperabilitu s databázemi, ať se již jedná o objektově-relační mapování, jmenné prostory nebo samotná databázová připojení. Zajímavé též je, že servery, které provozují J2EE aplikace, jsou též produktem specifikace J2EE. Jaké hlavní technologie J2EE lze vypíchnout (neboli, co bývá typicky třeba):

JSP, Java Server Pages (JSR 245 [44])

Technologie, která slouží k dynamické tvorbě HTML stránek. Poskytuje přístup k objektům (především jejich datům), které jsou do JSP procesoru dodány při tvorbě stránky.

JSTL, Java Standard Tag Library (JSR 52 [44])

Doplňuje JSP o algoritmické struktury (podmínky, cykly apod.), takže spolu s JSP dovoluje uvnitř vytvářené stránky přímo programovat.

JSTL používá obecný mechanismus *knihoven tagů* (tag libraries), který může využít každý programátor k zapouzdřování opakujících se částí svých stránek. To zjevně vede k vyšší přehlednosti a lepšímu strukturování kódu stránek.

Kromě JSTL jsou již hotovy a připraveny například knihovny tagů pro formátování výstupu, pro práci s XML a podobně.

Servlets (JSR 154 [44])

Abstrakce, která umožňuje JSP stránkám fungovat. Servlet je vnitřní část serveru, která vzniká při běhu (a případně se jen udržuje) z JSP souboru jeho překladem. Servlet je přeložený kód, který vlastnící server při požadavku od klienta spouští. Výsledek práce

servletu je klientovi odesílán jako HTML stránka.

Součástí technologie servletů je i jejich správa. To obnáší komunikaci s JSP stránkami, jejich překlad, životní cyklus servletů a jejich vazbu na další části J2EE aplikace. Pro běh servletů je nutný server, který takovou správu podporuje; jedná se například o server *GlassFish* (Sun) nebo *Tomcat* (Apache).

JDBC, Java Database Connectivity

Definice databázových zdrojů a správa připojení do databáze.

Common Annotations for the Java Platforms (JSR 250 [44])

Každá třída (člen třídy) v Javě může být opatřen *anotacemi*. Anotace je dodatečná, pomocná informace, která je typicky využita za běhu aplikace jako metainformace k ovládání nadřazených struktur (frameworků, dokumentačních nástrojů apod.). Anotace má podobu objektu, který se při překladu stane součástí *typové informace* (viz reflexi v kapitole „Jazyky třetí generace“) objektu. Kdokoli je tak později schopen v objektu anotaci najít a použít.

str. 13
1.4

JPA, Java Persistence API (JSR 220 [44])

Důležitá část databázové části aplikačních serverů, využívá anotačního mechanismu (viz předchozí bod). JPA pomocí anotací zapisuje u tříd a členů informace o způsobu jejich ukládání do databáze (perzistence). V J2EE je výchozí způsob ukládání definován, ale některé skutečnosti je dobré určit explicitně. Mimo to, složitější databázové vazby nelze jinak než přesným zápisem podchytit.

JPA je využíváno i jinými technologiemi, které slouží pro ukládání dat v J2EE aplikacích, například v Hibernate. Hibernate je svébytný a komplexní způsob databázové abstrakce a ukládání dat, který dodržuje J2EE specifikace. Je proto s JPA záměnný a často se také místo JPA používá.

SAAJ, SOAP with Attachments API for Java (JSR 67 [44])

Podpora pro SOAP. SOAP je protokol spravovaný konsorciem W3C (uvedený společností Microsoft), který slouží pro výměnu dat a vzdálená volání funkcí pomocí XML. SOAP se typicky přenáší v HTTP, ale není to nutnost. SOAP je používán především při výměnách strojových informací (při komunikaci programů).

WebServices (JSR 224 [44])

Jsou těsně sjpaty se SAAJ, WebServices slouží k obsluze vzdálených volání, jejich pu-

blikaci a popisu (pomocí WSDL popisovačů) a samozřejmě k překladu Java volání do SOAP zpráv a naopak. Jestliže SOAP slouží k přenosu informace mezi programy, tak WebServices jsou inteligentní konce tohoto přenosového kanálu, které o sobě umějí říci, co jimi může téci.

J2EE, neboli Java pro podniková řešení, je silně rozšířená a její použití stále narůstají. Důvod je zřejmý: J2EE nabízí pozadí, které dovoluje vytvářet aplikace velmi rychle a efektivně.

3.2 Kde J2EE rozběhnout

Běh J2EE programu je rovněž podchycen pomocí J2EE specifikací. Je třeba mít připraven program, který umí J2EE aplikaci vzít, vytvořit (všechna napojení na databázi, JSP a servlety i výkonné objekty) a podle požadavků klientů (WebServices, HTTP, tedy typicky HTTP) rozbíhat a vykonávat jednotlivé funkce. V praxi se objevily dva odlišné přístupy.

První varianta rozbíhá J2EE aplikaci pomocí jediného zastřešujícího programu, *aplikačního serveru*, který se stará o vše potřebné. Aplikační server je samonosný, po instalaci databáze již k běhu J2EE aplikace není nic více třeba. Tento přístup je výhodný, protože se programátor nemusí starat o nic více, než o svou aplikaci. Aplikační server převezme potřebné informace z anotací uvnitř aplikace. Současně je tento přístup nevýhodný, protože aplikační server musí obsahovat celou J2EE specifikaci a aplikace to nemusí potřebovat. Mohutný server se v takovém případě rozbíhá třeba jen proto, aby zajistil vznik a zánik funkčních objektů. To je samozřejmě neefektivní. Jiná nevýhoda je architektonická: anotace pro aplikační server a odkazy na API serveru jsou součástí zdrojového kódu programu. Program se tím stává závislý na konkrétní verzi J2EE specifikace, změny ve specifikaci mohou vynucovat nový překlad či dokonce úpravy programu. Aplikační servery vyrábí několik společností, svůj aplikační server má Sun, IBM či Oracle. Sun také inicioval vznik *open source* aplikačního serveru pojmenovaného *GlassFish*.

Druhá varianta není shrnuta do jediného programu, nepoužívá aplikační server. Prostředky a nástroje nutné pro J2EE aplikaci jsou zapouzdřeny do *modulu*, který má samostatnou konfiguraci a který může být rozběhnut uvnitř různých prostředí. Tato prostředí *modul* rozběhnou a *modul* sám se stará o J2EE logiku. Pochopitelně, každé prostředí má různou podporu pro různé kousky J2EE a modul to musí být schopen akceptovat. Součástí konfigurace modulu je proto vždy způsob napojení na běhové prostředí a výčet služeb a funkcí, které modul využívá. Tento přístup je výhodný, protože je možné připravit modul právě jen pro konkrétní aplikaci.

Neběží tedy zbytečně nic navíc. Přístup je výhodný také proto, že konfigurace může obsahovat J2EE informace, které **nemusejí** být ve zdrojovém kódu. Program není závislý na J2EE nástroji. Současně je tento přístup nevýhodný, protože vyžaduje kromě *modulu* ještě i nějaký další program, který je nutné instalovat. Jiná nevýhoda spočívá v samotné přítomnosti konfiguračního souboru: konfigurační soubor je soubor navíc a mimo to obsahuje informace o vnitřním uspořádání programu, takže se některé údaje musejí duplikovat (ve zdrojovém kódu a v konfiguraci). Příkladem *modulu*, který poskytuje J2EE podporu a který může být běžet v různých prostředích, je framework *Spring* [41].

Mezi oběma přístupy existuje soupeření (což je pro vývoj technologií nutné) a v mnoha případech je výběr jednoho nebo druhého důsledkem osobních preferencí architekta. Například, druhý přístup ruší závislost zdrojových kódu programu na okolní technologii. To je pro mnoho programátorů velmi důležitá věc a raději proto instalují něco navíc, než aby svůj kód s něčím provazovali. Nicméně, dosažený výsledek je v obou přístupech shodný.

4 Spring

4.1 Co je to Spring?

V roce 2002 Rod Johnson, architekt škálovatelných webových aplikací, vydal knihu „Expert One-on-One J2EE Design and Development“, ve které kriticky rozebral J2EE (zejména v té době používanou její část EJB 2.0). Stalo se totiž, že vývoj J2EE aplikací se zvrtl a návrhové vzory, sestavené jako soubor myšlenek a možných konceptů, začaly diktovat způsob návrhu a řešení. Došlo k významovým posunům, J2EE aplikace začaly být těžkopádné, neefektivní a obtížně udržovatelné. Například, jeden z významových posunů způsobil, že jednotlivé vrstvy uvnitř aplikační logiky (uvnitř jednoho serveru) spolu komunikovaly pomocí nástrojů pro vzdálené volání metod. Což je samozřejmě mrhání výkonem. Fakt, že J2EE tohle snadno umí, neznamená, že to je správný postup [12].

Rod Johnson analyzoval všechny vrstvy J2EE teoreticky i prakticky (sám je architekt) a vyhlásil ve své knize jakýsi návrat ke kořenům. Některé jeho myšlenky o J2EE jsou (a platí i obecně):

- J2EE byl měla být používána k vytváření programů podle připravených objektových návrhů, J2EE by objektový návrh neměla určovat.
- Distribuované řešení by se nemělo použít, dokud si je logika plynoucí ze zadání nevynutí.
- Při analýze řešení (aplikační logiky) by neměly být navrhovány ani třídy ani technologie, ale pouze funkční rozhraní.
- Ať se použije jakákoli strategie ukládání dat, vždy by měla být od aplikační logiky oddělena abstraktní vrstvou, aby došlo k úplnému skrytí detailů databázového zpracování.
- Jakákoli návrhová strategie selže, pokud není dobře sestaven objektový návrh. A naopak, dobrý, jednoduchý objektový návrh dokáže překlenout mnohá úskalí návrhových strategií.

Myšlenky z knihy [12] jsou vesměs jasné. Vtip je v tom, že ne vždy se myšlenky dostaví samy a dokud je někdo nezapíše, ignorují se.

Na základě uvedené knihy začal být vyvíjen framework Spring, který si klade za cíl být právě tak lehký, flexibilní a nevnucující se, jak je v knize psáno. Velmi dobře motivaci vzniku

Springu popisuje Spring sám [41]:

- J2EE by měla být snadno použitelná.
- Jediná správná cesta je vytvářet rozhraní, ne třídy. Spring snižuje složitost používání rozhraní k nule.
- JavaBeans představují skvělý způsob, jak konfigurovat aplikace.
- Objektový návrh je důležitejší, než jakákoli implementační technologie, včetně J2EE.
- Využívání výjimek je v Javě přemrštěné. Framework by neměl nutit programátora zpracovávat výjimky, když není zřejmé, jak se z nich zotavit.
- Testování je základ, a framework, jako je Spring, by měl testování usnadňovat.
- Používat Spring by mělo být příjemné.
- Zdrojový kód programu by **neměl** být závislý na API Springu.
- Spring by neměl soupeřit se stávajícími dobrými technologiemi, naopak, měl by podporovat integraci. (Například, JDO, Toplink, a Hibernate jsou skvělá objektově relační mapování. Není třeba vyvíjet žádná další.)

Jaká je tedy filozofie návrhu aplikace pro framework Spring? Jednoduchá.

- Objektový návrh, správně rozřazený do jednotlivých aplikačních vrstev definujících datové objekty, operace a funkční toky, bude implementován v čistých Java třídách (POJO) a rozhraních.
- Operacím a funkcím budou odpovídat rozhraní, která budou mít implementaci v nezávislé třídě. Tyto funkční implementace budou používat datové objekty a budou obsahovat jen logiku podle zadání.
- V programu se objeví rovněž rozhraní a nezávislé třídy s implementací, které oddělí aplikační logiku od databázového zpracování. A opět, třídy tohoto databázového rozhraní budou řešit pouze logiku ukládání a získávání dat.
- Aby mohl program komunikovat s uživatelem, bude nutnou součástí programu prezentační vrstva s jednou nebo více součástmi, podle toho, jací budou klienti. Prezentační vrstva bude velmi jednoduchá, protože navenek, směrem k uživateli, vždy bude pouze překládat požadavky nebo výsledky již hotových funkcí programu (tyto funkce jsou shrnuty v rozhraních aplikační logiky).

Stojí za povšimnutí, že se dosud vůbec neřešilo, jak který objekt vznikne, jak se spolu domluví, kdo vůbec všechny třídy oživí ani kde se vezme HTTP rozhraní, či jak se databázová logika spojí s databází. V modelovém kódu se také nijak neřešilo, že bude pracovat v nějakém

Springu, přestože pro něj byl kód navrhován. Žádné anotace nebo API. Kód programu obsahuje jen vlastní program. Spring je důležitý, ve zdrojovém kódu se o to ovšem nikdo nemusí starat.

4.2 Jak na Spring?

Nyní, když je program hotov, je třeba pro něj a podle něj vytvořit konfigurační soubor Springu. Konfigurace obsahuje formálně prvky stejného druhu (*beans*), přesto však není obtížné vyčlenit tři odlišné skupiny těchto prvků:

- V prvé řadě, konfigurace musí obsahovat položky pro všechny objekty aplikace, které implementují nějaké vnitřní či vnější rozhraní. Typicky se tak součástí konfigurace stanou objekty aplikační logiky a databázové abstrakce.
- V druhé řadě, v konfiguraci musí být zapsáno, jaké připravené části Springu, resp. produkty třetích stran, mají být provázány s objekty aplikace. Spring integruje prakticky všechny hlavní technologie používané v J2EE řešeních, takže se většinou bude jednat o připravenou část Springu. Pomocí těchto objektů se do aplikace dostane například Hibernate [18] (objektově-relační mapování, [32]), rozhraní WebServices (například open source implementace Apache-Axis), rozhraní pro web (například nativní Spring MVC controller) nebo správce transakcí (JTA jako součást Springu).
- Nakonec, součástí konfigurace se stanou rozličné pomocné drobnosti, které doplní jednotlivé deklarované objekty. To budou například různé interceptory volání, případně podpora logování a podobně.

Jak dochází k propojování objektů? Cituji opět motivaci Springu [41]: *JavaBeans představují skvělý způsob, jak konfigurovat aplikace.* Spring využívá dohody komunity Java vývojářů, že veřejné datové položky Java tříd budou dostupné pomocí čtecích (*getter*) a zapisovacích metod (*setter*) při dodržení jmenné konvence. Spring přečte svůj konfigurační soubor, nalezne odkazy na datové položky tříd a protože JavaBeans říkají, jak jsou tyto datové položky dostupné, rovnou data pomocí správné metody zapíše. Tahleta část Springu stojí a padá s klíčovou vlastností Javy jakožto jazyka třetí generace: bez **reflexe** by nebyly metody *setter* a *getter* dostupné a celou konfigurační stavbu by nebylo možné vybudovat.

Popsaný mechanismu konfigurace Springu je přirozený (přinejmenším proto, že tak byl v textu vysvětlen). Není to nicméně jediný způsob, jak seskládat J2EE aplikaci. Technologie EJB (vůči které se Rod Johnson vymezoval) nemá konfigurační soubor a informace o propojení

objektů vyčítá z anotací. Protože EJB byla na světě dříve, je tento způsob považován za *přímý*. Spring, který se objevil později, naproti tomu aplikuje postup *opačný*: nepřebírá informace z kódu, ale naopak je do kódu *dodává*. Protože Spring, ve srovnání s EJB, postupuje naopak byl jeho postup nazván *IoC*, *Inversion of Control* a technologie dodávání informací pak *DI*, *Dependency Injection* [20].

5 Java a mobilní zařízení

V kapitole o podnikových řešeních již bylo zmíněno, že Java získala převahu v serverových a mobilních nasazeních. Serverová nasazení byla vcelku probrána a zůstávají nasazení mobilní.

Operační systémy mobilních zařízení jsou ve srovnání s běžnými stolními operačními systémy primitivní. Většinou mají logikou práce blíž k jednoduchým systémům reálného času, než k univerzálním systémům. Časem nebude mezi operačním systémem v telefonu a systémem v PC rozdíl, dosud tam ale technologie nepokročila.

Mobilní telefony a jejich operační systémy tvoří skládačku. Všichni výrobci mají mnoho různých modelů telefonů s různými schopnostmi. A všichni výrobci mají jeden, dva, možná tři operační systémy, které do svých telefonů dávají. Aby výrobci dosáhli variability musejí mít operační systém modulární a proměnlivý.

Přesně podle tohoto vzoru je vybudována J2ME, *Java Platform Microedition*. Jako skládačka. Výrobci telefonů vlastní implementaci běhového prostředí Javy, ve kterém se spouštějí moduly — knihovny tvořící rozhraní J2ME. Běhové prostředí je přirozeně jednoduché, protože jediný jeho úkol je interpretace Java mezikódu. Podoba knihoven J2ME opět pochází z JCP (Java Community Process), a je proto konsenzem všech výrobců mobilních zařízení. Modularita zařízení se v J2ME projevuje přítomností více JSR specifikací; prakticky pro každou vlastnost zařízení existuje samostatně specifikované J2ME API.

Asi nebude překvapivé, že kromě modulů pokrývajících některou část funkce mobilních zařízení existují také moduly, které specifikují **základní platformu**, neboli API společné všem mobilním zařízením. Těchto základních API je málo, takže nebude problém je vyjmenovat:

CLDC, Connected Limited Device Configuration

Hlavní API pro malá zařízení s omezenými zdroji, to jest pro mobilní telefony. API obsahuje základní Javu (abstraktní datové typy, vstupně-výstupní operace a síťová připojení).

CDC, Connected Device Configuration

API pro zařízení s většími schopnostmi, s možností trvalého připojení. To v současnosti pokrývá *Smartphones* nebo zabudovaná zařízení se síťovým připojením. API přidává k CLDC plnou podporu síťování, certifikátů, reflexe či práci s textem.

MIDP, Mobile Information Device Profile

Druhá nejdůležitější část API pro mobilní telefony. MIDP dodává operace specifické pro

mobilní telefony, což je například podpora ukládání uživatelských dat, podpora pro GUI (abstrakce pro LCD obrazovky telefonů) nebo podpora pro ovládání médií v telefonu.

K těmto základním API, která musejí vždy být přítomna (v mobilních telefonech CLDC/MIDP), může každý výrobce telefonu přidat libovolný rozšiřující kousek Java API podle toho, co telefon umí. J2ME tak například definuje API pro OpenGL (JSR 239), API pro přístup k BlueTooth (JSR 82), WebServices API (JSR 172) nebo Advanced GUI (JSR 209) [44].

Některá z API jsou prakticky ve všech telefonech, jiná jen v některých. Například API pro vzdálená volání služby WebServices bývá dosud v telefonech zřídka, přestože se jinak jedná o výchozí způsob programové komunikace.

Pro tvorbu Java programů v mobilních telefonech je patrně nejdůležitější API MIDP, protože obsahuje jednak třídu `Midlet` (což je běhová jednotka, něco jako program či vstupní bod programu pro mobilní telefon) a jednak skupinu tříd pro tvorbu GUI. Veškerá komunikace s uživatelem telefonu je pomocí těchto tříd abstrahována do obrazovek (`Alert`, `Screen`, `Form`) a ovládacích prvků (`List`, `Image`, `TextBox`). Tvorba uživatelského rozhraní s využitím této abstrakce je velmi pohodlná. Prakticky postačuje pouze rozložit ovládací prvky do správných formulářů (obrazovek) a v reakcích na uživatelské zásahy zajistit jejich přepínání.

Mimo to, všechna stávající vývojová prostředí pro Javu (např. NetBeans) integrují balíčky pro vývoj MIDP aplikací, takže mnohdy ani programování přechodů mezi formuláři není nutné. Například, Mobility Pack pro NetBeans vytváří obrazovky a přechody mezi nimi pomocí grafického nástroje, takže programátor se může opět soustředit jen na svůj program [52].

PRAKTICKÁ ČÁST

1 O řešení obecně

Povaha a specifikace zadání poměrně silně předurčují, jakým způsobem by program měl být vyřešen. Řešení však není nalinkované, v mantinelech požadavků je stále dost prostoru, na druhou stranu je mnoho skutečností i bez další analýzy zřejmých:

- Řešení musí být vypracováno v jazyce **Java**.
- Systém bude mít mobilní část, takže bude třeba využít **J2ME** [46].
- Na jedné straně budou mobilní klienti, na druhé straně bude funkční server. Řešení bude **distribuované**.
- Kromě mobilních klientů bude možné se systémem pracovat přes web, bude proto existovat i druhý, **webový klient**.
- Existence dvou nezávislých klientů, distribuovanost a požadavek na ukládání data vedou k jasné aplikaci **třívrstvé architektury**.
- Java pro tvorbu serverových vícevrstvých řešení nabízí sadu specifikací, API a návrhových vzorů **J2EE**. To je přesně to, co požaduje zadání práce, a současně dochází k ideálnímu překrytí s dosud známými aspekty řešení.
- J2EE potřebuje běžet v nějakém běhovém prostředí, bude tedy třeba zvolit některý z možných přístupů pro J2EE řešení (viz též kapitolu „Implementace serveru“).

str. 42 5

Vývoj každého programového systému by měl splňovat určité formalizmy (metodologie, postupy). Formalizmy na jedné straně dovolují zadavatelům získávat zpětnou vazbu a na druhé straně zpřehledňují samo programování. Při řešení práce nebyla žádná konkrétní metodika použita, přesto byly obecné principy při návrhu objektových systémů dodrženy. Navíc, celý systém má několik vnějších rozhraní, má vnitřní strukturu a je distribuovaný, takže formální postup návrhu je prakticky nutnost.

Jak se při návrhu a vypracování řešení postupovalo?

- Byla zvážena východiska. Při vývoji každého programového systému je kromě specifikace zadání a řešení důležité prozkoumat motivace, cíle, případně uskutečnitelnost jednotlivých požadavků na řešení. Je zřejmé, že bez přihlédnutí k těmto skutečnostem nemusí řešení vyhovovat.
- Byl sestaven objektový návrh, aby se zjistilo, s jakými daty bude aplikace pracovat a jaké případy použití lze očekávat.
- Podle případů použití bylo navrženo funkční rozhraní aplikační logiky, rozhraní mo-

bilního a webového klienta.

- Byla připravena a v kostře sestavena konfigurace běhového prostředí (databázový server, HTTP-servlet server, J2EE framework a objektově-relační mapování), aby se vyvíjený systém mohl po částech ladit a testovat.
- Byla vytvořena a ověřena informační vrstva serverové aplikace (to jest vazba datových objektů na objektově-relační mapování a datové operace s datovými objekty).
- Vznikla mobilní část (mobilního klient) systému. Jako párová část v serveru současně vznikala implementace funkčních volání rozhraní pro tohoto klienta.
- Byly vytvořeny stránky webového rozhraní (to odpovídá případům použití objektového návrhu) a tyto stránky byly doplněny o další pomocné stránky (chybová hlášení, přechodové stavy apod.).
- Pro dané webové stránky byla navržena a vytvořena obsluhující vrstva (sazba HTML dokumentů, spolupráce s formuláři), k této vrstvě současně postupně vznikala implementace jednotlivých funkčních volání webového rozhraní.
- Stránkám webového rozhraní byla dána definitivní podoba, byly rozloženy prvky stránek a byly aplikovány CSS (kaskádové styly).

Výše popsané kroky byly vedeny v uvedeném pořadí. Následující popis řešení však toto pořadí implementace nerespektuje zcela přesně, protože bylo pro přehlednost a logiku výkladu lepší seskupit některé věci jinak.

Poslední obecnou věcí, která musela být během řešení rozhodnuta, je vzájemná vazba mezi telefonem (klientem) a serverem. V úvahu přicházelo přihlašování uživatelů z telefonu, použití telefonního čísla telefonu nebo nějaká jiná identifikace. Postupně se jako nevhodné ukázalo přihlašování (protože opakované přihlašování a vypisování údajů by v poměrně spartánském prostředí telefonu obtěžovalo a kromě toho by nebylo snadné tato přihlašovací data v přenosovém kanále skrýt) i použití telefonního čísla (tohle číslo není možné v prostředí Javy přečíst, patrně z bezpečnostních důvodů).

Zůstal *jiný způsob* identifikace. Bylo zvoleno spojovací číslo, které server generuje při založení uživatele a které je třeba zapsat do konfigurace telefonu. Telefon si spojovací číslo pamatuje, takže jeho konfigurační nastavení se typicky učiní jednou při prvním startu klientské aplikace. Z hlediska bezpečnosti přenosu dat číslo také vyhovuje, protože nenese žádná osobní data, nelze se pomocí něj nikam přihlásit, jeho platnost je omezená pouze na okamžik přenosu změřeného údaje.

2 Organizace řešení a vývojové nástroje

Pro práci bylo použito jediné vývojové prostředí NetBeans [52], což je univerzální, bohatě vybavené a konfigurovatelné IDE pro vývoj všech Java aplikací. Vývojové prostředí je stabilní, přehledné; jediné, co v něm působilo potíže, bylo inverzní barvení zdrojových textů (s černým pozadím).

Java organizuje své přeložené programy v knihovnách (JAR soubory, stromová adresářová struktura komprimovaná ZIPem) do *balíčků*, *packages*. Balíčky logicky člení řešení, mají však také funkční význam, protože pro jednotlivé Java třídy vytvářejí jmenný prostor. Třídy uvnitř jednoho balíčku se navzájem vidí přímo, bez uvádění dovozů a vazeb. Organizaci do balíčků využívá především programátor, nicméně stalo se zvykem, že stromová struktura jednotlivých *packages* je vyjádřena rovněž pomocí složek na disku. Důsledek této konvence je snadná orientace mezi fyzickým umístěním třídy (zdrojového souboru s třídou) a jejím logickým zařazením do programu (umístěním do příslušného balíčku). Ve své práci tuto konvenci dodržuji, takže všechny soubory na disku respektují vnitřní strukturu Java *packages* řešení.

Pokud je dále v popisu řešení zmíněn zdrojový soubor, nebo Java třída, je možné ji najít v souboru shodného jména ve složce, která odpovídá popisované části řešení (na složky řešení bude v popisu odkazováno). Protože zdrojové kódy jsou ve srovnání s hladkým textem řídké a dlouhé, nebudou v popisu práce použity. Každý čtenář má rychlou a snadnou možnost prohlédnout soubory ve svém oblíbeném textovém editoru.

Práce se logicky rozpadá do dvou bloků: mobilní část a serverová část. V samostatných projektech jsou dále odděleny dokumentace a objektový návrh. Protože bloky používají různé bázové technologie, jsou pro vývoj odlišeny druhem vývojového projektu:

Objektový návrh [30], projekt *MDGServerUML*

Projekt s objektovým návrhem, složka je přímo použitelná jako projekt pro NetBeans s rozšířením UML. Soubory nejsou moc zajímavé, NetBeans si je spravuje samo.

str. 40
4

Objektový návrh je popsán v kapitole „Objektový návrh“.

Serverová část, projekt *MDGServer*

Projekt pro celou serverovou část, složka je přímo použitelná jako projekt pro NetBeans. Ve složce <src> jsou uloženy zdrojové kódy členěné podle jednotlivých vrstev, složka <web/WEB-INF> obsahuje hlavní běhové konfigurace a složka <web/WEB-

-INF/jsp> obsahuje předlohy webových stránek.

str. 42 5
str. 47 6
str. 56 8

Řešení serveru je popsáno v kapitolách „Implementace serveru“(vlastní popis řešení, tj. rozložení do vrstev, použité vzory), „Běhové prostředí serveru“(jak a kde aplikaci rozbíhat, tj. co s webovým serverem, databází a J2EE podporou) a „Webový klient a jeho obsluha“(popis webového rozhraní aplikace).

Mobilní část, projekt *MDGClient*

Projekt pro mobilní aplikaci, složka je přímo použitelná jako projekt pro NetBeans. Zdrojové kódy jsou součástí složky `<src/webserviceclient>`.

str. 52 7

Implementace klienta je součástí kapitoly „Mobilní klient“.

Dokumentace, složka *MDGDoc*

Složka, která neobsahuje žádný projekt, pouze je v ní uložen zdrojový kód dokumentace projektu, což je zdrojový kód této práce v XML. Práce je zapsána v dokumentačním systému, jehož jsem autorem.

3 Východiska

Pro mobilní sběr glykemických dat včetně jejich zpracování na serveru s přístupem přes web bylo uváženo následující:

- Typický diabetik je člověk staršího věku. Aby bylo celé řešení použitelné, musí být rozhraní v mobilním telefonu natolik jednoduché a srozumitelné, aby nevytvořilo bariéru.
- Mobilní telefony současné doby nejsou univerzálně vybaveny žádným dostatečně abstraktním komunikačním rozhraním. Mobilní aplikace proto musí v komunikační části značně nezávislá, jinak by její nasazení bylo omezené jen na několik (většinou dražších) modelů telefonů.
- Dálkový bezdrátový přenos dat je potenciálně nespolehlivý, program v mobilním telefonu proto musí být na selhání připraven a především, musí o potížích uživatele jasně a srozumitelně informovat.
- Serverová část není vidět (vidět je její rozhraní), přesto je však rozumné snažit se o řešení, které nebude náročné na zdroje a bude nasaditelné potenciálně kdekoli (u velkého počtu poskytovatelů).
- Webové rozhraní aplikace nebude žádná pomocná část, naopak bude to hlavní část, kterou bude využívat lékař.
- Pro webové rozhraní platí částečně totéž, co pro mobilní část. Uživatelé nebudou počítačoví odborníci, takže obsah rozhraní by měl být jasný a informativní, forma přehledná a návodná a složitost minimální. (I když, taková rozhraní si zaslouží každý, i počítačoví odborníci.)
- Protože systém spojuje více osob v různých rolích, mělo by být webové rozhraní patřičně strukturováno. V rozhraní musí být dovoleny jen takové vztahy a operace, které neporuší soukromí uživatelů a které zamezí neoprávněným zásahům.
- Jelikož není předem známo, kde by měl systém běžet, má smysl uvažovat i o dobré abstrakci databázového připojení [18].

4 Objektový návrh

Přestože by se východiska z předešlé kapitoly mohla zdát složitá, není tomu tak. Objektový návrh, který definuje rozložení dat a funkcí systému, je totiž relativně jednoduchý.

Návrh byl proveden v jazyce UML (Unified Modeling Language, [49]). Byly vytvořeny diagramy tříd a případů použití. Diagram tříd popisuje datové objekty (*domain objects*) a případy použití, které, jak se později uvidí, odpovídají obrazovkám webového rozhraní.

4.1 Doménové objekty

‡ Doménové objekty jsou součástí složky `<src/domain>`.

Všechny doménové objekty mají unikátní identifikaci (člen `Id` tříd) a všechny jsou v kódu zapsány jako jednoduché Java třídy (POJO) bez funkčních metod. Každému datovému členu třídy odpovídají metody *getter* a *setter*. Všechny doménové objekty implementují rozhraní `DomainObject`.

`Role`

Třída definující roli uživatele, tedy zdali je uživatel klient (diabetik), lékař nebo administrátor. Přestože je teoreticky možné pracovat i s jinými rolemi, systém v současné podobě další role neumí spravovat. Třída role je propojena s třídou `User` vazbou 1:N.

`User`

Třída představující uživatele systému. Druh, role uživatele vzniká propojením s objektem `Role`. Každý uživatel může být spojen s mnoha svými měřeními (třída `Measurement`) vazbou 1:N.

`Measurement`

Třída obsahující jeden změřený údaj, to jest okamžik pořízení údaje a vlastní hodnotu údaje. Třída je propojena s třídou `User` vazbou N:1.

`String roleName`	vnitřní jméno role
`String description`	popis role
`boolean predefined`	identifikace druhu role, zabudované jsou *predefined*
`List users`	seznam uživatelů vystupujících v této roli

Tabulka 1 Členové třídy `Role`

`String firstName`	křestní/první jméno
`String surname`	příjmení
`int birthYear`	rok narození (viditelné pouze pro roli *klient*)
`String login`	přihlašovací jméno
`String password`	přihlašovací heslo
`String phone`	číslo telefonu (informativní)
`String phoneId`	identifikace telefonu (nutná pro roli *klient*)
`Role role`	role uživatele
`List Measurements`	seznam měření (s daty je pro roli *klient*)
`List clients`	seznam klientů (s daty jen pro roli *lékař*)[1]
`List physicians`	seznam lékařů (s data jen pro roli *klient*)[1]

Tabulka 2 Členové třídy `User`

`Date instant`	okamžik pořízení měření
`Double measuredValue`	velikost změřeného údaje
`boolean valid`	platnost měření, pole může ovlivnit pouze *lékař*
`String note`	poznámka k měření, pole ovlivní *lékař*, vidí je i *klient*
`User user`	uživatel, jemuž měření patří

Tabulka 3 Členové třídy `Measurement`

Doménové objekty se dále v kódu vyskytují prakticky jen na místech parametrů či při získávání a plnění dat. Slouží jako nosiče údajů.

[1] Třída `User` obsahuje dvě vazby 1:N sama do sebe, což není doporučený postup při návrhu 4NF databázové struktury [15]. V daném případě se však pracuje s abstrakcí objektově-relačního mapování, které si s 4NF potíže nedělá a cyklické odkazy správně převede do korektního tabulkového vyjádření. Nicméně, z hlediska objektového návrhu, ale i z pohledu J2EE, tyto věci nemají být na takto vysoké úrovni řešeny.

5 Implementace serveru

Server, jeho výkonná část, je největší částí řešení. Server podle logiky třívrstvé architektury zajišťuje veškeré výkonné akce, musí zajistit obsluhu klientů a korektní úschovu pracovních dat. Vnitřní architektura serveru sleduje logiku řešení. Jednotlivé části serveru a problematika spojená s jejich tvorbou bude popsána postupně v samostatných sekcích.

5.1 Informační vrstva, objektově-relační mapování

Informační (databázová) vrstva serveru má za úkol odstínit databázové operace od logiky aplikace. Odstínění je v principu dvojí. Za prvé, vrstva překládá výsledky databázových dotazů do datových objektů a zpětně datové objekty propaguje zpět do databáze. Za druhé, vrstva izoluje vlastní databázové dotazy, takže všechny věci spjaté s výběry, řazením, filtrací a podobně zůstávají aplikační vrstvě skryty.

Překlad objektového rozhraní do relační databáze zajišťuje objektově-relační mapování. Bylo by možné vyvinout vlastní způsob mapování, ale to není cílem této práce. Při výběru technologie mapování bylo třeba zvolit mezi Java Persistence API (JSR 220) a Hibernate. JPA je JCP implementace části J2EE, Hibernate je nezávislý produkt. Hibernate vyhrál, protože existuje déle (je stabilnější a prověřený), protože má volnější vazbu na J2EE (takže může být interoperabilnější a přenositelnější) a protože pro Hibernate poskytuje Spring plnou podporu. Je třeba podotknout, že Hibernate plně respektuje definice, anotace a rozhraní Java Persistence API, takže na úrovni nutných anotací v kódu jsou obě technologie zaměnitelné.

‡ Informační vrstva je uložena ve složce `<src/dao>`.

Informační vrstva aplikace je složena z DAO (*Data Access Object*) rozhraní a tříd. Pro každou doménovou třídu zde existuje odpovídající DAO rozhraní, které vyváží operace pro danou doménovou třídu potřebné. DAO rozhraní jsou všechna potomkem (rozšiřují je) abstraktního `DaoObject` rozhraní, takže společné funkce jsou předem vytknuty. Přítomná DAO rozhraní: `DaoObject` (abstraktní), `DaoRole`, `DaoUser`, `DaoMeasurements`.

Abstrakce pomocí DAO rozhraní je důležitá pro možné změny prostředí. V této práci to není klíčová věc, ale zkušenost učí, že dobře strukturovaná a oddělená rozhraní jsou vždy ku prospěchu věci. Pokud by hypoteticky došlo ke změně objektově-relačního mapování (například z Hibernate na JPA), v DAO rozhraních a DAO klientech se nic nezmění. Jediné, co by muselo být přepsáno, jsou DAO implementační třídy.

‡ Hibernate implementace DAO je obsahem složky <src/dao/hibernate>.

Implementační třídy DAO v této práci používají Hibernate, a jsou proto tak i pojmenovány. Pro každý doménový objekt (a tedy i DAO rozhraní) existuje Hibernate implementace: HibernateDaoObject (abstraktní), HibernateRole, HibernateUser, HibernateMeasurements. Hibernate řeší a zapouzdřuje všechny úkony, které jsou pro úschovu dat nutné. Funkční implementace jednotlivých Hibernate-DAO tříd jsou proto velice jednoduché.

5.2 Vrstva aplikační logiky

Aplikační vrstva obsahuje funkční kód aplikace. To obnáší způsob vzniku datových objektů, jejich provazování s jinými objekty, případně kontrolu platnosti dat či vazeb. Aplikační vrstva není vázána na žádnou pomocnou technologii. To je v pořádku, protože aplikační vrstva by měla být od cizích nástrojů absolutně odstíněna.

‡ Aplikační logika je umístěna do <src/business>.

Aby bylo možné aplikační logiku snadno propojovat s vyššími vrstvami, je použit podobný vzor jako v případě DAO vrstvy. Napojení aplikační vrstvy určují *rozhraní* [21], implementace je skryta do jiných tříd (které by tak případně bylo možné snadno přepsat bez změny okolí). Je zvykem, že rozhraní funkčních vrstev jsou nazývána *fasády*, *facades* [13]. Tento zvyk je dodržen i v této práci.

‡ Fasády aplikační logiky jsou ve složce <src/business/facade>.

Aplikační logika opět víceméně respektuje dělení podle doménových objektů. V tomto případě to však není způsobeno logikou struktury dat, ale logikou funkční, byť je tato velice podobná. Funkční logika je v řešení práce jednoduchá a proto datovou strukturu odráží. Je-li třeba pracovat s měřeními, použije se logika pro měření, je-li třeba pracovat s uživateli, použije se logika pro uživatele.

Aplikační logika obsahuje třídy: UserLogic, MeasurementLogic, RoleLogic, které implementují fasády (navenek viditelná rozhraní): UserLogicFacade, MeasurementLogicFacade, RoleLogicFacade.

5.3 Rozhraní mobilního klienta

Rozhraní mobilního klienta je připojeno na jedné straně k aplikační logice a na straně druhé

k síťovému přenosu dat.

Data přenášená mezi serverem a klientem mohou obecně mít různou formu. J2EE doporučuje (a pro programátora je to i příjemné), aby se na obou stranách distribuované aplikace objevovala data ve stejných objektech a aby funkční rozhraní byla stejná (jakoby prodloužená). Prodloužená funkční rozhraní jsou nazývána *delegáty*, *delegates*. Datové objekty nemají speciální pojmenování, obyčejně jsou stejně jako v serveru nazývány *doménové objekty*.

Přenos informací a funkcí z klienta do serveru je v takovém uspořádání snadný: klient použije datový objekt (formálně i obsahově shodný s týmž objektem na serveru), zavolá funkci delegáta a zapouzdřené algoritmy uvnitř delegáta zajistí přesunutí volání do serveru. V serveru existují podobné zapouzdřené algoritmy, které zajistí převod přijatého volání do funkčních volání rozhraní.

Pro převod datových objektů a volání do přenosové podoby se používá jejich serializace a deserializace, celý mechanismus distribuovaného volání se pak nazývá *RPC*, *remote procedure call*, či *RMI*, *remote method invocation* (ale RMI je současně název konkrétní Java technologie), česky *vzdálené volání metod* či procedur [37].

Základním mechanizmem pro distribuovaná vzdálená volání v J2EE jsou WebServices, využívající protokol SOAP. Tato možnost byla první volbou, která byla prověřena. Na straně serveru bylo velmi snadné zařadit do aplikace WebServices modul, v daném případě to byl Apache Axis. Bohužel, vyskytly se potíže při použití WebServices v mobilní části aplikace, protože většina současných mobilních telefonů (přes 80 %) JSR 172 (WebServices) nepodporuje [45]. To by byl pro řešení podstatný problém, protože použití systému nemůže být omezeno jen na několik drahých telefonů; účelem je zasažení co největšího počtu uživatelů, kteří typicky drahé telefony nemají.

Proto začaly být studovány alternativní způsoby serializace. Spring jako zvolená J2EE platforma, podporuje nativně (neboli už to v něm je naprogramováno) HTTP calls, Caucho Burlap a Caucho Hessian. Další způsoby jsou také možné, ale vyžadovaly by programování adaptéru pro Spring. Mezi třemi zmíněnými způsoby serializace byl zvolen *Hessian*. HTTP calls byl zamítnut, protože by vyžadoval poměrně složitou obsluhu na straně klienta. Burlap pracuje podobně jako Hessian, ale je textový (XML). Binární Hessian při stejné funkci jako Burlap přenáší výrazně méně dat a je tak efektivnější (pro mobilní komunikace i cenově). Výhodné na Hessianu a Burlapu je také to, že jejich součástí jsou přímo třídy pracující v J2ME [17].

Hessian je RMI knihovna, která pro přenos dat využívá jednoduchého protokolu, takže je rychlá a efektivní. Více na straně serveru nebylo nutné s Hessianem řešit, protože Spring veškerá spojení s rozhraním klienta ve svém kódu poskytuje.

‡ Rozhraní mobilního klienta je přítomno ve složce `<src/webservice>`.

‡ Fasády rozhraní jsou umístěny do `<src/webservice/facade>`.

Vytvoření vlastního rozhraní pro mobilního klienta bylo po výběru způsobu serializace již jednoduché. Rozhraní je opět organizováno do fasády a implementační třídy. Protože mobilní klient má jedinou úlohu, pořizovat data, má i fasáda s implementací třídy jedinou metodu (`addMeasurement`).

Rozhraní mobilního klienta je skutečné rozhraní, nemá v sobě žádné funkce. Jedinou součástí jeho implementace je volání příslušného rozhraní aplikační vrstvy. Rozhraní mobilního klienta se jmenuje `MeasurementLogicFacade`.

5.4 Rozhraní webového klienta

Rozhraní pro obsluhu serveru přes web je v serveru obsaženo celé, nemá žádné delegáty, serializaci a jiné protějšky na klientské straně. Webové rozhraní musí navenek poskytovat data zcela oproštěná od konkrétní logiky, protože klienti (internetové prohlížeče) s žádnou konkrétní logikou pracovat nemohou. Musejí být obecné. Webové rozhraní musí zajistit tvorbu HTML dokumentů a zpětné spojování odpovědí klientů (tj. výsledků formulářů) s aplikačními daty.

Sestavování HTML stránek a komunikace s formuláři je pochopitelně vyřešená věc, alespoň co se abstraktní podpory týče. V řešení byl pro tento účel použit modul Springu, *Spring MVC*. MVC znamená *Model-View-Controller*, což je ve zkratce posloupnost kroků, kterými se v dynamicky generovaných webových rozhraních oddělují jednotlivé obslužné aktivity. *Model* znamená sadu datových objektů, které mají být ve webové stránce zobrazeny, jsou to data již připravená pro konkrétního klienta. *View* představuje šablonu (programovatelnou), která modelová data převede na výsledný HTML kód. Nakonec *Controller* slouží k obsluze stránek, to jest k propojení URL (požadavku klienta) s konkrétním kusem rozhraní, které připraví modelová data a postoupí je k tvorbě HTML stránky do patřičného *View* [27]. Většina současných MVC *Controller* chápe v různých podobách šířeji a zařizuje s jejich pomocí kompletní obsluhu formulářů. Nejinak tomu je i ve *Spring MVC* [41].

‡ Rozhraní webového klienta patří složka `<src/webinterface>`.

Podobně jako v případě mobilního klienta slouží i webový klient k překladu HTTP komunikace do funkčních volání aplikační logiky. A stejně jako v případě mobilního klienta, mimo vyřešení aktivit rozhraní jsou přítomna pouze volání do aplikační vrstvy.

Protože je blok obsluhy webového klienta poměrně rozsáhlý a řídí se svou logikou (která je odlišná od zbytku aplikace), je jeho podrobný popis odložen do kapitoly „Webový klient a jeho obsluha“.

str. 56
8

6 Běhové prostředí serveru

str. 26 3.2

Jak již bylo v kapitole „Kde J2EE rozběhnout“uvedeno, potřebuje každý J2EE systém prostředí, které zajistí společné funkce, integraci všech jeho částí a oživí jeho komponenty. Pro řešení práce byl vybrán framework Spring, protože je konfigurovatelnější než aplikační servery, protože je schopen pracovat v jednodušších prostředích (takže je snadněji nasaditelný), protože vlastní podporu pro serializaci Hessian (nutnou pro přenos dat do mobilního klienta) a protože jeho MVC modul je přehledný a snadno použitelný. Více o Springu viz kapitolu „Spring“.

str. 28 4

Spring sám o sobě může být spuštěn pomocí další aplikace. To může být jakýkoli Java program, ale může to také být rovnou prostředí HTTP a servlet serveru, který je pro řešení také potřeba. Jako ideální byl vybrán open source server Apache Tomcat [3], který je jednou z referenčních aplikací J2EE servlet technologie. Spring je v rámci Tomcatu rozběhnut jako jeho část.

6.1 Apache Tomcat

Apache Tomcat implementuje J2EE technologie servlet a JSP, je tedy *servlet container*. Tomcat se instaluje jako samostatný produkt, je nezávislý. Samozřejmě, po instalaci je Tomcat prázdný a žádné služby nenabízí, žádný server v něm neběží. Každá aplikace se do něj musí instalovat (*deployment process*), teprve pak ji Tomcat začne světu nabízet. Deployment a ostatní konfigurační akce jsou integrovány do Tomcatu jako jeho součást.

‡ Předobraz WAR souboru je přítomen ve složce `<web>` serveru.

Formát instalace programu, soubor *WAR* (WebARchive), který je Tomcatu srozumitelný, rovněž přesně určuje J2EE. WAR musí obsahovat správně pojmenované složky se soubory (JSP stránky, servlety, knihovny) a musí obsahovat hlavní konfigurační soubor (*deployment descriptor*) `<WEB-INF/web.xml>`. Předem dané soubory (konfigurace, JSP) lze nalézt přímo jako součást projektu, ostatní věci (knihovny, přeložený kód a samotný WAR) se vytvoří automaticky během překladu [50].

`<WEB-INF/web.xml>` obsahuje:

Popis aplikace a její základní WWW konfigurace

To představuje jméno aplikace (`<display-name>`), její popis (`<description>`), parametry spojení (`<session-config>`, `<welcome-file-list>`) nebo procesní

změny během HTTP zpracování (sekce `<filter>`).

Podporu pro tvorbu servletů

Pokud nějaká existuje, v mém podporu tvoří definice knihoven tagů pro tvorbu servletů z JSP (`<jsp-config>`).

Definici služeb serveru

Servlet může vykonávat veškeré logické zpracování v sobě, ale protože servlety jsou cíleny do oblasti tvorby HTML stránek (poskytování dat webovým klientům), není to dobré. Popřelo by to částečně logiku třívrstvých aplikací. Je proto vhodné rozběhnout stálý kód (logiku serveru, J2EE aplikaci) ne jako servlet, ale jako *službu*, kterou pochopitelně servlety využití. Pro definici služeb v souboru `<web.xml>` slouží prvky `<listener>` (definice služeb) a `<context-param>` (parametry služeb).

V aplikaci jsou definovány dvě služby: *logger* pro zápis zpráv a *Spring* (neboli tohle je místo, které rozbíhá stroj pro mou J2EE aplikaci)

Definici servletů

Přesněji tříd, které se chovají jako servlety. Logika servletů je taková, že server (Tomcat) podle klientem požadovaného URL vytvoří patřičnou obslužnou instanci servletu a předá jí řízení. Jednoduchý servlet by byl zapsán přímo, zde se však používá MVC a specializovaný mobilní klient. Místo přímého odkazu na třídu servletu se proto používá *Spring dispatcher*, který se chová jako servlet, ale vevnitř je konfigurovatelný a nabízí podstatně více. Pro definici servletů slouží prvky `<servler>` a `<servlet-mapping>`.

Aplikace definuje dva servlety: *web* pro webové klienty a *remoting* pro mobilní klienty. Každému z obou servletů odpovídá samostatný konfigurační soubor (`web-servlet.xml` a `remoting-servlet.xml`). Protože jsou tyto soubory obsluhovány Springem, budou pospány v následující sekci.

Vytvořením WAR souboru se správným obsahem a korektním popisem konfigurace a jeho instalací je práce s Tomcatem skončena.

6.2 Spring

‡ Konfigurační soubory Springu jsou uloženy ve složce `<web/WEB-INF>` serveru.

Konstrukci serverové části, její start a běh, obsluhu webových klientů pomocí MVC a obsluhu mobilního klienta, to vše má v aplikaci na starost Spring [41]. Jeho konfigurace je rozložena

do třech souborů:

<spring-application.xml>

Hlavní konfigurační soubor pro logiku aplikace (což je mimo mobilní a webové rozhraní vše), pomocí tohoto souboru se konfiguruje jádro Springu. V konfiguraci Tomcatu je jádro spuštěno jako služba (viz kapitolu „Apache Tomcat“).

str. 47 6.1

<remoting-servlet.xml>

Soubor konfigurující serializaci a rozhraní pro mobilní klienty.

<web-servlet.xml>

Soubor konfigurující webové rozhraní, to jest MVC.

Všechny konfigurační soubory Springu mají jednotnou podobu XML souborů. Jak již bylo psáno v kapitole „Jak na Spring?“, v konfiguraci Springu dochází ke spojení jednotlivých vlastností tříd programu. Spring podle konfigurace vytváří *objekty*, v konfiguraci tomu odpovídají XML prvky `<bean>`, a propojuje je s využitím jejich *setter* metod. V konfiguraci je proto možné nalézt jen `<bean>` prvky patřičných tříd spolu s podřízenými prvky s konfigurací a spojeními těchto tříd s jinými třídami.

str. 30 4.2

Hlavní konfigurační soubor `<spring-application.xml>` obsahuje:

Prvky DAO vrstvy

To jsou všechny již představené objekty sloužící k oddělení databáze a aplikační vrstvy. Všechny DAO prvky (`daoUser`, `daoMeasurement`, `daoRole`) jsou propojeny vlastností `sessionFactory` s Hibernate.

Hibernate, stroj pro objektově relační mapování

Prvek se jménem `hibernateSessionFactory`, s připojenou konfigurací v souboru `<hibernate-config.xml>`. Tento prvek zajišťuje kompletní databázovou podporu. Hibernate je rovněž doplněn správcem transakcí (prvek `transactionManager`), který zaručí, že data v případě chyby zůstanou konzistentní.

Prvky bussiness vrstvy

Neboli instance všech objektů s aplikační logikou (`userLogic`, `measurementLogic`, `roleLogic`). Tyto prvky mají typicky propojení do objektů DAO vrstvy; současně je v konfiguraci těchto prvků uvedeno, že mají být spravovány pomocí správce transakcí. Operace objektů aplikační logiky proto vždy běží jako transakce.

Konfigurační soubor `<remoting-servlet.xml>` obsahuje:

...převzaté informace

V souboru samotném to není zaznamenáno, avšak za běhu aplikace jsou všechny objekty definované v hlavním konfiguračním souboru dostupné stejně, jako by byly zapsány přímo zde.

Řadič URL

Třída `BeanNameUrlHandlerMapping`, která uvnitř servletu zajistí, že příchozí klientský požadavek bude zpracován správným objektem. V případě `<remoting--servlet.xml>` je zajištěna pouze jediná vazba na objekt `/MDGService`.

Hessian server

Představovaný objektem `/MDGService`. Služba Hessianu vnitřně, dostane-li se k ní řízení, zajistí serializaci a volání objektu `mdgService`, tak jak je to zapsáno v prvcích `<property>` služby.

Implementaci mobilního rozhraní

Neboli instanci objektu `webservice.MDGService`, objektu popsaného v kapitole „Rozhraní mobilního klienta“.

str. 43 5.3

Konfigurační soubor `<web-servlet.xml>` obsahuje různá nastavení související s MVC vrstvou. Jiný pohled a podrobnější popis této části nabízí také kapitola „Webový klient a jeho obsluha“. Obsahem souboru je:

str. 56 8

...převzaté informace

V souboru samotném to není zaznamenáno, avšak za běhu aplikace jsou všechny objekty definované v hlavním konfiguračním souboru dostupné stejně, jako by byly zapsány přímo zde.

Řadič URL

Objekt `urlMapping`, v konfiguraci objektu je zapsáno, které obslužné objekty získají řízení při přijetí různých URL. Objekt spolupracuje s pomocným `openSessionIn-ViewInterceptor`, což je objekt, který zajistí přežívání jednotlivých Hibernate sezení během postupných vytváření HTML stránek.

Řadič pohledů (*View*)

Objekt `viewResolver`, objekt nedělá prakticky nic jiného, než že převádí symbolic-

ká jména pohledů (např. *viewdata*) na jména příslušných JSP souborů (např. `</WEB-INF/jsp/viewdata.jsp>`).

Objekt sezení

Neboli objekt, který uchovává data spojená s jedním přihlášeným klientem.

Obslužné objekty požadavků

Nejzajímavější část konfigurace, která poskytuje objekty pro vytváření HTML stránek. Jsou to `loginController`, `userController`, `mainController`, `viewDataController` a `viewUsersController`. Tyto objekty jsou zpravidla spojeny s patřičným objektem aplikační logiky. Pokud objekty spolupracují s formulářem, mají u sebe konfiguraci pro formuláře. Všechny objekty jsou spojeny s objektem sezení.

Validátory dat přicházejících z formulářů

Objekty `loginValidator` a `userValidator`; objekty slouží ke kontrole dat přicházejících z formulářů. Objekty jsou spojeny pouze s aplikační logikou, která slouží bezprostředně pro kontrolu dat formulářů.

6.3 Databáze

Řešení práce je z principu postaveno tak, aby na databázi nebylo závislé. Nezávislost zajišťuje několik vnitřních abstrakcí: za prvé, pomocí JDBC je odstíněno fyzické databázové připojení od logických databázových pokynů; a za druhé, tyto logické databázové pokyny jsou filtrovány a zapouzdřeny uvnitř objektově-relačního mapování ([32], [18]).

Abstrakce je důležitá, pro vývoj a běh však musí být použito konkrétní řešení. V případě této práce byla aplikace laděna a testována v prostředí databáze PostgreSQL [34].

Díky popsané abstrakci postačuje pouze PostgreSQL instalovat, definovat v ní jednu databázi a jednoho uživatele [34], pomocí kterého bude JDBC do databáze přistupovat. Tentýž uživatel pak musí být zapsán rovněž do konfigurace Hibernate (viz kapitolu „Spring“). Všechny databázové operace (založení tabulek, sloupců apod.) následně zajistí Hibernate ve své režii, takže není třeba nic ručně nebo poloautomaticky konfigurovat.

str. 48
6.2

7 Mobilní klient

Část systému, která běží v telefonu, je v porovnání se serverovou částí značně odlišná. Prakticky se zde nevyskytuje žádný kód, hlavní díl práce tvoří uživatelské rozhraní a jeho obsluha. To je v pořádku, neboť podle logiky třívrstvých aplikací [28] by se na klientovi žádná aplikační logika neměla vyskytovat.

Mobilní klient musí:

- nabídnout uživateli formuláře, povely a ovládací prvky,
- zajistit spojení se serverovou částí,
- ukládat systémovou konfiguraci potřebnou k běhu.

‡ Zdrojové kódy klienta jsou v `<src/webserviceclient>` projektu *MDGClient*.

7.1 Uživatelské rozhraní

Grafické uživatelské rozhraní je vytvořeno pomocí RAD (rapid application development) nástroje FlowDesigner, který je součástí NetBeans IDE [52]. S pomocí tohoto nástroje byly sestaveny formuláře a obrazovky podle toku operací, které jimi procházejí. FlowDesigner automaticky nabízí jen takové ovládací prvky, které jsou součástí specifikace MIDP (viz kapitolu „Java a mobilní zařízení“). Jednotlivé obrazovky GUI, a zejména tok operací mezi nimi, ukazuje obrázek 1.

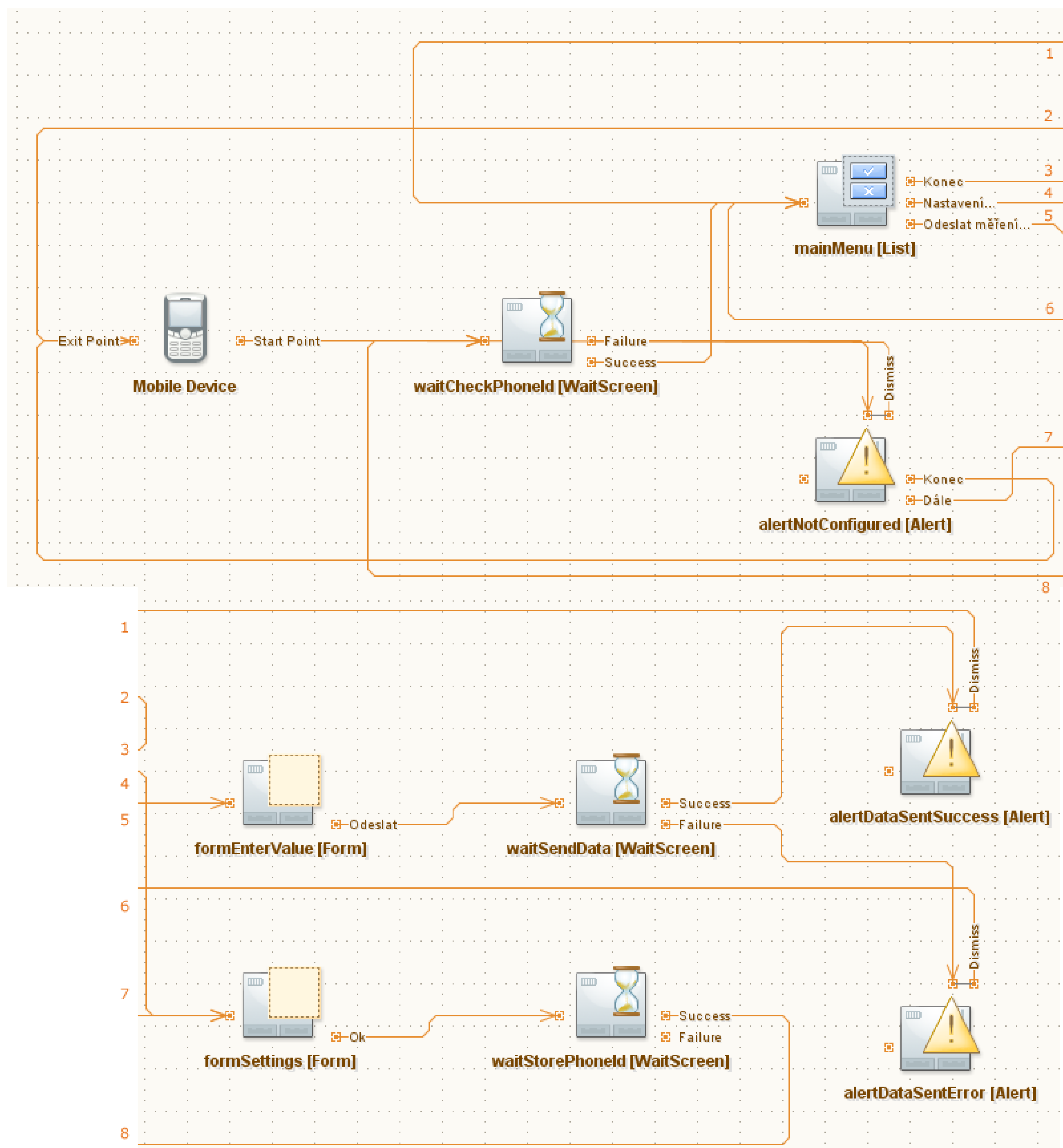
str. 32
5

Množství jednotlivých obrazovek lze rozdělit a popsat podle jejich druhu:

Obrazovky s delšími operacemi

Tyto obrazovky představují úlohy, které aplikace vykonává pro splnění nějakého požadavku uživatele. Přítomny jsou obrazovky `waitCheckPhoneId` (slouží k počáteční kontrole a nastavení identifikace telefonu), `waitStorePhoneId` (slouží k uložení systémové konfigurace) a `waitSendData` (je viditelná během odesílání údaje do serveru).

Všechny tyto obrazovky mají uživatele informovat, že něco probíhá. Také tomu tak je, protože funkční volání (např. odesílání dat do serveru) běží uvnitř těchto obrazovek. Každá probíhající akce může skončit úspěšně anebo neúspěšně. Proto má každá z obrazovek oba výstupní body zapojeny (kromě `waitStorePhoneId`, které data kontroluje následující `waitCheckPhoneId`), přičemž neúspěšná ukončení přecházejí do obrazovek s hlášením.



Obrázek 1 Obrazovky a jejich vzájemné vztahy

Obrazovky s hlášením

Informují uživatele, aplikace obsahuje tři takové obrazovky: `alertNotConfigured` (ukáže se, pokud dosud není telefon nastaven), `alertDataSentError` (informuje o chybě při přenosu dat), `alertDataSentSuccess` (oznamuje uživateli, že jeho měření úspěšně dorazilo do serveru).

Hlavní menu

Jednoduchá obrazovka (`mainMenu`), která je pro práci s klientem výchozí. Z menu je možné vstoupit do systémové konfigurace a do odeslání změřených dat.

Formuláře pro editaci údajů

V aplikaci musí být přítomen formulář pro zadávání systémové konfigurace (`formSettings`, slouží typicky pouze při prvním spuštění aplikace) a formulář pro vkládání změřených dat (`formEnterValue`). Po opuštění formulářů vždy následují výkonné akce, takže v GUI dochází k přepnutí do obrazovek s delšími operacemi.

7.2 Spojení se serverem

str. 43
5.3

V klientovi, tak jak bylo popsáno v kapitole „Rozhraní mobilního klienta“, musejí být v souladu s J2EE přítomny speciální objekty, které zařídí přenos funkčních volání do serveru. Tyto objekty se nazývají *delegáty* [9].

Ve stejné kapitole se rovněž píše o výběru serializéru Hessian [17], který se dobře hodí do mobilního prostředí díky své minimalističnosti. Mobilní klient využívá tento Hessian a konstruuje delegáta, který zajišťuje přenos jediného funkčního volání (`addMeasurement`).

Delegát je rozložen do tří tříd:

`HessianResult`

Třída sloužící k přenosu výsledků volání `HessianClient`.

`HessianClient`

Hlavní část implementace Hessianu. Třída poskytuje metodu `Transact`, která pomocí HTTP protokolu a Hessian serializéru zavolá funkci serveru a přečte výsledek s návratovou hodnotou. Metoda `Transact` je chráněná a nelze ji použít jinak než zevnitř třídy dědící `HessianClient`.

`MobileHessianClient`

Dědic třídy HessianClient, který poskytuje požadované funkční volání `addMeasurement`. Třída uvnitř volá výše zmíněnou metodu `Transact`.

Abstraktní části delegáta (`HessianResult` a `HessianClient`) jsou navrženy tak, aby je případně bylo možné využít i v jiných projektech.

7.3 Konfigurace

Mobilní telefon provozující klientskou část systému (neboli telefon zařazený do aplikace), musí udržet dva konfigurační údaje : URL k serverové službě a identifikaci telefonu (spojovací číslo). Je zjevné, že oba údaje jsou vyžadovány pro komunikaci se serverem a že oba údaje

typicky bude třeba upravovat pouze jednou při prvním spuštění aplikace. Vždy je však možné údaje upravit později.

Spojovací číslo je vytvořeno při zařazení uživatele do aplikace a je viditelné v uživatelských datech ve webovém klientu (vidí ji uživatel v rolích *klient* a *administrátor*).

Pro ukládání a čtení konfigurace je v klientovi přítomna třída `Storage`, která je poměrně samostatná (se statickou inicializací) a dovnitř klientské aplikace poskytuje metody pro čtení a zápis konfiguračních dat.

8 Webový klient a jeho obsluha

str. 48 6.2 Webový klient byl z pohledu konfigurace Springu popsán v kapitole „Spring“. Cílem této kapitoly je popsat řešení z funkčního hlediska. Základní část Springu, která je pro webový str. 45 5.4 klient použita, byla rovněž zevrubně popsána v kapitole „Rozhraní webového klienta“; jedná se o *Spring MVC*.

Stručně, MVC znamená *Model-View-Controller*, kdy příchozí požadavek je zpracován řadičem (*Controller*), doplněn modelovými daty (*Model*) a pomocí pohledu (*View*) je odeslán ke klientovi. Tato funkční posloupnost je v zásadě shodně vyjádřena v konfiguraci i odpovídajících třídách webového rozhraní [27].

8.1 Implementace řadičů

‡ Zdrojové kódy řadičů jsou ve složce `<src/webinterface>`.

Konfigurační soubor Springu přiřazuje vstupním URL (podle obsahu adresy) výkonné objekty — řadiče. Řadiče jsou sdílené objekty, které dokáží reagovat na více vstupních volání. Z hlediska implementace jsou řadiče Java třídy, které dědí některý z abstraktních řadičů definovaných ve Springu a podle potřeby implementují jednotlivé jeho části. Třídy s řadiči existují za běhu aplikace trvale, v konfiguračním souboru Springu existují proto jejich protějšky v podobě elementů `<bean>`.

Aplikace používá dva druhy řadičů: řadič pro tvorbu stránek a jednoduchou úpravu jediné existující datové položky a řadič pro komunikaci s formuláři.

Řadiče pro tvorbu stránek jsou celkem tři:

`MainController`

Řadič pro hlavní stránku a různá hlášení. Řadič je ryze jednosměrný, nedokáže upravovat žádná data.

`ViewDataController`

Řadič pro zobrazování a editaci změřených údajů. Údaje jsou editovatelné částečně a jen uživatelem v roli *lékař*. Řadič především data zobrazuje, ale dokáže také v rámci zobrazených údajů přijímat požadavky na jednoduchou editaci (např. editaci poznámky lékaře).

`ViewUsersController`

Řadič pro zobrazování a editační operace s celými uživateli. Za tyto operace považuji *zrušení* uživatele, *přiřazení* lékaře ke klientovi a *zrušení* tohoto přiřazení. Řadič opět slouží hlavně pro zobrazení, editační operace jsou omezeny jen na tři uvedené.

Řadiče pro komunikaci s formuláři jsou dva:

`LoginController`

Řadič pro obsluhu přihlašování uživatelů, a případně opakovaná přihlašování při neúspěchu. Tehdy je formulář zobrazen s již zadanými daty a doplněný o chybová hlášení.

`UserController`

Řadič pro položkovou editaci uživatelů. `UserController` se používá pro formuláře s různými cíly, jednak pro zakládání nových uživatelů, jednak k editaci vlastních uživatelských dat přihlášeného uživatele a jednak pro obecnou editaci uživatele (to může pouze uživatel v roli *administrátora*). Stejně jako `LoginController` i tento řadič kontroluje platnost dat přicházejících z formuláře a případně formulář s chybovými text odmítá.

Práce s formuláři je složitější, nejde o pouhé vytvoření stránky. Každý formulář je vnitřně propojen s *datovým objektem*. Tento objekt se při odesílání dat ke klientovi používá jako *modelový objekt* a při přijímání dat z klienta jako objekt pro úschovu dat. Elegantní řešení formulářů pro editaci používá na místě datového objektu přímo doménové objekty. To značně zjednodušuje práci: před odesláním formuláře stačí datový objekt získat z databáze a po přijetí změn tentýž objekt pouze v databázi aktualizovat. Samozřejmě, při zakládání nových uživatelů je třeba vždy vyrobit datový objekt nový.

Ve formulářích pro editaci bylo třeba kromě editovaných doménových objektů udržet také informaci o způsobu použití formuláře (jestli se vytváří nový uživatel, edituje stávající apod.) Proto se jako datový objekt ve formulářích používají objekty třídy `UserCommand`, která agreguje třídu `User` (doménový objekt) a přidává datový člen se způsobem užití formuláře. Mimo to je do třídy `UserCommand` také zařazen (vkládaným pro kontrolu).

Každý formulář může být vyplněn nesprávně. V případě přihlašovacího formuláře může být zadáno nesprávné jméno či uživatel; při editaci uživatele musí být definováno alespoň přihlašovací jméno. Spring MVC pro řešení těchto situací nabízí abstrakci *validátoru* [41]. Validátor je objekt, který dostává data přijatá od uživatele ještě dříve, než jsou postoupena k dalšímu zpracování řadičem. Validátor může data odmítnout; pak řadič vůbec data nezpracovává a

okamžitě generuje novou stránku s pohledem na stávající data doplněnou o chybová hlášení. V klientovi se používají dva validátory `LoginValidator` a `UserValidator`, které jsou spojeny v konfiguračním souboru Springu s řadiči formulářů (`LoginController` a `UserController`). Použití validátorů šetří prostředky serveru a samozřejmě také zrychluje zpracování odpovědi.

Oba druhy řadičů (zobrazovací i formulářové) generují HTML stránky pro uživatele po svém zpracování stejným způsobem. Připraví modelová data (to jsou data, která mají být ve stránce zobrazena) a postoupí tato modelová data objektu, který je aplikuje na šablonu pohledu. Modelová data mohou být jakákoli, v této aplikaci se nejčastěji používají informace o přihlášeném uživateli a dále seznamy uživatelů a dat.

V MVC Springu je postoupení řízení pro tvorbu pohledu implementováno pomocí třídy `ModelAndView`, kterou v kódu často používám. `ModelAndView` pracuje s oběma druhy dat: s požadovaným pohledem i sadou modelových objektů. Nejjednodušší použití `ModelAndView` plní obě tyto informace přímo do konstruktoru objektu.

8.2 Implementace pohledů

‡ Zdrojové kódy pohledů jsou ve složce `<web/WEB-INF/jsp>`.

Pohled, či šablona stránky je JSP soubor. To znamená, že může obsahovat odkazy na datové objekty a může obsahovat algoritmické operace. JSP soubor vypadá jako běžná HTML stránka (což je předobraz budoucí stránky), do níž jsou zahrnuty pomocí speciálních tagů JSP výrazy a algoritmy. Pro zápis algoritmů bylo použito JSTL (část J2EE, [22]) a pro práci s daty pak knihovna tagů pro Spring a formuláře Springu. Datové objekty, se kterými pohledy pracují, jsou právě ty modelové objekty, které řadiče připravují.

Jednotlivé pohledy jsou čistě logická věc. Jejich jména nijak nesouvisejí se jmény stránek (přiřazení URL řadičům je vnější operace), přesto však pohledy byly podle stránek pojmenovány, aby zůstala zřejmá jejich vzájemná vazba. Existují tak následující pohledy, které odpovídají řadičům: `login`, `main`, `user`, `viewdata` a `viewusers`.

Kromě toho však existují další pohledy:

`badrequest`

Obecná stránka, která se zobrazuje při chybách, tzn. při neočekávaných stavech nebo neúplných operacích.

`entry`

Úvodní stránka, ze které se přechází k přihlášení.

`created`

Pohled s informací o založení uživatele. Mimo jiné je zde na tomto místě poprvé zobrazeno spojovací číslo pro konfiguraci telefonu.

S implementací pohledů rovněž souvisí bezpečnost editace dat a koherence pohledů a webových stránek. MVC v principu umožňuje pod jednu stránku (pod jedno URL) ukrývat libovolný počet pohledů. Například, bylo by možné, aby se po úpravě uživatele (URL `<user.html>` a *View* `user`) přešlo zpět na hlavní stránku, aniž bych změnil URL. Uživatel by viděl pohled `main` uvnitř stránky `<user.html>`. To je zmatek, a proto byly všechny přechody mezi pohledy doplněny změnou URL: ve jménech nových pohledů se zapisuje metainformace `redirect:`, stejného výsledku by bylo lze dosáhnout také použitím pomocné Spring třídy `RedirectView`.

Důležitý důsledek tohoto přístupu je také to, že přechodem k jinému URL prohlížeč ztrácí informaci o datech odeslaných ve formuláři. To je nutné, aby se zamezilo náhodnému opakování operace obnovením stránky. Prohlížeč by tehdy odeslal do serveru již jednou zpracovaná data, a to není v pořádku. Patrně by se nic nestalo, ale jestliže je možné zabránit nesprávnému toku dat, je třeba tak učinit. Zmíněným přechodem k novému URL se tento nedostatek řeší.

8.3 Chování a ovládání

Při tvorbě webového rozhraní bylo dbáno na to, aby se rozhraní pokud možno vždy dokázalo zotavit se z nečekaných stavů samo. O některých věcech se již psalo: při přechodech mezi pohledy se důsledně dbá na rozvázání vazby s formulářem a na párování URL a pohledů. Jiné věci dosud nebyly zmíněny. Tedy:

- Celé webové rozhraní je vybudováno jako **tříúrovňové**. Systém rozeznává role *administrátor*, *klient* a *lékař*, role nelze spojovat. Není tedy možné, aby byl tentýž uživatel současně byl lékařem i klientem. Tento přístup je zamýšlený, protože jasně vymezuje, kdo je kdo a co může.
- Pomocí webového rozhraní je možné pracovat se systémem pouze po **přihlášení**. Systém kontroluje stav přihlášení aktivně při každém přístupu na stránku. Pokud není nikdo přihlášen, či pokud přihlášení vypršelo, stránka je nedostupná a systém zobrazí stránku s chybovým hlášením.

- Systém pečlivě kontroluje **konzistenci** editačních dat. Pokud například při přijetí dat z formuláře chybí pokyn nebo nějaká identifikace, je celá operace odmítnuta a systém zobrazí stránku s chybovým hlášením. Dále, systém zamezuje některým operacím v určitých stavech systému. Například, není možné vůbec vstoupit do editace uživatelů, pokud dosud žádný neexistuje apod.
- Co nejvíce dat jdoucích od klienta bylo **skryto** do HTTP komunikace tak, aby nebyla přímo viditelná v URL (neboli nepoužívá se HTTP metoda *GET*, ale *POST*). *GET* se používá pouze při zahajování editací pro přenesení editačních informací uvnitř URL [19].

Stránek, které má uživatel možnost vidět, není mnoho, přesto však stránky dodržují vnitřní logiku:

- Z každého pohledu je možné vrátit se **zpět** (ne tlačítkem prohlížeče, ale odkazem) a vždy je možné se **odhlásit**. V určitých pohledech se nabízejí také odkazy na důležité stránky.
- V každé stránce je jasně zřejmé, co lze udělat. Při jednorázových editacích lze údaj upravovat přímo na místě, při editacích více dat je vždy nabídnut samostatný formulář.
- Pokud je třeba se rozhodnout, jsou nabídnuty jen dvě varianty.

Struktura a přechody stránek jsou voleny tak, aby dávaly logický smysl:

- Při prvním přístupu k serveru je vždy zobrazena **úvodní stránka**.
- Následuje **přihlašovací stránka**, kde se uživatel snaží přihlásit.
- Po úspěšném přihlášení se každý uživatel dostane na **hlavní stránku**, kde je vždy možné upravit svá data a odhlásit se. *Klienti* mohou zobrazit seznam svých měření a seznam lékařů, kteří o nich mají vědět; *lékaři* mohou zobrazit seznam svých klientů a *administrátoři* mohou vidět seznam všech uživatelů.
- **Editace vlastních dat** se po potvrzení změn vrací zpět na hlavní stránku.
- Klienti svůj **seznam měření** mohou pouze prohlížet. Vidí však i poznámky k měřením, které doplňuje lékař.
- V **seznamu lékařů** se mohou klienti přiřazovat (a vyřazovat) ke svým ošetřujícím lékařům.
- V **seznamu klientů** mohou lékaři prohlédnou všechny klienty, kteří si lékaře vybrali jako ošetřujícího. Z tohoto seznamu klientů může lékař přejít na stránku s měřeními klienta.

- Lékař v **seznamu měření** klienta může dopisovat k měřením poznámky a případně označovat měření za neplatná (například když jsou zjevně nesprávná).
- Administrátor může ze **seznamu uživatelů** přejít k **editaci** libovolného uživatele, na stejné stránce může též vytvořit nového uživatele.

Jiné možnosti přechodu mezi stránkami a možnosti použití systému v různých rolích nejsou možné.

9 Možnosti rozvoje

Přestože je systém vybudován v rámci logiky řešení kompaktně a soběstačně, je možné již nyní vidět, jaké další možnosti úprav či rozvoje jsou možné:

- Je možné dále vyvíjet prezentační schopnosti webové části, například doplnit grafické zobrazení dat. Otázka je, zdali má takové zobrazení při očekávané četnosti pořizování dat smysl.
- Patrně by nebyl problém systém mírně zobecnit (hlavně na straně popisů) a používat jej pro komunikaci a zpracování jiných dat, než glykemických (například EKG).
- Možné velké zvýšení užitné hodnoty by mohlo přinést propojení systému s online měřením a přenášením dat. Glykemické křivky jsou pro lékaře zajímavé také při průběžném vývoji v čase a kdyby existovalo automatické měřicí zařízení (například spojené s inzulinovou pumpou), mohla by být data pomocí telefonu odesílána serveru například každou půlhodinu. Je zřejmé, že toto řešení by patrně vynutilo vytvoření grafické prezentace dat (viz první bod).
- Při nasazení serveru by mělo být použito zabezpečení kanálu (TLS apod.). Není to ani tak možnost rozvoje, jako správný postup. Mé řešení toto nepokrývá, protože zabezpečený přístup k serveru přes HTTPS je věcí konfigurace serveru, ne jeho aplikace [48].
- Systém by nebylo složité rozšířit o zpětnou vazbu od lékaře. Již nyní může uživatel číst poznámky, ale možná by mělo smysl, aby lékař mohl například poznámku podle potřeby rovnou klientovi odeslat jako SMS či e-mail.
- Mobilní klient by rovněž mohl být schopen číst poznámky zapsané lékařem; například poznámku zapsanou k poslednímu měření, apod.
- Do webového rozhraní by mohlo být doplněno také online vkládání měřených dat. Najednou by se mohlo vložit více dat a nemuselo by se operovat s mobilním telefonem.

Závěr

Distribuovaný systém, který řeší zadání této práce, vypadá při použití jednoduše jak v mobilním telefonu tak i ve webovém rozhraní. To je důležité, neboť věci by měly být tak jednoduché, nakolik jen mohou být. Jednoduchost navenek však nemusí znamenat jednoduchost vevnitř: v popise praktického řešení je uvedeno mnoho případů, kdy bylo třeba řešit různé otázky a nejasnosti. Je třeba říci, že prakticky všechny byly vyřešeny uspokojivě.

Celý systém je vytvořen v jazyce Java. Použití Javy významně urychlilo vývoj, protože použité vývojové nástroje a postupy jsou vysoce sofistikované. Prezentační vrstvy mobilního klienta podléhají standardům a existuje pro ně návrhář; totéž v abstraktnější a složitější podobě pak existuje i na straně serveru: Java definovala pomocí svých standardizačních procesů specifikaci J2EE, která (nejen) na ideové úrovni popisuje doporučené postupy při tvorbě podnikových řešení. S pomocí této specifikace byla navržena přehledná, třívrstvá aplikace klient-server, která je i ve svých vnitřních částech formálně čistá a řádná.

J2EE je obecný standard, který lze implementovat v různých podobách. V řešení byl využit framework Spring, který se od roku 2003 systematicky snaží připravovat pro vývoj J2EE aplikací postupy a API, které dovolí J2EE použít jednoduše a efektivně. Spring je elegantní, vysoce konfigurovatelný a integrující nástroj, ve kterém je možné vyřešit i speciální serializace a komunikaci s limitovaným mobilním zařízením.

Při samotné implementaci byl kladen důraz na správné a doporučené vývojové postupy, mezi něž lze počítat analýzu a návrh, dodržování důsledného oddělení výkonných vrstev, separaci API od implementace pomocí rozhraní, dodržování konvencí kódování a také udržování vnitřní konzistence jmen a umístění (podobné nechť se jmenuje podobně).

Stejný důraz byl kladen rovněž na návrh struktury uživatelského rozhraní, protože to je nakonec to jediné, co z programu bude vidět. Dobré uživatelské rozhraní nesmí uživateli překážet, nebo ho mást. Součástí tohoto přístupu je také snaha implementovat webového i mobilního klienta robustně, to znamená učinit ho odolným vůči neočekávaným stavům a datům.

Práce vznikla jako příprava na možné praktické pilotní nasazení v diabetické poradně (o tomto nasazení však dosud není rozhodnuto). Současně je povaha řešení natolik obecná, že myšlenky a postupy ověřené v této práci mohou být rychle využity a implementovány v jiných projektech.

Conclusion

Distributed system, which solves thesis assignment, seems simple and usable either in its mobile part either through its web interface. It is important, because things should be as simple as they can be. But, the outer simplicity need not imply simplicity inside, there were described many of cases in practical solution, where questions and problems had to be solved. It must be said that almost all were solved in a satisfactory manner.

Whole application is built in Java language. The usage of Java significantly speeded up development, because used tools and practices are highly sophisticated. Presentational tier of mobile client is standardized and there is RAD tool for it; the same in a complex fashion exists for server solutions too: Java defined with the help of self standardatization mechanisms J2EE specification, which is (not only) a source of ideas and recommendations how to create enterprise solutions. Using the specification clear, three-tier and client-server application was designed.

J2EE is general standard, which can be implemented by many ways. For this solution, the Spring framework was used. Spring, from year 2003, systematically aims to prepare solutions, procedures and APIs, which allow to use J2EE simply and effectively. Spring is elegant, highly configurable and integrating tool, which is capable to solve even special serialization or communication with limited mobile device.

During implementation itself, there was placed emphasis on correct and recommended development processes. They cover, for instance, analysis and desing, keeping of strict separation of executive layers, separation of API and implementation using interfaces, usage of coding conventions and keeping of internal consistency of names and places (let similar is named similarly).

The same emphasis was placed on user interface design too, because at the end it is the only thing, which will be really visible. Correct user interface must not interfere with user or confuse him. The effort to implement web and mobile client to be robust is also a part of this approach. Robust for this case means resistant to unexpected states or data.

The thesis was introduced as preparation of practical pilot solution for diabetic specialist. Simultaneously, the solution is so much general, that ideas and processes proved by the thesis can be very quickly reused and implemented in another projects.

Literatura

1 Kiszka, B.: *1001 tipů a triků pro programování v jazyce Java*. Computer Press, Praha, 2003.

2 Wikipedia, EN: *Ada (programming language)*.
http://en.wikipedia.org/wiki/Ada_programming_language,
19. května 2007.

3 Apache Software Foundation, ASF: *Apache Tomcat*.
http://tomcat.apache.org/,
19. května 2007.

4 Wikipedia, EN: *Basic programming language*.
http://en.wikipedia.org/wiki/BASIC_programming_language,
19. května 2007.

5 Wikipedia, EN: *Buffer overflow*.
http://en.wikipedia.org/wiki/Buffer_overflow,
19. května 2007.

6 W3C, consortium: *Cascading style sheets*.
http://www.w3.org/Style/CSS/,
19. května 2007.

7 Wikipedia, EN: *Client-server*.
http://en.wikipedia.org/wiki/Client_server,
19. května 2007.

8 Wikipedia, EN: *Comparison of C# and Java*.
http://en.wikipedia.org/wiki/Comparison_of_C_Sharp_and_Java,
19. května 2007.

9 Wikipedia, EN: *Delegation (programming)*.
http://en.wikipedia.org/wiki/Delegation_(programming),
19. května 2007.

10 Wikipedia, CZ: *Diabetes mellitus*.
http://cs.wikipedia.org/wiki/Diabetes_mellitus,
19. května 2007.

11 Wikipedia, EN: *Dynamic allocation*.
http://en.wikipedia.org/wiki/Dynamic_allocation,
19. května 2007.

12 Johnson, Rod: *Expert One-on-One J2EE Design and Development*. Wrox Press, Somerset, 2002.

13 Wikipedia, EN: *Façade pattern*.
http://en.wikipedia.org/wiki/Facade_pattern,
19. května 2007.

14 Wikipedia, EN: *Fortran*.
http://en.wikipedia.org/wiki/Fortran,
19. května 2007.

15 Wikipedia, EN: *Fourth normal form*.
http://en.wikipedia.org/wiki/4NF,
19. května 2007.

16 Wikipedia, EN: *Garbage collection*.
http://en.wikipedia.org/wiki/Garbage_collection,
19. května 2007.

17 Caucho, Inc.: *Hessian Binary Web Service Protocol*.
http://www.caucho.com/hessian/index.xtp,
19. května 2007.

18 JBoss, RedHat: *Hibernate*.
http://www.hibernate.org/,
19. května 2007.

19 W3C, consortium: *Hypertext Transfer Protocol*.
http://www.w3.org/Protocols/,
19. května 2007.

20 Wikipedia, EN: *Inversion of control*.
http://en.wikipedia.org/wiki/Inversion_of_control,
19. května 2007.

21 Wikipedia, EN: *Interface (computer science)*.
http://en.wikipedia.org/wiki/Interface_(computer_science),
19. května 2007.

22 Sun Microsystems, Corp.: *Java EE at a Glance*.
http ://java.sun.com/javaee/index.jsp,
19. května 2007.

23 Brůha, L.: *Java — hotová řešení*. Computer Press, Brno, 2004.

24 Wikipedia, EN: *Mariner I*.
http ://en.wikipedia.org/wiki/Mariner_I,
19. května 2007.

25 Microsoft, Corp.: *Memory Management Rules*.
http ://msdn2.microsoft.com/en-us/library/ms686638.aspx,
19. května 2007.

26 Microsoft, Corp.: *Microsoft .Net homepage*.
http ://www.microsoft.com/net/,
19. května 2007.

27 Wikipedia, EN: *Model-view-controller*.
http ://en.wikipedia.org/wiki/Model-view-controller,
19. května 2007.

28 Wikipedia, EN: *Multitier architecture*.
http ://en.wikipedia.org/wiki/Tier_(computing),
19. května 2007.

29 Pecinovský, R.: *Myslíme objektově v jazyku Java 5.0*. Grada, Praha, 2004.

30 Wikipedia, EN: *Object oriented modeling*.
http ://en.wikipedia.org/wiki/Object-Oriented_Modeling,
19. května 2007.

31 Wikipedia, EN: *Object oriented programming*.
http ://en.wikipedia.org/wiki/Object-oriented_programming,
19. května 2007.

32 Wikipedia, EN: *Object-relational mapping*.
http ://en.wikipedia.org/wiki/Object-Relational_Mapping,
19. května 2007.

33 Neumann, Ulrich a Lewis, J. P.: *Performance of Java versus C++*.
http ://www.idiom.com/~zilla/Computer/javaCbenchmark.html,
19. května 2007.

34 Momjian, B.: *PostgreSQL — praktický průvodce*. Computer Press, Brno, 2003.

35 Wikipedia, EN: *Programming language*.
http://en.wikipedia.org/wiki/Programming_language,
19. května 2007.

36 Wikipedia, EN: *Reflection (computer science)*.
http://en.wikipedia.org/wiki/Reflection_(computer_science),
19. května 2007.

37 Wikipedia, EN: *Remote procedure call*.
http://en.wikipedia.org/wiki/Remote_procedure_call,
19. května 2007.

38 Wikipedia, EN: *Service-oriented architecture*.
http://en.wikipedia.org/wiki/Service-oriented_architecture,
19. května 2007.

39 Wikipedia, EN: *Simple object access protocol*.
http://en.wikipedia.org/wiki/SOAP,
19. května 2007.

40 Wikipedia, EN: *Software componentry*.
http://en.wikipedia.org/wiki/Component_software,
19. května 2007.

41 Interface21, Ltd.: *Spring framework*.
http://www.springframework.org/,
19. května 2007.

42 Wikipedia, EN: *Structured programming*.
http://en.wikipedia.org/wiki/Structured_programming,
19. května 2007.

43 Sun Microsystems, Corp.: *The History of Java Technology*.
http://www.java.com/en/javahistory/,
19. května 2007.

44 JCP, members: *The Java Community Process Program*.
http://www.jcp.org/en/home/index,
19. května 2007.

45 Sun Microsystems, Corp.: *The Java ME Device Table*.
http://developers.sun.com/techtopics/mobility/device/device,
19. května 2007.

46 Sun Microsystems, Corp.: *The Java ME Platform — the Most Ubiquitous Application Platform for Mobile Devices*.
http://java.sun.com/javame/index.jsp,
19. května 2007.

47 Wikipedia, EN: *Total cost of ownership*.
http://en.wikipedia.org/wiki/Total_cost_of_ownership,
19. května 2007.

48 Wikipedia, EN: *Transport Layer Security*.
http://en.wikipedia.org/wiki/Transport_Layer_Security,
19. května 2007.

49 Wikipedia, CZ: *Unified modeling language*.
http://cs.wikipedia.org/wiki/Unified_Modeling_Language,
7. května 2007.

50 Wikipedia, EN: *WAR (file format)*.
http://en.wikipedia.org/wiki/WAR_(file_format),
19. května 2007.

51 Wikipedia, EN: *Web Services*.
http://en.wikipedia.org/wiki/Webservices,
19. května 2007.

52 Sun Microsystems, Corp.: *Welcome to NetBeans*.
http://www.netbeans.org/,
19. května 2007.

53 W3C, consortium: *XML Schema*.
http://www.w3.org/XML/Schema,
19. května 2007.