UNIVERZITA PARDUBICE
Fakulta elektrotechniky a informatiky

Funkční testování online aplikace pro vedení plánu obsazení kolejí v železniční stanici

Jakub Doležal

Bakalářská práce
2014

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: 2013/2014

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Jakub Doležal**
Osobní číslo:	**I09091**
Studijní program:	**B2646 Informační technologie**
Studijní obor:	**Informační technologie**
Název tématu:	**Funkční testování online aplikace pro vedení plánu obsazení kolejí v železniční stanici**
Zadávající katedra:	**Katedra informačních technologií**

Zásady pro vypracování:

Pro větší osobní železniční stanice jsou vedeny plány obsazení kolejí. Cílem bakalářské práce je připravit a aplikovat testovací scénáře pro online aplikaci pro vedení plánu obsazení kolejí ve vybrané osobní železniční stanici.
Pro vývoj samotné aplikace, a tedy i pro vývoj testů, je předpokladem využití jazyka Java EE a dalších vhodných webových technologií.
Bakalářská práce bude zpracována pro diplomovou práci, která by měla sloužit vlakovým dispečerům jako pomůcka k operativnímu vyhodnocování aktuálních situací.

Rozsah grafických prací:

Rozsah pracovní zprávy:

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

Seznam odborné literatury:

1. Tacy Adam, Hanson Robert, Essington Jason, Tokke Anna. GWT in Action. Manning Publications, 2012. ISBN: 978-1935182849.
2. Mojžíš V., Molková T. Technologie a řízení dopravy I. Univerzita Pardubice, 2002.
3. Ash Lydia. The Web Testing Companion: The Insider's Guide to Efficient and Effective Tests. Wiley, 2003. ISBN-13: 978-0471430216.

Vedoucí bakalářské práce: **Ing. Michael Bažant, Ph.D.**
Katedra softwarových technologií

Datum zadání bakalářské práce: **20. prosince 2013**
Termín odevzdání bakalářské práce: **9. května 2014**

L.S.

prof. Ing. Simeon Karamazov, Dr.
děkan

Ing. Lukáš Čegan, Ph.D.
vedoucí katedry

V Pardubicích dne 31. března 2014

Prohlášení autora

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

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

Souhlasím s prezenčním zpřístupněním své práce v Univerzitní knihovně.

V Pardubicích dne 5. 5. 2014 Jakub Doležal

Poděkování

Děkuji své rodině, že mi pomohla připravit ideální prostředí pro studium a přípravu této bakalářské práce. Děkuji za pomoc a cenné rady od zaměstnanců školy, zvláště pak od Dany Jablonské, Renaty Kalhousové a Petry Jiřištové.

Hlavně bych chtěl poděkovat svému vedoucímu práce Ing. Michaelu Bažantovi, Ph.D. za veškerý čas, který mi věnoval při konzultacích a při administrativních problémech vyskytujících se nejen s odevzdáním bakalářské práce.

Anotace

Pro větší osobní železniční stanice jsou vedeny plány obsazení kolejí. Cílem bakalářské práce je připravit a aplikovat testovací scénáře pro online aplikaci pro vedení plánu obsazení kolejí ve vybrané osobní železniční stanici. Pro vývoj samotné aplikace, a tedy i pro vývoj testů, je předpokladem využití jazyka Java EE a dalších vhodných webových technologií. Bakalářská práce bude zpracována pro diplomovou práci, která by měla sloužit vlakovým dispečerům jako pomůcka k operativnímu vyhodnocování aktuálních situací.

Klíčová slova

železniční, stanice, plán, obsazení, kolejí, testování, web, aplikace

Title

Functional testing of the on-line application for graphical timetable in a railway station

Annotation

For greater passenger railway stations are prepared plans that keep track occupancy. Aim of this work is to prepare and apply test scenarios for online application management plan contained in the selected tracks passenger rail station. For the development of the application itself, and thus also for test development is a prerequisite for using Java EE and other relevant web technologies. Bachelor thesis will be prepared for a thesis, which should serve as a train dispatcher aid for the operational assessment of the current situation.

Keywords

Railway, station, schedule, cast, rails, testing, web, aplikace

Obsah

Seznam zkratek

API	Application Programming Interface
CSS	Cascading Style Sheets
DTO	Data Transfer Object
GVD	Grafikon vlakové dopravy
GWT	Google Web Toolkit
HTML	HyperText Markup Language
JRE	Java Runtime Environment
JSNI	JavaScript Native Interface
JRE	Java Runtime Environment
RIA	Rich Internet Applications
RCP	Regionální centrum provozu
SVG	Scalable Vector Graphics
W3C	World Wide Web Consortium
W-POK	webový plán obsazení kolejí
XML	Extensible Markup Language

Seznam obrázků

Seznam tabulek

Úvod

Testování je jeden ze základních problémů nejen při programování, ale vlastně každodenního života. Vše, co v životě děláme, je dobré si po sobě zkontrolovat a přesně tak to funguje i při programování. Testování je tedy kontrola, zda program, či jen část kódu, funguje přesně tak, jak jsme chtěli a podle očekávání. Naše očekávání pak definuje textová dokumentace ke konkrétnímu programu zvaná také jako specifikace, která se u větších programů píše ještě před začátkem jakéhokoli programování.

Pro programování máme velké množství možností, jak napsat program. Ať už výběr správného programovacího jazyka, nebo prostředí, ve kterém se bude programovat. Jednotlivé části webové aplikace se pak skládají z různých programovacích jazyků, které spolu ve výsledku dávají potřebné vlastnosti, které očekáváme a které jsou také sepsány ve specifikaci aplikace.

Cílem této práce je provést testování konkrétních částí kódu v internetové online aplikaci W-POK pro vedení plánu obsazení kolejí ve vybrané osobní železniční stanici. Tato aplikace byla vytvořena v diplomové práci: „Webová aplikace pro vedení plánu obsazení kolejí v železniční stanici“ od Bc. Jana Žampacha. (1)

Pomocí různých nástrojů tato bakalářská práce dojde k výsledkům, které dále pomohou při zpracování a posuzování, jak dále vylepšit tuto aplikaci při zavedení do praxe a ke skutečnému používání v železniční stanici.

1 Testování

Testování je nutnou součástí vývoje každého programu. Programátor chce, aby jeho program byl správný a fungoval tak, jak původně zamýšlel a proto se využívá testování, aby se dohledali chyby. Jednotlivé chyby se musí posoudit, jestli jsou pro program závažné či nikoliv.

Provádí se různými způsoby a vždy záleží na velikosti týmu, který projekt bude vytvářet a dále na velikosti projektu, který se bude vytvářet. Podle toho se volí vhodné techniky a metody pro vývoj softwaru a tedy i testování.

1.1 Proč testovat?

V této kapitole bude uvedeno, jakým způsobem se přistupuje k nalezené chybě z hlediska její váhy v programu a jeden příklad z praxe, kdy se doopravdy nevyplatilo chybu zanedbat. Z tohoto všeho by potom mělo čtenáři vyplynout, proč je testování důležité a že se nedoporučuje ho zanedbávat. Informace v této kapitole pochází ze zdroje (2).

1.1.1 Definice chyby

Jak už bylo řečeno v úvodu, testování má být kontrolou pro programátory, zda dělají vše podle specifikace, nebo také jinak řečeno, že nedělají chyby. Taková chyba se pak dá definovat různými způsoby například:

1. software dělá něco, co by podle specifikace dělat neměl,
2. software nedělá něco, co by podle specifikace dělat měl,
3. software dělá něco, o čem se specifikace nezmiňuje,
4. software nedělá něco, o čem se specifikace nezmiňuje, ale měla by se zmiňovat,
5. software je obtížně srozumitelný, těžko se s ním pracuje, je pomalý, nebo se s ním koncovému uživateli bude těžko pracovat.

Tyto chyby se snaží ve velké firmě najít tester, nebo skupina testerů, kteří mají za úkol příslušný software otestovat, přičemž platí, že čím dříve se chyba odhalí, tím méně nákladů a úsilí stojí její oprava.

- Př.: Takovým příkladem je třeba chyba nalezená v době, kdy se ještě rozhoduje o specifikaci aplikace a tato chyba většinou nestojí nic, popřípadě nový papír a čas, který tým stráví v důsledku vyřešení chyby a její opravy.

- Př.: Naproti tomu, když se objeví chyba v té části vývoje, když už se má produkt dávat na trh a je to chyba doopravdy taková, která nutí produkt stáhnout z trhu, taková chyba pak stojí velké množství peněz jak na opravu, tak i režii a všechny věci, které jsou chybou postižené, se musí buď vyhodit, nebo se na ně musí udělat nějaká „záplata“.

1.1.2 Rozhodování nad závažností chyby

Každý tester se pak potýká s problémem, co všechno se musí otestovat a co všechno se testovat nemusí. K neopravení chyby může být několik důvodů:

1. není na to čas – chyba má takový rozsah testování, který nelze splnit v reálném termínu,

   (Př.: při testování nástroje kalkulačka ve Windows XP nelze v reálném čase otestovat všechny případy sčítání libovolného počtu čísel a všech jejich kombinací)

2. ve skutečnosti to ani není chyba – jedná se třeba o vlastnost systému, tedy se s tím musí počítat, že to tak bude a je třeba i nutné to zpětně zařadit do specifikace,

   (Př.: jak se má program zachovat při různých stavů počítače, při kterých se operační systém zachová jinak, než to aplikace požaduje, atd.)

3. oprava je příliš riskantní – stává se často, když se při opravě jedné chyby objeví chyby jiné, které se pod tlakem časově napjatého plánu k vydání konečné verze produktu mohou jevit jako skutečně příliš riskantní. V takovém případě může být vhodnější chybu v produktu ponechat, upozornit na ni jako na známou chybu a vyhnout se riziku zavlečení nových neznámých chyb, které by se zavedly opravením této chyby,

4. oprava tzv. „nestojí za to“ – toto je pro případ, že chyby se objevují velice zřídka, nebo na málo používaných funkcích a lze je opominout, nebo rozumným způsobem obejít.

Tedy nejlépe je k chybám přidělit jejich váhu. Většinu závažných chyb pak spolu konzultují všichni, co s vývojem produktu mají co dočinění hlavně pak testeři, programátoři a manažeři projektu, kteří každý z nich můžou mít jiný názor na konkrétní chybu a jiné možné řešení.

1.1.3 Příklad z praxe

Příklad z praxe, kdy se doopravdy nevyplatilo vynechat nalezenou chybu, který je převzat z (2):

„Chyba v dělení s pohyblivou řádovou čárkou u procesoru Intel Pentium, 1994

Napište do kalkulačky svého osobního počítače následující výpočet

*(4195835 / 3145727) * 3145727 – 4195835*

Pokud je výsledek nula, je počítač v pořádku. Jestliže ale dostanete jako odpověď cokoliv jiného, znamená to, že máte v počítači starý procesor Pentium s chybou v dělení čísel s pohyblivou řádovou čárkou – to je softwarová chyba, kterou se podařilo vypálit do počítačového čipu neboli integrovaného obvodu a v procesu výroby ji tak mnohokrát zopakovat.

Na to, že při dělení vznikají problémy s neočekávanými výsledky, přišel při jednom ze svých experimentů, řešeném na počítači s procesorem Pentium , Dr. Thomas R. Nicely z Lynchburg College (Virginia) dne 30.října 1994. Svoje zjištění ihned zveřejnil na Internetu a záhy s sebou strhl doslova bouři, protože stejnou chybu dokázalo navodit mnoho jiných lidí, kteří přišli i na jiné situace, jež vedly k nesprávným výsledkům. Tyto situace byly naštěstí poměrně vzácné a nesprávnými výsledky trpěly jen určité vědeckotechnické výpočty, extrémně náročné na matematické operace. Většina lidí se při běžné práci, výpočtech daní nebo v podnikání s touto chybou nesetká.

Na tomto příběhu není ani tak zajímavá chyba samotná, jako způsob, s jakým se Intel s touto situací vypořádal:

- *Inženýrům pro testování softwaru se podařilo na tento problém při prováděných testech přijít, a to ještě před uvedením čipu na trh. Vedení firmy Intel ale usoudilo, že zjištěný problém není dostatečně závažný ani pravděpodobný, že nestojí za to jej ani opravovat, ani publikovat.*
- *Po oznámení chyby se Intel prostřednictví tiskových zpráv a veřejných prohlášení snažil zmírnit přisuzovanou závažnost zjištěné chyby.*
- *Pod dalším tlakem nabídl Intel nakonec výměnu vadných čipů, ale jen těm uživatelům, kteří dokázali, že byli chybou nějakým způsobem poškozeni.*

Strhl se veřejný křik. Internetové diskusní skupiny byly zahlceny rozhořčenými zákazníky, kteří po Intelu požadovali opravu problému. Novinové články líčily společnost Intel jako nezodpovědnou a nedůvěryhodnou. Nakonec se Intel za způsob, jakým se k chybě postavil, veřejně omluvil, a uvolnil přes 400 miliónů dolarů jako náklady na náhradu vadných čipů. Dnes již Intel oznamuje známé problémy přímo na svém webovém serveru a pečlivě sleduje připomínky zákazníků i na internetových diskusních skupinách.“

1.2 Modely životního cyklu vývoje softwaru z hlediska testera

V této kapitole bude uvedeno, jak může vypadat proces životního cyklus programu, kde v tomto procesu vývoje zasahuje tester a jak obtížná z hlediska ostatních modelů je práce testera. Informace v této kapitole jsou převzaty z (2).

Proces, kterým je vytvářen softwarový produkt od úplného začátku do úplného konce (veřejného uvedení na trh), se nazývá model životního cyklu vývoje softwaru. Tento proces probíhá různými způsoby. Některý software se vyvíjí s přísnou disciplínou pečlivého pracovníka, některý software vzniká pod vládou mírně řízeného chaosu a jiný software je doslova živelně slepen a sdrátován. Zákazník obvykle pozná, jaký postup byl při tvorbě softwaru skutečně použit.

Takže při výrobě softwaru je možné použít celou řadu různých metod. Nikdy není možné určit, která z metod je ta jedině správná. Většina metod vychází ze čtyř nejčastěji používaných modelů, které jsou dále rozvedeny v následujících podkapitolách:

1. model velkého třesku
2. model „opravuj a programuj“
3. model vodopádu
4. spirálový model

Každý z modelů má svoje výhody a nevýhody. Tester musí vždy danému projektu přizpůsobit způsob testování.

1.2.1 Model velkého třesku

Tato metoda je vlastně velmi jednoduchá. Na jedné straně stojí velké množství peněz a lidé a na druhé straně stojí velké množství energie. Z toho všeho možná vznikne dokonalý

softwarový produkt, nebo ne. Veškeré úsilí se soustředí na vývoj softwaru a psaní jeho programového kódu. Většinou nejsou přesně známé vstupní požadavky, taky proto tato metoda velmi často postrádá plánování a rozvrh práce. V průběhu práce se téměř neprovádí žádné testování a dokonce není ani přesně stanoveno datum odevzdání softwaru.

Tester mívá s tímto projektem snadnou i obtížnou práci. Projekt, když je prakticky hotov, tak není možné do něj zasahovat anebo opravovat chyby. Tester se tedy musí omezit jen na sepsání nalezených chyb, aby se zákazník s chybami mohl předem seznámit a naučil se s nimi žít. Pokud se snaží tester najít všechny chyby, dostane se většinou do konfliktu s dodavatelem, který spěchá na expedici projektu. Proto se začínajícímu testerovi doporučuje, aby se testování projektu vytvářeného tímto modelem raději vyhnul, jelikož se jedná o náročnou práci, kde se tester dostává k projektu až na konci, tedy projít vše je velice obtížné a vše se musí od začátku učit.

1.2.2 Model „programuj a opravuj“

Tým začíná hrubou představou výsledného produktu a udělá si jednoduchý návrh. Potom se pustí do dlouhého procesu programování, testování a opravování. Tento model může velmi brzy ukázat výsledek své práce, je i výhodný pro malé projekty, které pak slouží jako prototypy a demonstrační příklady. Ani u tohoto modelu se žádné velké testování neprovádí a tester spolupracuje s programátory. Každý den tester dostává novou verzi softwaru, kterou bude stále dokola testovat a zkoušet. Mezitím ale programátoři pokračují dále v programování a nalezené chyby testera se tedy nestíhají opravovat pro všechny verze programu. Chyb bude ale stále ubývat, dokud nenastane čas na expedici softwaru. Takovouto prací se tester setká nejčastěji. Tento model je určitě dobrý úvod do vývoje softwaru a k získání znalostí testera a pohybování se v týmu a spolupráci mezi programátory a testery.

1.2.3 Model vodopádu

Používá při výuce programování a přináší velmi dobré výsledky.

Jednotlivé kroky: nápad, analýza, návrh, vývoj testování, výsledný produkt.

Tento model postupuje od jednoho kroku k dalšímu až po zhodnocení práce a uzavření daného kroku, ke kterému se už později nevrací.

V tomto modelu jsou tři důležité věci:

a) velký důraz je kladen na specifikaci výsledné podoby produktu,

b) jednotlivé kroky jsou diskrétní a nepřekrývají se,

c) není možné se vracet zpět.

V dnešní době se ovšem můžeme setkat s tím, že než se dostaneme k dalšímu kroku, může pozbýt platnosti prvotní důvod pro vytvoření softwaru. Z technického pohledu má modul velkou výhodu, protože v každém kroku je všechno detailně popsáno a specifikováno. Tester tedy přesně ví, co má testovat a může si vytvořit jasný plán a rozvrh testů. Nevýhodou tohoto modelu je to, že testování přichází až před odevzdáním projektu a při testování se může přijít na chybu, která vznikla na začátku vývoje a odstranění této chyby může být velmi nákladné.

1.2.4 Spirálový model

Tento model zavedl Barry Boehm v roce 1986 a nazval jej „A Spiral Model of Software Development and Enhancement“ (Spirálový model vývoje a zdokonalení softwaru). Ukázalo se, že tento model je velmi efektivní. Úplně ze začátku se začne se stručnou definicí nejdůležitějších funkcí, vyzkoušejí se a vyžádají se připomínky od zákazníků. Potom postup opakujeme, dokud se nedostaneme k výslednému produktu. Při každém průchodu spirálou se provádí těchto 6 kroků:

1. určení cílů, alternativ a omezení,
2. rozpoznání a řešení rizik,
3. vyhodnocení alternativ,
4. vývoj a testování aktuální úrovně,
5. plánování další úrovně,
6. rozhodnutí o postupu na další úroveň.

Vlastně tento model spojuje všechny předcházející modely. Tester v něm může ovlivnit projekt hned na začátku testování a provádí se průběžně. Na závěr se už jen vše prověří, zda všechny testy fungují správně podle očekávání, zda se nic nezměnilo.

1.3 Test černé, bílé skříňky, statické a dynamické testování

Při testování černé skříňky ví tester jen to, co má testovaný software dělat. Nemůže se podívat dovnitř a neví tedy, jak software pracuje. Jestliže tedy napíše nějaký údaj na vstupu, na výstupu dostane odpovídající výsledek a může tedy jen tento výsledek pozorovat. Jako softwarový tester si pak může zjištěný výsledek ověřit na jiném „certifikovaném“ zařízení a vzájemným porovnáním odvodit, jestli software pracuje správně.

Při testování bílé skříňky má tester k dispozici zdrojový kód programu. Jeho zkoumání mu může pomoci při testování, ale bohužel je zde skryto jedno riziko. Člověk by mohl přizpůsobit testování činnosti programového kódu, což by znamenalo, že test nebude objektivní.

Statické testování se provádí na něčem, co neběží. Zkoumaný objekt se jen prohlíží a reviduje, aniž by se cokoliv zapisovalo do paměti přístroje, nebo v jakýkoli moment měnilo svůj stav či uložené údaje.

Dynamické testování probíhá se spuštěným softwarem, který se postupně zastavuje v námi zvolené chvíli a v té chvíli se zkoumá, zda program dělá, ukládá, vybírá to, co má a nic jiného. Informace v této kapitole jsou převzaty z (2).

1.3.1 Statické testování černé skříňky: testování specifikací

Specifikace mohou mít různou podobu. Můžou mít podobu psaného dokumentu nebo grafických schémat. Někdy se může stát, že testovaný projekt nemá specifikaci napsanou na papíře, ale členové týmu mají určitou představu o projektu, který budou vytvářet. Pak se

jich budeme dotazovat a můžeme sesbírané informace systematicky zapsat a dát je kolovat k revizi.

Při testování nás nezajímá, jak byly jednotlivé informace získány, ale stačí nám pouze, že byly převedeny do výsledné specifikace produktu. Jako tester můžeme tedy převzít tuto specifikaci, provést nad ní statické testování černé skříňky a pečlivě prozkoumat možné chyby.

1.3.2 Dynamické testování černé skříňky

Říká se mu také testování softwaru s klapkami na očích. Dynamické testování probíhá na běžícím programu, kdy s tímto programem pracujeme stejným způsobem jako zákazník. Neznáme však způsob práce testovaného softwaru. Potřebujeme znát určitou definici činnosti softwaru. Jakmile známe vstupy a výstupy, můžeme se pustit do definice testových případů. Nejdůležitějším úkolem je volba vhodné množiny testových případů, kde se v této práci píše později při definování hraničních podmínek, atd.

1.3.3 Statické testování bílé skříňky: zkoumání návrhu a programového kódu

Jedná se o proces pečlivé a metodické revize návrhu, architektury a kódu programu, přičemž se hledají chyby bez spuštění programu. Takovému testování se říká strukturální analýza.

Smyslem statického testování bílé skříňky je včasné nalezení chyb a možnost nalezení i takových chyb, které by se při spuštěném programu hledaly obtížně. Čím více nezávislých osob provádí testování, tím lépe, zejména pokud se jedná o revizi na nízké úrovni a je prováděna v časných stádiích vývojového cyklu. Mnohé týmy podceňují statické testování bílé skříňky, protože se domnívají, že testování zbytečně prodlužuje proces vývoje a považují jej za příliš časově náročné, příliš nákladné a nepříliš produktivní. Lze porovnat tuto metodu testování s metodou dříve zmíněnou, kdy se testování provádí úplně na konci projektu.

Formální revize je pak proces, pod nímž probíhá statické testování bílé skříňky. Formální revize má 4 základní prvky:

- identifikace problémů (hledání chybných věcí a chybných prvků),
- dodržování pravidel (určených pravidel a zásad pro určitý programovací jazyk, …),
- příprava,
- písemná zpráva.

Formální revize fungují, když se dodržuje pevně stanovený postup a potom správně dokáží chyby odhalovat chyby včas.

Vedení formální revize má ještě další nepřímé důsledky:

- komunikace,
- kvalita,
- pospolitost týmu,
- řešení.

Revize partnerem – nejméně formální revize softwaru. I při této kamarádské revizi je nutné dodržovat čtyři klíčové elementy revize.

Průchody kódu jsou další zvýšení formálnosti revize. Programátor prezentuje svůj kód dalším kolegům. Oponenti mají možnost se předem seznámit se softwarem, připravit si komentáře a otázky. Je důležité napsat o revizi zprávu, která se pak dále dokládá pro budoucí další testování.

Inspekce jsou nejformálnější revize. Jsou vysoce strukturované a provádějí je vyškolení pracovníci. Kód prezentuje překladatel, tedy ne tvůrce kódu. Revize se účastní inspektoři, kteří mají revidovat programový kód z různých hledisek. Po skončení schůzky se inspektoři setkají a společně proberou nalezené vady a připraví společně s moderátorem písemnou zprávu. Inspekce se ukazuje jako velice efektivní prostředek pro vyhledávání chyb.

1.3.4 Dynamické testování bílé skříňky

Při dynamickém testování zkoumáme programový kód za běhu programu. Rozhodujeme se, co testovat a co netestovat a jak k testování přistupovat podle informací, které získáme z pozorování kódu programu a jeho činnosti. Běžně se toto testování označuje jako strukturální testování, protože při návrhu a vlastním provádění testů vidíme a využíváme podkladovou strukturu kódu programu. Znalost způsobu práce softwaru má vliv na postup při testování. Při dynamickém testování můžeme software nejen testovat, ale i řídit.

Dynamické testování bílé skříňky zahrnuje čtyři následující okruhy činností:

- přímé testování funkcí, procedur, podprogramů a knihoven na nízké úrovni,
- testování softwaru na nejvyšší úrovni, jako kompletního programu. Testované případy při tom upravíme podle toho, co víme o činnosti softwaru,
- jestliže získáme přístup pro čtení proměnných a stavových informací, můžeme si snáze ověřit, jestli prováděné testy skutečně dělají to, co od nich očekáváme. Zároveň tak můžeme donutit software k takovým věcem, které by bylo obtížné navodit při normálním testování,
- měření množství konkrétních částí testovaných kódů, podle něhož upravíme testy a testové případy.

Dynamické testování bílé skříňky nesmíme zaměňovat s klasickým laděním. Cílem dynamického testování bílé skříňky je vyhledávání chyb, zatímco cílem ladění je i jejich opravení. Nicméně musíme připustit, že se obě činnosti překrývají a to v procesu rozpoznání místa a příčiny vzniku chyby. V této fázi je nutné oddělit práci programátora,

který píše kód programu a následně nalezené chyby opravuje, a práci testera, který chyby vyhledává. Při této činnosti musí i tester napsat část programového kódu pro řízení testů.

Jestliže tester provádí testování na nízké úrovni, bude pracovat s celou řadou stejných nástrojů jako programátor.

Po otestování dat je nutné také vyzkoušet všechny stavy programu a přechody mezi nimi. Musíme se pokusit o úplnou analýzu programovacího kódu. Znamená to otestovat vstup a výstup z každého modulu, prověřit každý jednotlivý řádek programu a jít v softwaru po každé cestě v programové a rozhodovací logice.

Nejjednodušší formou úplné analýzy programového kódu je krokování programem po jednotlivých řádcích, při čemž navštívené řádky kódu sledujeme pomocí debuggeru vestavěného do kompilátoru. Tento způsob naprosto dostačuje pro malé programy nebo jednotlivé moduly. Pro většinu softwaru se používá úplný analyzátor kódu, který se napojí na testovaný software. Při testování běží transparentně na pozadí a zaznamenává informace o běhu programu.

Dynamické testování bílé skříňky je velice silným nástrojem. Ušetří testerovi hodně práce, protože z interních informací snadno zjistí, co je potřeba testovat. Zvýší tak značně efektivitu testování.

1.4 Testování stavů, modulů, test splněním, test selháním, třídy ekvivalence, hraniční podmínky, testování stavů a řízení toku softwaru

V této kapitole bude uvedeno, jakými způsoby lze testovat a na co se zaměřit při testování. Nelze totiž otestovat vše a důležité je si určitě co vše testovat, aby se pak testy dokázaly považovat za užitečné a ne za zbytečné. Informace v této kapitole jsou převzaty z (2).

1.4.1 Testy splněním a testy selháním

Test splněním (test-to-pass) je ve skutečnosti kontrola, jestli je vůbec software funkční. Tento test se provádí hlavně v počáteční fázi. Můžeme být překvapeni, kolik chyb se najde při normálním běžném používání softwaru.

Teprve potom se provádí testy selháním (test-to-fail). Přejde se k úskočným pokusům, při nichž se tester pokouší chyby vyprovokovat, a tím chyby vlastně nachází. Záměrně se volí případy se zřetelem na vyzkoušení známých slabých míst v softwaru.

1.4.2 Rozdělení tříd ekvivalentních případů

Při výběru vhodné množiny testových případů se metodicky redukuje množina všech možných případu do menší podmnožiny. Té se pak říká třída ekvivalence, která se redukuje dál až do té doby, než se naleznou případy, které by mohli nějak narušit funkčnost programu.

(Př.: při testování sčítání dvou čísel na kalkulačce se stačí omezit na otestování hraničních případů, jako jsou třeba 0+0, 0+1, 1+0, … nebo z druhé strany 1+99999…, atd.).

(Př.: nebo při testování vkládání znaků do pole se dá testovat velikost tohoto pole, tedy zda jde zadat 0 znaků, nebo třeba až 255 znaků či více (používají se právě mocniny dvou,

protože počítač pracuje ve dvojkové soustavě). Potom se nebudou testovat všechny jednotlivé znaky na klávesnici, ale třeba jen ty speciální jako jsou: /, \, :, *, …).

1.4.3 Hraniční podmínky

Hraniční podmínky jsou dost zvláštní, protože programování je na hraniční problémy velice citlivé. Jak už bylo řešeno, software je ve své podstatě binární – buď je určitá věc pravda, nebo nepravda. Pokud programátor pracuje s číselným intervalem, může se stát, že chybně podchytí čísla na hranicích intervalu a právě tam pak může být chyba.

Máme různé typy hraničních podmínek, které se mnohdy odhalí až při hlubším zkoumání.

Nejvíce chyb se najde, když se vytvoří dvě množiny ekvivalentních případů, kde první podmnožina bude obsahovat data, u kterých se očekává správní činnost programu (jsou to hodnoty, které leží uvnitř hranic intervalu) a druhá podmnožina obsahuje data, která mohou způsobovat chyby (jsou to údaje, které leží vně mimo hranice intervalu). Tester se vždy snaží najít platný údaj těsně v hranici intervalu, dále poslední možnou platnou hodnotu, a neplatnou hodnotu těsně za hranicemi intervalu.

Existují ale ještě další interní hraniční podmínky, tak zvané subhraniční podmínky. Tyto podmínky pak pomáhají odhalit samotní programátoři. I tyto podmínky musí tester vzápětí zkontrolovat.

1.4.4 Testování stavů

Druhou důležitou stránkou testování je ověřování logiky toku řízení programů prostřednictvím jeho různých stavů. Pod pojmem stav softwaru se rozumí stav nebo režim, v němž se software momentálně nachází.

Počáteční stav je stav, ve kterém se software nachází po spuštění. Kdykoliv pak vybereme některý z nástrojů a příkazů, při kterém software změní svůj vzhled, skladbu nabídek nebo svoji činnost, tak se fakticky změní jeho stav. Program projde určitou posloupnost částí svého programového kódu, přepne vybrané bity, nastaví příslušné proměnné, načte nějaká data a nakonec přijde do jiného stavu. Tester musí vždy testovat stavy programu a přechody mezi nimi.

1.4.5 Testování logiky toku řízení softwaru

S testováním stavů softwaru a logiky řízení jsou spojeny stejné problémy. Navštívit všechny stavy programu je možné, ale je nemožné projít všechny cesty všech změn stavů. Řešením je vhodnou metodou rozdělit třídy ekvivalence na stavy programu a cesty mezi nimi. Bereme na sebe riziko, že některé stavy a přechody neotestujeme. Toto riziko snížíme vhodnou a inteligentní volbou.

Prvním krokem je vytvoření vlastní mapy stavů a přechodů. Taková mapa může být součástí specifikace produktu.

Každý diagram k testování logiky toku by měl zobrazovat:

- každý stav, v němž se software může nacházet,
- vstup nebo podmínku, při které se přechází z jednoho stavu do druhého,
- nastavení podmínek a generovaný výstup při vstupu nebo výstupu z daného stavu.

2 Práce s testy

Zde v této kapitole bude uvedeno, jaké konkrétní testy se používají pro testování v programovacím jazyce Java a použité konkrétní metody pro testování v této bakalářské práci. Informace v této kapitole jsou převzaty ze zdrojů (4, 7).

2.1 Testy prováděné v java aplikaci NetBeans

V programu NetBeans lze psát projekty přímo celých webových aplikací, což ulehčuje práci programátorům a na tyto aplikace lze pak nasazovat jednotlivé testy, nebo lze i naopak vytvářet testy, podle kterých se pak vytvoří šablona celé třídy, kterou si už programátor poupraví podle svých představ.

Z druhé strany je tedy možné použít průvodce i pro vytváření testů už z naprogramovaného kódu, který se samozřejmě musí doplnit o různá další specifika, aby mohl správně fungovat a testoval doopravdy to, co potřebujeme testovat. Samotný takto vytvořený test má na svém konci metodu false("zpráva"), která vyvolá chybu a vypíše přednastavenou zprávu na výstup konzole.

2.2 Přehled testovacích frameworků

Program NetBeans, který se tedy hodí k vývoji aplikace W-POK, v základu podporuje používání dvou testových frameworků, kterými jsou:

- JUnit,
- TestNG.

Hlavní rozdíl mezi nimi je v pojetí přidávání testů do testovacích skupin a prací s takto vytvořenými skupinami. Jelikož JUnit to defaultně neumí a musí si k tomu vzít na pomoc třídu TestSuite a v té to potom spouštět, TestNG zvládá přidat jen do anotace @Test další parametr. Tedy se takovéto vytváření skupin zdá jednodušší, ale možná v některých situacích méně přehledné.

Vedle tohoto hlavního rozdílu jsou zde už jen malé rozdíly, jinak jsou testovací frameworky velmi podobné.

Lze najít a doinstalovat také další testovací frameworky, které jsou podporovány pro testování Java aplikací. V bakalářské práci je použitý testovací framework JUnit.

2.2.1 Testovací třídy – junit.framework.Assert

Metody, které se využívají k testování, jsou převážně z knihovny junit.framework.Assert. Těmito metodami jsou:

- assertEquals(x,y) – pro kontrolu, zda objekty x a y jsou stejné (porovnává se podle metody equals()). Metoda je mnohonásobně přetížená, lze tedy pracovat s nejrůznějšími datovými typy,
- assertSame(x,y) – pro kontrolu, zda objekty x a y ukazují na stejný prvek (porovnání se zde provádí pomocí logického operátoru rovnosti),

- assertFalse(b) – pro kontrolu, zda výsledek výrazu b je False,
- assertTrue(b) – pro kontrolu, zda výsledek výrazu b je True,

Všechny tyto metody při chybě vyvolávají výjimku typu AssertionFailedError. Metody mohou mít navíc ještě jeden parametr typu String pro vypsání konkrétní zprávy, při vyvolání výjimky. Příklad je v příloze D.

Anotace @Test, která se píše před každou testovou třídu, poví JUnit, že public void metoda, ke které je anotace @Test přidána může běžet jako test. Všechny výjimky vyhozené testem budou vyhodnoceny JUnit jako chyba. Pokud žádná výjimka nebyla vhozena, test skončí jako úspěšný.

Tato anotace supluje 2 další funkční parametry, které jsou:

1. expected – metoda může vyvolat výjimku. Pokud nemůže výjimku vyvolat nebo vyvolá jinou, než je deklarovaná, tak test skončí jako s metodou fail("zpráva"), např.:

```
@Test(expected=IndexOutOfBoundsException.class)
public void outOfBounds()
{
      new ArrayList&lt;Object&gt;().get(1);
}
```

2. timeout – zavolá metodu fail("zpráva"), pokud překročí definovanou délku času výpočtu, než je délka stanovená (v milisekundách). Př.:

```
@Test(timeout=100)
public void infinity()
{
      while(true);
}
```

2.2.2 Souhrnný test – junit.framework.TestSuite

Dále lze pak testy vložit do jednoho konkrétního souhrnného testu, který tyto testy spustí všechny najednou. Zjednoduší to práci, při postupném testování, jestliže chceme zpětně zkontrolovat, zda se z původních testů nic nezměnilo a zda programátor nevytvořil svojí novou změnou nějakou chybu v už naprogramovaném softwaru.

Tento souhrnný test má pak v sobě metodu, která vrací typ třídy TestSuite z balíčku junit.framework.TestSuite. Tato třída v sobě může mít několik testových tříd, u kterých vždy zavolá jejich konstruktor. Má jednoduché použití pro vkládání testů pomocí metody addTestSuite(), která jako parametr dostane testovou třídu. Podmínkou ale je, že každá třída, která má být přidána do TestSuite, tak musí dědit ze třídy junit.framework.TestCase, aby mohla být přidána do souhrnného testu.

Př.:

```
public class AllTests {
      public static TestSuite suite(){
            TestSuite suite = new TestSuite();
            suite.addTestSuite(Trida1Test.class);
            suite.addTestSuite(Trida2Test.class);
            return suite;
      }
}
```

Vždy je dobré po dokončení konkrétního testu jej vložit do souhrnného test, aby se na tento nově vytvořený test nezapomnělo.

3 Webová aplikace W- POK

Zde v této kapitole bude uvedeno něco málo o testované webové aplikaci W-POK. Obecné informace proč se vůbec aplikace začala vyvíjet, její výhody oproti tomu co místo ní bylo, použité technologie na který aplikace staví a které využívá ke své práci, popis některých tříd, a další potřebné věci, které dají čtenáři vhodný základ k pochopení problému. Informace v této kapitole jsou převzaty z (4, 7).

3.1 W-POK obecně

Diplomová práce, podle které byla vytvořena aplikace W-POK, si kladla za cíl navrhnout vhodný softwarový nástroj, který by umožnil částečnou automatizaci procesu tvorby plánu obsazení kolejí a sloužil by jako pomůcka staničním zaměstnancům. Slouží tedy především jako pomůcka staničním dispečerům k operativnímu řízení provozu ve stanici. Jeho přínos je jednak při řešení běžných situací, jako jsou zpoždění vlaků nebo při sestavě vlakové cesty jednotlivých vlaků. Další přínos je pak zejména v případě řešení nestandardních situací, jako je např. výluka koleje, kdy musí být všechny vlaky z inkriminované koleje přemístěny jinam.

„V současné době nejsou pro sestavu POK používány jednotné software nástroje a POK nejsou sestavovány centrálně. Každá větší vlaková stanice má své konvence. To je způsobeno jednak tím, že neexistuje žádný specializovaný software ani podpůrná pomůcka pro tvorbu plánů a jednak neexistencí interního předpisu, který by problém tvorby POK nějakým způsobem upravoval. To sice dává možnosti dispečerům upravovat si POK specificky dle svých potřeb, ale tím pádem neexistuje mezi POK žádná jednotnost.“ (1)

Výhodou potom tedy bude jednotné ošetřování chyb, následné vytvořené novější verze, jednotlivá vylepšení a přidávání nových technologií a to už může být aplikace zavedena v provozu. Bude navíc stačit jen jeden vyškolený centrální tým, který bude vědět, jak podpořit správnou funkčnost aplikace, jelikož bude jednotná a mnoho dalších výhod vyplývajících z centralizovaného provedení.

Data pro W-POK jsou z aktuálního grafikonu vlakové dopravy, přičemž jsou převedena do specifické grafické podoby, která přehledně koncentruje data vztažená ke konkrétní železniční stanici. Soubor dat pro vytvoření plánu tvoří seznam dopravních kolejí a seznam vlaků, které projíždí v železniční stanicí v průběhu jednoho dne. Důležitými atributy vlaku jsou druh, číslo, výchozí pobytová kolej a časové údaje o pobytu vlaku ve stanici.

Aplikace vytváří novou koncepci, jejíž hlavní součástí bude dynamický plán, který poskytuje možnosti interaktivních zásahů do plánu a umožní tak dispečerům využít lépe jeho potenciál, než je tomu ve statickém pojetí plánu obsazení kolejí. Pomocí aplikace bude tento sestavený plán veden a spravován. Umožní dispečerům tuto aplikaci operativně využívat a zadávat zpoždění vlaků, změny doby pobytů na kolejích, změny pobytových kolejí. Tyto změny budou proveditelné jak na úrovni grafické – přímo v plánu, tak na úrovni seznamu všech vlaků. Aplikace by měla sloužit jako interaktivní podpůrná pomůcka pro rozhodování při řešení nastalých nestandardních situací. Aplikace vychází ze stávající zavedené pomůcky, protože by bylo velmi obtížné obsáhnout v plánu obsazení kolejí všechny zavedené konvence.

Navrhovaná aplikace ve své podstatě podporuje dosavadní zvyklost, kdy měl každý dispečer k dispozici svou kopii plánu obsazení kolejí. Aplikace bude spuštěna na serveru

společnosti a dispečeři si budou moci kdykoliv zobrazit aktuální stav plánu obsazení kolejí, popř. jej editovat pod svým uživatelským účtem a se svými oprávněními. Tato architektura řešení má určitě nespornou výhodu v tom, že aplikaci si může zobrazit jakákoli oprávněná osoba s připojením do firemní sítě. Jediným požadavkem na klientské počítače tak zůstává mít aktualizovaný prohlížeč a funkční připojení k síti. Nevýhodou architektury klient-server může být ve chvílích přetížení sítě, kdy samotná komunikace po síti, kdy server bude vytížen při komunikaci s ním, se mohou vyskytnout prodlevy. Další nevýhodou zmíněné architektury je poměrně náročnější implementace, jak z hlediska použitých technologií, tak z hlediska návrhu a samotného ladění a nasazení aplikace. Informace v této kapitole jsou převzaty z (1).

3.2 Zvolené technologie aplikace

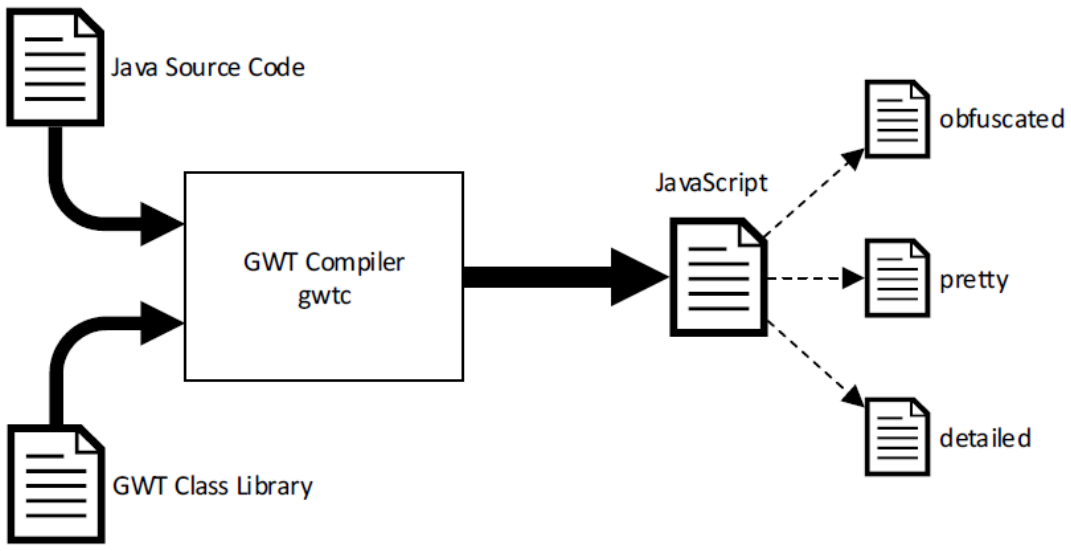
Technologie, které využívá aplikace, jsou zvoleny s ohledem na to, co aplikace potřebuje ke svému chodu. Jednotlivý popis technologií dávají následující odstavce. Informace v této kapitole jsou ze zdroje (1).

3.2.1 GWT

GWT (Google Web Toolkit) je sada vývojářských nástrojů, která kompletně v jazyce Java umožňuje vytvářet RIA aplikace. GWT dohromady spojuje výhody jazyka s výhodami JavaScriptu (samozřejmě i jejich vlastnosti). Poskytuje kompilátor, který zvládá vyprodukovat JavaScript kód, má možnost psaní nativního JavaScript kódu (JSNI) a emulované knihovny JRE. Nad těmito funkcionalitami GWT je poskytováno zdokumentované API.

Úkolem GWT kompilátoru je převést Java kód na JavaScript podobně, jako když se kompiluje Java kód do byte kódu. Kompilátor poskytuje tři módy kompilace, které určují výsledný vzhled výsledného JavaScript kódu.

Obrázek 6 znázorňuje proces kompilace GWT kompilátorem a poukazuje na možné módy kompilace GWT kompilátorem.

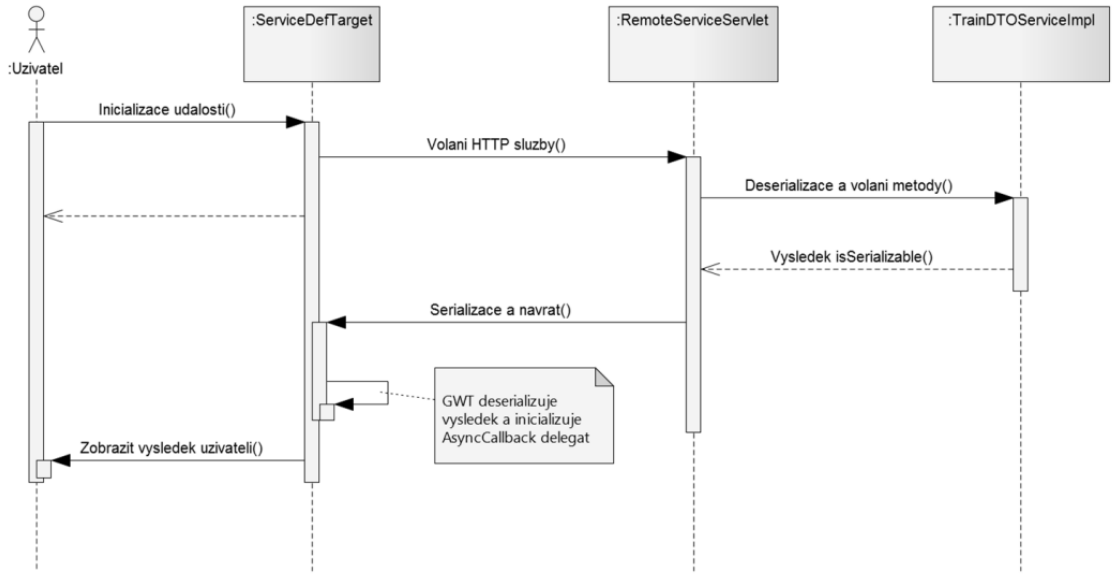


Obrázek 1 – Proces GWT kompilace do jazyka JavaScript a její možné úrovně. Zdroj: (3)

Další důležitou vlastností GWT kompilátoru je to, že jeho zdrojem mohou být pouze Java zdrojové soubory.

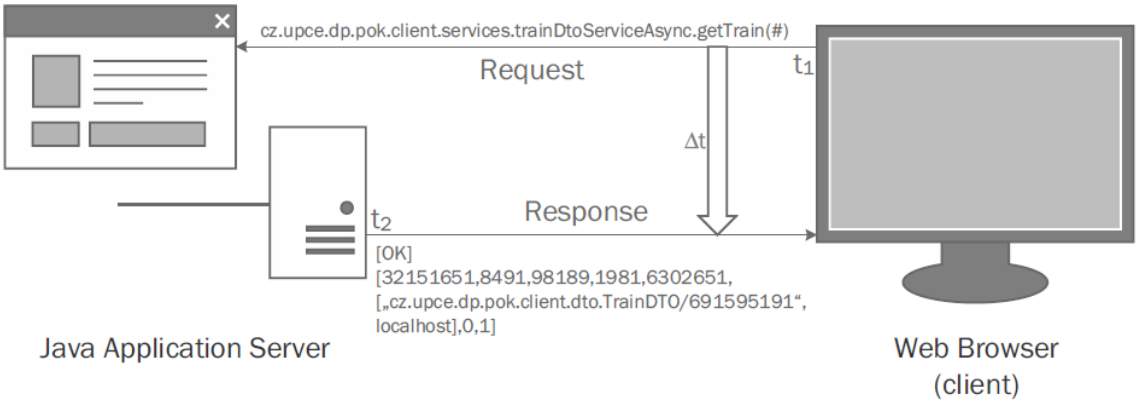
Výstupem kompilátoru je JavaScript kód a výsledkem kompilace je několik samostatných souborů, které jsou pro dosažení maximální kompatibility mezi prohlížeči. Každý tento soubor je optimalizovaný pro daný prohlížeč. Toto nastavení lze specifikovat v konfiguračním XML souboru nastavení kompilátoru. Výhoda této koncepce je, že data, která jsou nutná pro správné fungování aplikace v konkrétním prohlížeči, si každý prohlížeč načte sám.

„Přenos informací ze strany klienta na server a naopak je základním předpokladem většiny webových aplikací a zde tomu pomáhají RPC komunikace v GWT To poskytuje dva nástroje komunikace založené na rozhraní `XMLHttpRequest`*, jednak třídu* `RequestBuilder`*, která v podstatě zapouzdřuje toto rozhraní a dále mechanizmus GWT RPC (Remote Procedure Call). Ten je z praktického hlediska více použitelný, protože dovoluje posílat a přijímat reálné Java objekty mezi klientem a serverem.“* (1)



Obrázek 2 – Sekvenční diagram RPC přenosu dat. Zdroj: (5)

Vzdálené volání metod (probíhající asynchronním způsobem) je důležitým aspektem RPC komunikace. V překladu to znamená, že se zavolá metoda vzdálené služby a na straně klienta mezitím provádí kód bez čekání na návratovou hodnotu. Tedy mezi tím uplyne chvíle času, viz obrázek 3.



Obrázek 3 – Znázornění časového průběhu GWT RPC komunikace. Zdroj: (3 str. 349)

3.2.2 SVG grafika

Tato technologie pomáhá vyřešit jeden z požadavků aplikace, kterým je, aby grafická číst W-POK byla vykreslena vektorovou grafikou. Což poskytuje výhody oproti rastrové grafice. Klíčové vlastnosti SVG (Scalable Vector Graphics) jsou:

- používáno pro definici vektorově založené grafiky na webu,
- je definováno pomocí XML formátu,
- možnost libovolného zvětšení a zmenšení bez ztráty kvality vykreslení,
- s každým objektem (vlaky, koleje) lze pracovat samostatně a vázat na něj určité dynamické akce (zpoždění, změna koleje apod.),
- text v SVG grafice lze vyhledat,
- vektorová grafika je podporována moderními prohlížeči a lze provázat s jinými standardy,
- elementy lze stylovat pomocí kaskádových stylů,
- paměťová náročnost výsledné reprezentace je mnohem menší,
- standard SVG je zaštítěn konsorciem W3C.

Kromě Internet Explorer verze 8 a nižší je SVG technologie plně podporovaná v prohlížečích, a tedy je možné tento kód vložit do stránky, kde nejvhodnějším způsobem se jeví vložit SVG element přímo do HTML kódu. (6)

Bylo nutné najít způsob, jakým lze generovat SVG elementy přímo do HTML stránky dynamicky pomocí programovacího jazyka. Byla k tomuto účelu zvolena knihovna `lib-gwt-svg`, díky které lze vytvářet libovolné SVG elementy pomocí konstrukcí jazyka Java. Knihovna `lib-gwt-svg` je určena pro Framework GWT, který sám o sobě poskytuje pouze grafické komponenty pro rastrové kreslení na HTML Canvas. Výsledný kód je kompilován jako součást klientské části aplikace. Práce s knihovou se využívá třeba pro vytvoření

transformace skupiny, ve které bude umístěna šipka, která má za úkol ukazovat směr příjezdu a odjezdu vlaku. (1)

3.3 Realizace aplikace

Funkční požadavky byly rozčleněny podle logického členění na moduly. Hlavní modul POK obsahuje vykreslený plán obsazení kolejí. Další modul je seznam vlaků. Vlaky může řadit podle různých požadavků. Modul přihlášení zabezpečuje přihlašování uživatelů a modul administrace zabezpečuje správu uživatelů.

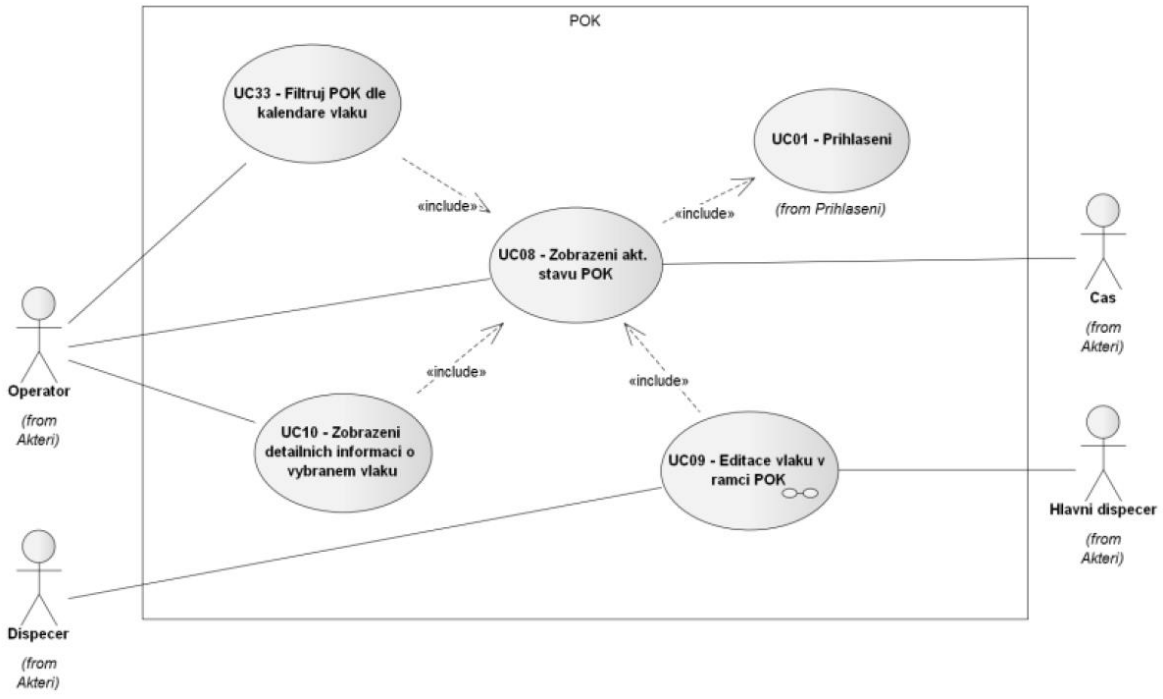
Modul POK zobrazuje aktuální stav, umožňuje rychle měnit koleje, doby pobytu, filtrovat POK dle data. Dokáže upozorňovat na konflikty. Zobrazí detailní informace o vlaku.

Modul „vlaky“ zobrazuje seznam vlaků a u jednotlivých vlaků zobrazí detaily. Umožňuje přiřazovat výluky, zpoždění, vložit nový vlak a to na lokální i centrální úrovni.

Modul „přihlášení“ umožňuje uživateli vstoupit do systému a měnit si heslo.

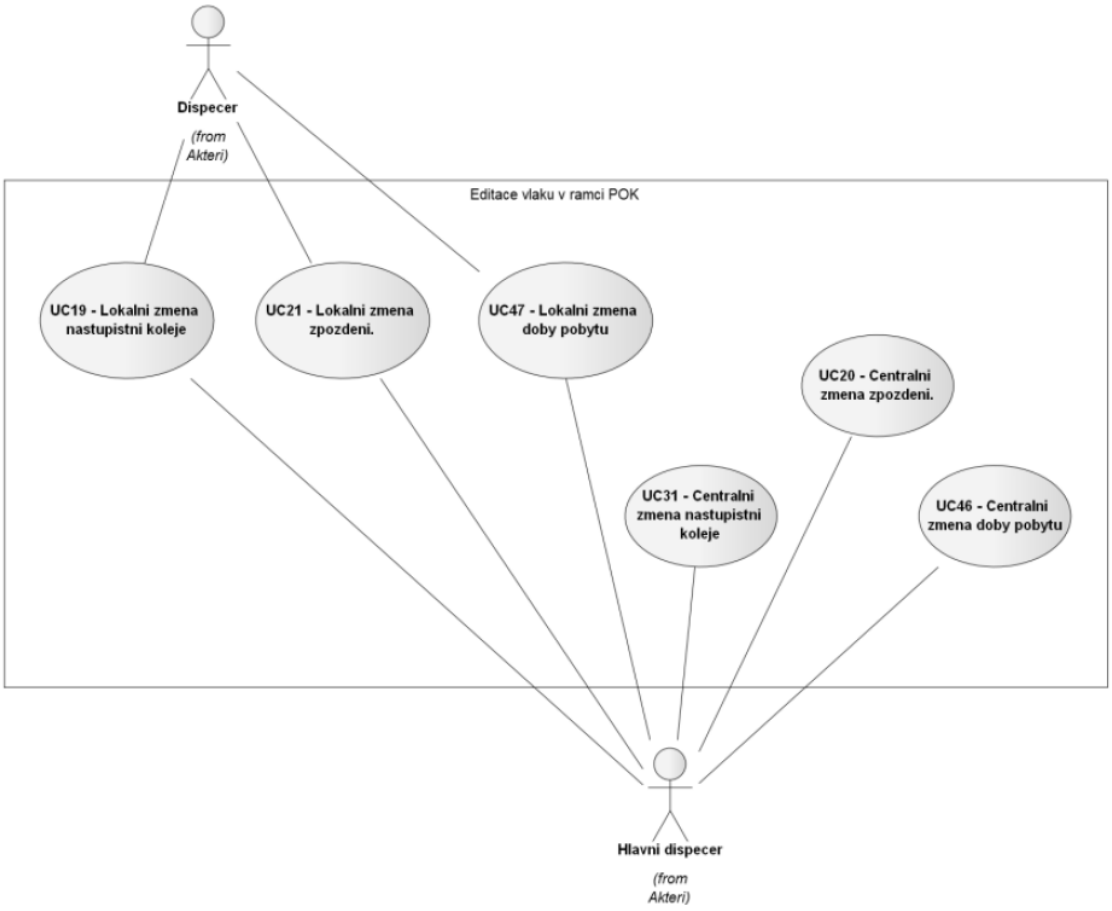
Modul „administrace“ umožňuje správu uživatelů.

Nefunkční požadavky systému zahrnují požadavky na výkon, spolehlivost, škálovatelnost, bezpečnost a použitelnost.



Obrázek 4 – Realizace případu užití pro modul POK. Zdroj: (1)

Po přihlášení do systému se zobrazí aktuální stav POK. Do systému vstupuje operátor, který může filtrovat plán podle různých požadavků a najetím kurzoru nad vybraný vlak se zobrazí všechny informace o tomto vlaku. Dále může do systému vstoupit „dispečer“ a „hlavní dispečer“, kteří mohou editovat s různou úrovní oprávnění. Informace v této kapitole jsou převzaty z (1).



Obrázek 5 – Editace vlaku v rámci POK. Zdroj: (1)

3.3.1 Datový model

Datový model splňuje požadavky na editaci vlaků, přičemž musí být možné se vždy vrátit do původního stavu. Takže struktura musí umožňovat uchovávat všechny změna v datech a zároveň zobrazovat stav po úpravách.

Struktura datového modelu:

Tabulka „Train“ – uchovává všechny důležité atributy, které se vážou ke konkrétnímu vlaku. Pro identifikace vlaku byl zvolen alfanumerický typ, aby bylo možné k číslu vlaku přidat i písmeno, pro lepší specifikaci. Pro evidenci časových údajů byl zvolen alfanumerický typ „number“, který se zobrazuje jako počet sekund od půlnoci. Vše je dále uloženo v databázi.

Tabulka „TrainCalendar“ – je prezentována jako pole bitů, kde je pro den, kdy jede vlak, nastaven bit na hodnotu: „1“. Nevýhodou této tabulky je, že při restrikci je nutné načíst všechny vlaky s jejich kalendáři a filtraci provést až s použitím bitové logiky.

Tabulka „Station“ – uchovává fyzicky dostupné okolí železniční stanice. Tyto stanice udávají výsledný „tvar“ vlaku. Tento „tvar“ je uchován ve formě výčtového typu v samotné třídě. Tabulka je napojena na tabulku „train“ pomocí dvou klíčů: předchozí a následná stanice.

Obdobným způsoben jsou utvořeny i další tabulky:

- „Rails“ – seznam dostupných kolejí v dané železniční stanici,
- „LocoType“ – typy hnacích vozidel,
- „TrainType“ – typy vlaků,
- „TrainName“ – názvy vlaků,
- „Delay“ – údaje o zpoždění.

Tabulka „PokChanges“ – slouží pro uchování provedených změn v rámci POK. Obsahuje primární klíč a atributy, které jsou tvořeny cizími klíči. Jsou to vazby na tabulku „train“ a další tabulky. Je zde uchována vazba na původní needitovaný vlak, na nový editovaný vlak, evidence typu změny, měřítko změny (jestli je změna lokální, nebo globální pro všechny uživatele) a uživatele, jenž změnu provedl. Tak je možné, aby určitý uživatel získal pro něho aktuální stav POK.

Tabulky „ChangeType“ a „ChangeScope“ uchovávají informace o provedených editací vlaků.

Tabulka „Users“ – uchovává informace o uživatelích.

3.3.2 Import dat

Je nutné uzpůsobit importovaná data a grafickou část aplikace pro konkrétní železniční stanici.

Struktura importovaných dat:

- druh – označuje typ vlaku,
- číslo – označení vlaku,
- kolej – pobytová kolej,
- příjezd – příjezd do železniční stanice,
- pobyt – délka pobytu na koleji,
- odjezd – odjez ze železniční stanice,
- poznámka,
- název – název vlaku,
- předDopr – předchozí železniční stanice (směr šipky),
- náslDopr – následující železniční stanice (směr šipky),
- délka – délka vlaku (nepovinný údaj),
- hmotnost – hmotnost vlaku (nepovinný údaj),

- řada – řada hnacího vozidla (nepovinný údaj).

3.3.3 Uživatelské role

Při vytváření uživatele je mu vytvořena role, podle čísla v intervalu 1-15, kde zvolené číslo vyjádřené ve dvojkové soustavě dá dohromady všechna práva, která mu k příslušné roli náleží. Oprávnění jednotlivých rolí vystihuje tabulka. V tabulce X je L a G je bráno jako úroveň přístupu (lokální a globální).

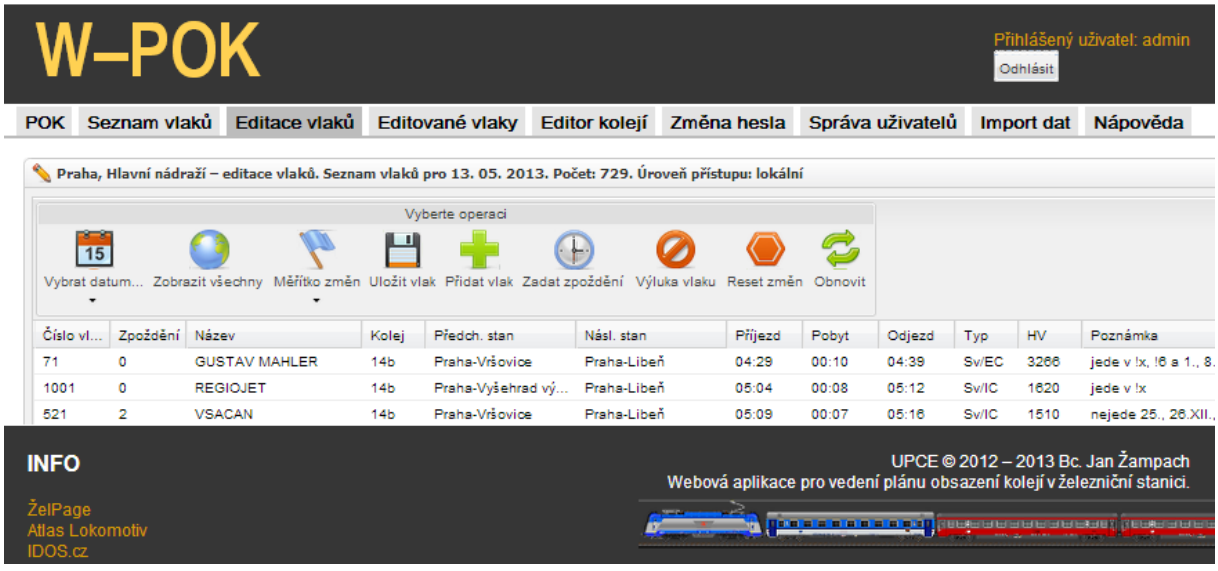
Tabulka 1 – Oprávnění uživatelských rolí. Zdroj: (1)

Číselný kód	Název role	POK		Seznam vlaků	Editace vlaků		Editované vlaky	Editor kolejí	Změna hesla	Správa uživatelů	Import dat
Přístup		L	G	–	L	G	–	G	–	–	–
1	Administrátor	✗		✗	✗		✗	✗	✓	✓	✗
2	Hl. dispečer	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓
4	Dispečer	✓	✗	✓	✓	✗	✓	✗	✓	✗	✗
8	Operátor	✓		✓	✗		✗	✗	✓	✗	✗

3.3.4 Návrh grafického rozhraní

Vychází z požadavků na aplikaci. Byly využity kaskádové styly CSS (Cascading Style Sheets) a rozvržení pomocí HTML elementů. Design aplikace byl zvolen s ohledem na jednoduchost, přehlednost a použitelnost.

Celkový náhled na aplikaci v okně zobrazuje obrázek č.:6.



Obrázek 6 – Celkový náhled na aplikaci v okně prohlížeče. Zdroj: (1)

V horní části je název aplikace, informace o přihlášeném uživateli a tlačítko pro odhlášení uživatele. Následuje seznam dostupných záložek, mezi kterými lze přepínat. Uživateli se zobrazí ty záložky, které jsou mu přiřazeny podle jeho role. Šablona je uzavřena informacemi o autorovi aplikace.

4 Nalezené chyby a návrhy na jejich opravu

Testování proběhlo vzhledem k malému množství času, které bylo pro testování vytyčeno, hlavně nad kalendářovou strukturou aplikace, na kterou bylo upozorněno od vedoucího práce.

Omezení tedy proběhlo v první řadě na třídy, které se nemusely připojovat žádným způsobem k serveru, což v první řadě znamenalo, vyhledat třídy, které nevyužívají třídy Session.

4.1 Konstruktor ve třídě cz.upce.dp.pok.client.dto.TrainDTO

Konstruktor třídy TrainDTO(int id, String rails, String prevstation, String nextstation, String traintype, String locotype, String trainname, String trainnumber, String note, Time arrival, Time staytime, Time departure, int tweight, int tlength) vytváří vlak, který má id zapsané v jiné proměnné, než se kterou se pracuje. Proměnná, se kterou se pracuje má název id a proměnná, do které se píše, se nazývá trainid.

Tato chyba má pak za následek další hlášené chyby. Pokud u vlaku nedokážeme přečíst správně jeho id, nemůžeme pak tento vlak porovnávat s dalšími vlaky, nelze potom odebírat vlaky, získat jejich id. Proto vznikají další chyby způsobené touto chybou:

ve třídě cz.upce.dp.pok.client.dto.MultiDTO vznikly chyby pro metody:

- remove(),
- update(),
- getMap(),
- getTrains(),

ve tříde cz.upce.dp.pok.client.dto.TrainDTO vznikly chyby pro metody:

- getId(),
- equals().

Většina těchto metod porovnává jednotlivé vlaky pomocí implementované metody equals a jelikož i tato metoda hlásí chybu, skončí metody chybně. Jelikož tato jedna chyba opraví o tyto další, nebudu je zde dále rozebírat.

Návrh řešení: změnit název proměnné z “trainid“ na “id“ v konstruktoru TrainDTO(int id, String rails, String prevstation, String nextstation, String traintype, String locotype, String trainname, String trainnumber, String note, Time arrival, Time staytime, Time departure, int tweight, int tlength) ve třídě cz.upce.dp.pok.client.dto.TrainDTO. Je to nejlepší a nejjednodušší řešení vzniklé chyby.

4.2 Konstruktor ve tříde cz.upce.dp.pok.client.dto.ServerException

Při vytvoření třídy jakýmkoli konstruktorem se jediná privátní proměnná error neinicializuje. Jelikož zde třída nemá ani setter pro tuto proměnou, tak nelze žádným

způsobem tuto proměnnou zadat. Tato chyba se dá považovat za nedůležitou spíše jen za informativní chybu.

Návrh řešení: jelikož se proměnná v programu nepoužívá, lze tuto chybu ignorovat, nebo v lepším případě proměnnou vymazat a metodu tedy také, protože metoda dědí ze třídy Exception a využívá její metody a proměnné.

4.3 Omezení a chyby třídy cz.upce.dp.pok.server.util.calendar.BytesCalendarTool

Dny v roce se v této třídě se zapisují pomocí pole bytů. Konstruktor BytesCalendarTool (byte[] pole); vezme pole, u kterého pozmění první prvek na hodnotu aktuálního roku, kde rok reprezentuje číslo, od kterého se odečte báze (zde je báze nastavena na rok 2128, tedy 2000+218, kde -218 je minimální možná hodnota datového typu byte). Rok 2014 by zde byl zapsán jako hodnota -114.

Tedy tato třída funguje jen pro roky 2000-2255. Jiné roky tato třída nahrazuje příslušnými roky z tohoto intervalu, kde rok 1999 je nahrazen rokem 2255, dále pak rok 1998 rokem 2254, 2256 rokem 2000, atd.

4.3.1 Chyba metody setDaysInYear()

Metoda setDaysInYear(int daysInWeek, boolean state) podle javadoc autora má dělat, že: „Nastaví pro daný den všechny dny v roce (např. všechny středy) na hodnotu state.“ Tato metoda volá metodu setDayInMonth(int month, int day, boolean state), která nastavuje konkrétní den podle čísla (tedy pokud zadáme 2, nastaví pro každého druhého každý měsíc na logickou 1 nebo 0 podle hodnoty state).

Návrh řešení: volat metodu setDaysInMonth(int month, int daysInWeek, boolean state) místo metody setDayInMonth(int month, int day, boolean state). Nebo přepsat javadoc podle metody, jak doopravdy funguje.

4.3.2 Chyba metody setMoFriDaysInMonth()

Metoda setMoFriDaysInMonth(int month, boolean state) podle javadoc autora: „Nastaví všechny Po-Pa dny v měsíci. Nebere ohled, zda je v daném dnu svátek.“ Bohužel metoda nastavuje dny špatně, jelikož metoda CzechWorkCalendar.getWorkDays() bere pracovní dny jako den 2., 3., 4., 5. a 6. nýbrž metoda setDaysInMonth(int month, int daysInWeek, boolean state), se kterou se dny nastavují, bere pracovní dny jako 0., 1., 2., 3. a 4.

Návrh řešení: změnit nastavení jednotlivých pracovních dnů ne podle metody CzechWorkCalendar.getWorkDays(), ale doopravdy jako výčet jednotlivých dnů branný podle metody convertIndexToDay(int index) v této třídě.

4.3.3 Chyba metody setIntervalDays()

Metoda setIntervalDays(int fromMonth, int fromDay, int toMonth, int toDay, boolean state) nemá javadoc, ale podle názvu metody a kódu se dá odvodit, že se autor snažil nastavit jednotlivé dny v intervalu dní v měsících, které jsou v konkrétním intervalu měsíců. Autor ale zapomněl po průchodu vnitřním cyklem znovu inicializovat proměnnou vnitřního cyklu, tedy pro ostatní průběhy vnějšího cyklu vnitřní cyklus neproběhne. Z toho

plyne, že se vždy nastaví jen konkrétní dny v prvním měsíci a v dalších měsících už se nenastaví.

Návrh řešení: pro vnitřní cyklus for použít jinou nově vytvořenou proměnnou, nebo si proměnou fromDay zálohovat do jiné proměnné a po průchodu vnitřním cyklem proměnou fromDay ze zálohy znovu nastavit.

Druhá chyba metody:

Jelikož se tato metoda používá ve třídě cz.upce.dp.pok.server.util.calendar.TimeInterval v metodě getInterval(BytesCalendarTool calTool, boolean state), která také nemá javadoc, tedy nelze přesně říci, co má metoda dělat, ale zde se dá z kódu odvodit, že má metoda nastavit do calTool konkrétní dny v intervalu, který nám nastaví metody setFrom a setTill, což nekoresponduje s metodou naprogramovanou metodou setIntervalDays(int fromMonth, int fromDay, int toMonth, int toDay, boolean state). Ta má další problém v tom, že není schopna nastavovat data mezi jednotlivými měsíci např.: nelze nastavit dny mezi 27.2. a 2.3., protože metoda je naprogramovaná, aby den první byl menší hodnoty, než den druhý, protože nelze logicky projít skrze cyklus for(int i = 27; i <= 2; i++){}.

Návrh řešení: napsat javadoc, aby bylo jasné, co metody mají dělat, nebo ho přizpůsobit tomu, co doopravdy dělají.

Návrh řešení další: předělat metodu tak, aby volala metodu setIntervalDays(int from, int to, boolean state), pro kterou se tedy musí zjistit konkrétní čísla konkrétních dní a potom lze nastavit jednotlivé dny po sobě jdoucí.

4.4 Chyby ve třídě cz.upce.dp.pok.server.util.calendar.TimeInterval

4.4.1 Chyba metody getInterval()

Metoda getInterval(BytesCalendarTool calTool, String[] tokens, boolean state) má javadoc spíše formou poznámky, ze které je pochopitelné, co by metoda měla dělat, ale ve výsledku testování dělá jen polovinu z toho. S metodou lze nastavit jednotlivé pracovní dny, prázdninové dny včetně nedělí, konkrétní dny v týdnu v intervalu pondělí až neděle, nebo i jen jeden den v týdnu pomocí vylepšení, při kterém se do parametru tokens přidá pole o dvou prvcích, kde oba dva prvky mají stejný parametr

(např.:

```
new String[]{"6","6"}
```

)

Dále by měla ale metoda umět ještě nastavit i inverzi těchto údajů, tedy všechny nepracovní dny, atd. Bohužel to metoda neumí a ve výsledku to ani umět nemusí, ale jelikož zde není pořádně napsán javadoc, není jasné, co má tedy metoda dělat.

Návrh řešení: přepsat a připsat do javadoc, že lze vytvářet jen pracovní dny, prázdniny včetně nedělí, konkrétní dny v týdnu, třeba všechny středy až čtvrtky. Dále pak odstranit zbytečné řádky metody, tedy i jednotlivé metody ze třídy cz.upce.dp.pok.server.util.calendar.BytesCalendarTool, které kvůli funkci inverze zadání byly vytvořeny (tedy zbytečně vytvořeny a testovány):

- setAllNonWorkDays(),
- setAllNonHolidays(),
- doAND(),
- doOR().

Návrh řešení další: naprogramovat metodu podle javadoc tedy pro začátek možnost nastavit proměnnou mask, pro přijímání zadávání inverzní formy údajů a přizpůsobit metodu, aby podle toho fungovala.

Závěr

Tato bakalářská práce se zabývá funkčním testováním aplikace, tedy nikoliv uživatelským testováním. Zvolená velikost testovaných metod byla omezena z hlediska termínu odevzdání testů a testy se tedy prováděly nad dohodnutými třídami.

Hlavním úkolem tedy vzápětí vzniklo otestovat převážně kalendářovou strukturu a tedy se muselo omezit množství testů prováděného nad aplikací. Tato volba byla učiněna zejména z toho důvodu, že některé metody potřebovaly aktivní připojení k serveru, který nechtěl komunikovat při spuštěném debuggeru.

Je tedy ještě potřeba otestovat jinými způsoby třídy, které se aktivně připojují k serveru.

V rámci bakalářské práce se podařilo objevit mnoho chyb, které dříve či později by se projevily v aplikaci vzhledem k datu. Tyto chyby by chtělo co nejdříve opravit a opravit s nimi i třídy, které třídy s chybami využívají.

Nejvíce kritickou nalezenou chybou byla záměna jednotlivých dní v týdnu v metodě setMoFriDaysInMonth() ve třídě BytesCalendarTool. Jelikož autor chybu neobjevil, s metodou potom v projektu dále nepočítal, což je vidět při zobrazení využité této metody, že se používá doopravdy jen ve vytvořeném testu.

Další možnosti testování aplikace spočívají v dodělání testových případů pro třídy, kde se aktivně přistupuje k serveru (tedy hlavně k zápisu, nebo čtení z databáze serveru), potom to chce aplikaci otestovat ne funkčně, ale uživatelsky (z pohledu uživatele, který bude aplikaci používat), jednotlivé stavy aplikace a nakonec ještě vše otestovat jestli aplikace uspěje při zátěžových testech při připojení více uživatelů, jak se jim budou zobrazovat stejně data a jaká bude prodleva mezi jednotlivými kroky jednotlivých uživatelů (za které bude moct nejen připojení k serveru, ale i kód aplikace).

Literatura

1. ŽAMPACH, Jan. *Webová aplikace pro vedení plánu obsazení kolejí v železniční stanici*. Pardubice, 2013. Diplomová práce. Univerzita Pardubice, Fakulta elektrotechniky a informatiky, Katedra softwarových technologií. Vedoucí práce Ing. Michael Bažant, Ph.D.

2. PATTON, Ron. *Testování softwaru*. Vyd. 1. Praha: Computer Press, 2002, 313 s. ISBN 80-722-6636-5.

3. HANSON, Robert a Adam TACY. *GWT in action: easy Ajax with the Google Web toolkit*. 2007 [cit. 2014-02-27]. ISBN: 19-339-8823-1.

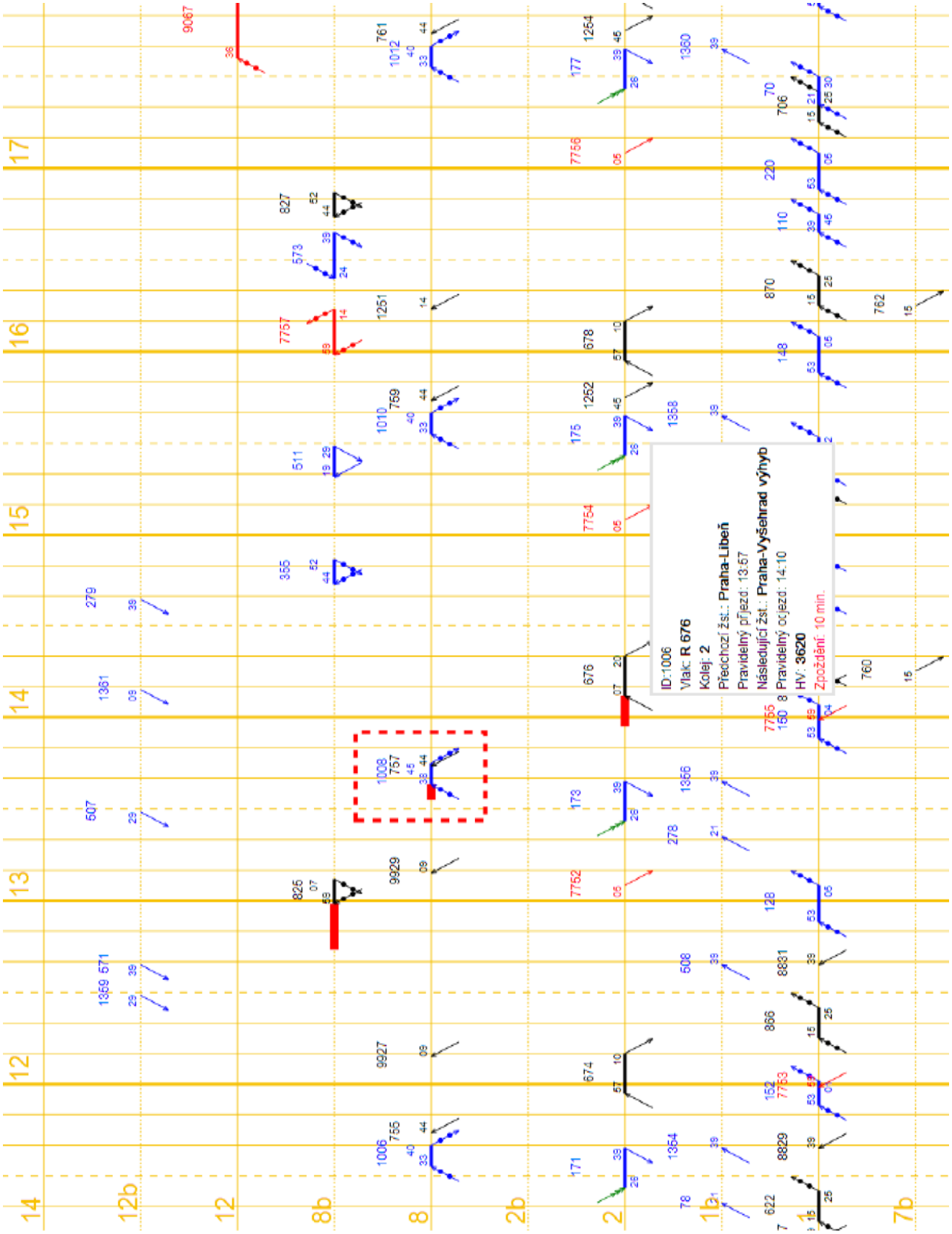
4. *Junit.sourceforge.net* [online]. 2014 [cit. 2014-03-28]. Dostupné z: http://docs.oracle.com/javase/7/docs/api/.

5. RASOLO, Franck. Calling the server from the client. *We like GWT presentation* [online]. c2010, poslední revize 4.2.2010 [cit. 2014-02-28]. Dostupné z: https://code.google.com/p/welikegwt-presentation/wiki/GwtRpc.

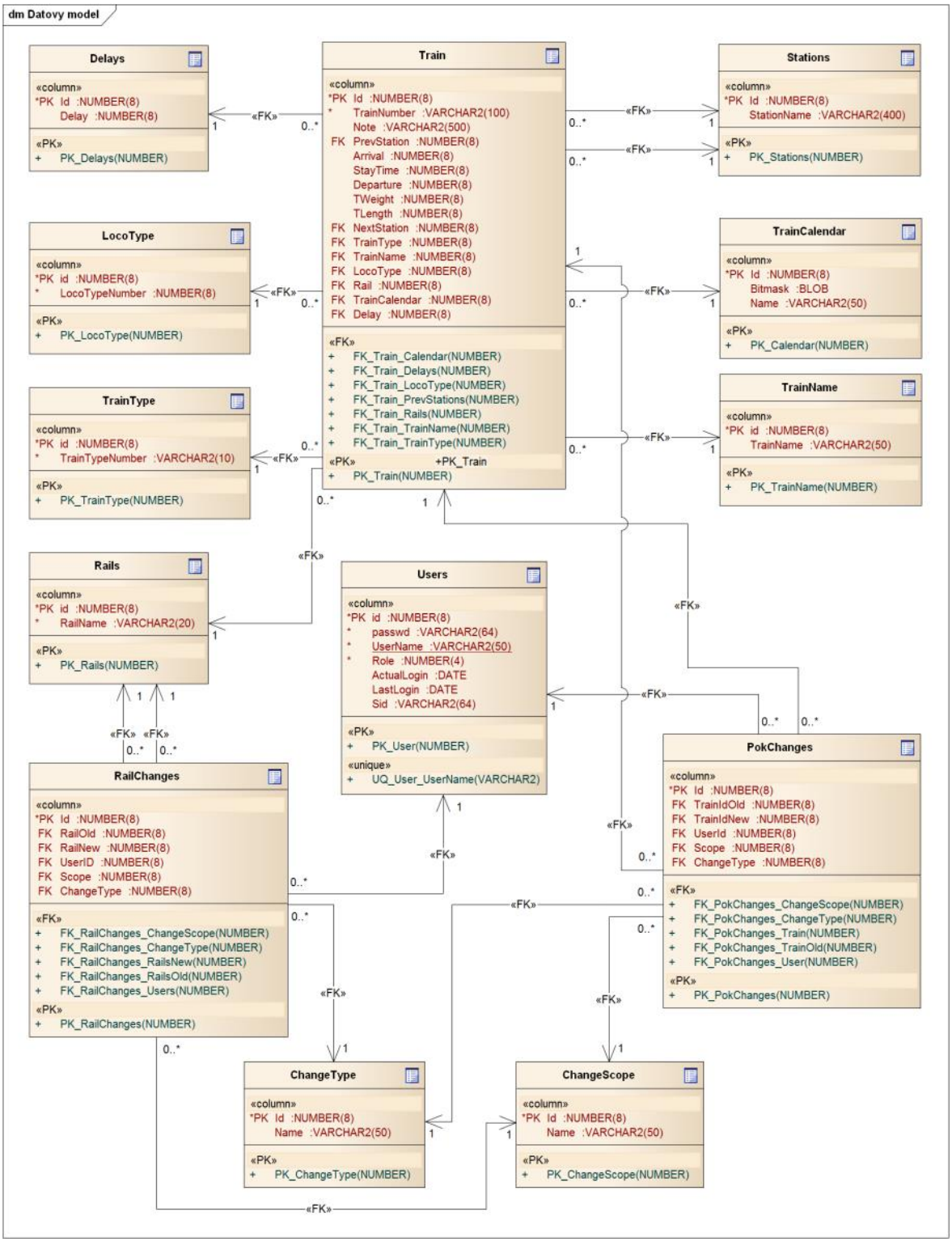
6. W3C. *Options for using SVG in Web pages* [online]. W3C, 16. 8. 2011 [cit. 2014-03-29]. Dostupné z: http://www.w3schools.com/svg/svg_inhtml.asp.

7. Mkyong.com. *JUnit 4 Vs TestNG – Comparison* [online]. 2009-2013 [cit. 2014-04-10]. Dostupné z: http://www.mkyong.com/unittest/junit-4-vs-testng-comparison/.

Příloha A – Fragment vytvořeného plánu obsazení kolejí



Příloha B – UML datového modelu



Příloha C – Zdrojový kód testové metody doOR

```
/**
* Test of doOR method, of class BytesCalendarTool.
* Test pomocí Boolovského zákonu A || !A = 1
*/
@Test
public void testDoOR() {
      System.out.println("doOR");
      BytesCalendarTool instance_pomocna = new BytesCalendarTool();
      //A
      instance_pomocna.setAllWorkDays(true);
      BytesCalendarTool secondCalendar = instance_pomocna;
      BytesCalendarTool instance = new BytesCalendarTool();
      //!A
      instance.setAllNonWorkDays(true);
      BytesCalendarTool expResult = new BytesCalendarTool();
      //1
      expResult.setAllDays(true);
      //A || !A
      BytesCalendarTool result = instance.doOR(secondCalendar);
      //prověřuje se, jestli je každý den stejný, tedy pro oba dva
      //případy je logická 1
      for (int i = 0; i < 47 * 8 - 2; i++) {
            assertEquals(expResult.getDay(i), result.getDay(i));
      }
}
```

Příloha D – Příklad testu pomocí junit.framework.Assert

```
class Trida {

	int promenna1;

	public Trida(int promenna) {
		promenna1 = promenna;
	}

	@Override
	public boolean equals(Object obj) {
		return promenna1 == ((Trida)obj).promenna1;
	}
}

@Test
public void testNazevTestovaneMetody() {
	Trida stejna1 = new Trida(1);
	Trida stejna2 = new Trida(1);
	Trida stejna3 = stejny1;
	Trida jina1 = new Trida(2);

	assertNotSame(stejny1, jiny1);
	assertSame(stejny1, stejny1);
	assertSame(stejny1, stejny3);
	assertEquals(stejny1, stejny2);
}
```