TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií

Principy vytváření robustního softwaru v týmu programátorů

Diplomová práce

Studijní program: N2612 – Elektrotechnika a informatika
Studijní obor: 1802T007 – Informační technologie

Autor práce: **Bc. Tomáš Lád**
Vedoucí práce: Ing. Roman Špánek, Ph.D.

Liberec 2014

TECHNICAL UNIVERSITY OF LIBEREC
Faculty of Mechatronics, Informatics
and Interdisciplinary Studies

Principles for creating robust software in a team of programmers

Diploma thesis

Study programme: N2612 – Electrotechnology and informatics
Study branch: 1802T007 – Information technology

Author: **Bc. Tomáš Lád**
Supervisor: Ing. Roman Špánek, Ph.D.

Liberec 2014

TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky a mezioborových studií
Akademický rok: **2013/2014**

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Tomáš Lád**
Osobní číslo:	**M11000239**
Studijní program:	**N2612 Elektrotechnika a informatika**
Studijní obor:	**Informační technologie**
Název tématu:	**Principy vytváření robustního softwaru v týmu programátorů**
Zadávající katedra:	**Ústav mechatroniky a technické informatiky**

Zásady pro vypracování:

1. Seznamte se s principy návrhu softwaru, na který jsou kladeny velké nároky na robustnost.
2. Seznamte se s oblastí softwarového inženýrství, především pak s tvorbou softwaru v rámci skupiny programátorů, jak řešit jejich kooperaci, rozdělení problému na moduly.
3. Demonstrujte zjištěné postupy na pilotní implementaci software.
4. Zhodnoťte využitelnost jednotlivých postupu a případně navrhněte možné modifikace.

Rozsah grafických prací: **dle potřeby dokumentace**

Rozsah pracovní zprávy: **40–50 stran**

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

Seznam odborné literatury:

[1] **The Unified Modeling Language User Guide. Grandy Booch, James Rumbaugh, Ivar Jacobson, Addison-Wesley, ISBN 0-321-26797-4**

[2] **Design Patterns: Elements of Reusable Object-Oriented Software. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, ISBN-10: 0201633612, ISBN-13: 978-0201633610**

[3] **The Mythical Man-Month: Essays on Software Engineering. Frederick P. Brooks Jr., ISBN 0201835959 (ISBN13: 9780201835953)**

[4] **Test Driven Development: By Example. Kent Beck, ISBN 0321146530 (ISBN13: 9780321146533)**

[5] **Řízení softwarových projektů, Božena Mannová, Karel Vosátka, ISBN 8001032973, 788001032978**

Vedoucí diplomové práce: **Ing. Roman Špánek, Ph.D.**
Ústav mechatroniky a technické informatiky

Konzultant diplomové práce: **Ing. Pavel Tyl**
Ústav mechatroniky a technické informatiky

Datum zadání diplomové práce: **10. října 2013**

Termín odevzdání diplomové práce: **16. května 2014**

prof. Ing. Václav Kopecký, CSc.
děkan

L.S.

doc. Ing. Milan Kolář, CSc.
vedoucí ústavu

V Liberci dne 10. října 2013

Prohlášení

Byl jsem seznámen s tím, že na mou diplomovou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci (TUL) nezasahuje do mých autorských práv užitím mé diplomové práce pro vnitřní potřebu TUL.

Užiji-li diplomovou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti TUL; v tomto případě má TUL právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Diplomovou práci jsem vypracoval samostatně s použitím uvedené literatury a na základě konzultací s vedoucím mé diplomové práce a konzultantem.

Současně čestně prohlašuji, že tištěná verze práce se shoduje s elektronickou verzí, vloženou do IS STAG.

Datum:

Podpis:

Abstrakt

Tato diplomová práce pojednává o principech týmového vývoje softwaru, u kterého jsou kladené velké nároky na stabilitu a rozšiřitelnost. V práci jsou popsány metody řízení projektů včetně jejich vzájemného porovnání, nástroje podporující týmovou spolupráci a standardní postupy při návrhu, implementaci a testování softwaru. Popsané postupy jsou pak využity při realizaci referenčního projektu v podobě databázového informačního systému.

Klíčová slova

Softwarové inženýrství, Objektově orientované programování, Testování softwaru, Systémy pro správu verzí, Maven

Abstract

This thesis deals with the principles of team software development with high requirements on the stability and scalability of the final software product. The thesis describes methods of project management and discusses their advantages and disadvantages. Moreover, tools for team cooperation and widely used techniques for design, implementation, and testing are also overviewed. The described procedures are used for a pilot implementation of a database information system.

Key words

Software Engineering, Object oriented programming, Software testing, Version control systems, Maven

Poděkování

Rád bych poděkoval vedoucímu mé práce Ing. Romanu Špánkovi, Ph.D. za jeho rady a připomínky k práci. Dále chci poděkovat za spolupráci všem kolegům, kteří mi předali své zkušenosti, názory a poznatky z praxe.

Obsah

Seznam obrázků

Seznam tabulek

Seznam zkratek

IS	Informační Systém
API	Application Programming Interface
GUI	Graphical User Interface
XML	eXtensible Markup Language
UML	Unified Modeling Language
DAO	Data Access Object
JNLP	Java Network Launching Protocol
DSL	Domain Specific Language
DB	DataBase
SQL	Structured Query Language
ORM	Object Relational Mapping
DAO	Data Access Object
PS	Port Service (systém jenž je předmětem této práce)
BHT	Bremer HafenTelematik (systém přístavu Bremerhaven)
SIS	Bremer Schiffsmeldedienst
FT	File Transfer (označení pro výměnu souborů mezi PS a BHT)
JDK	Java Development Kit
IDE	Integrated Development Environment
SVN	SubVersioN

Kapitola 1

Úvod

Vývoj softwarového řešení je dlouhodobý a náročný proces a týmová kooperace je při vývoji nezbytná. Není časově reálné obsáhnout všechny úkony v jedné osobě, tj. být analytikem, programátorem, obchodníkem atd.

Cílem této práce je uvést čtenáře do problematiky týmového vývoje softwaru. Téma bude pojato spíše z technického úhlu pohledu na úkor sociálních nebo tzv. soft skills. V práci budou představeny metody řízení projektů a nezbytné nástroje pro jednodušší kooperaci v týmu. Paralelně se studiem teoretických informací bude vyvíjen informační systém zajišťující elektronické zpracování transakcí mezi klientem a přístavem Bremerhaven.

Za implementační jazyk referenčního projektu je zvolen objektově orientovaný jazyk Java, neboť platforma Java je hojně využívanou značkou v oblasti vývoje kvalitních a konkurenceschopných aplikací.

Kapitola 2

Robustní software

Software považujeme za robustní, pokud se rozumným způsobem vyrovná s nepřípustnými stavy a dalšími neočekávanými situacemi. Pokud vykazuje sám o sobě odolnost vůči běžným a nekritickým chybám.

Pokud software označíme za korektní, máme na mysli, že správně vykonává jen a pouze úkol, ke kterému byl navržen a pro všechny ostatní případy vyhodí např. výjimku. Korektní přístup razí heslo, že žádný výsledek je lepší než špatný výsledek.

Jakkoliv si tyto dva principy jdou svojí myšlenkou proti sobě, oba hrají velmi důležitou roli při vývoji softwaru. Každý rozsáhlejší softwarový systém je možné separovat na menší části, které jsou čistě robustního nebo korektního charakteru.

Problém	**Robustní řešení**	**Korektní řešení**
Nesprávným znakem zakomentované řádky v konfiguračním souboru	Pokusit se rozpoznat nejčastěji používané znaky pro komentář a takové řádky ignorovat	Ukončit při čtení konfiguračního souboru a nahlásit problém
Uživatel, který zadá datum v podivném formátu	Pokusit se rozpoznat řetězec a datum od uživatele převzít	Nahlásit neplatné datum
Přehrát video se špatnými snímky	Přeskočit vadné snímky a pokračovat v přehrávání	Zastavit přehrávání a nahlásit poškozené video
Přidané mezery na konci HTTP hlaviček	Odebrat mezery a zpracovat požadavek	Vrátit chybové hlášení HTTP 400 Bad Request

Tabulka 2.1: Příklady robustního a korektního řešení problému

Z výše uvedených příkladů by se otázka robustnosti a korektnosti dala shrnout do následujících dvou bodů.

- **Robustnost** - Externí rozhraní (GUI, vstupní soubory, konfigurace, API apod.) existují primárně, aby sloužily uživatelům. Musí se dělat co nejvstřícnější, s očekáváním, že se na vstupu budou zadávat nesmyslné údaje.
- **Korektnost** - Interní model projektu by měl být tak jednoduchý, jak to je jen možné, a být vždy ve stoprocentně validním stavu. Vyhodit výjimku, informovat uživatele, ukončit aplikaci vždy, když se narazí na něco, co není v pořádku.

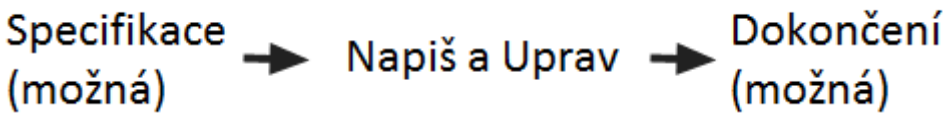
Je vhodné vkládat dělící vrstvu mezi interní model a externí rozhraní. Dělící vrstva bude opravovat „divné“ hodnoty a udrží interní model v konzistentním stavu. Našim cílem by mělo být, aby externí rozhraní byla plně robustní a interní model stoprocentně korektní. Naši uživatelé i kolegové budou mít příjemnější život.

Kapitola vychází z knihy *Introduction to Programming Using Java* [3], kde jsou vysvětleny pojmy robustnost a korektnost i s praktickými doporučeními zaměřenými na jazyk Java. Dále kapitola bere inspiraci z článku *Software Product Qualities* [19], který je zaměřen spíše teoreticky na problematiku spolehlivosti softwaru, kam pojmy robustnosti a korektnosti také patří.

Kapitola 3

Principy vývoje softwaru

V počátcích softwarového vývoje se používal triviální model tzv. napiš a oprav. Model vznikl zcela přirozeně a své jméno dostal až s odstupem času. Jeho princip spočívá v naprogramování aplikace, jejím odevzdáním zákazníkovi a následném opravováním chyb. Brzy začalo být všem zřejmé, že udržet vývoj softwaru tímto způsobem není možné. Takto jednoduchý model totiž postrádá fáze analýzy a návrhu a rovnou se pouští do implementace požadavků.



Obrázek 3.1: Napiš a uprav model

3.1 Tradiční modely

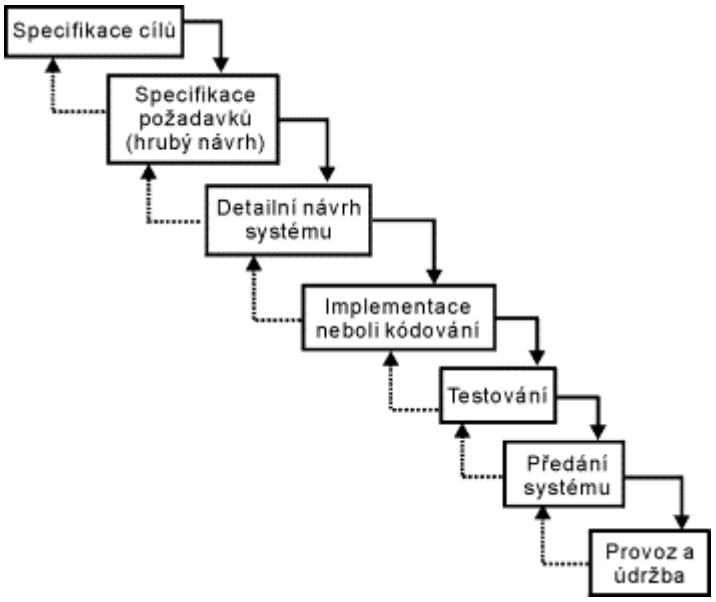
3.1.1 Vodopádový model

Vodopádový model [6] je historicky jedním s nejstarších a byl představen roku 1956 B. W. Boehmem. Model spočívá v rozložení procesu výroby do jednotlivých, pevně vymezených etap. Začíná analýzou požadavků. Dle analýzy se systém naimplementuje. Výsledný produkt se otestuje a nasadí k zákazníkovi. Cílem jeho vzniku bylo zavést do vývoje systémů jednotný řád, umožnění řešení komplexnějších problémů díky dekompozici a snížení množství chyb precizní kontrolou všech výstupů jednotlivých etap. Obrázek 3.2 vyjadřuje návaznost jednotlivých fází modelu vodopád.

Nevýhody vodopádového modelu

- Prodleva mezi zadáním a výsledným produktem. Během vývoje zákazník nic nevidí, a ani my nedokážeme odhalit výslednou kvalitu produktu.
- Na začátku je nutné vytvořit přesné zadání, aby byl výsledný produkt v pořádku.

Z těchto důvodů byl vodopádový model dále modifikován a vznikly nové modely jako např. Prototypový model nebo Spirálový model. Model vodopád lze chápat jako univerzální model, který má své nevýhody, ale je podstatně lepší než náhodný, metodicky neřízený přístup k řešení systému.



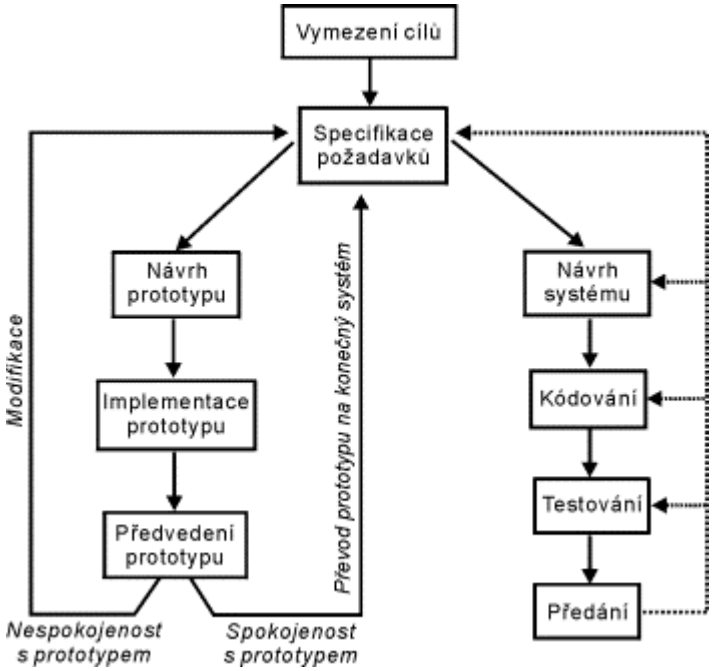
Obrázek 3.2: Vodopádový model [6]

3.1.2 Prototypový model

Prototypový model [6], nebo též model iterativního cyklu s přírůstky. Základní charakteristikou prototypového modelu je předpoklad změn výchozích požadavků zákazníků a umožnění reakce na tyto změny, čímž se liší od modelu vodopád. Tento model se začal prosazovat v 80. letech. Jeho hlavním cílem je urychlení vývoje IS využitím prototypů a seznámení zákazníka s prvními verzemi systému v co nejkratší době. Prototyp můžeme chápat jako zjednodušenou implementaci celého systému nebo jako plnou implementaci části systému. Tato implementace je provedena v co nejkratším čase a v takové funkčnosti, která prezentuje veškerá vnější rozhranní a umožňuje zákazníkovi reagovat na výsledky. Na základě připomínek zákazníků jsou upřesňovány požadavky a modifikován prototyp do té doby, dokud zákazník není spokojen. Poté následuje samotný návrh a implementace celého systému.

Výhody prototypového modelu

- Již v časných fázích vývoje je věnována pozornost použití SW.
- Chyby a nevyhovující postupy jsou odhaleny co nejdříve.



Obrázek 3.3: Prototypový model [6]

3.1.3 Spirálový model

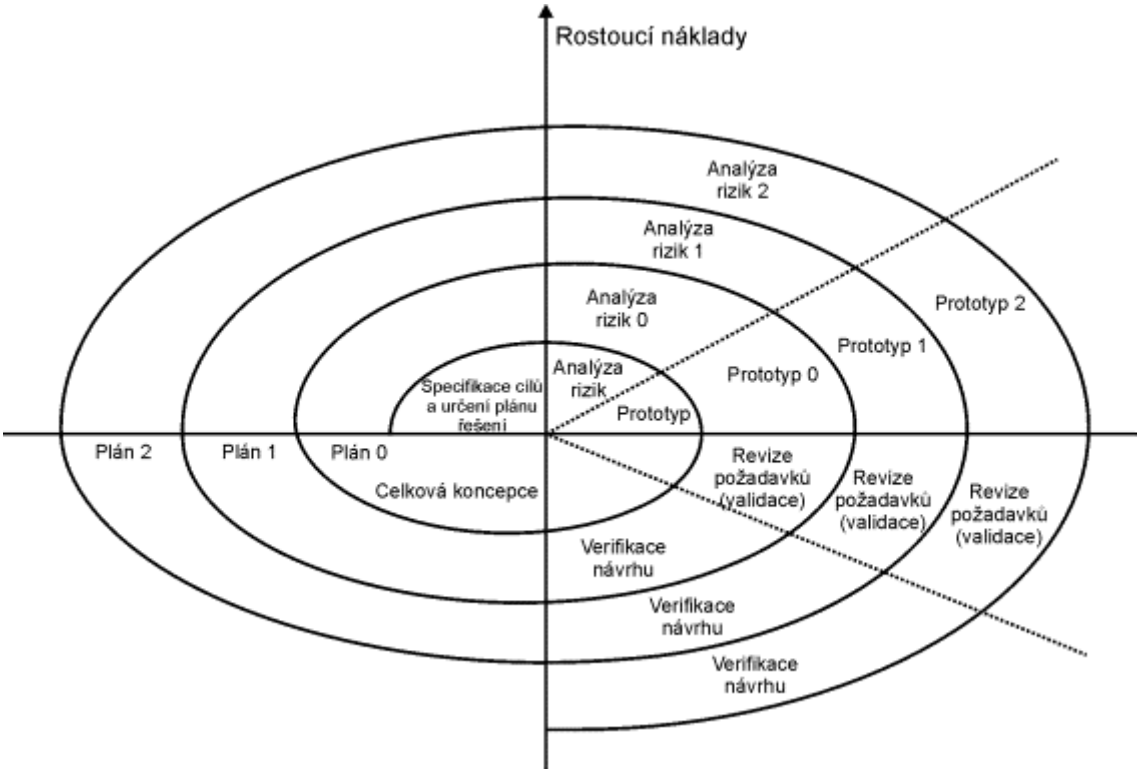
Spirálový model [6] byl představen opět B. W. Boehmem v roce 1988 a je kombinací prototypového přístupu a analýzy rizik. Základem celého modelu je neustálé opakování vývojových kroků tak, že v každém dalším kroku se k již ověřené části systému přidají další části na vyšší úrovni. Postup vývoje v jednotlivých krocích je shodný s původním modelem vodopád a během každého cyklu spirály jsou tak spouštěny čtyři základní fáze (kvadranty).

- Analýza – stanovení cílů, alternativ a rozsahu iterace.
- Vyhodnocení – vyhodnocení alternativ, identifikace a řešení rizik.
- Vývoj – vývoj produktu a kontrola očekávaných výsledků.
- Plánování – plán pro příští iteraci.

Výhody spirálového modelu

- Model využívá ověřené kroky vývoje a analýzou rizik předchází chybám.
- Umožňuje konzultovat požadavky zákazníků v jednotlivých krocích a modifikovat systém podle upřesněných požadavků.
- První verze systému je možné sledovat a hodnotit při jejich postupném vzniku.

Na obrázku 3.4 je znázorněn princip spirálového modelu. Náklady a čas nutný na realizaci jednotlivých částí projektu, či na řešení celého projektu, jsou patrné z obrázku.



Obrázek 3.4: Spirálový model [6]

3.2 Agilní metodiky

Základem agilního přístupu je úzká spolupráce programátorů s koncovými uživateli systému, kteří spolupracují při všech fázích vývoje. Dotaženo do extrému, uživatelé se na vývoji přímo podílejí, protože jako součást vývojového týmu poskytují programátorům okamžitou zpětnou vazbu. Z pohledu vývojového týmu je hlavní změnou ochota kdykoli přistoupit na změnu zadání a řešení předělat v nejlepším zájmu klienta.

O agilním přístupu k vývoji software hovoříme od sepsání zakládajícího dokumentu The Agile Manifesto [16]. Tento dokument, vypracovaný v roce 2001 skupinou předních teoretiků a vývojářů je tak stručný, že jej můžeme ocitovat v jeho úplném znění.

> Objevujeme lepší způsoby vývoje software tím, že jej tvoříme a pomáháme při jeho tvorbě ostatním. Při této práci jsme dospěli k těmto hodnotám.
>
> - **Jednotlivci a interakce** před procesy a nástroji.
> - **Fungující software** před vyčerpávající dokumentací.
> - **Spolupráce se zákazníkem** před vyjednáváním o smlouvě.
> - **Reagování na změny** před dodržováním plánu.
>
> Jakkoliv jsou body napravo hodnotné, bodů nalevo si ceníme více.

3.2.1 Extrémní programování

Extrémní programování [7] zavedl Kent Beck a metodika byla poprvé použita v roce 1996. Jejím základem jsou standardní postupy (psaní kódu, testování atd). Co dělá model extrémním, jsou tyto standardní postupy dotažené do extrémů. Díky tomu je možné dosáhnout vyšší kvality a pružněji reagovat na změny klientových požadavků během vývoje. Manažeři, zákazníci i programátoři tvoří jeden tým. Ten se samoorganizuje tak, aby bylo vyřešení problémů co nejefektivnější.

Nejjednodušší návrh, který ještě funguje

Klasický návrh plánuje do budoucnosti. Do návrhu se snažíme zahrnovat i vlastnosti, u kterých se domníváme, že je budeme potřebovat v budoucnosti. Teorie extrémního programování ovšem říká, že tato strategie funguje pouze v případě, pokud se nic nezmění mezi přítomným a budoucím časem.

Extrémní programování radí aplikovat jednoduché řešení problému tak, aby to ještě fungovalo. Tímto způsobem nikdy neplatíme za flexibilitu, kterou nevyužijeme a budeme mít tendenci činit systém pružnější tam, kde pružný má být. Návrh bude jednodušší, bude mít menší sklony k chybám. Pokud se chyby vyskytnou, budou se lépe opravovat. V případě, že v budoucnosti bude nutné systém rozšířit, rozšíříme ho pouze o vlastnosti, které skutečně potřebujeme.

Testování

Testy se v případě extrémního programování stávají přímo součástí programu a mnoho programátorů píše testy ještě dříve než samotný program. Není extrémního programování bez automatizovaných testů. Kent Beck je jedním z průkopníků jednotkového testovaní, když položil základ rodině populárních xUnit testovacích frameworků. Konkrétně testovacím frameworkem SUnit pro jazyk Smalltalk. Testy v extrémním programování mají několik pravidel.

- Žádné dva testy se nesmí ovlivňovat navzájem.
- Testy jsou maximálně automatické. Když testy spouštíme, nemusíme nad nimi přemýšlet.
- Testy nás mají potěšit. Mají nám ukázat, že naše práce je kvalitní a software dělá přesně to, co dělat má.

Párové programování

U jednoho počítače by vždy měli při programování sedět dva vývojáři s jasně rozdělenými rolemi. První programátor píše kód a druhý přemýšlí o celku a dohlíží na kvalitu napsaného kódu. Jejich role se mění během dne a to jak v rolích jednotlivých programátorů, tak se mohou vzájemně prohazovat programátoři v různých párech.

Tato technika je vhodná zejména při práci na velmi složitých úkolech, u kterých je vysoká šance na vytvoření chybného kódu. Naopak je tato technika spíše kontraproduktivní pro jednoduché úkoly.

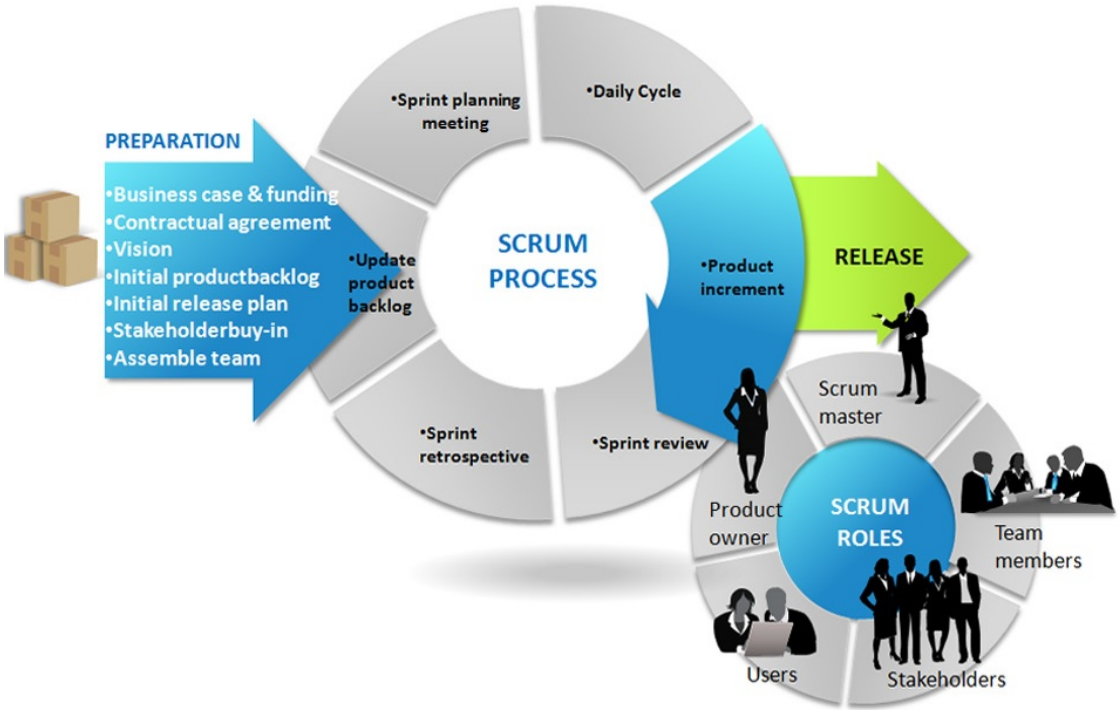
3.2.2 Scrum

Název metodiky Scrum [8] pochází z terminologie hry ragby. Označuje shluk hráčů, kteří se snaží dotlačit míč do cíle. Celý vývojový proces si je možné představit jako ragbyový zápas, přičemž vítězstvím se myslí spokojný zákazník.

Scrum je metodika vývoje softwaru, určená pro menší a zkušené pracovní týmy. Umožňuje optimálně vytěžit schopnosti všech pracovníků v týmu a dodat tak kvalitní výsledek práce v krátkém čase.

Základem Scrumu je sprint. Sprint je periodická činnost (s periodou obvykle 2-4 týdny), která je každý den kontrolována. Vstupem sprintu je seznam návrhů na vylepšení stávajícího produktu tzv. features, příp. chyby. Výstupem je produkt s novou funkcionalitou, příp. s opravenými chybami. Každý sprint je zahájen podrobným rozplánováním vylepšení a chyb, včetně časových odhadů.

Důležitým prvkem zajišťujícím funkčnost celého procesu jsou pravidelné schůzky, které probíhají každý den a neměly by být delší jak 15 minut. Na setkání jednotliví členové shrnou co udělali od posledně, na čem se chystají pracovat a co je zdržuje v práci. Scrum Master[1] tak má možnost sledovat, zda se naplánované odhady shodují s realitou a při případném zdržení (podhodnocený odhad, výskyt nečekaných problémů) přesunout část práce do dalšího sprintu.



Obrázek 3.5: Jak funguje Scrum proces [8]

[1]Scrum Master je osoba, jejíž úkolem je zajistit hladký průběh sprintu, organizovat schůzky, řešit případné administrativní problémy

3.3 Porovnání tradičních a agilních přístupů

Když se bavíme o agilních technikách, určitě nás napadnou různé otázky. K čemu to je dobré? Jak nám přesun k agilnímu řízení pomáhá? Může agilní řízení koexistovat s tradičním přístupem? A mnoho dalších otázek. Než si některé otázky zodpovíme, shrneme si základní faktory, kterými se od sebe odlišují.

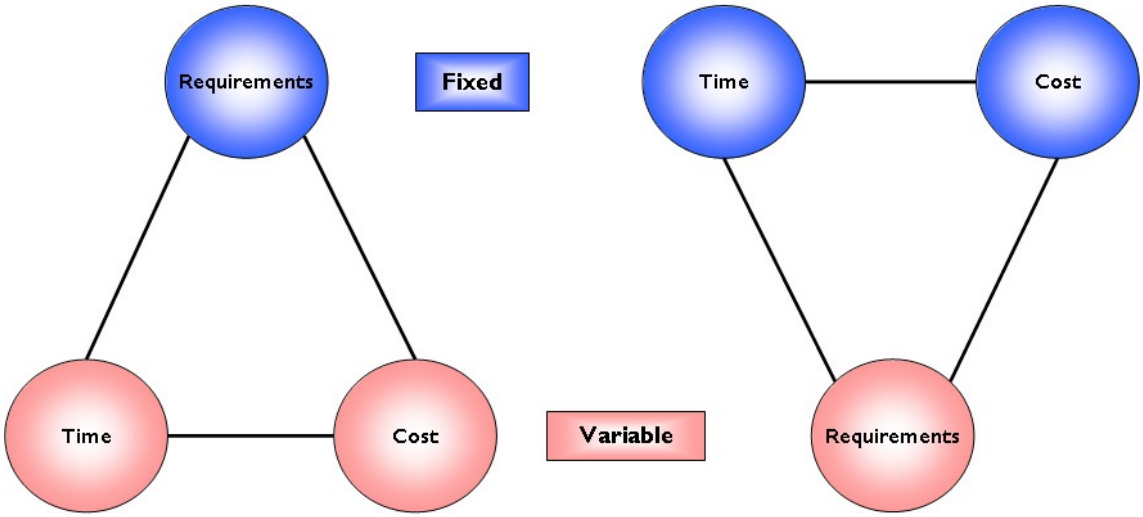
Tradiční	**Agilní**
Důraz na procesy a nástroje	Komunikace, individualita, kreativita
Uzavírání smluv s restrikcemi	Spolupráce se zákazníkem
Obsáhlá dokumentace	Provozuschopný software
Plánování je mohutná činnost na samém počátku projektu	Plánování je evoluční proces. Plán se postupně vyvíjí a stává podrobnější
Klasická hierarchická struktura shora dolů	Samospravný charakter tj. týmy přijímají rozhodnutí sami za sebe

Tabulka 3.1: Porovnání tradičních a agilních přístupů k vývoji softwaru

Množina požadavků na funkcionalitu je jednoznačně stanovena na začátku vývoje. Prioritou tradičního přístupu vývoje zůstává tedy naplnění funkčnosti, přičemž potřebné náklady na projekt stejně jako čas potřebný k jeho dokončení se jen odhadují, nikoliv přesně stanovují. Formulace požadavků na funkcionalitu zůstává fixní, variabilní jsou zdroje a čas. Metodiky agilního vývoje naopak přesně vymezují konečný termín dokončení pro realizaci projektu a nejvyšší možné zdroje na začátku projektu. Předmětem změn se stávají jednotlivé funkčnosti aplikace. Požadavky na funkcionalitu se přizpůsobují v průběhu vývoje, zatímco čas a zdroje zůstávají neměnné [5, s. 119].

Agilní metodiky jsou využívány na projektech, kde se zadání často mění nebo není přesně specifikováno. Právě z důvodu častých změn je nutné průběžné a opakované ověřování správnosti zdrojového kódu. Automatizované testování by mělo předcházet implementaci. Silný důraz je kladen na psaní programového kódu, který začíná ihned po vymezení cílu projektu a tvorbě testů. Agilní přístupy kladou důraz na komunikaci mezi procesy (např. párové programování), proto jsou chyby a problémy dříve odhaleny. Nové funkčnosti jsou implementovány poměrně v krátkých dodávkách, proto je iterativní a inkrementální způsob vývoje

> spojen s velmi krátkými iteracemi. Zákazník má možnost monitorovat a upravovat průběžné konfigurace. Obsahují podstatně menší objem dokumentů, jejich dokumentace není tak rozsáhlá jako u rigorózních metodik [5, s. 119].



Obrázek 3.6: Priority při tradičním a agilním vývoji softwaru

Koexistence tradičního a agilního řízení

Přestože se oba přístupy zdají být v jasném kontrastu, tak mohou v jednom týmu koexistovat. Není nutné k oběma přistupovat takto vyhraněně. Můžou koexistovat a také se vzájemně doplňovat. Inteligentní agilní manažer dělá nevědomě spoustu rozhodnutí, o kterých ani netuší, že jsou součástí tradičních definic řízení. Platí to samozřejmě i opačně.

Kapitola 4

Nástroje podporující týmový vývoj

Při vývoji ve více lidech je kritickým faktorem úspěchu sdílení informací mezi všemi členy týmu. Informacemi se myslí zdrojové kódy aplikace, evidence chyb, evidence požadavků, monitorování stavu vývoje, rozdělení práce a tak dále. Výběrem správných nástrojů a jejich správným používáním si velmi usnadníme práci.

4.1 Verzovací systém Git

Při programování ve skupině vznikají různé verze. Každý napíše něco, změní určitou část, a pak je potřeba vše spojit dohromady. Jednotlivé verze je možné si posílat třeba e-mailem. Ale jsou i vyspělejší způsoby, které nazýváme verzovací systémy. Hlavní výhody plynoucí z používání verzovacího systému.

- Uchování historie verzí.
- Týmová spolupráce.
- Možnost souběžné práce na jednom projektu (tzv. ve více větvích).
- Redukce konfliktů v kódu na minimum.

Verzovací systémy rozdělujeme na centralizované (Subversion) a decentralizované (Mercurial, Git). U centralizovaných systémů máme kompletní historii změn pouze na serveru, zatímco na klientech je pouze poslední verze programu a rozpracované soubory. Jsou zde jasně rozdělené role na server a klienty. V případě decentralizovaných (distribuovaných) systémů jsou všechny uzly rovnocenné, na všech máme kompletní historii a můžeme se vrátit k libovolné verzi, aniž bychom se museli dotazovat serveru. Role nejsou pevně dané a můžou se dynamicky měnit.

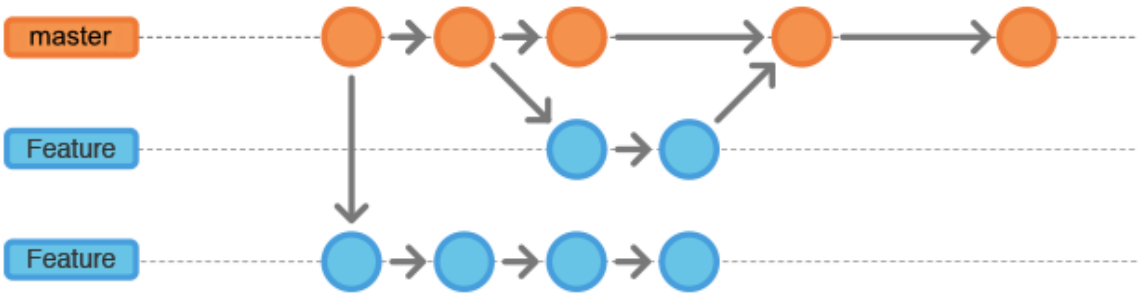
Git je možné používat stejným způsobem jako centralizované verzovací systémy, ale jeho síla vynikne, až když se ponoříme více do hloubky. Modelů, jak správně pracovat s Gitem existuje mnoho. Já mám dobré zkušenosti s modelem Git-Flow [9], který nachází uplatnění především při vývoji středně velkých projektů. K porozumění jak funguje Git-Flow, je nezbytné se seznámit též s modely Feature Branch a Pull Request.

4.1.1 Feature Branch

Master větev představuje hlavní historii projektu. Ovšem pokaždé, když se začne pracovat na nové funkci (feature), se vytvoří nová větev. Nové funkce musí mít popisné názvy. Záměrem je poskytnout jasný a srozumitelný účel každé větvi. Jednotlivé funkce by měly být malé, nejlépe aby netrvaly déle než jeden pracovní týden.

Když je nová funkce hotova, musí být zařazena zpět do master větve, aby k ní měl přístup i zbytek týmu. Připojení feature větve k master větvi je nejlepší provést skrze tzv. Pull Request. Přínos Feature Branch spočívá v tom, že v hlavní větvi máme jen funkční a otestovaný kód.

Technicky Git nerozlišuje mezi hlavní a vedlejší větví. Vývojáři mohou na feature větvi vykonávat úplně ty samé operace jako tomu je u hlavní větve.

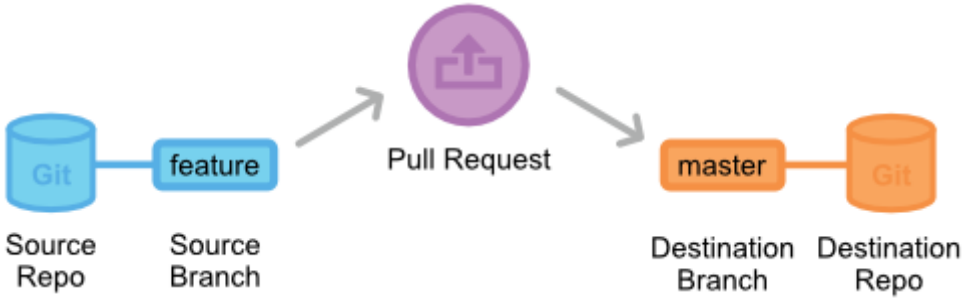


Obrázek 4.1: Feature Branch [9]

4.1.2 Pull Request

Pull Request je mechanismus pro autora feature větve, jak oznámit ostatním členům týmu, že je se svojí prací hotov. Při vykonání pull requestu se všichni zainteresovaní dozvědí, že je potřeba podrobit tuto větev drobnohledu a začlenit ji do hlavní větve.

Pull Request je ale více než jen oznámení pro ostatní. Je to i vyhrazené fórum pro diskuzi o dané větvi. Pokud ostatní členové naleznout problematické části, mohou zde vznášet dotazy, diskutovat a navrhovat zlepšení. Všechny tyto aktivity jsou zaznamenávány přímo v konkrétním pull requestu.



Obrázek 4.2: Pull Request [9]

4.1.3 Git-Flow

Git-Flow je sada pravidel jak pracovat s Gitem. Nepřidává žádný nový koncept, ani nerozšiřuje jeho vlastnosti. Pouze stanovuje pevný a osvědčený rámec pro práci na větších projektech. Přiřazuje každé větvi konkrétní roli a definuje jak, kdy a které větve by měly mezi sebou interagovat. Kromě Feature Branch přidává další role pro údržbu, opravy a vydávání nových verzí vyvíjeného softwaru.

Git-Flow stále používá centrální repozitář jako komunikační uzel pro všechny vývojáře. A stejně jako u jiných postupů, vývojáři pracují na lokální úrovni a přidávají nové větve do centrálního repozitáře. Hlavní rozdíl je, že každá větev má přiřazenou nějakou roli a podle toho se s danou větví i pracuje.

Historical Branch

Tento postup pro uchování historie projektu používá namísto jedné hlavní větve rovnou dvě hlavní větve. Master větev uchovává historii software tak, jak je uvolňován do produkce. Naopak develop větev slouží jako integrační větev pro nové funkce. Pro pořádek je vhodné každé přidání do master větve označovat číslem aktuální verze produktu.

Na začátku vývoje vytvoříme master větev. Poté z master větve provede klon a máme develop větev. A toto je poprvé a naposled, s výjimkou hotfix větví, co byl tok dat z master větve ven.

Feature Branch

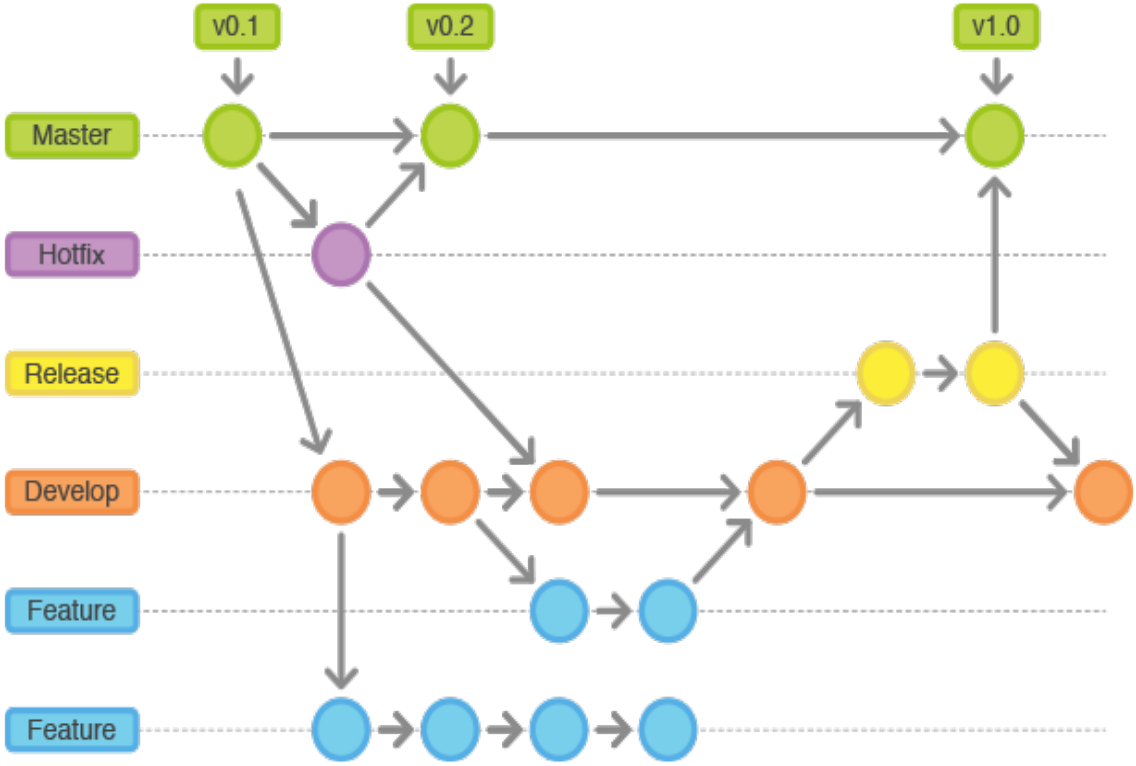
Co je Feature Branch již víme. Jediný rozdíl je, jak s ní pracovat. Pravidlo je jednoduché. Feature větev nikdy nekomunikuje s master větví (ani release nebo hot-fix větví). Vždy jen s develop větví. Když vytváříme novou větev, vždy jako klon develop větve. Když chceme přidat novou funkci do systému, opět ji slučujeme jen s develop větví.

Release Branch

Jakmile se nám blíží datum vydání nové verze softwaru, uděláme klon z develop větve a novou větev pojmenujeme jako release s příslušným číslem verze. Vytvořením této větve začíná nový cyklus před oficiálním vydáním. Do realese větve už nesmějí být přidávány žádné nové funkce. Pouze opravy, doplnění dokumentace a podobné úkony. Jakmile je vše připraveno, otestováno, release větev se sloučí do master větve a je otagována. Poté je ještě sloučena zpět do develop větvě. V develop větvi během release cyklu normálně pokračuje vývoj nových vlastností.

Hot-Fix Branch

Hot-Fix větve se používají pro rychlou opravu vydaných verzí. Hotfix větve jsou jediné, které by měly být vytvářeny jako klon přímo z master větve. Jakmile oprava je kompletní, hotfix by měl být sloučen do obou hlavních větví master a develop (nebo aktuální release větve).



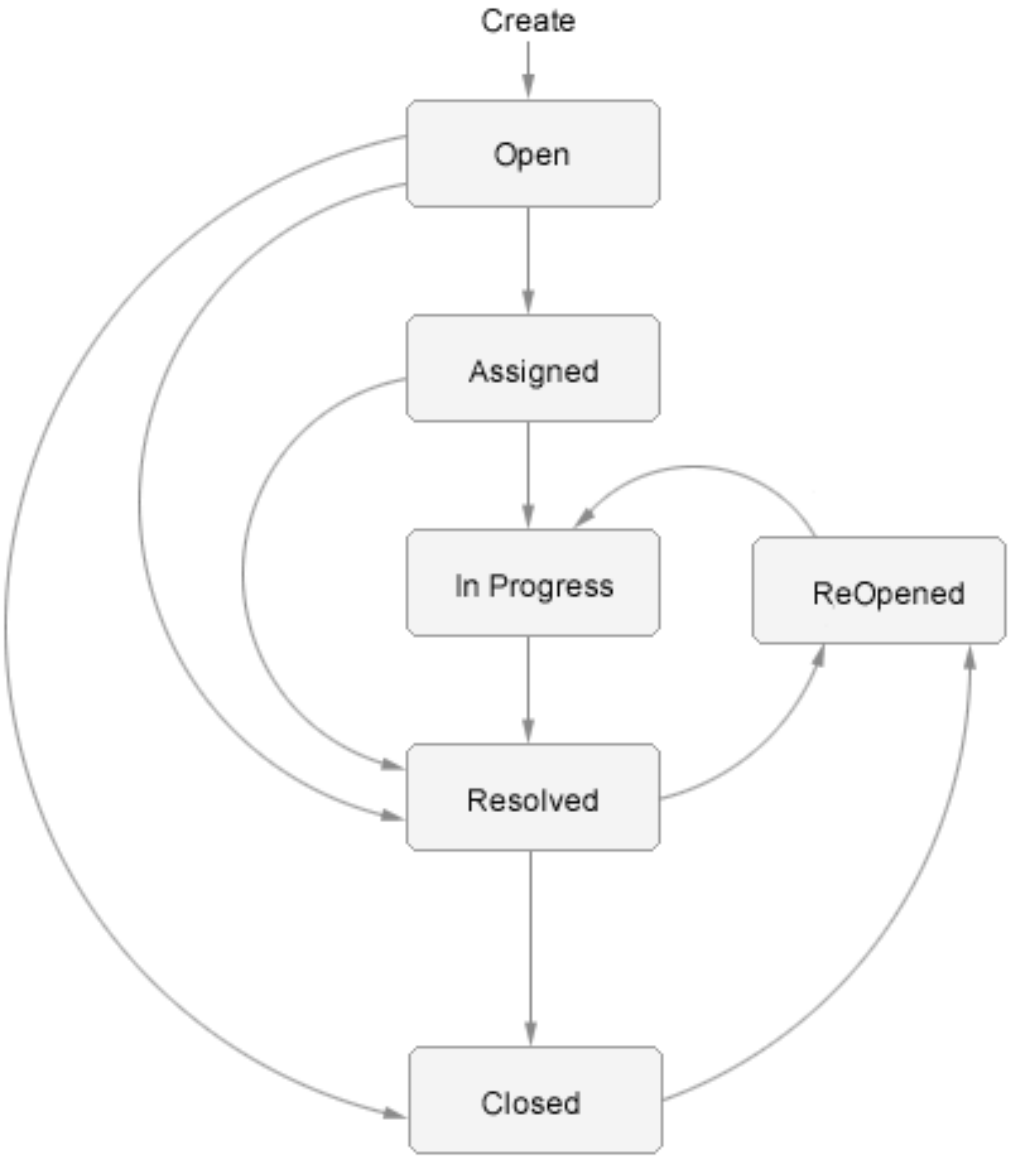
Obrázek 4.3: Git-Flow [9]

4.2 Správa chyb a požadavků

Při vývoji se stále objevují nové požadavky, nebo se nalézají nové chyby. K evidenci těchto věcí stále mnoho lidí používá Excel nebo e-mail. Ale tyto taktiky jsou neefektivní, a nebo přinejmenším náchylné k chybám. Dobré automatizované řešení pro správu chyb a požadavků by mělo zjednodušit proces získávání, řízení a stanovení problémů. Co by takový nástroj měl přinejmenším sledovat.

- Co má být opraveno nebo vytvořeno.
- Co to je za chybu a jaké jsou její okolnosti. Co to vlastně nefunguje.
- Jak by to mělo fungovat správně.
- Kdo vydal požadavek, kdo ho potvrdil, analyzoval, implementoval řešení a kdo ověřil jeho funkčnost.
- Kdy byla chyba nahlášena, kdy byla opravena a kdy ověřena.
- Co vedlo k rozhodnutí vybrat tohle řešení před jiným.
- Jaké změny v kódu byly provedeny.
- Jak dlouho trvalo vyřešit nahlášený problém.

Na druhou stranu by systém pro správu chyb a požadavků neměl být zbytečně komplikovaný. Práce s takovým systémem by měla být příjemná a intuitivní. Ať si vybereme Redmine, Pivotal Tracker nebo jakýkoliv jiný úspěšný trekovací systém, vždy se bude držet (přibližně) pracovního postupu znázorněného na obrázku 4.4.



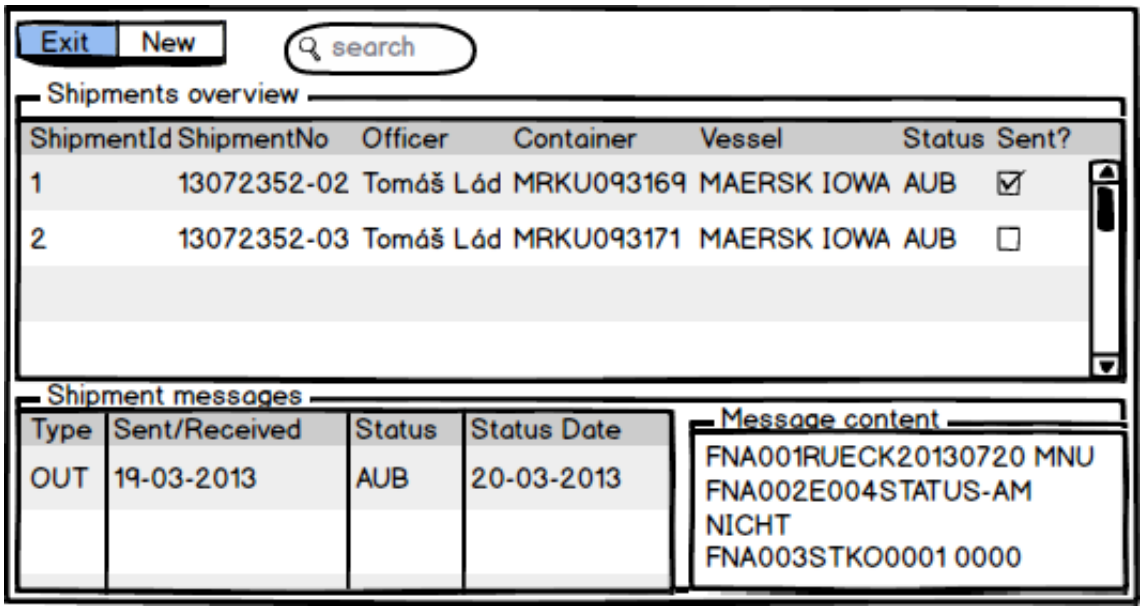
Obrázek 4.4: Pracovní postup se systémem pro správu chyb a požadavků

4.3 Návrh GUI

Každá dobrá aplikace musí být nejen dobře naprogramována, ať už z hlediska efektivity fungování, nebo vnitřní architektury, ale musí být také jednoduše použitelná. Dobré uživatelské rozhraní umožňuje uživateli pracovat s aplikací účelně a pohodlně. Rozhraní, které navíc dodržuje předepsaná pravidla, pomáhá uživateli vytvářet

správné stereotypy práce se systémem. Správně navržené uživatelské rozhraní je také přínosem pro programátory, neboť dobré řešení problému bývá zároveň jednodušší než špatné řešení. Nemusí se tak psát složité a rozsáhlé manuály, protože uživatelé mohou uplatnit dobré návyky z dosavadních aplikací.

Při návrhu uživatelského rozhraní je velmi důležité začít tzv. skicováním[1] obrazovek namísto ostrého programování. Skici [2] se lépe konzultují se zadavatelem, neboť zadavatel chápe, že se jedná pouze o hrubý náhled na obrazovky a nesnaží se jít tolik po detailech. Též je vhodné vytvářet skici černobílé, aby nedošlo k záměně za finální vzhled. Při prvním nástřelu obrazovek bezprobléму postačí tužka a papír. Pro další a lepší návrh je vhodné využít specializovaný software. Vhodným nástrojem je například Balsamic Mockups [13].



Obrázek 4.5: Ukázka skici vytvořené v programu Balsamic Mockups

[1]Slovo skica je spojováno se snahou o vyjádření myšlenky či první náhled. V malování to je základ pro následný obrázek.

[2]V softwarové branži se skici též označují jako wireframy.

Kapitola 5

Vývoj na Java platfomě

Java patří mezi celosvětově nejpopulárnější platformy pro vývoj desktopových, webových a mobilních aplikací. Je to tak díky jednoduchosti, výkonu a pružnosti jazyka Java na straně jedné a díky spoustě knihoven, nástrojů a návrhových vzorů na straně druhé, které zefektivňují práci programátora.

Java platforma je ověřená velkými firmami jako jsou Oracle, IBM, Google a mnoha dalšími. Jazyk Java byl navržen tak, aby byl snadno pochopitelný, dalo se v něm jednoduše vyvíjet a programátoři mohli brzy prezentovat výsledky své práce.

5.1 Buildovací nástroje

Buildovací nástroje jsou primárně určeny ke kompilaci a sestavení projektu ze zdrojových kódů do výsledné podoby. Ať už se jedná o webové stránky, desktopové aplikace, či knihovny. Buildovací nástroje se v průběhu let postupně stávají sofistikovanějšími a přibírají nové funkce, jako například schopnost spravovat závislosti projektu, nebo automatizaci vlastně definovaných úkonů nad rámec základního sestavovacího procesu. Typický buildovací nástroj obsahuje dvě základní komponenty - konfiguraci a interpreter. Moderní buildovací nástroj by měl být dobrý především v následujících činnostech.

- Sestavit projekt ze zdrojových kódů do finální podoby.
- Umět spravovat různé profily (development, staging, production atd).
- Spravovat závislosti v projektu.
- Být na zvolené platformě nezávislý.
- Jednoduše umožňuje automatizovat vše, na co si vývojář vzpomene.

Při výběru vhodného nástroje je potřeba zvážit i další důležité faktory. Například zda se jedná o open-source projekt, nebo jakým způsobem se píší vlastní skripty. V rámci Java platformy dominují především tyto tři - Ant, Maven a Gradle.

5.1.1 Ant

Ant není nejstarším buildovacím nástrojem Java platformy, ale byl první, který byl považován de-fakto jako standard. Jeho sláva začala okolo roku 2000. Z hlediska konfigurace je to těžkopádný systém, který vyžaduje, abyste definovali všechny jeho akce explicitně. Vzhledem k absenci přednastavenému způsobu užití, je Ant ten, u kterého vám nastavení zabere nejvíce času.

Nevýhody Antu

- Chybí přirozená správa závislostí (musí být použit s dalším systémem).
- Zdlouhavé počáteční nastavení.
- Nedostatek výkonu.

5.1.2 Maven

Maven je mladší a zvolil zcela jiný přístup než Ant. Snaží se prosazovat konvence před konfigurací, ovšem s dovětkem, že si můžete vše upravit podle svého. Snaží se být snadno rozšiřitelný pomocí plug-inů. Jako konfigurační jazyk bylo zvoleno XML. Základem jsou tři pluginy - Package, Clean a Test.

Výhody Mavenu

- Aktualně má dominantní podíl mezi uživateli (je jednoduché získat pomoc od zkušenějších).
- Automatizovaná správa závislostí pomocí online repositářů.
- Obrovské množství plug-inů.

Nevýhody Mavenu

- Pro větší úkoly poměrně pomalý.
- Občas je docela těžké prolomit konvence.

5.1.3 Gradle

Gradle je jedním z nejnovějších. Jeho první verze byla uvolněna na konci února 2011. Na rozdíl od Antu a Mavenu, které měly konfiguraci založenou na XML, je zde použit DLS a to Groovy[1]. Gradle, s pomocí Groovy, může snadno využívat všechny existující Maven plug-iny, nebo jakékoliv třídy jazyka Java, což z něj činí jeden z nejvíce výkonných buildovacích nástrojů.

[1]Groovy je objektově orientovaný programovací jazyk pro platformu Java. Jde o alternativu k programovacímu jazyku Java. Můžeme se na něj dívat jako na skriptovací jazyk pro javovskou platformu.

Výhody Gradle

- Postaven na Groovy, což mu propůjčuje úplnou kontrolu.
- Jednoduché vytváření vlastních úkonů.
- Minimalistická konfigurace.

Nevýhody Gradle

- Zatím není tak rozšířen mezi komunitou.
- Pro složitější definování vlastních úkolů je nutné se naučit základy jazyka Groovy.

5.2 Objektově relační mapování

V současnosti je drtivá většina aplikací vyvíjena v objektově orientovaném stylu, ale data těchto aplikací jsou ukládána do relačních databází. To může přinášet problémy a je vhodné navrhnout takové mapování, aby jeden záznam tabulky byl reprezentován jako perzistentní (trvale uložený) objekt příslušné třídy. Tento proces může zabrat 50 až 70 procent času vývoje aplikace.

Objektově relační mapování (ORM) je programovací technika, která tento problém řeší. Na Java platformě je ORM popsáno pomocí rozhraní Java Persistence API a je součástí standardní knihovny. Implementací existuje celá řada např. referenční implementace: EclipseLink, nebo velmi populární Hibernate.

Výhody ORM

- Výsledný kód je přehlednější a tedy méně náchylný k chybám.
- Možnost používat srozumitelnější dotazy.
- Vysoká úroveň nezávislosti na použité databázi a centralizované definování vztahů mezi objekty.

Nevýhody ORM

- Další vrstva abstrakce v aplikaci, tj. další nároky na hardware.
- Pro zpracování velkého množství dat budou vždy vítězit uložené procedury.

Nejrozšířenější implementací standardu JPA v současné době je Hibernate [12]. Hibernate poskytuje ještě další nástroje. Například objektově orientovaný jazyk (HQL), který na základě objektové notace dokáže uložená data z databáze získávat v podobě objektů, nebo generátor mapovacích tříd. Ukázka jedné mapovací třídy pomocí anotací je v příloze A.1. Mapování databázových tabulek je možné též popsat v XML, ale anotace jsou obecně považované za modernější a přehlednější.

5.2.1 Návrhový vzor DAO

Návrhový vzor Data Access Object (DAO) [4] nás abstrahuje od implementačních detailů persistentního řešení. Myšlenka vzoru tkví v tom, že se mezi mapovací třídy a persistentní mechanismus vloží další vrstva. Tedy mapovací třídy již nekomunikují přímo s persistentním řešením (například databází). Komunikují přes tuto nově vloženou vrstvu.

Přínos této vrstvy je, že pokud potřebujete změnit základní persistentní mechanismus, postačí provést změny jenom v jedné vrstvě a nikoli na všech místech, kde se pracuje s daty. DAO vrstva si obvykle vystačí s menším souborem tříd, než je počet doménových tříd, ale není to žádné pravidlo. Všechny naše DAO třídy jsou odvozeny z následujícího rozhraní AbstractDAO, kde T je generický parametr. V tomto kontextu to bude doménová třída.

```
public interface AbstractDAO<T extends Object, ID extends Serializable> {

  T findById(ID id);

  List<T> findAll();

  T save(T entity);

  T update(T entity);

  void delete(T entity);

  List<T> findByCriteria(Criteria criteria);
}
```

5.3 Spring Framework

Spring Framework [22] (dále pouze Spring) je modulární Java/JEE aplikační rámec, jenž také bývá někdy označován jako odlehčený kontejner. Spring Framework je základní součástí Spring IO platformy [23].

Spring je využíván, stejně jako Enterprise JavaBeans [24], zejména pro tvorbu webových aplikací, ale lze jej použít v podstatě pro jakýkoliv typ aplikace včetně klasických desktopových aplikací.

Klíčovou vlastností Springu je umožnit čistým a pohodlným způsobem vzájemně oddělit (z hlediska vzájemné závislosti) nejen jednotlivé vrstvy, ale dokonce i jednotlivé objekty, což je výraznou výhodou například pro jednotkové testování.

Drtivá většina existujících frameworků se zaměřuje na podporu jen určité vrstvy aplikace. Například Wicket, který usnadňuje vývoj webové prezentační vrstvy aplikací, nebo objektově-relačně mapovací nástroj Hibernate, orientující se na perzistenční vrstvu. Spring se naproti tomu snaží konzistentním způsobem propojit všechny vrstvy aplikace.

5.4 Desktopové aplikace

Desktopové aplikace jsou jedním z hlavních typů softwaru. Java GUI knihovna by měla být standardizována a být součástí platformy. Nicméně různé operační systémy mají různé styly a různé sady komponent. Jsou tam některé prvky, které existují na všech platformách a mají podobný vzhled i ovládání. Tyto společné prvky, jako jsou tlačítka, textová pole, zaškrtávací políčka atd. jsou nazývány standardní komponenty. Ovšem různé GUI toolkity poskytují různé sady komponent. Stejné komponenty z různých sad můžou mít jiný vzhled i ovládání. Při posuzování GUI toolkitu se obvykle hledí na dvě měřítka. Prvním měřítkem jsou typy nabízených komponent a druhým měřítkem jsou funkce daných komponent.

Abstract Window Toolkit

Abstract Window Toolkit (AWT) má pouze společný soubor komponent, které existují na všech platformách. Například nenajdete pokročilé komponenty, jako stromy atd., protože tyto komponenty nejsou podporovány všemi platformami. To samé platí pro funkce komponent. AWT podporuje pouze ty funkce, které jsou k dispozici na všech platformách. Díky chabé podpoře komponent a funkcí si AWT k sobě nikdy nepřitáhlo velkou pozornost. Nyní je AWT označené jako zastaralé (deprecated). Ovšem stále bude součástí Javy kvůli zpětné kompatibilitě.

Swing

Swing vychází z AWT, ale je bohatší na komponenty a funkce. Využívá stejné události a je postaven na podobných komponentách. Při vývoji Swingu byl ovšem obětován výkon. Všechny komponenty jsou emulovány. Takže aplikace napsané ve Swingu budou vypadat stejně napříč operačními systémy.

Standard Widget Toolkit

Standard Widget Toolkit (SWT) je základní grafická knihovna pro Javu, která nemá žádnou závislost na standardním grafickém API Javy. SWT je sice napsáno v Javě, ale nesmí používat ochranou známku Java, protože v sobě obsahuje nativní kód z jednotlivých operačních systémů, na které je portováno.

SWT má podobný přístup jako AWT. Komponenty jsou svázány s nativními službami příslušného OS. Pouze v případě, kdy příslušný OS danou službu nepodporuje, tak ji SWT emuluje. Zřídka užívané vlastnosti jsou vynechané.

JavaFX

JavaFX je bohatá klientská platforma pro vytváření cross-device aplikací, tj. aplikací běžících na různých zařízeních. První verze vyšla v roce 2008 a nebyla přijata s velkým nadšením. Důvodem byl problematický JavaFX Script. S příchodem JavaFX 2.0 se toto mění. JavaFX 2.0 je implementována jako standardní knihovna a s příchodem JDK 7u15 se stala součástí standardního balíku.

5.4.1 Java Web Start

Java Web Start je rámec, který umožňuje instalaci, spouštění i aktualizaci javových aplikací doslova jedním kliknutím v internetovém prohlížeči. Programy spouštěné přes Web Start však neběží v prostředí prohlížeče (jako například applety), ale jsou zcela samostatné, přesně tak, jak je má v sobě zažité běžný uživatel. Avšak v jednom se od těch lokálních liší. Protože jsou spouštěné ze sítě (nejčastěji z internetu), musí podléhat zvláštním bezpečnostním restrikcím. Běží pouze v tzv. pískovišti (sandbox), na kterém mají pevně dané hranice. Tyto hranice lze libovolně nastavovat, takže pokud je prohlášena za důvěryhodnou, může bez větších problémů přistupovat na disk hostitelského počítače apod.

Web Start ale přináší i další podstatné vylepšení, které webová aplikace nemůže nabídnout. Tím je spouštění bez nutnosti připojení k síti. Pokud už máme program jednou stažený, můžeme se rozhodnout, jestli jej chceme spustit ze sítě (bude ověřeno, zda máme nejaktuálnější verzi a pokud ne, bude stažena a nainstalována) nebo z lokální instalace.

Aby mohl být vzdálený program identifikován a nainstalován, disponuje Web Start speciálním protokolem JNLP (Java Network Launching Protocol). Jádrem protokolu je soubor JNLP, což je XML dokument uložený na serveru. Obsahuje informaci o spouštěcích třídách programu, jeho autorovi apod. Zájemce o spuštění jednoduše zadá ve webovém prohlížeči adresu tohoto souboru a správně nastavený prohlížeč jej spustí ve speciálním programu (javaws), který se postará o zbytek.

Konečně, pokud se vývojář rozhodne svou aplikaci distribuovat prostřednictvím technologie Java Web Start, musí mít stále na paměti, že všechna data (program samotný, obrázky, zvuk) musí být umístěna v JAR archívech a tyto archívy musejí být z důvodu bezpečnosti digitálně podepsány.

Kapitola 6

Automatizované testování

Testování je důležitou fází životního cyklu vývoje softwaru. Je to série procesů, která zajišťuje, aby software dělal to, pro co byl navržen, a aby nedělal nic, co dělat nemá. Testování má za úkol nacházet chyby v softwaru a umožnit jejich opravení pokud možno dříve, než bude hotový produkt uvolněn do produkce, kde by svými chybami mohl způsobit nemalé problémy. O softwarové chybě hovoříme tehdy, pokud je splněna jedna nebo více z následujících podmínek [2].

- Software nedělá něco, co by podle specifikace produktu dělat měl.
- Software dělá něco, co by podle údajů specifikace produktu dělat neměl.
- Software dělá něco, o čem se produktová specifikace nezmiňuje.
- Software nedělá něco, o čem se produktová specifikace nezmiňuje, ale měla by se zmiňovat.
- Software je obtížně srozumitelný, těžko se s ním pracuje, je pomalý, nebo, podle názoru testera softwaru, jej koncový uživatel nebude považovat za správný.

6.1 Proč automatizované testy

Cílem automatizovaných testů je rychlejší vývoj s vysokou kvalitou. Ovšem nic není zadarmo. Aby automatizované testy byly opravdu přínosem, musí celý tým táhnout za jeden provaz a být přesvědčen o jejich přínosu. Tři důvody, proč se automatizací testů zabývat.

- Na konci každého vývojového cyklu spustíme testy a ušetříme spoustu času zjišťováním, zda jsme něco nerozbili.
- Pomáhají psát kód a celkově lépe specifikovat chování systému.
- Jednou napsaný test sdílíme s celým týmem a každý člen týmu si ho může jednoduše spustit.

6.2 Úrovně automatizovaných testů

Zřejmě neexistuje přesná specifikace, jak by se úrovně automatizovaných testů měly nazývat. Nicméně napříč vývojáři se dohodneme na minimálně následujících třech úrovních.

Jednotkové testy

Je jich hodně, jsou malé a rychlé. Předmětem jednotkových testů je separovat a otestovat chování jedné třídy. Neobsahují žádné integrační body jako jsou například komunikace s databází, volání externích API atd. Jednotkové testy se rychle píší a lehce udržují.

Integrační testy

Mají zajistit, že integrační body uvnitř systému fungují. Integrační testy jsou obecně poměrně snadné na implementaci, ale jejich údržba může být časově náročná. Jsou důležité pro zajištění, že jednotlivé komponenty uvnitř budovaného systému budou spolu i nadále interagovat dle očekávání.

Akceptační testy

Akceptační testy jsou nejvyšší úrovní automatizovaných testů. Výše je pouze manuální testování. Většinou se jedná o testy s uživatelským rozhraním. Akceptační testy jsou velmi křehké a nechají se snadno rozbít nekontrolovanými zásahy například do onoho uživatelského rozhraní. Časově se jedná o velmi drahou záležitost. Ovšem jsou vysoce efektivní při zajišťování, zda systém funguje jako celek z pohledu koncového uživatele.

6.3 Testovací strategie

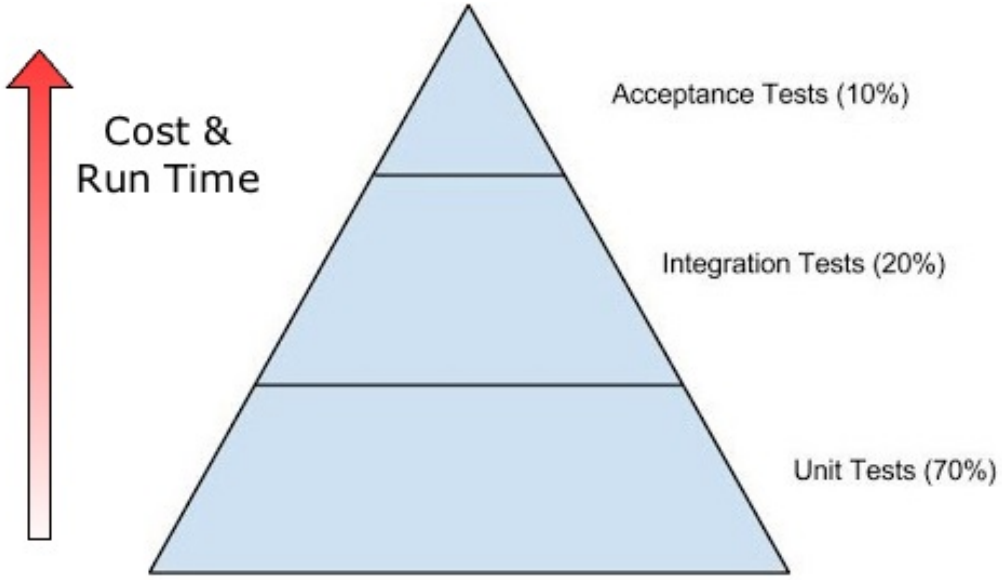
Jedno z mnoha doporučení je namíchat si testovací směs, kde budou majoritní jednotkové testy, méně integračních testů a ještě méně akceptačních testů. Doporučovaný poměr mezi jednotlivými úrovněmi je procentuálně vyjádřen na obrázku testovací pyramidy 6.1.

Integrační testy jsou pomalé a obtížně se udržují (v porovnání s jednotkovými testy), takže je lepší jich mít méně a zaměřit se výslovně na pokrytí problematických integračních bodů. Není efektivní testovat logiku systému s integračními testy, protože jsou pomalé a stejně nebudeme vědět, jestli selhal kód nebo integrační bod.

Akceptační testy jsou nejnáročnější na programování, provoz i údržbu. To je také důvod, proč je důležité, aby se minimalizoval jejich počet. Měli bychom se snažit, pokud to je možné, takový akceptační test rozbít na menší části a testovat na nižších úrovních. Akceptační testy poskytují výborný přehled o tom, jak systém pracuje jako celek. Obecně platí, že jsou nejlepší pro testování scénářů. Takový scénář může být například, že se student přihlásí na portál, zapíše si dva nové předměty, jeden

předmět si odepíše a pak odhlásí. Takových scénářů by mělo být ale jen pár a měly by v sobě zahrnovat více na sebe navazujících činností. Činnosti, které očekáváme od budoucího uživatele. Naopak nemá smysl udržovat jednoduché scénáře. Jednoduché scénáře je lepší rozbít na menší části a otestovat na nižší úrovni.

Automatizované testy píšeme proto, aby jsme mohli rychleji vyvíjet. O kladném přínosu jednotkových a integračních testů nemůže být v dnešní době spor. U akceptačních testů to chce pečlivě uvážit. Vždu tu bude k dispozici manuální testovaní a pokud otestovat jistý scénář manuálně trvá tři minuty a z automatizování by zabralo týden, pak je volba jasná. Přesto je nanejvýš žádoucí se snažit, aby manuální testování bylo omezeno na minimum.



Obrázek 6.1: Testovací pyramida

6.4 Testování na Java platformě

Mezi uživateli jazyka Java se obzvlášť velké oblibě těší testovací framework JUnit [21] s přímou podporou v hlavních IDE.

JUnit se v současnosti převážně vyskytuje ve dvou verzích - ve verzi 3 a ve verzi 4. Hlavní rozdíl mezi těmito verzemi je ve využití anotací jazyka Java. Anotace se používají k přidávání metainformací k elementům jazyka Java a součástí syntaxe jsou od verze Java 1.5, proto jsou použity až ve čtvrté verzi.

6.4.1 Mockito

Při testování řešíme často otázku konfigurace a vůbec zajištění prostředí pro běh testů. Někdy se nám samotné nastavení prostředí může zdát náročné a testy odložíme, což je samozřejmě špatně. Tento problém se dá lehce obejít pomocí tzv. mockování objektů (vytváření zástupných/nepravých objektů).

Mock je dynamicky vytvořený objekt s rozhraním dané třídy. Mockování je definice chování mocku při interakci s okolním světem. Například volám na mocku metodu X a očekávám, že se vrátí hodnota 1. Mocky nám také zaručují izolovanost testu vůči ostatnímu kódu.

Mockito je mockovací framework, který vymyslel Szczepana Faber z důvodu nespokojenosti s ostatními mockovacími frameworky. Tento framework je relativně mladý a snaží si vzít to nejlepší ze svých předchůdců EasyMock a JMock. Autor Mockita měl pro jeho zavedení následující důvody [20].

> - Ať je to jednoduché – není důvod vymýšlet nějaký složitý a sofistikovaný jazyk pro volání metod v Javě, proto již existuje jazyk, jmenuje se Java.
> - Čím méně specifického jazyka, tím lépe. Interakce ať je jenom volání metod. Volání metod je jednoduché, učit se nový jazyk je složité.
> - Nepoužívat řetězce pro metody. Strávil jsem více času čtením zdrojového kódu než psaním nového. Proto chci, aby zdrojový kód byl čitelný.
> - Nepoužívat anonymní třídy – přináší více složených závorek, více odsazení, více kódu atd. Už jsem zmínil, že jsem strávil více času čtením než psaním?
> - Jednoduchý refaktor – změna jména metody, by neměla rozbít test.

Abychom knihovnu mohli použít, musíme ji přilinkovat do projektu. Vše je řešeno jedním statickým importem. Mocky se vytváří pouze jediným způsobem, a to voláním metody mock. Dále se nastaví, jaké hodnoty budou vráceny pro příslušné volání pomocí metody when a jaká hodnota bude vrácena. Následuje ukázka použití Mockito frameworku.

```
import static org.mockito.Mockito.*;
import static org.junit.Assert.*;

@Test
public void test() {
  Comparable c = mock(Comparable.class);
  when(c.compareTo("Test")).thenReturn(1);
  assertEquals(1, c.compareTo("Test"));
}
```

6.4.2 Testování DAO vrstvy

Testování DAO vrstvy se neobejde bez databáze, nad kterou se bude tato vrstva testovat. V projektu PortService je použita instance PostgreSQL - kopie produkční instance očištěná o veškerá data s výjimkou číselníků[1]. Lepší alternativou se zdají být tzv. In-Memory databáze (IMDB)[2]. Konkrétně pro Javu je možné doporučit

[1] V databázových schématech představují číselníky způsob, jak zaznamenat seznam vyjmenovaných přípustných hodnot.

[2] In-Memory databáze používá jako primární úložiště dat operační paměť počítače.

databázi HSQLDB [17]. Argumenty pro užití In-Memory databází při automatizovaném testování perzistenční vrstvy.

- Vykonávané operace jsou rychlejší.
- Větší míru nezávislosti, které testy potřebují ke svému běhu.
- Jednodušší opakovatelnost testů, resp. opakovatelnost očekávaného výsledku. Testy nad cílovou databází jsou citlivější na změny a chyby v testovacích datech, pokud už data před testem obsahují.

Následuje ukázka vybraného testu, kde se testuje uložení a následné smazání řádku v tabulce Address. Celá akce se odehrává ve Spring kontejneru, který se mimo jiné postará například o rollback stavu databáze, pokud test neuspěje. Nutno poznamenat, že kód tohoto testu by vypadal vždy stejně, nezávisle na zvoleném typu testovací databáze.

```
@Transactional
@TransactionConfiguration(defaultRollback = true)
@ContextConfiguration({"classpath:applicationContext-test.xml"})
@RunWith(SpringJUnit4ClassRunner.class)
public class AddressDAOHibernateTest {

  @Autowired
  private AddressDAO addressDAO;

  @Test
  public void test() {
    int size = addressDAO.findAll().size();

    // test whether the address will be created
    Address address = new Address();
    address.setCity("Liberec");
    address.setStreet("Hluboka");
    address.setZip("606 55");
    addressDAO.save(address);
    assertEquals(size + 1, addressDAO.findAll().size());

    // return the database to its origin state
    addressDAO.delete(address);
    assertEquals(size, addressDAO.findAll().size());
  }
}
```

6.4.3 Testování uživatelského rozhraní

Pro správné otestování GUI musíme často psát kód, který simuluje činnost uživatele. Než začneme psát testy, potřebujeme znát strukturu a očekávané chování uživatel-

ského rozhraní. Potřebujeme vědět, z jakých komponent se GUI skládá, jaké jsou jejich unikátní identifikátory, aby bylo možné se na ně v testech odkazovat.

GUI se skládají z objektů, které mají množství různých vlastností, jejichž nastavení je také třeba při testování ověřit. Objekty mohou být v závislosti na dalším kontextu programu aktivní či neaktivní, mohou mít různý vzhled (velikost písma, umístění na obrazovce, rozměry, barva, (ne)viditelnost i obsah (text, číselné a logické hodnoty či výběr ze seznamu hodnot). Některé vstupní formuláře také mohou v rámci ovladače události validovat zadaný vstup.

Knihovna FEST [18] umožňuje snadné vytváření testů pro aplikace postavené na Java Swing. Zkratka FEST pochází z anglického Fixtures for Easy Software Testing, což může být volně přeloženo jako vybavení pro snadné testování softwaru. Oproti ostatním frameworkům, které byly vytvořeny pro testování GUI, má FEST několik vlastností, kterými se vyznačuje. API je napsané v Javě a poskytuje tzv. Fluent Iterface[3].

V příloze je ukázka jednoho GUI testu, který ověřuje zda formulář správně validuje své vstupy A.2.

[3]Termín označující druh návrhu API, který umožňuje psát lehce čitelný kód tak, že všechny metody vracejí takovou hodnotu, aby je bylo možné řetězit za sebou.

Kapitola 7

Referenční projekt

PortService je informační systém zajišťující elektronické zpracování transakcí mezi klientem a přístavem Bremerhaven.

7.1 Požadavky na systém

7.1.1 Funkční požadavky

Funkční požadavky jsou výroky, co by systém měl obsahovat a jak by se měl chovat v určitých situacích.

- Všichni uživatelé aplikace se musejí přihlásit pomocí uživatelského jména a hesla.
- Někteří uživatelé mohou mít administrátorské oprávnění.
- Všichni uživatelé mohou vytvářet/upravovat/rušit zásilky.
- Uživatelé s administrátorským oprávněním mohou spravovat zbylé uživatele a mají přístup ke konfiguraci IS.
- Aplikace musí být schopná komunikovat s přístavem Bremerhaven.
- Nové zásilky se budou vytvářet buď jako zcela nové, nebo jako klon již existující zásilky.
- Již odeslaná zásilka musí jít stornovat.
- Změny v odeslaných zásilkách se budou provádět jako storno již odeslané zásilky a následné odeslaní nově vytvořené zásilky.
- Rozpracovaná zásilka musí jít uložit (bez odeslání).
- Ke každé zásilce se musí uchovávat kompletní historie zpráv od systému BHT.
- Zprávy od systému BHT musejí být přijímány asynchronně na pozadí.

- Pro některé typy zpráv (závažné chyby v zásilce atd.) jsou vyžadované notifikace na určené e-mailové adresy.
- Aktualizace jízdního řádu lodí musí probíhat každý den.

7.1.2 Nefunkční požadavky

Nefunkční požadavky jsou vlastnosti, jenž jsou systémem nabízeny. Například požadavky na výkonnost, standardy kvality, nebo designové omezení.

- Uživatelské rozhraní musí být jednoduché a přehledné. Preferován je styl MS Office (Ribbon).
- Je požadována desktopová (úplná) a webová (odlehčená) verze uživatelského rozhraní.
- Desktopové rozhraní musí být implementováno jako Java Web Start.
- Webové rozhraní ve stylu bohatých internetových aplikací (dynamické tabulky, stromy atd).
- Aplikace musí být schopná nepřetržitého provozu v řádu dní.

7.2 Analýza

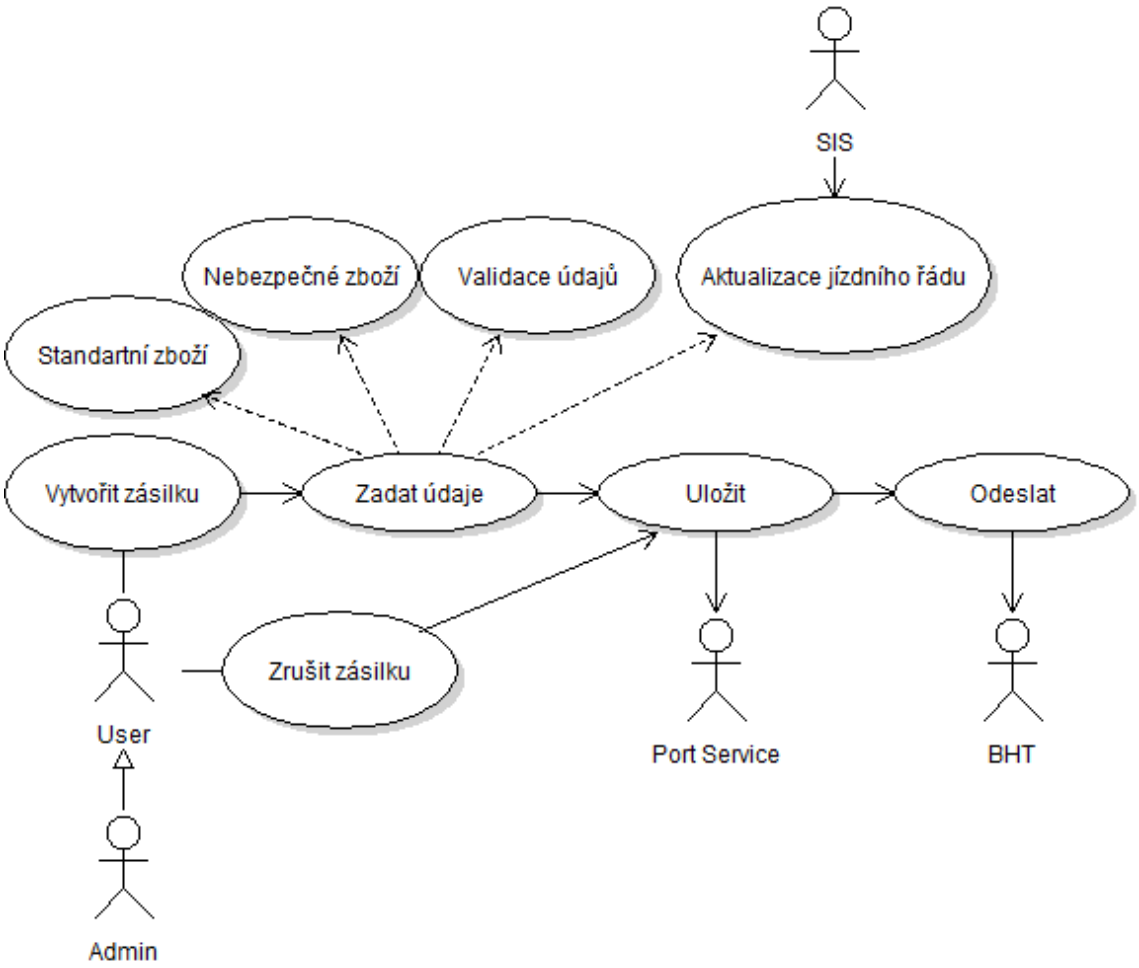
7.2.1 Diagramy případů užití

Z funkčních požadavků aplikace vyplývají základní funkce aplikace a nutnost rozlišení. Diagramy případů užití (Use-Case) zobrazuje chování systému tak, jak ho vidí uživatel. Účelem diagramu je popsat funkcionalitu systému, tedy co od něj klient nebo my očekáváme. Diagram vypovídá o tom, co má systém umět, ale neříká jak to bude dělat. Součástí diagramu jsou také tzv. aktéři.

- *User/Admin* - uživatel pracující s IS mající uživatelské nebo administrátorské opravnění.
- *BHT* - systém přístavu Bremerhaven.
- *Port Service* - databáze IS.
- *SIS* - systém pro řízení lodních cest v Brémách.

Export zásilky do systému BHT

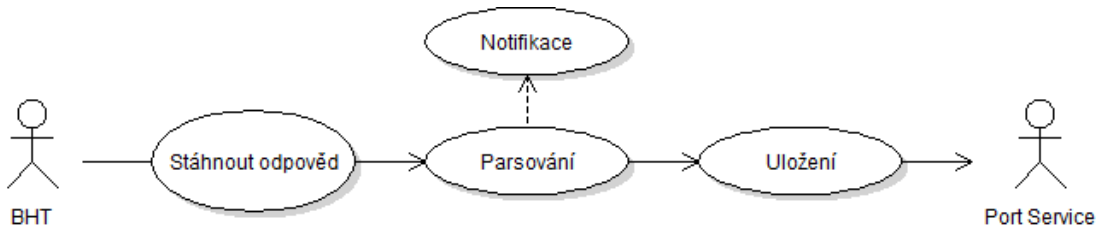
Na obrázku 7.1 je diagram případů užití zobrazující tok událostí vztahujících se k manipulaci se zásilkami. Diagram zahrnuje celý proces od vytvoření zásilky až po její odeslání do BHT.



Obrázek 7.1: Diagram případů užití týkajících se exportu zásilky do BHT

Zpracování zpráv od systému BHT

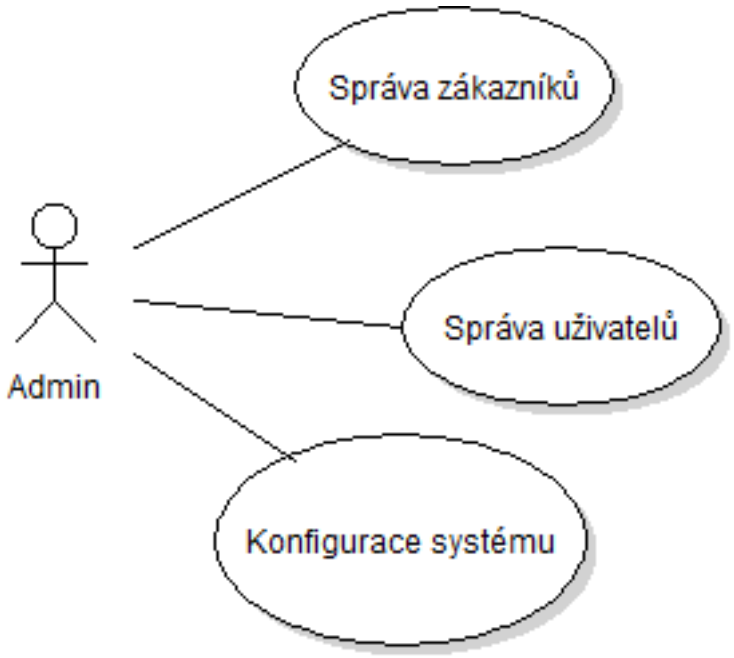
Na obrázku 7.2 je diagram případů užití týkajících se zpracování zpráv od BHT.



Obrázek 7.2: Diagram případů užití týkajících se zpracování zpráv od BHT

Uživatelé jako administrátoři

Na obrázku 7.3 je diagram případů užití zobrazující rozšířené možnosti uživatelů, kteří mají administrátorské oprávnění.



Obrázek 7.3: Diagram případů užití týkajících se uživatelů jako administrátorů

7.2.2 Sledovaná data

- Konfigurační parametry aplikace.
- Logy aplikace.
- Seznam rejdařů působících v Bremerhavenu.
- Seznam zákazníků společnosti.
- Záznamy o kontejnerech (přepravních prostředcích) přepravovaných v rámci zásilky (výška, váha, ISO typ, vlastník atd).
- Údaje o způsobu dopravy zboží do přístavu nakládky.
- Přepravované zboží (typ, váha, počet kusů, balení atd).
- Záznamy o nebezpečném zboží (hořlavost, expolosivita atd).
- Celní údaje o zásilkách.
- Použité plavby ze systému SIS.
- Historii zásilky jak putovala k zákazníkovi.

7.2.3 Export zásilek do systému BHT

V této sekci jsou popsány základní předpoklady a pravidla pro elektronický export údajů o zásilkách mezi systémem PS a systémem BHT. Popis exportu v této práci je zjednodušen, jen aby bylo jasné co se zhruba v aplikaci odehrává, a aby bylo možné vysvětlit architekturu aplikace. Plné znění dokumentace lze nalézt mezi použitými zdroji [14].

Data se mezi systémy PS a BHT přenášejí prostřednictvím textových souborů s využitím protokolu FTP. Server BHT vyčkává na příchozí spojení klienta, který zadává požadovanou akci (poté, co se tento uživatel úspěšně k serveru přihlásil).

Klient vkládá své datové soubory se zprávou (Auftrag) do domovské složky na serveru BHT, který tuto složku v pravidelných intervalech kontroluje. Nově nalezené soubory jsou přeneseny do systému BHT a tam zpracovány. Po přijetí souboru a jeho zpracování je do domovské složky vložen soubor obsahující potvrzení o přijetí (Rückmeldung). Soubory s potvrzeními by měly být klientem po svém zpracování ze serveru BHT vymazány.

Pro obsah přenosového (FT) souboru musí být použito ASCII kódování (veškerou diakritiku je tedy třeba z textů odstranit - to se týká zejména vlastních jmen). Implicitní hodnota textových položek je mezera, číselných pak 0 (u reálných čísel se nepoužívá desetinný oddělovač). Čísla se doplňují na předepsanou délku nulami zleva, řetězce mezerami zprava.

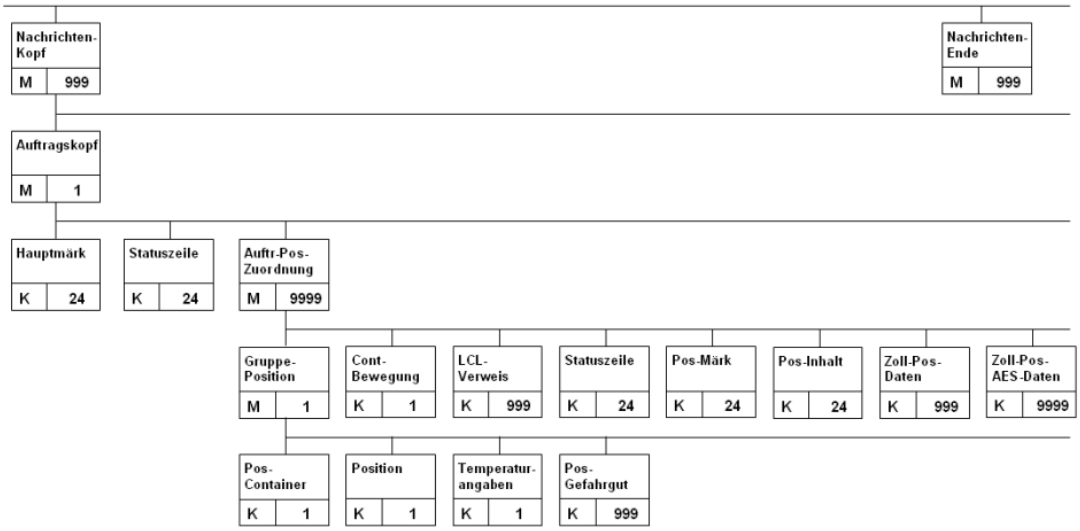
7.2.4 Vytvoření exportního souboru pro systém BHT

Přenos začíná hlavičkovým segmentem FTKO a je ukončen uzavíracím segmentem FTEN. FTKO otevírá datový přenos, FTEN jej uzavírá. Každý přenos (FT soubor) obsahuje jednu či více zpráv reprezentovaných segmenty NAKO, NAEN. Každá zpráva obsahuje alespoň tři datové segmenty.

Segmenty představují sestavu datových atributů. Každý segment sestává z datové skupiny segment-kopf obsahující atributy segment-id, segment-versions-nr a dalších skupin a atributů. Je-li datová skupina označena jako povinná (M), pak musí být ve zprávě uveden alespoň jeden atribut z této skupiny. Zpráva začíná hlavičkovým segmentem NAKO, který identifikuje typ zprávy. Po hlavičce následuje jeden či více segmentů, které vlastní zprávu popisují. Zpráva je ukončena uzavíracím segmentem NAEN. Strukturu zprávy popisuje obrázek 7.4. Písmeno M značí, že segment je povinný a ve zprávě se musí vyskytovat. Písmeno K pak označuje segmenty, které nemusí být ve zprávě přítomny. Číslo určuje, kolikrát se segment může ve zprávě opakovat. Segmenty nachrichten-kopf (NAKO) a nachrichten-ende (NAEN) v podstatě jen obalují vlastní příkaz. Jeden řádek zprávy odpovídá jednomu segmentu. První čtyři písmena každého řádku identifikují segment.

7.2.5 Zpracování potvrzení ze systému BHT

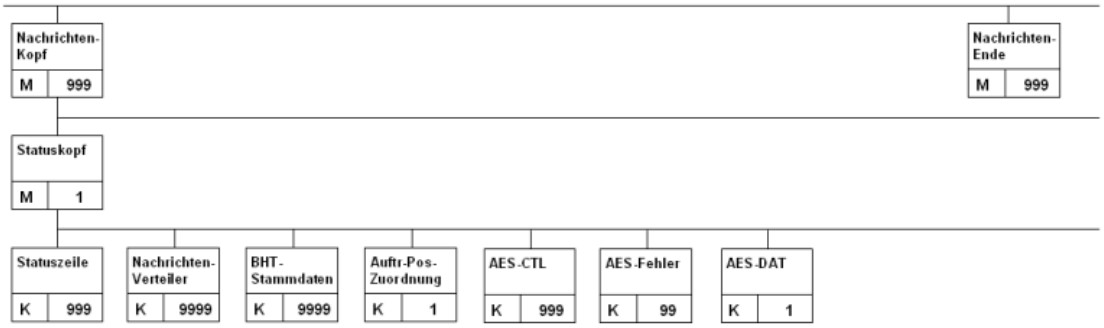
Potvrzovací zpráva (Rückmeldung) generována systémem BHT je odpověď na příkazy klientem BHT k založení, změně či stornování zásilky. V potvrzovací zprávě



Obrázek 7.4: Struktura zprávy Auftrag

systém BHT potvrzuje klientovi úspěšné přijetí přenesených dat. Kromě přidělení identifikátoru jsou zasílány též název lodi, kód přístavu, kód terminálu či např. sdělení o tom, že makléřská kopie nebyla vytvořena, chybové zprávy apod.

Potvrzovací soubor je vygenerován systémem BHT a umístěn do klientské složky na serveru BHT. Tento soubor je třeba stáhnout na lokální disk a zpracovat. Strukturu potvrzovacího souboru zachycuje obrázek 7.5.

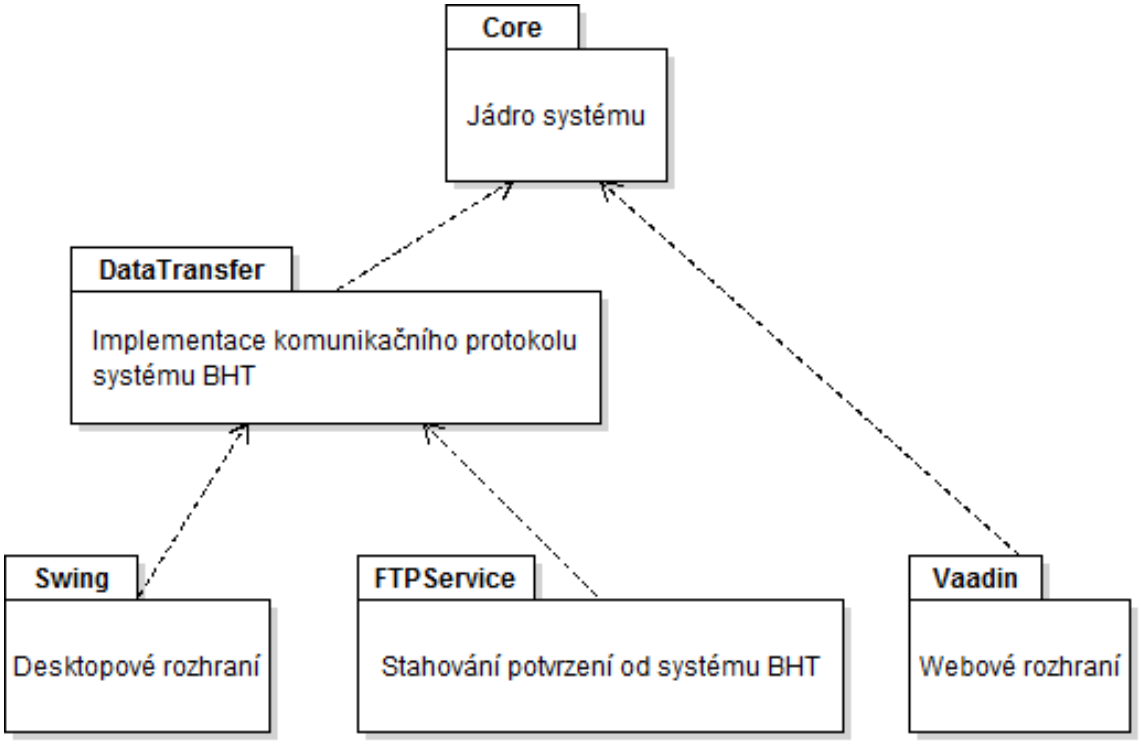


Obrázek 7.5: Struktura zprávy Rückmeldung

7.3 Realizace

Větší systémy rozdělujeme do samostatných modulů s jasně definovanými necyklickými závislostmi, což nám umožňuje rozdělit vývoj jednotlivých částí mezi více lidí s jasně vymezenou zodpovědností a hovoříme o tzv. modulární architektuře.

Tímto pravidlem se řídí také IS PortService, který je rozdělen do modulů dle obrázku 7.6. Jednotlivé muduly jsou představeny v navazujících sekcích.



Obrázek 7.6: Rozdělení systému PortService do modulů

7.3.1 Jádro systému

Modul Core obsahuje především vrstvu pro perzistenci dat. Perzistenční vrstva je v architektuře vícevrstvých aplikací zodpovědná za poskytování dat vyšším vrstvám a zajištění jejich perzistence v databázi. Podrobněji o problematice viz kapitola o objektově-relačním mapování 5.2. Modul Core obsahuje ještě další funkcionalitu nezbytnou napříč celým systémem.

Porovnání dvou zásilek, zdali se shodují

Ke každé třídě modelu existuje porovnávací třída, která implementuje rozhraní IEquality. Porovnávací třídy nám řeknou, jestli jsou dva objekty shodné z hlediska logiky aplikace, nikoliv z pohledu uložení v paměti počítače. Porovnávající třídy jsou v sobě vnořené ve stejném pořadí jako třídy modelu a v aplikaci stačí pouze volat porovnání pro třídu Shipment. Důvodem k takovémuto členění je přehlednější testování.

```
public interface IEquality<T> {

  /**
   * @return true if the given objects are considered equivalent.
   */
  boolean areEqual(T a, T b);
}
```

Validace, je-li zásilka správně vyplněna

Validační třídy kontrolují dodržování nastavených podmínek a implementují rozhraní IValidator. Validační třídy nevracejí pouhou informaci true/false, ale objekt ValidationResult, který je možné rozšířit a obohatit o informace dle potřeby.

```
public interface IValidator<T> {

  void validate(T t, ValidationResult result) throws ValidationException;
}
```

A dále

- Řadící algoritmy dané potřebami systému. Např. zboží musí být při exportu odesíláno ve stejném pořadí, v jakém jsou seřazeny v rámci kontejneru atd.
- Ověřování uživatele.
- Posílání e-mailů.

7.3.2 Implementace komunikačního protokolu systému BHT

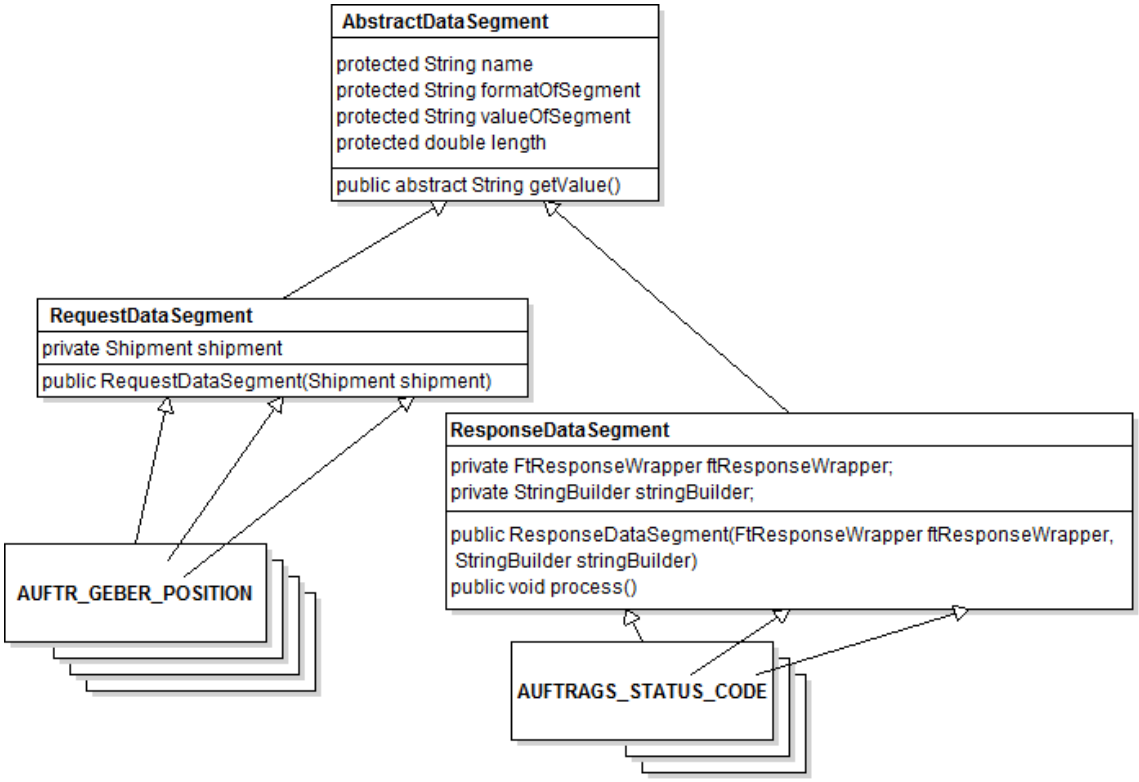
Modul DataTransfer je zodpovědný za sestavení zprávy pro systém BHT (request) a za čtení zpráv od systému BHT (response). Při sestavování jsou vstupem data z PS databáze a výstupem je textový soubor. Naopak při čtení zprávy od BHT je vstupem textový soubor a výstupem jsou uložená data v PS databázi. Tento modul není zodpovědný za samotný přenos textových souborů. V modulu se vyskytují dva typy segmentů a oba typy segmentů mají svoji request a response část.

Datové segmenty

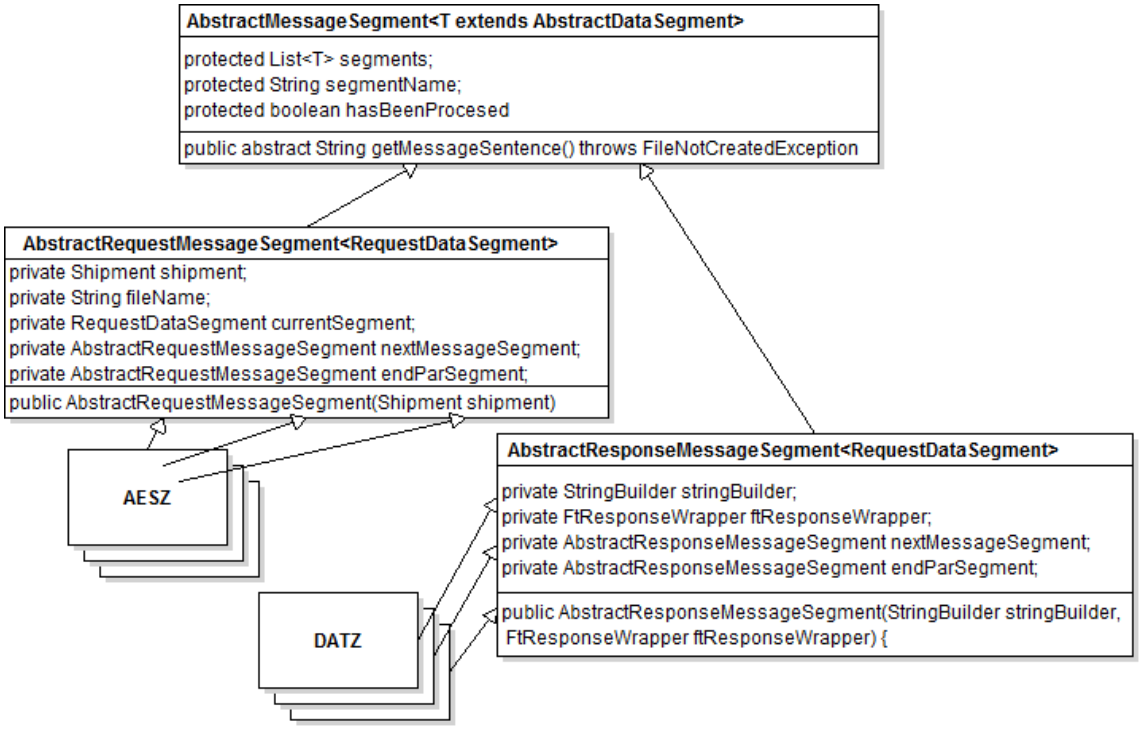
Datové segmenty obsahují konkrétní informace o zásilce jako např. kolik váží kontejner, nebo zda byla zásilka odeslána. Jeden takový segment obsahuje vždy jednu konkrétní informaci. Datové segmenty o sobě navzájem nevědí. Nemají informaci o svém pořadí, ani který segment následuje. Tohle obstarávají hlavičkové segmenty. Každý datový segment rozšiřuje třídu AbstractRequestDataSegment nebo AbstractResponseDataSegment viz obrázek 7.7.

Hlavičkové segmenty

Hlavičkové segmenty jsou nositelé datových segmentů. Dle typu hlavičkového segmentu rozlišujeme jaké informace nám předává, neboli jaké datové segmenty v sobě zahrnuje. Každý hlavičkový segment rozšiřuje třídu AbstractRequestMessageSegment nebo AbstractResponseMessageSegment viz obrázek 7.8.



Obrázek 7.7: Datové segmenty



Obrázek 7.8: Hlavičkové segmenty

7.3.3 Stahování potvrzení od systému BHT

Modul FTPService je implementován jako služba (daemon) běžící na pozadí operačního systému Microsoft Windows, která v pravidelných intervalech stahuje potvrzení z FTP serveru společnosti BHT. Jak jsou potvrzení dále zpracována, to již řeší modul DataTransfer.

Operační systém Microsoft Windows nemá nástroj, kterým by bylo možné spustit Java aplikaci jako službu. Nicméně existují způsoby, jak tento nedostatek vyřešit. Systém PS využívá balíček aplikací Apache Commons Daemon, který reší registraci služby v operačním systému. Je to balíček multiplatformní. Podporuje platformy Windows a UNIX. Součástí balíčku Apache Commons Daemon je aplikace Procrun[1], která umí zavést JAR soubor (java aplikaci) do paměti a spustit v ní požadovanou metodu s požadovanými parametry. Jak se nástroj Apache Commons Daemon používá je názorně popsáno na webových stránkách Apache Commons Daemon [15].

7.3.4 Desktopové rozhraní

Pro desktopové rozhraní byl vybrán framework Swing, neboť to je léty prověřený framework.

7.3.5 Webové rozhraní

Webové rozhraní je vytvořeno ve stylu bohatých internetových aplikací a je postaveno na frameworku Vaadin [25]. Vaadin je Java knihovna, která je navržena pro jednoduché vytváření a správu webového uživatelského rozhraní. Vaadin poskytuje architekturu, při kterém můžete zapomenout na klasické vytváření internetových stránek. Pomocí Vaadinu programujete web ve stejném stylu jako desktopové aplikace a velmi se podobá právě frameworku Swing.

Webové rozhraní slouží jen a pouze pro prohlížení stavu a historie zásilek. Jednoduše kopíruje funkcionalitu úvodní obrazovky. I vzhledově si drží stejný styl a rozmístění komponent. Výjimku tvoří menu, kde nalezneme pouze tlačítko na odhlášení uživatele.

[1] Pro UNIXové platformy se tato aplikace jmenuje JSVC.

Kapitola 8

Závěr

Diplomová práce přibližuje principy při týmovém vývoji softwaru na platformě Java. Součástí práce je referenční projekt, na kterém byly ověřovány nabyté teoretické znalosti. Referenční projekt je informační systém zajišťující elektronické zpracování transakcí mezi klientem a přístavem Bremerhaven. Informační systém byl vyvíjen jako komerční produkt pro společnost Port Service s.r.o. Produkt je již nasazen u zákazníka.

Pokud má být projekt úspěšně doveden k cíli, jsou zásadní především dva faktory. Oba tyto faktory jsou důležitější než samotná volba platformy pro vývoj.

Za prvé si vybrat vhodnou metodiku práce pro daný projekt. V práci jsou popsané tradiční a agilní přístupy k řízení projektů, včetně jejich vzájemného porovnání. Platí přitom, že žádný z možných přístupů není univerzální a samospásný. Pokud ale charakter připravovaného projektu dovolí, je vhodné upřednostnit agilní metodiky.

Za druhé zvolit distribuovaný verzovací systém a dobře ovládnout jeho pracovní postupy. V práci jsou popsány stav a možnosti současných verzovacích systémů. Velký prostor je věnován především distribuovanému verzovacímu systému Git a pracovnímu postupu nazývaném Git-Flow. V počátku projektu byl zvolen centralizovaný verzovací systém SVN. Jak projekt postupně rostl, narůstal i počet konfliktů v kódu. Poté jsme přešli na populárnější Git. Konfliktů se nikdy nezbavíte. Ovšem dají se systematicky minimalizovat, a poté se ušetřený čas měří v hodinách.

Vytvoření této práce mě obohatilo, zejména ve smyslu sbírání zkušeností v oblasti platformy Java, návrhu systému a kooperaci s ostaními členy týmu. Studování technik předcházení problémům je totiž bezpředmětné, pokud se člověk s daným problémem alespoň rámcovitě nesetkal. To se dělo v praktické části velmi často, takže o hledání způsobů, jak řešit problémy, nebyla nouze.

Literatura

[1] E. Gamma, R. Helm, R. Johnson a J. M. Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software* Addison Wesley Professional, 1994, ISBN 0-201-63361-2

[2] R. Patton: *Testování softwaru* Computer Press, Brno, 2002, ISBN 80-7226-636-5

[3] D. Eck: *Introduction to Programming Using Java* O'Reilly Media, USA, 2011, ISBN 978-1441419767

[4] A. Bien: *Real World Java EE Patterns Rethinking Best Practices* O'Reilly Media, USA, 2009, ISBN 978-0-557-07832-5

[5] V. Kadlec: *Agilní programování: Metodiky efektivního vývoje softwaru* Computer Press, Brno, 2004, ISBN 80-251-0342-0

[6] *Životní cyklus informačního systému* [online]
URL:<http://www.fi.muni.cz/ smid/mis-zivcyk.htm>

[7] *Extrémní programování* [online]
URL:<http://www.extremeprogramming.org>

[8] *Průvodce Scrumem* [online]
URL:<https://www.scrum.org/scrum-guide>

[9] *Jak pracovat s Gitem* [online]
URL:<https://www.atlassian.com/git/workflows>

[10] *Introduction to the Java Persistence API* [online]
URL:<http://docs.oracle.com/javaee/6/tutorial/doc/bnbpy.html>

[11] *PostgreSQL* [online]
URL:<http://www.postgresql.org>

[12] *Hibernate* [online]
URL:<http://hibernate.org/orm>

[13] *Balsamiq Mockups* [online]
URL:<http://balsamiq.com>

[14] *Dokumentation Datenschnittstelle BHT* [online]
URL:<http://www.dbh.de/uploads/media/BHT.pdf>

[15] *Apache Commons Daemon* [online]
URL:<http://commons.apache.org/daemon>

[16] *Manifest Agilního vývoje software* [online]
URL:<http://agilemanifesto.org/iso/cs>

[17] *HyperSQL DataBase* [online]
URL:<http://hsqldb.org>

[18] *GUI Functional Swing Testing* [online]
URL:<https://code.google.com/p/fest>

[19] *Software Product Qualities* [online]
URL:<http://ix.cs.uoregon.edu/michal/Classes/W98/LecNotes/01-Qualities.pdf>

[20] *Mockito* [online]
URL:<http://monkeyisland.pl/2008/01/14/mockito>

[21] *JUnit* [online]
URL:<http://junit.or>

[22] *Spring Framework* [online]
URL:<http://projects.spring.io/spring-framework>

[23] *Spring IO* [online]
URL:<https://spring.io>

[24] *Enterprise JavaBeans Technology* [online]
URL:<http://www.oracle.com/technetwork/java/javaee/ejb/index.html>

[25] *Vaadin Framework* [online]
URL:<https://vaadin.com/home>

Příloha A

Zdrojové kódy

A.1 Mapovací třída Shipment

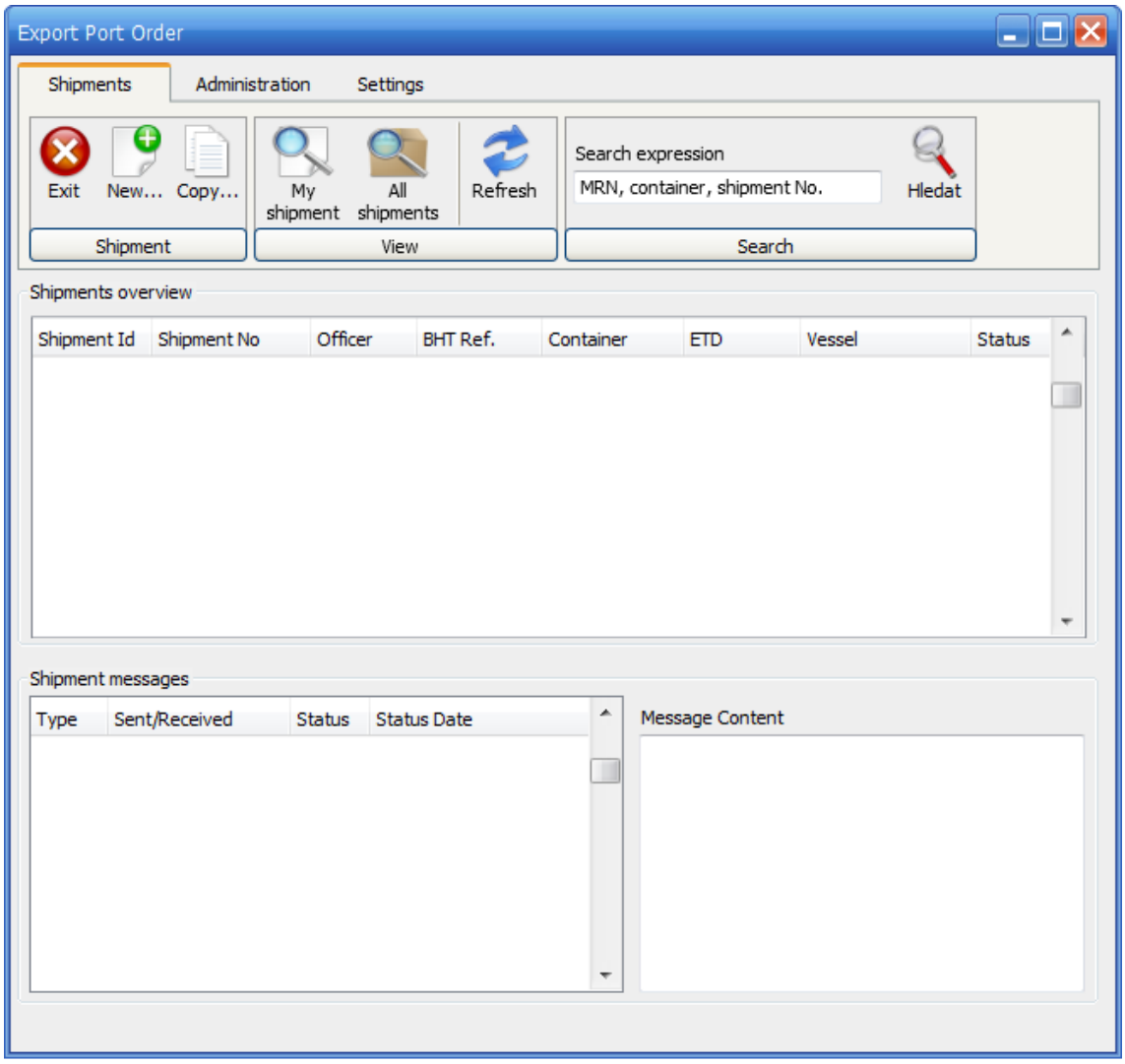
```
@Entity
@Table(name = "shipment",
  schema = "public",
  uniqueConstraints = {
    @UniqueConstraint(columnNames = "shipmentno"),
    @UniqueConstraint(columnNames = "customerref")
  })
@SequenceGenerator(name = "shipment_id_seq",
  sequenceName = "shipment_id_seq",
  initialValue = 1,
  allocationSize = 1)
public class Shipment implements java.io.Serializable {

  private int id;
  private Customer customer;
  private Set<Declaration> declarations = new HashSet<Declaration>(0);

  public Shipment() {}

  @Id
  @Column(name = "id", unique = true, nullable = false)
  @GeneratedValue(strategy = GenerationType.SEQUENCE,
    generator = "shipment_id_seq")
  public int getId() {
    return this.id;
  }

  public void setId(int id) {
    this.id = id;
  }
}
```

A.2 Automatizovaný test formuláře ContainerForm

```
@RunWith(GUITestRunner.class)
public class ContainerFormTest extends FestSwingTestCaseTemplate {

  private Container container;
  private ContainerForm form;
  private TestFrame frame;
  private FrameFixture frameFixture;
  private JPanelFixture panelFixture;

  @Before
  public void setUp() throws Exception {
    setUpRobot();
  }

  @Test
  @GUITest
  public void testChangedValues() {
    container = new Container();
    container.setContainerno("containerno");
    container.setShippersowned(false);
    container.setGrossweight(new Float("9587"));

    form = new ContainerForm();
    form.init(container);

    frame = new TestFrame();
    frame.add(form);

    formFixture = WindowFinder.findFrame("frameId").panel("formId");
    formFixture.textBox("tfContainerNo").setText("K01");
    formFixture.checkBox("cbShipperOwned").check();
    formFixture.textBox("tfGrossWeight").setText("4569.9");

    try {
      form.update(container);
    } catch (UpdateItemException e) {
      fail(e);
    }

    assertThat(container.getContainerno()).isEqualTo("K01");
    assertThat(container.getShippersowned()).isEqualTo(true);
    assertThat(container.getGrossweight()).isEqualTo(new Float("4569.9"));
  }
}
```

Příloha B

Obrázky



Obrázek B.1: Úvodní obrazovka desktopového rozhraní

Příloha C

Obsah přiloženého CD

Soubor	Obsah
`DiplomovaPrace`	diplomová práce ve formátu PDF
`latex/`	text diplomové práce ve formátu LaTeX
`java/`	zdrojové kódy systému PortService