VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

VÝVOJ SOFTWARE ŘÍZENÝ TESTEM

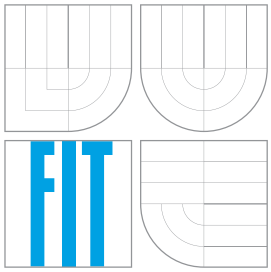
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

DUŠAN NAVRÁTIL

BRNO 2008

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

VÝVOJ SOFTWARE ŘÍZENÝ TESTEM

TEST-DRIVEN SOFTWARE DEVELOPMENT

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — DUŠAN NAVRÁTIL
AUTHOR

VEDOUCÍ PRÁCE — JAROSLAV ZENDULKA, doc., Ing., CSc.
SUPERVISOR

BRNO 2008

Zadání bakalářské práce/5874/2007/xnavra34

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2007/2008

Zadání bakalářské práce

Řešitel: **Navrátil Dušan**
Obor: Informační technologie
Téma: **Vývoj software řízený testem**
Kategorie: Softwarové inženýrství

Pokyny:

1. Seznamte se s problematikou agilního vývoje software se zaměřením na vývoj řízený testem.
2. Seznamte se s dostupnými prostředky na podporu vývoje řízeného testem.
3. Po dohodě s vedoucím navrhněte ukázkovou aplikaci pro zvolené implementační prostředí.
4. Na ukázkové aplikaci ilustrujte vývoj řízený testem.
5. Zhodnoťte dosažené výsledky a diskutujte další možné pokračování projektu.

Literatura:

- Kadlec, V.: Agilní programování. Computer Press, 2004.
- Martin, R.C.: Agile Software Development.Prentice Hall, 2003.

Při obhajobě semestrální části projektu je požadováno:

- 1 až 3.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Zendulka Jaroslav, doc. Ing., CSc.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2007
Datum odevzdání: 14. května 2008

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Ing. Jaroslav Zendulka, CSc.
vedoucí ústavu

LICENČNÍ SMLOUVA
POSKYTOVANÁ K VÝKONU PRÁVA UŽÍT ŠKOLNÍ DÍLO

uzavřená mezi smluvními stranami

1. Pan

Jméno a příjmení: **Dušan Navrátil**
Id studenta: 78839
Bytem: Domamyslická 52, 796 01 Prostějov
Narozen: 20. 12. 1985, Martin
(dále jen "autor")

a

2. Vysoké učení technické v Brně

Fakulta informačních technologií
se sídlem Božetěchova 2/1, 612 66 Brno, IČO 00216305
jejímž jménem jedná na základě písemného pověření děkanem fakulty:

.....
(dále jen "nabyvatel")

Článek 1
Specifikace školního díla

1. Předmětem této smlouvy je vysokoškolská kvalifikační práce (VŠKP):
bakalářská práce

Název VŠKP: Vývoj software řízený testem
Vedoucí/školitel VŠKP: Zendulka Jaroslav, doc. Ing., CSc.
Ústav: Ústav informačních systémů
Datum obhajoby VŠKP: .....

VŠKP odevzdal autor nabyvateli v:

tištěné formě počet exemplářů: 1
elektronické formě počet exemplářů: 2 (1 ve skladu dokumentů, 1 na CD)

2. Autor prohlašuje, že vytvořil samostatnou vlastní tvůrčí činností dílo shora popsané a specifikované. Autor dále prohlašuje, že při zpracovávání díla se sám nedostal do rozporu s autorským zákonem a předpisy souvisejícími a že je dílo dílem původním.
3. Dílo je chráněno jako dílo dle autorského zákona v platném znění.
4. Autor potvrzuje, že listinná a elektronická verze díla je identická.

Článek 2
Udělení licenčního oprávnění

1. Autor touto smlouvou poskytuje nabyvateli oprávnění (licenci) k výkonu práva uvedené dílo nevýdělečně užít, archivovat a zpřístupnit ke studijním, výukovým a výzkumným účelům včetně pořizování výpisů, opisů a rozmnoženin.
2. Licence je poskytována celosvětově, pro celou dobu trvání autorských a majetkových práv k dílu.
3. Autor souhlasí se zveřejněním díla v databázi přístupné v mezinárodní síti:
   - ☐ ihned po uzavření této smlouvy
   - ☐ 1 rok po uzavření této smlouvy
   - ☐ 3 roky po uzavření této smlouvy
   - ☐ 5 let po uzavření této smlouvy
   - ☐ 10 let po uzavření této smlouvy

   (z důvodu utajení v něm obsažených informací)
4. Nevýdělečné zveřejňování díla nabyvatelem v souladu s ustanovením § 47b zákona č. 111/ 1998 Sb., v platném znění, nevyžaduje licenci a nabyvatel je k němu povinen a oprávněn ze zákona.

Článek 3
Závěrečná ustanovení

1. Smlouva je sepsána ve třech vyhotoveních s platností originálu, přičemž po jednom vyhotovení obdrží autor a nabyvatel, další vyhotovení je vloženo do VŠKP.
2. Vztahy mezi smluvními stranami vzniklé a neupravené touto smlouvou se řídí autorským zákonem, občanským zákoníkem, vysokoškolským zákonem, zákonem o archivnictví, v platném znění a popř. dalšími právními předpisy.
3. Licenční smlouva byla uzavřena na základě svobodné a pravé vůle smluvních stran, s plným porozuměním jejímu textu i důsledkům, nikoliv v tísni a za nápadně nevýhodných podmínek.
4. Licenční smlouva nabývá platnosti a účinnosti dnem jejího podpisu oběma smluvními stranami.

V Brně dne: .....

....

Nabyvatel Autor

Abstrakt

Tato bakalářská práce představuje agilní metodologii zvanou vývoj řízený testy a ilustruje její použití na ukázkovém příkladě pomocí testovacího nástroje. Během ilustrace jsou vysvětleny techniky sloužící k dosažení cíle softwarového projektu. Dále je diskutován vliv vývoje řízeného testy na kvalitu sofwarového produktu.

Klíčová slova

vývoj řízený testy, testování, refaktorování, agilní metodiky, softwarové inženýrství

Abstract

This bachelor's thesis introduces an agil method called test-driven development and illustrates it by an example using a testing tool. The way to reach objectives of software project is explained during the illustration. Further, the effect of test-driven development on the quality of software product is discussed.

Keywords

test driven development, testing, refactoring, agile methods, software engineering, tdd

Citace

Dušan Navrátil: Vývoj software řízený testem, bakalářská práce, Brno, FIT VUT v Brně, 2008

Vývoj software řízený testem

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Jaroslava Zendulky. Použil jsem pouze podklady uvedené v přiloženém seznamu.

.....
Dušan Navrátil
12. května 2008

Poděkování

Děkuji vedoucímu práce panu Jaroslavovi Zendulkovi za cenné připomínky a odbornou pomoc. Také bych rád poděkoval mé rodině a mým blízkým za velkou podporu.

© Dušan Navrátil, 2008.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

Kapitola 1

Úvod

Softwarové inženýrství je poměrně mladý obor, který se neustále vyvíjí. Vznikl jako reakce na softwarovou krizi, jejíž následky jsou patrné i v současnosti. Softwarové inženýrství nabývá na důležitosti a hledá nové způsoby vývoje softwaru. Klasické metodologie se prediktivním způsobem snaží plánovat detaily vývoje. Tento plán je optimalizován pro původní zadání projektu a jakékoliv změny či výchylky od původního cíle pro něj mohou znamenat zahození veškeré práce a nutnost začít znovu. Striktní oddělení fází vývoje způsobuje, že závazky vůči zákazníkovi jsou usneseny příliš brzo a je poté těžké reagovat na změny v požadavcích.

Naproti „plány řízenými“ metodikami vznikájí koncem minulého století metodologie, které se snaží řešit tyto problémy. Narozdíl od klasických metodik neplánují vývoj na měsíce dopředu. Jejich snahou je uspokojit měnící se požadavky zákazníka a zároveň udržet náklady projektu na předem stanovené hranici.

V druhé kapitole se seznamíme s agilními metodikami. Zdůrazníme jejich rozdíly oproti tradičním postupům. Zaměříme se na jednoho ze zastupců těchto metodik – vývoj řízený testy. Popíšeme proces a přínos TDD[1].

Ve třetí kapitole si uvedeme dostupné prostředky na podporu vývoje řízeného testy. Jedná se o hlavně o rodinu testovacích rámců xUnit.

V čtvrté kapitole si názorně demonstrujeme použití TDD v praxi. Ukázková aplikace bude implementována v programovacím jazyce C++. Uvedeme si techniky typické pro TDD. Předvedeme si také použití testovacího rámce cppUnit.

[1] Test Driven Development - testy řízený vývoj

Kapitola 2

Testy řízený vývoj (TDD)

V této kapitole charakterizuji základní rysy testy řízeného vývoje. Nejdříve se zmíním obecně o agilních metodologiích, z nichž TDD vychází. Při zpracování této kapitoly jsem čerpal hlavně z [7] a [9].

2.1 Agilní metodiky

Jeden z důležitých požadavků na vývoj softwaru je rychlost dodání softwarového produktu. Klasické metodiky softwarového inženýrství ale kladou akcent na specifikaci požadavků, podrobnou analýzu a perfektní návrh. Tento přístup znatelně zpomaluje vývoj softwarového produktu. Tyto fáze totiž obvykle trvají dlouhou dobu a jakékoliv pochybení v některé z nich může mít v pokročilých fázích projektu fatální důsledky. Přesto tyto metodiky pomohly překonat softwarovou krizi v 70. a 80. letech. Nicméně ukázalo se, že nedovedou řešit určité typy problemů. Situace na trhu byla tehdy jiná než dnes. Požadavek na rychlost nebyl tak markantní. Dodavatele sofwaru často nedodržovali termín dodávky a zákazník si raději počkal na kvalitní produkt. Vývoj softwarového produktu se také rapidně prodražoval.

Když to srovnáme se současnou situací na trhu, všimneme si mnoha rozdílů, které nás nutí pohlížet na vývoj softwaru z jiného úhlu. Konkurence na poli softwarových produktů je nyní čím dál více. Vývojem se zabývá mnohem více lidí. Vytvořit dokonalou aplikaci už často není tím nejdůležitějším vodítkem při vývoji softwaru. Použitím klasických metodik můžeme sice vytvořit aplikaci se skvělou funkcionalitou, ale než projdeme kroky klasického vývoje softwaru, konkurence může mít již dávno vyvinutou aplikaci, na kterou byly kladeny stejné požadavky. Pro určité typy produktu (např. webová aplikace) je požadavek na rychlost velice důležitý a může často rozhodnout o tom, bude-li mít produkt úspěch či nikoli. Je důležité si uvědomit, že příčina případného neúspěchu projektu pak ale netkví v softwarovém inženýrství ale v managementu projektu, zvolení nevhodného postupu či metodiky.

> *Klasické softwarově-inženýrské techniky jsou založeny na rigorózních postupech, které vycházejí z důkladné analýzy problému a propracovaného, neprůstřelného návrhu. Nelze zpochybnit, že tento model je v obecném případě nejlepší. Vždy lze očekávat, že výsledek vzešlý z podrobného splynutí s problémem bude kvalitnější než plod vypěstovaný na neprozkoumané půdě.* [7]

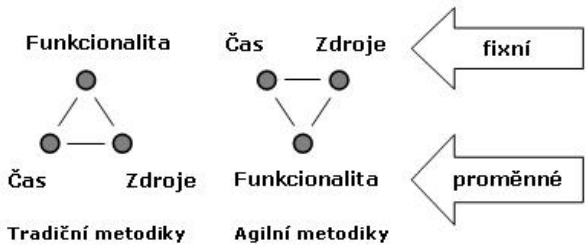
Naproti klasickým metodikám existují metodiky *agilní*, které se nesnaží plánovat vývoj dopředu. U těchto metodik je analýza samozřejmě také velice důležitá. Způsob, jak se k ní

přistupuje, je ale odlišný. Agilní metodiky se snaží, aby analýza byla kvalitní a zároveň rychlá, a tvrdí že se tyto přívlastky nevylučují. Typicky se nejdříve vytvoří hrubý návrh - zakladní objekty. Průběžně se tento návrh i konkrétní funkčnost mění podle pozadavků klienta. Klasické fáze jako analýzu a návrh se snaží integrovat s implementací a otevřít tak vývoj změnám, které mohou znamenat jednodušší přesto fungující variantu.

Tyto metodiky vychází z přirozených principů, které jsou vývojářům blízké. Dobrý pocit z vykonané práce je pro programátora důležitý, podporuje jeho sebedůveru a odvahu provádět změny.

2.1.1 Rozdíl oproti tradičnímu přístupu

Odlišné přístupy lze nejlepé ilustrovat následujícím obrázkem, který porovnává pohled na základní proměnné při vývoji softwaru.



Obrázek 2.1: Rozdílné pojetí proměnných při vývoji softwaru(převzato z [7])

Při vývoji použitím klasických metodik jsou požadavky na funkcionalitu od začátku přesně definovány a hlavním cílem je splnit tyto požadavky. U takového projektu se těžko odhaduje cena výsledného produktu a doba potřebná pro vývoj.

Naopak u agilního vývoje jsou stanoveny nejvyšší možné náklady a termín, kdy nejpozději má být aplikace hotova. Požadavky na aplikaci se průběžně konzultují se zákazníkem, který má tak možnost je přehodnotit a určit priority tak, aby se projekt vešel do předem definovaných atributů.

2.1.2 Základní principy

Na základě manifestu[1] agilního vývoje softwaru se v následujících odstavcích pokusím formulovat zakladní principy agilních metodik, které jsou společné pro konkrétní zástupce.

Inkrementální vývoj s častými dodávkami Hlavním cílem je uspokojit zákazníka. Vývoj je prováděn inkrementálně s krátkými iteracemi. Fungující software je dodáván často, aby zákazník mohl mít z něho již nějaký užitek a poskytoval nám zpětnou vazbu. Vývoj je otevřen změnám, neprovádí nic, co není nutné. Změny mohou totiž pro zákazníka znamenat konkureční výhodu.

Spolupráce se zákazníkem Narozdíl od tradičních přístupů se specifikace požadavků nepovažuje za nějaký neměnný dokument. Agilní metodiky kladou důraz na spolupráci zákazníka s vývojáři. Ideálním způsobem, jak toho docílit, je zařazení zákazníka do vývojového týmu. Tím nemůže dojít k tomu, že by výsledný produkt nesplňoval očekávání zákazníka.

[1] Přesné znění je uvedeno v [1]

Zdrojový kód jako nositel informace K dokumentaci vývoje a aktuálního návrhu nejsou používány UML diagramy či jiné modelovací techniky. Jediným spolehlivým nositelem informace je zdrojový kód. S jeho psaním se začíná již velmi brzy, ihned po vytyčení základních požadavků na produkt potřebných pro hrubý návrh. Z tohoto důvodu je doporučeno používat jednotná pravidla pro psaní kódu.

Vzájemná konverzace v týmu Narozdíl od psaní rozsahlých dokumentací, se porozumění problému docílí vzájemnou konverzací. Vyvojáři musí být schopni přímé a osobní komunikace v týmu. Naprostá většina problémů spojená s vývojem tkví právě v nefungujicí komunikaci. V některých agilních metodikách je dokonce jeden člen týmu pověřen udržováním komunikace a důvery v týmu a odstraněním komunikačních bariér.

Kvalitní návrh I v agilním vývoji je perfektní návrh velice důležitý. Není však samostatnou etapou procesu ale je integrován do každodenní činnosti. Je nutné jej měnit a to se samozřejmě projeví i ve zdrojovém kódu. S ohledem na tuto skutečnost je snaha, aby byl návrh co nejjednodušší. Proto je implementováno jenom to, co je opravdu potřeba, o nic více, o nic méně.

Předvídatelný vývoj Tyto metodiky mají za cíl udržet vývoj se stanovenými atributy(čas a náklady). Nepočítá s presčasy, pretěžováním pracovní síly, které nakonec vedou k nižší produktivitě.

Metodiky nejsou přesně definovány, protože se neustále vyvíjejí. Častou praktikou v týmech bývá organizování schůzek, na kterých se diskutují možnosti efektivnější práce.

2.2 Testy řízený vývoj

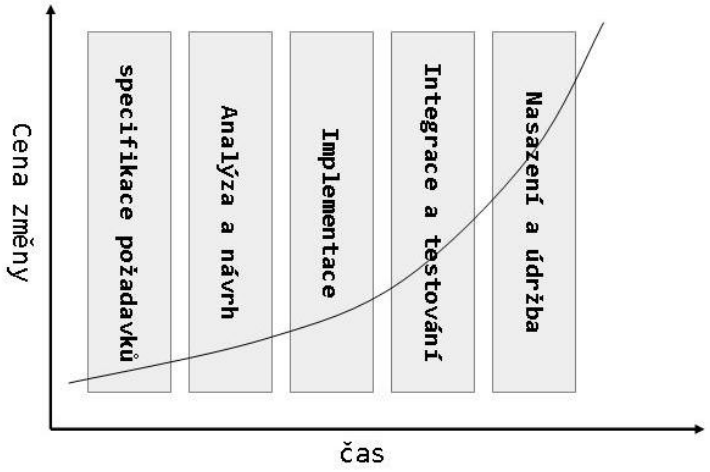
Testování je zajisté nezbytná činnost při vývoji softwaru, ať již používáme jakoukoli metodiku. Nemůžeme dovolit aby výsledný produkt otestoval až zákazník. Je snaha zajistit jeho kvalitu a testovaní je pro tento účel jedna z nejúčinnějších metod. Můžeme na testování nahlížet ze dvou různých pohledů. Zatímco tradiční metodiky chápou testování jako jednu z fází vývoje, u TDD je testování spjato s celým procesem a je základní technikou pro dosažení cíle.

V klasickém vodopádovém modelu je testování zasazeno jako předposlední fáze vývoje, která v porovnání s ostatnímy fázemi trvá dlouhou dobu. Na obr.2.2 je vidět graf křivky, která znázorňuje „cenu změny“ požadavků na softwarový produkt. Tedy pokud by zákazník změnil požadavky na aplikaci, nebo by se zjistilo, že analýza problému neproběhla korektně a návrh tudíž neodpovídá požadavkům na aplikaci, cena na „překopání“ projektu by byla dosti vysoká.

TDD pomáhá tuto cenu pomocí agilních technik redukovat. Je to, jako bychom obrázek otočili o 90°, ale přesto postupovali zleva doprava. To znamená, že bychom provedli z každé etapy vývoje vždy jen malou část. Testování, stejně jako i třeba návrh, je proloženo celým vývojem. TDD nám tedy dovolí být flexibilnější vůči změnám.

2.2.1 Charakteristika TDD

Základní činností při použití metodiky TDD je psaní testů. TDD zachází v tomto smyslu až do takových důsledků, kdy říká, že první krokem k přidání nové funkcionality je přidání



Obrázek 2.2: Růst ceny SW produktu v jednotlivých fázích klasického vývoje

testu, který přislušný kód ověří. Ačkoliv tento krok připadá zdánlivě nelogický, má svůj opodstatněný důvod. Jakmile totiž musíme napsat nejprve test, nutí nás to promyslet architekturu, predtím než začneme „zběsile“ programovat. Tento krok pomáhá uvědomit si, co si vlastně pod danou funkčnosti představujeme, a vede tak ke kvalitnějšímu návrhu.

Jakmile je přidán nový test, můžeme začít resp. pokračovat s implementací. Kód upravíme tak, aby pokryl pouze tento test, o nic více, o nic méně. Snažíme se najít co nejrychlejší způsob jak napsat příšlušný kód tak, aby test prošel úspěšně.

V porovnání s tradičním testováním, se TDD snaží u rizikových projektů o důsledné testování každé řádky kódu. Takové jistoty mohou tradiční metodiky dosáhnout jen stěží. Pokud jde o pokrytí testy, má TDD lepší výsledky než klasický vývoj.

Ron Jeffries vyjadřuje vztah TDD k návrhu a testování takto:

> *„ Cílem TDD je čistý kód, který funguje.“* [5]

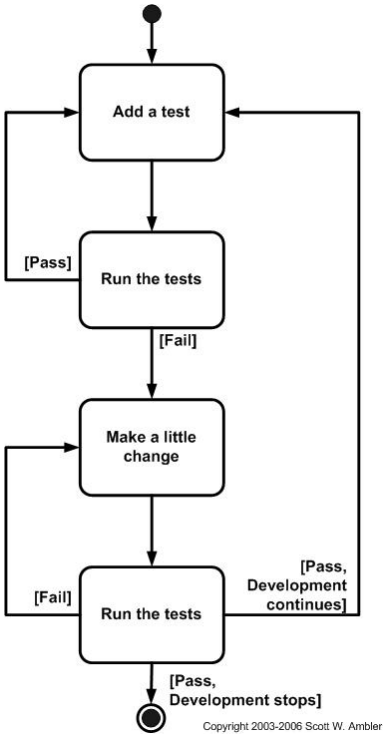
Pravidla pro vývoj TDD jsou možná jednoduchá k porozumění, ale není tak snadné si na ně navyknout a dodržovat je. Programátory často svádí připsat kousek kódu bez toho, aniž by pro něj nejdříve napsali test a spustili jej. To znamená, že je velice důležité mít disciplínu a odhodlání používat TDD.

2.2.2 Cyklus TDD

Proces TDD prochází nasledujicími kroky:

1. **Přidání testu**
   Prvním krokem je vytvoření testu, který by měl být jednoduchý a u kterého víme, že selže. Mělo by být na první pohled zřejmé, co testuje, jaké jsou jeho spravné výstupy k odpovídajícím vstupům. Pokud se nám zdá kód testu příliš dlouhý (např. dlouhé nastavování objektů), indikuje to nekvalitní návrh programu.

2. **Spuštění všech testů**
   Tento krok by měl sloužit k utvrzení toho, že test selže, a že je tedy potřeba upravit kód tak, aby při příštím spuštění test prošel. Vzhledem k rozsáhlosti projektu je často lepší spouštět jen sadu testů týkajicích se jen jednoho menšího celku aplikace.



Obrázek 2.3: TDD cyklus (převzato z [3])

3. **Úprava kódu**
   Máme selhávající test. Nyní ho potřebujeme co nejrychleji zprovoznit. Tento krok by neměl trvat příliš dlouho. Snažíme se jen o to, abychom co nejrychleji získali „zelený ukazatel“. Přidaný kód ve skutečnosti nemusí představovat akce provádějící výpočet. Často se v tomto kroku používají techniky jako *falešná implementace* či *triangulace*. Tyto techniky budou popsány a demonstrovány v kapitole 4 při ilustraci vývoje použitím TDD.

   V zásadě nejde o to, aby byl kód elegantní. Vyčištění kódu a odstranění duplicit nalezá v cyklu TDD také své místo, ale ne zde.

   Kent Beck v [5] uvádí tyto zásady na pravou míru s tím, že délku tohoto kroku si můžeme zvolit podle toho, jak si věříme. Někdy si budeme jisti, jaký kód danou funkčnost představuje. Potom není důvod nenapsat zřejmou implementaci. Techniky jako falešná implementace tu nejsou od toho, aby nás zpomalovaly, ale proto, aby nám pomohly provádět menší a stabilnější kroky. Často jsme ale nemile překvapeni, když je následné spuštění testu neúspěšné. Poté je potřeba se opět vrátit k malým krůčkům.

4. **Spuštění automatizovaných testů**
   Tento krok oveřuje, zda provedená úprava kódu byla úspěšná či nikoli. Pokud test selhal musíme opravit implementaci tak, aby test prošel. Jinak nemůžeme pokračovat dále.

5. **Refaktorizace**
   Do této fáze vstupujeme s pocitem, že máme zdánlivě funkční kód. Prvotní imple-

mentace funkčnosti byla provedena s cílem co nejdříve získat „zelený ukazatel“. Kód obsahuje duplicity, používá konstanty místo proměnných apod. Refaktorizace znamená jakoukoli změnu, která přispěje k čitelnosti či lepší struktuře.

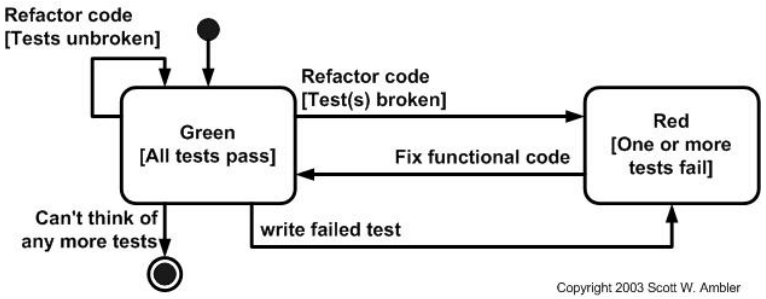
> Refaktorizace je provádění změn v již existujícím, funkčním kódu bez změny jeho vnějšího chování. Cílem je zlepšit vnitřní strukturu. [4]

TDD pojímá refaktorizaci zajimavým, trochu odlišným způsobem. Odporuje pravidlu, že refaktorizace nemění sémantiku programu, a zachovává pouze sémantiku testovacích případů. Uvažuje pouze testy, které máme napsané a které prošly a ostatní testy, které by neprošly, jsou pro tuto chvíli nezajímavé. To znamená, že když refaktorováním chceme odstranit nějaký problém, který je nám zřejmý i bez použití testů, nezabýváme se tím, dokud nebudeme mít tento problém pokrytý testem.

Pro provádění těchto kroků je k dispozici mnoho nástrojů, které umožňují automatizaci těchto činností. Mnoho z nich je přímo integrováno do vývojových prostředí. Pro efektivní refaktorizaci je výhodou znalost alespoň základních návrhových vzorů.

2.2.3 Červený vs. zelený ukazatel

TDD nám pomocí testů dává možnost vidět, v jaké fázi se nachazíme. Na obrázku 2.4 je stavový diagram, znazorňující přechody mezi zeleným a červeným stavem.



Obrázek 2.4: Přechody mezi stavy TDD (převzato z [3])

Na začátku vývoje nemáme vytvořen žádný test a vstupujeme tedy do „zeleného stavu“, který nám říká, že všechny testy proběhly úšpěšně. Napsáním a spuštěním nového testu se dostáváme do „červeného stavu“, kdy nám alespoň jeden test neprošel. Úpravou kódu a tedy zprovozněním testu se dostaneme zpátky. Při refaktorizaci se snažíme provádět malé změny tak, aby další spuštění testu bylo úspěšné. Pokud ale některý z testů neprojde, vracíme se do „červeného“ stavu a musíme kód upravit tak, aby test prošel. Testovací proces je ukončen, když nás již nenapadá nějaký další test.

2.2.4 Testy jako dokumentace

TDD jakožto zástupce agilních metodik vychází z principů, které jsou vyvojáři blízké. Programátor často nerad čte dokumentaci, aby zjistil co dělá daný modul, třída či funkce. Raději poohlédne po nějakém ukázkovém příkladě, který si může spustit a z jehož zápisu se dozví, jak se tato třída používá. Testovací případ je přesně takovým ukázkovým příkladem. Zároveň mohou testy představovat součást specifikace požadavků. Deklarují přesně to, co od dané aplikace zákazník požaduje.

Testy možná nemohou zcela nahradit dokumentaci ale zajisté mohou být jejím důležitým doplňkem. Zvyšují efektivitu čtení dokumentace a porozumění obsahu.

2.2.5 Omezení

Použití TDD je obtížné při programování uživatelského rozhraní, kdy je nutný nějaký nástroj pro automatizované spouštění testů založených na uživatelských událostech. Jeden z dostupných nástrojů uvádím v 3.2.

Další obtíž naskytne v případech, kdy není možné oddělit programové jednotky a testovat je izolovaně. Příkladem může být například použití databáze, webové služby či nějakého jiného externího procesu. Tento problém lze řešit pomocí rozhraní pro externí přístup. Toto rozhraní bude implementováno dvěma způsoby: jedna implementace bude přistupovat ke skutečnému zdroji, např. databázi, ta druhá bude bude využívat falešný objekt. V anglickém žargónu se falešný objekt rozlišuje na:

Fake object implementuje stejné rozhraní jako objekt, který simuluje, a vrací předem připravené výsledky.

Mock object se liší tím, že navíc obsahuje aserce, které kontrolují např. validitu vstupních argumentů při volání nějaké metody.

Oba typy objektů, vracející zdánlivě data zadaná uživatelem, či data z databáze, můžeme použít k izolaci testovaného kódu od externího přístupu.

Ačkoli tímto způsobem dosáhneme lépe testovatelného a znovupoužitelného kódu, nevýhodou tohoto řešení je to, že implementace externího přístupu není pokryta testy. Proto je zapotřebí vytvořit další, *integrační testy* (viz kap. 3.1), které ověří skutečný přístup k externímu procesu. [12]

Kapitola 3

Nástroje pro TDD

Smysluplné používání TDD požaduje nějaký specifický nástroj pro testování. V této kapitole se zaměřím na to, jakou podporu by měl takovýto nástroj poskytovat a s jakými druhy testů se v TDD setkáváme. Výběr kvalitního testovacího rámce má při vývoji zajisté velký vliv, protože nám může ušetřit spoustu času.

Dále uvedu dostupné nástroje, jejich výhody a nevýhody oproti ostatním. Nakonec přiblížím koncept asi nejpoužívanějšího testovacího rámce – xUnit.

3.1 Požadavky na nástroj TDD

U testovácího nástoje používáného pro TDD je potřeba, aby umožnil při spuštění provést všechny testy. Takový *automatizovaný* nástroj je pro programátora nepostradatelný, neboť mu ulehčuje pravidelné a časté spouštění testů.

Dalším důležitým hlediskem je strukturalizace testů. V rámci TDD mluvíme o *testování jednotek*. Jde o způsob, jakým testovat individuální jednotky zdrojového kódu. Cílem je izolovat jednotlivé části programu a otestovat funkčnost každé této části zvlášť. Programovou jednotkou se rozumí nejmenší testovatelná část aplikace. Obvykle záleží na používaném programovacím paradigmatu, jakou jednotku máme přesně na mysli. V procedurálním programování si pod tímto pojmem můžeme představit funkci, proceduru nebo samotný program. V objektově orientovaném programování je to většinou třída. Testy pro určitou programovou jednotku by nikdy neměly překročit hranice této jednotky. Jiným způsobem je testování spolupráce jednotlivých částí systému neboli *test integrity*.

Testy se mohou také lišit na základě viditelnosti vnitřní struktury. *Black-box testy* (také *funkční*) kontrolují pouze, jestli výstupy dané funkce či metody odpovidají zadaným vstupům. Již se nezajímají o to, jakým způsobem bylo těchto vysledků dosaženo. Není známo, kterými větvemi běh programu procházel. Naproti tomu, se *white-box testy* (také *strukturní*) snaží odstranit pomyslnou roušku dané jednotky. Testuje všechny cesty, které mohou být vykonány. A to i odezvu na chybné vstupy. Psaní těchto testů je mnohem náročnější avšak poskytují více informací. Při ilustraci vývoje TDD v kap. 4 se podíváme na některé techniky, které takové testování umožňují.

Testování integrity představuje ověření spravného propojení menších celků aplikace – modulů, jednotek. Následuje po testování jednotek a předchází testování systému. Při testování integrovaných částí je použita metoda black-box testing.

TDD nástroj se často využívá v kombinaci se systémem správy verzí. Pokud testy selžou, je možné se vrátit k dřívější verzi programovacího kódu, která úspěšně prošla testy. Tento

způsob může být často produktivněší než použití ladícího nástroje. [8]

3.2 Dostupné nástroje

Pro vývoj řízený testy je zřejmě nejúspěšnějším zástupcem rodina testovacích nástrojů xUnit. Tento nástroj je integrován do některých vývojových prostředí. Typickým příkladem je *NetBeans*, který je určen hlavně pro programovací jazyk Java. Komponenta *jUnit* je však přítomna pouze v komerčním balíku. Od společnosti Microsoft je možné použít nástroj *NUnit* pro programovací jazyk C#. Komponeta je integrována do vývojového prostředí Visual Studio, kde kromě podpory testování, je prostor i pro snadnou a automatickou refaktorizaci. Existují vsak i open-source projekty. Jako připojitelná knihovna pro jazyk C++ je imlementován *cppUnit*. Je možnost tyto knihovny zavést do vývojových prostředí jako například C++ Builder. Podobně je tomu tak i u *phpUnit* pro jazyk PHP nebo *Test::Unit* pro Ruby.

Používání TDD je často trochu závislé na vývojovém prostředí, ve kterém aplikaci vytváříme. Dále se liší typem projektu. Pokud se jedná o webovou nebo formulářovou aplikaci, je dobré mít testy pro uživatelské rozhraní, které bychom mohli spouštět automatizovaně. Takovéto rozšířené možnosti testování nabízí například MS Visual Studio Team System. Návod, jak jednoduše vytvořit testový případ webové aplikace, je uveden v [10]. Princip spočívá v záznamu událostí vyvolaných uživatelem a jejich přehrání při spuštění testu.

3.2.1 xUnit

Rodina testovacích ramců xUnit je již v oblasti testování jednotek známa dlouho. Koncept xUnit je jednoduchý, přesto poskytuje spoustu možností pro efektivní a důkladné testování. Mnoho nástrojů(viz [11]) pro testování jednotek je založeno právě na tomto konceptu.

Obvykle poskytuje tyto prostředky:

Test case: Jde o *případ*, který testuje konkrétní chování části aplikace. Typicky je prezentován jako funkce, v jejimž těle se vyskytují různé druhy asercí.

Test fixture: tvoří jakousi základnu – *přípravu* pro testovací případy. Její součástí jsou metody `setUp()` a `tearDown()`. Metoda `setUp()` slouží k inicializaci společných objektů pro více testů a k alokaci zdrojů. Je spuštěna vždy před samotnou sadou testovacích případů. Naopak metoda `tearDown()` je provedena po ukončení sady testů a slouží k uvolnění zdrojů.

Test suite: je sada či množina sad testů. Poskytuje možnost shromáždit testy, které se váží k dané programové jednotce.

Test runner zajišťuje *vykonávání testů* a zobrazení výsledků v textovém nebo grafickém režimu.

Ukázka použití těchto prostředků v jazyce C++ bude uvedena v kapitole 4 při ilustraci vývoje řízeného testy. Konkrétním nástrojem bude knihovna cppUnit (viz [2]).

Snahou xUnit je jednoduché používání. Jeho implementace by neměla být příliš složitá. Kent Beck v [5] dokonce dopuručuje při seznamování s novým programovacím jazykem

implementovat vlastní xUnit i v případě, existuje-li již dostupná verze. Pokud si totiž vytvoříme takový nástroj, jeho ovládání pro nás určitě nebude problémem. Tímto způsobem se lze také rychle seznámit s nástroji a vlastnostmi daného jazyka.

Kapitola 4

Ilustrace vývoje TDD na ukázkové aplikaci

Následující ukázková aplikace je implemetována výhradně pro demonstrační účely. Její funkčnost proto nebude úplná. Pro praktické použití by byla nutná optimalizace pro výkon. Taktéž by bylo nutné zajistit bezproblémový překlad na více překladačích.

Implementace některých úseků aplikace bude přeskakována, aby nedošlo ke zbytečnému opakování postupů. Následující výklad je pouze částí aplikace. Vývoj aplikace resp. některých jeho částí se může zdát poněkud zdlouhavý a některé obraty triviální a trochu zbytečné, avšak cílem této kapitoly je ukázat způsob, jak TDD dokáže postupovat po malých implementačních změnách. Porozumění hlubšího smyslu těchto technik lze poté využít i v náročnějších projektech.

Ilustrační výklad je členěn tak, že každá podkapitola představuje určitou techniku TDD, doporučení nebo pouze provádění typického implementačního postupu. Pro názvy identifikátorů jsem zvolil český jazyk, aby byl výklad lépe čitelný a bylo snadnější jej porozumět. Následující úseky kódu slouží zároveň jako demonstrace testovacího rámce cppUnit.

Vzhledem k typu ukázkové aplikace, je nutná pouze znalost programovacího jazyka C++ a matematická znalost lineární algebry.

Metodika TDD je úzce spjata s implementací. Proto uvedení alespoň jednoduché ukázky programovacího kódu, zejména psaní testových případů, je pro správné pochopení dosti důležité. Návrh se vyvíjí postupně. Není proto na začátku uveden žádný UML diagram ale výklad obsahuje konkrétní úseky kódu z ukázkové aplikace. Pro představu je alespoň uvedena velmi stručná specifikace požadavků. Ta ale ovšem při vývoji použitím TDD nebude brána jako pevná směrnice, ale bude možné ji v procesu vývoje měnit.

Během ilustrace vývoje se dopustím několika programátorských chyb. Neopomenu však demonstrovat, jakým způsobem TDD s těmito chybami zachází a jak je řeší.

Potřebné informace o předvedených technikách jsem čerpal z [5]. Jejich souhrn je uveden na konci kapitoly společně s uvedením možného pokračování vývoje aplikace. Dále jsou zde diskutovány dosažené výsledky na základě osobních zkušeností s TDD během vývoje ukázkové aplikace.

4.1 Specifikace požadavků

Je požadováno vytvoření dynamicky připojitelné knihovny, která poskytuje základní operace s vektory a maticemi. Tato knihovna je určena pro práci s celými čísly. Pro uchování

dat má být použit datový typ `integer`.

Budeme předpokládat, že fiktivní zákazník nemá přesnou představu o aplikaci a tak se budou další konkrétnější požadavky na aplikaci rysovat až v průběhu vývoje knihovny na základě konzultací s tímto zákazníkem.

Prvně by knihovna měla splňovat tyto základní požadavky:

- Možnost přístupu(čtení resp. zápis) k vektorům a maticím.
- Přítomnost základních operací – sčítání, odčítání, součin matic resp. vektorů. Dále velikost vektorů.
- Porovnání na rovnost vektorů resp. matic.
- Použití vektorových operací u řádku, resp. sloupce matice.

4.2 Postup implementace knihovny

Aplikace je vyvíjena ve Visual Studiu 2005. Podpora pro testování v této verzi není integrována, takže bylo nutné použít knihovnu pro testovací rámec – cppUnit. Koncept xUnit je popsán v 3.2.1. K testování jednotky bude sloužit jako testovací přípravek třída `TestFixture`. Její metody představují jednotlivé testovací případy. Specializací této třídy dostaneme tedy náš vlastní testovací přípravek. Příkladem může být třída `Vektor_Test`, která testuje metody třídy `Vektor`.

4.2.1 Úvodní test

Jako první test, kterým začínáme cyklus TDD, je dobré zvolit něco jednoduchého. Operace, kterou budeme testovat by neměla být náročná na implementaci. Nejlepší je zvolit operaci, která nic nepočítá. Při psaní úvodního testu chceme řešit co nejméně otázek. Krok TDD by měl zabírat pouze minuty. To, co musíme řešit téměř vždy, je otázka: „Kam operace patří?“. Můžeme se ale vyhnout úvaze nad výpočtem, který operace ukrývá. Krok TDD cyklu tak můžeme zkrátit volbou vstupů a výstupů, které je jednoduché odhadnout. V ukázkové aplikaci chceme například nejdřív implementovat operaci, která nám vrátí rozměr daného vektoru. Nezapomeňme, že první, co v TDD cyklu provedeme, je psaní nového testu. Při tomto kroku si představíme základní strukturu aplikace.

```
void Vektor_Test::TestRozmeruVektoru()
{
  CPPUNIT_ASSERT( 1 == Vektor(1).Rozmer() );
}
```

Nyní spustíme test a sledujeme, jak selže. Na první dojem se spuštění testu, který nemůže fungovat, zdá dosti nesmyslné. Kent Beck v [5] říká, že tento krok nám pomůže nalézt *konkrétní měřítko neúspěchu*. Naš cíl se zúží na zprovoznění tohoto testu.

Tento test dokonce ani nejde zkompilovat. Z výpisu překladače zjistíme, které chyby musíme odstranit a co je potřeba k tomu, aby se kód zkompiloval.

```
Vektor_test.cpp(12) : error C2228:
    left of '.Rozmer' must have class/struct/union type is ''unknown-type''
Vektor_test.cpp(12) : error C3861:
    'Vektor': identifier not found
```

Musíme zavést třídu `Vektor` a její konstruktor. Dále je nutné přidat ke třídě Vektor metodu `Rozmer()`. Provádíme co nejmenší kroky, a tak implementuje jen to nejnutnější, a to i na úkor návrhu či skutečné funkčnosti. Cílem je získat „zelený ukazatel průběhu“, abychom mohli pokračovat.

```
class Vektor
{
public:
  Vektor(int rozmer) {};
  int Rozmer() { return 1; };
public:
  ~Vektor(void);
};
```

Nyní po úspěšném spuštění testu můžeme provést úpravu kódu zavedením soukromé složky `rozmer`. V implementaci metody `Rozmer()` vrátíme hodnotu této složky. Pokud test projde, můžeme se pustit do dalšího testu.

4.2.2 Testování výjimek

Z definice vektoru je známo, že rozměr vektoru může být pouze přirozené číslo. V našem příkladě bychom tedy chtěli aby byla při zadání záporného nebo nulového rozměru vektoru vyvolána výjimka.

Testování výjimky se trochu liší od klasických testů. Jeden ze způsobů, jak toto chování naimplementovat, je zachytit konkrétní výjimku a ignorovat ji. Pokud operace nevyvolá očekávanou výjimku, necháme test „selhat“ zavoláním cppUnit metody `fail()`. Obdobného výsledku lze dosáhnout také použitím makra `CPPUNIT_ASSERT_THROW`.

```
  CPPUNIT_NS::Message cpputMsg_( "ocekavana vyjimka nebyla vyvolana" );
  try {
     Vektor v(0);
     CPPUNIT_NS::Asserter::fail( cpputMsg_, CPPUNIT_SOURCELINE() );
  }
  catch (NeplatnyRozmerVektoru &) {
  }
```

Test při spuštění selže a jako příčina neúspěchu se objeví námi zadaná chybová zpráva. Tělo konstruktoru doplníme o if konstrukci a se zprovozněným testem můžeme pokračovat dál.

```
Vektor::Vektor(int rozmer) : rozmer(rozmer)
{
  if (rozmer < 1) trow NeplatnyRozmerVektoru();
}
```

Vektor již obsahuje důležitou vlastnost „rozmer“ ale zatím neobsahuje žádné složky. Zkusíme to otestovat. Použijeme metodu `setUp` třídy `TestFixture`, abychom alokovali pomocné pole, které budeme používat i v dalších testech. Připomenu, že tato metoda je volána před spuštěním sady testů. Je tedy vhodná k přípravě a nastavení zdrojů(např. otevření souboru nebo alokace paměti). Její použití ale není vždy vhodné. Často nechceme aby se stav

alokovaných objektů měnil podle toho, v jakém pořadí jsou spuštěny testy. Testové případy by na sobě neměly být závislé. Uvedení inicializace v testovém případu také umožňuje lepší čitelnost testu.

```
void Vektor_Test::setUp()
{
  poleCisel = new int[2];
  poleCisel[0] = 4;
  poleCisel[1] = 3;
}

void Vektor_Test::TestObsahuSlozek()
{
  Vektor v1(2, poleCisel);
  CPPUNIT_ASSERT_EQUAL( 4, v1.slozky[0] );
  CPPUNIT_ASSERT_EQUAL( 3, v1.slozky[1] );
}
```

Rychle zprovozníme test tak, že přidáme veřejnou položku `slozky` do třídy `Vektor`. V konstuktoru potom tento atribut inicializujeme.

```
class Vektor {
  ...
  public:
   int *slozky;
  ...
}

Vektor::Vektor(int rozmer, int *pole) : rozmer(rozmer), slozky(pole)
{ ... }
```

Oba testy proběhly úspěšně. Naše implementace se opírá o vytvoření mělké kopie pole čísel reprezentujících vektor. Vytvoření mělké kopie může být promyšleným záměrem, ale často bývá spíše programátorskou chybou. Jelikož víme, že závislost mezi pomocným polem a nově vytvořeným vektorem bude s největší pravděpodobností dělat v budoucnu problémy, měli bychom ihned implementaci opravit. Při vývoji softwaru pomocí TDD není striktním pravidlem refaktorovat po každém zprovoznění testu. Nevíme dopředu, jak bude naše třída `Vektor` vypadat v budoucnu a jestli se závislost skutečně negativně projeví. Zatím nám to totiž žádný test nepotvrdil. Budeme tedy ignorovat možné důsledky tohoto činu a necháme vývoj pokračovat dál s přesvědčením, že nás na to jeden z budoucích testů upozorní.

Abychom mohli dále pokračovat, není nutné mít správný návrh, ale je nutné mít zprovozněné všechny testy. Dále jsme nechali atribut `slozky` verejně přístupný, což odporuje pravidlu správného návrhu v OOP – zapouzdření objektu. Tento atribut nám bude prozatím užitečný pro testování, dokud nebudeme mít implementovány vlastní přístupové metody. Nyní si jej tedy nebudeme všímat, ale tuto vadu si poznamenáme, abychom později nezapoměli refaktorovat.

4.2.3 Triangulace

Technika triangulace patří mezi metody, kdy se snažíme rychle „získat zelený ukazatel“ a abstrahovat výpočet po malých krocích.

Další operací, kterou bude třída `Vektor` vybavena, bude rovnost dvou vektorů. Můžeme využít vlastnost jazyka C++ – přetěžování operátorů. Nejdříve budeme postupovat od té nejjednodušší podmínky pro rovnost dvou vektorů: jejich rozměry musí být stejné. Test pro operaci bude následující:

```
void Vektor_Test::TestRovnosti()
{
  Vektor v1(2, poleCisel);
  Vektor v2(2, poleCisel);
  CPPUNIT_ASSERT_EQUAL( true, v2 == v1 );
}
```

V tělu funkce vrátíme očekávaný výsledek abychom si byli jisti, že test projde úspěšně.

```
bool Vektor::operator==(Vektor & v)
{
  return true;
}
```

Princip triangulace je velice podobný falešné implementaci demonstrované v 4.2.4. Jedná se o techniky užívané pro úpravu kódu po neúspěšném spuštění nově přidaného testu. Jakmile nám však všechny testy proběhly úspěšně, cyklus nekončí. Nastává čas na skutečnou implementaci. V triangulaci abstrahujeme výpočet pouze tehdy, když máme dva a více příkladů. Pro náš test rovnosti vektoru to tedy znamená, že musíme napsat další aserci. Ta bude například znázorňovat situaci, kdy mají dva vektory odlišný rozměr.

```
void Vektor_Test::TestRovnosti()
{
  ...
  Vektor v3(1, poleCisel);
  CPPUNIT_ASSERT_EQUAL( false, v2 == v1 );
}
```

Test selhal. Nyní je potřeba zobecnit implementaci:

```
bool Vektor::operator==(Vektor & v)
{
  if (rozmer != v.rozmer) return false;
  return true;
}
```

Potřebný počet příkladů pro testování nás utvrzuje v tom, že máme správnou abstrakci pro náš výpočet. Triangulace je hojně využívána pro svou jednoduchost. Častěji se ale používá falešná nebo zřejmá implementace.

4.2.4 Falešná implementace

Jedná se zřejmě o nejrychlejší způsob, jak zprovoznit test a přitom se nezabývat skutečným výpočtem testovaného kódu.

V ukázkovém příkladě přistoupíme k porovnávání obsahu složek. Test pro porovnání dvou vektorů na základě jejich rozměrů máme již napsaný. V metodě, kterou testujeme, vrátíme jako výsledek konstantu.

```
bool Vektor::operator==(Vektor & v)
{
  ...
  return true;
}
```

Všimněme si, že hodnota kterou vracíme z metody se přesně shoduje s očekávaným výstupem v testu. To znamená duplicitu mezi testem a kódem, kterou musíme v dalším kroku TDD cyklu odstranit. Falešná implementace je pouze technika, jak se dostat do pevného bodu, od kterého se můžeme „odrazit“ a refaktorovat. Tělo metody můžeme postupně zobecňovat, abychom dostali kód, který provádí skutečný výpočet.

Zkusme tedy napsat skutečnou implementaci metody jako rovnost složek:

```
bool Vektor::operator==(Vektor & v)
{
  ...
  if (this->slozky != v.slozky ) return false;
  return true;
}
```

Test byl úspěšný. V dobré víře jsme se pokusili implementovat porovnávací operátor. Dochází ale k porovnávání ukazatelů, nikoli k porovnávání jednotlivých složek vektoru. Tuto chybu, která se projeví až za běhu programu, můžeme lehce přehlédnout. Ukážeme si další často používanou techniku v TDD, která hlavně pomáhá odhalit chyby v implementaci a jejímž cílem je opět co nejdříve získat „zelený ukazatel“.

4.2.5 Návratový test

Přidáme test, který porovnává dva vektory, jejichž složky jsou od sebe různé. Složku druhého vektoru nastavíme na jinou hodnotu:

```
void Vektor_Test::TestRovnosti()
{
  ...
  v1.slozky[0] = 5;
  CPPUNIT_ASSERT_EQUAL( false, v2 == v1 );
}
```

Zjistíme, že tento test při spuštění neuspěl. Implementaci přetíženého operátoru opravíme. Uděláme však jen to nejnutnější, aby test prošel. Pozměníme implementaci tak, abychom měli jistotu, že příště test dopadne úspěšně. „Opíšeme“ to z testu. Omezíme se na složku, která je u těchto dvou vektorů různá:

```
bool Vektor::operator==(Vektor & v)
{
  ...
  if (this->slozky[0] == v.slozky[0] ) return true;
  else return false;
}
```

I po tomto kroku test selhal. Jsme si jisti, že test je napsán správně, ale nevíme, jak jej zprovoznit. Můžeme test rozdělit, abychom se dozvěděli, ve které části kódu je chyba. Náš test, který porovnává dva nestejné vektory, je částí testovacího případu `TestRovnosti`. Na prvním řádku nastavujeme nultou složku prvního vektoru tak, aby byla odlišná od nulté složky druhého vektoru. Očekávanou sémantiku tohoto příkazu můžeme otestovat přidáním dalšího testovacího případu do naší sady testů. Jednodušším způsobem bude přidání další aserce. Zároveň otestujeme, zda-li hodnota nulté složky druhého vektoru je opravdu jiná. Cílem této techniky je rychle analyzovat místo v kódu, kde dochází k neočekávanému chování. Rozdělíme-li test na menší části – tzv. dceřiné testy, pomůže nám to zaměřit pozornost na menší část kódu, kde se chyba vyskytuje a rychleji ji opravit. Jinými slovy, dostáváme se na nižší úroveň testování.

```
void Vektor_Test::TestRovnosti()
{
...
  v1.slozky[0] = 5;
  CPPUNIT_ASSERT_EQUAL( 5, v1.slozky[0] );
  CPPUNIT_ASSERT_EQUAL( 4, v2.slozky[0] );
...
}
```

Aserce neuspěla a výpis testovácího rámce cppUnit nás upozorní, že nultá složka vektoru `v2` je ve skutečnosti rovna 5, tedy stejné hodnotě nulté složky vektoru `v1`. Tento jev v nás nutně evokuje podezření na závislost těchto dvou objektů na jednom zdroji, v tomto případě paměti. Vrátíme se tedy k části kódu tam, kde je objekt inicializován, tedy k implementaci konstruktoru. Zde vidíme, že ukazatel `slozky` je inicializován pomocí jiného ukazatele. Nejsme-li si jisti, že náš předpoklad původu chyby je správný, můžeme pro utvrzení přidat další testovací případ. Závislost odstraníme tak, že kazdý nově vytvořený objekt bude mít alokovaný svůj vlastní prostor, který v destruktoru objektu uvolníme. Opravíme tedy implementaci:

```
Vektor::Vektor(int rozmer, int *pole) : rozmer(rozmer)
{
  if (rozmer < 1) throw NeplatnyRozmerVektoru();
  slozky = new int[rozmer];
  slozky[0] = pole[0];
  slozky[1] = pole[1];
}
```

Všechny testy proběhly úspěšně. Všimněme si, že nemáme pokrytou situaci, když bude hodnota parametru `pole` `NULL`. Následný přístup přes `NULL` ukazatel, by způsobil zhroucení programu. Měli bychom tento případ nějak ošetřit. Pravidla TDD ale říkají něco jiného.

Máme implementovat pouze to, pro co jsme napsali test. Zatím ani nemáme definované chování pro tuto situaci, nebudeme ji tedy prozatím řešit. Tuto skutečnost si pouze poznamenáme, abychom na ni během vývoje nezapomněli.

Jelikož se nám vyskytuje duplicita mezi přiřazením nulté a první složky vektoru, refaktorujeme. Pokud chceme postupovat po opravdu malých krocích, můžeme si nejdřív vytvořit proměnnou, kterou inicializujeme a použijeme ji jako index v poli `slozky`. Zkontrolujeme, zda testy proběhnou úspěšně a teprve poté příkaz přiřazení obalíme do cyklu. Po úspěsném spuštění testu lze pokračovat v implementaci přetíženého operátoru pro porovnání, kde provedeme tutéž refaktorizaci.

```
bool Vektor::operator==(Vektor & v)
{
  ...
  for (int i = 0; i < rozmer; i++)
    if (this->slozky[i] != v.slozky[i] ) return false;
  return true;
}
```

4.2.6 Změna přístupu ke složkám

Přístup ke složkám vektoru pomocí veřejného atributu `slozky` není znakem kvalitního návhu a již jsme se přesvědčili o tom, že takovéto obcházení zapouzdření může způsobit těžko odhalitelné chyby. Vytvoříme tedy přístupovou metodu pro získání složky vektoru. Nejdříve však pro tuto metodu napíšeme test. Postupujeme nejdříve od té jednodušší operace - tedy čtení. Implementujeme metodu `Slozka()` tak, aby vrátila hodnotu složky na určitém indexu. Poté napíšeme další test, který očekává při zadání indexu mimo rozsah vyvolání výjimky. Upravíme kód tak, aby pokryl tento test.

Teprve poté otestujeme zápis do složky vektoru. Až nebudeme mít již další test pro tuto metodu a budeme si tedy jisti, že je máme implementovány správně, nahradíme všechny výskyty přístupu k veřejnému atributu za nově implementovanou metodu. Během nahrazování spouštíme naši sadu testů, abychom si byli jisti, že refaktorizace je prováděna korektně. Teprve poté můžeme viditelnost atributu `slozky` změnit na soukromou.

Protože snadno můžeme při refaktorizaci udělat chybu, takto provedený postup je typickým příkladem refaktorování pomocí menších kroků. Také to zvyšuje naši sebedůvěru při „čištění“ kódu.

4.2.7 Zpět ke konstruktoru

Při používání TDD nemáme od počátku přesnou představu, jak mají být objekty stavěny. Na začátku jsme vývoj nezačali testováním konstruktoru. Neměli jsme představu o tom, jak má vypadat. Vědeli jsme pouze, že třída `Vektor` by měla poskytnout informace o svém rozměru. A tak jsme do konstruktoru třídy přidali parametr, který inicializuje rozměr vektoru. Stejně jsme to provedli s obsahem složek. Chtěli bychom pokrýt situaci, kdy je v parametru konstruktoru zadán `NULL` tak, že složky vektoru budou inicializovány nulou. Při nezadání tohoto parametru bude `NULL` implicitní hodnotou. Bude to elegantní způsob vytvoření nulového vektoru. Až teď tedy přidáme pro konstruktor další test a implementací jej pokryjeme.

4.2.8 Od jednoho objektu k více

Následující technika demonstruje způsob, jak obecně pracovat s kolekcemi objektů pokud chceme postupovat opatrně. Ve třídě `Vektor` budeme chtít implementovat metodu pro výpočet velikosti vektoru. Nejdříve si vytvoříme metodu, která nepracuje s polem, v našem případě se složkami vektorů. Budeme chtít začít od jediné hodnoty, kterou předáváme v parametru metody. Test pro velikost vektoru vypadá následovně:

```
void Vektor_Test::TestVelikosti()
{
  Vektor v(1);
  v.Slozka(0) = 1;
  CPPUNIT_ASSERT_EQUAL( 1.0, v.Velikost(1) );
}
```

Nejdříve použijeme falešnou implementaci k zprovoznění tohoto testu:

```
double Vektor::Velikost(int slozka) {
  return static_cast<double>(slozka);
}
```

Postupně abstrahujeme výpočet pro tuto jedinou hodnotu:

```
double Vektor::Velikost(int slozka) {
  return sqrt(static_cast<double>(slozka*slozka));
}
```

Tento způsob testování je možnost, jak *izolovat změny* v implementaci od testovacího případu. V metodě máme nyní přístupné jak složky daného vektoru, tak náš parametr sloužící k testování jediné hodnoty. Můžeme tedy upravovat kód bez vlivu na testovací případ.

Upravíme metodu tak, aby pracovala s více složkami:

```
double Vektor::Velikost(int slozka) {
  int suma = 0;
  for (int i = 0; i < rozmer; i++)
    suma += slozky[i]*slozky[i];
  return sqrt(static_cast<double>(suma));
}
```

Nyní odstraníme nepoužívaný parametr z metody a tedy i z testovacího případu. Na provedený krok můžeme také nahlížet jako na izolovanou změnu. Provedli jsme změnu testovacího případu bez vlivu na implementaci.

```
void Vektor_Test::TestVelikosti()
{
  ...
  CPPUNIT_ASSERT_EQUAL( 1.0, v.Velikost() );
}
```

Ted již můžeme pokračovat v testování operace nad kolekcí tím, že budeme testovat operaci s vyšším rozměrem:

```
void Vektor_Test::TestVelikosti()
{
  Vektor v(2, poleCisel);
  CPPUNIT_ASSERT_EQUAL( 5.0, v.Velikost() );
}
```

Podobným způsobem můžeme implementovat i další operace pro vektor – součin, skalární součin dvou vektoru apod.

4.2.9 Operace s řádkem matice

Pokud již máme většinu operací pro vektor hotových, můžeme se pustit do složitějšího shématu – matice. Na základě předem vytvořených testů implementujeme základní metody jako konstruktor, přístupovou metodu, metody pro získání základních informací o matici. Postup jejich implementace bude obdobný jako u vektoru, proto jej zde nebudu uvádět. Deklarace třídy `Matice` vypadá následovně:

```
class Matice
{
  int _radku;
  int _sloupcu;
  int **_pole;
public:
  Matice(int, int, int *pole = NULL);
  int pocetSloupcu() {return _sloupcu;}
  int pocetRadku() { return _radku;}
  int & Prvek(int, int);
public:
  ~Matice(void);
};
```

Nyní bychom chtěli naimplementovat spolupráci matice s vektorem. Konkrétně bychom chtěli, abychom mohli s řádkem nebo sloupcem pracovat jako s vektorem. Nejdříve budeme uvažovat pouze manipulaci s řádkem. Naše prvotní představa může být taková, že jeden z konstruktorů třídy `Vektor` by byl inicializován odkazem na matici. Nechceme vytvořit kopii řádku a vytvořit z něj vektor, ale chceme, aby ostatní operace dokázaly rozpoznat, jestli jde o opravdový vektor nebo odkaz na řádek matice. Testový případ bude náležet v přípravku `linearniAlgebra_Test` testujícím integraci jednotek. Reprezentace samotného testu je následující:

```
void linearniAlgebra_Test::TestPraceSRadkemMatice()
{
  int pole[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
  Matice m(3, 3, pole);
  Vektor v(&m, 2);
  CPPUNIT_ASSERT_EQUAL(7, v.Slozka(0));
}
```

Kompilace neproběhla úspěšně. Není implementován konstruktor s takovými parametry. Ve skutečnosti potřebujeme provést i další úpravy kódu, aby se nám test nezhroutil.

4.2.10 Odklonění

Při práci s takto vytvořeným vektorem, který reprezentuje řádek v matici, je nutné zajistit správnou komunikaci s maticí. Cílem TDD je provádět co nejmenší kroky. Budeme chtít nejdříve otestovat tuto komunikaci tak, že testovaný objekt bude komunikovat se samotným testem místo objektem. V našem případě chceme, aby vektor volal metodu testu, místo toho, aby volal metodu matice. Pozměníme tedy testovací případ:

```
void linearniAlgebra_Test::TestPraceSRadkemMatice()
{
  Vektor v(this, 2);
  CPPUNIT_ASSERT_EQUAL( 7, v.Slozka(0) );
}
```

Nejdříve implementujeme nový konstruktor ve třídě `Vektor`. Přidáme to této třídy nový atribut, který bude obsahovat odkaz na objekt, se kterým bude komunikovat. V konstruktoru, který jsme vytvořili na začátku, naplníme tento ukazatel `NULL` hodnotou:

```
class Vektor
{
  ...
  linearniAlgebra_Test *ukMatice;
  public:
  Vektor(int rozmer, int *pole) : rozmer(rozmer),
                                  ukMatice(NULL) {...}
  Vektor(linearniAlgebra_Test *matice, int cisloRadku = 0) :
                                  rozmer(0), slozky(NULL),
                                  ukMatice(matice) {}
  ...
}
```

V přístupové metodě `Slozka()` upravíme implementaci tak, aby všechny předchozí testy prošly bez problémů a zároveň uspěl i nově napsaný test:

```
int & Vektor::Slozka(int index) {
  if (ukMatice != NULL) {
    return 7;
  }
  else {
    ...
  }
}
```

Použili jsme falešnou implementaci v metodě `Slozka()` k tomu, abychom test zprovoznili. Vektor prozatím ale s žádným objektem nekomunikuje. Pomocí malých kroků se budeme blížit s cílenému záměru a během postupu se utvrzujeme spouštěním testů:

1. Implementujeme požadovanou metodu v testovacím případu resp. přípravku[1]. V tomto případě se jedná o přístupovou metodu `Prvek()`.

```
int & linearniAlgebra_Test::Prvek(int radek, int sloupec)
{
  return 7;
}
```

2. V přístupové metodě `Slozka()` třídy `Vektor` nahradíme falešnou implementaci voláním metody přes uložený odkaz na objekt.

```
int & Vektor::Slozka(int index) {
  if (ukMatice != NULL) {
    return ukMatice->Prvek(2, 0);
  }
  else {
    ...
  }
}
```

Teď po spuštění testu jsme naprosto přesvědčeni, že vektor komunikuje správným způsobem s objektem, v tomto případě testem. Není totiž žádná jiná možnost, jak by vektor vrátil správný výsledek. Navíc se samotný testovací případ stává čitelnější, neboť volaná metoda je součástí testu.

Pokud nechceme měnit typ předávaného ukazatele, je možné vyčlenit rozhraní. Vytvořili bychom tedy abstraktní třídu, která by obsahovala metodu, přes kterou chceme komunikaci s objektem testovat. Testový případ i náš objekt by tuto třídu dědil. Vytvořili bychom tak prostor pro polymorfní chování a mohli bychom tak testovat komunikaci s objektem pomocí testu kdykoli potřebujeme.

Nyní se můžeme vrátit k testu, který testuje komunikaci vektoru přímo s maticí, ne tedy s testem. Parametry metody `Prvek()` abstrahujeme na příslušné proměnné. Nejprve přidáme do třídy `Vektor` atribut `cisloRadku`, reprezentující číslo řádku v odkazované matici. Inicializujeme jej v konstruktoru. A poté postupně nahradíme parametry volání metody `Prvek()`:

```
int & Vektor::Slozka(int index) {
  ...
    return ukMatice->Prvek(cisloRadku, index);
  ...
}
```

4.2.11 Vyčlenění rozhraní

Máme implementován způsob, jak pracovat s řádkem matice jako s vektorem. Nyní bychom mohli upravit další metody třídy `Vektor`. Všude bychom použili if konstrukci k tomu, abychom odlišili, zda se jedná o opravdový vektor nebo o řádek matice. Těchto podmínek

[1] Testovací přípravek v cppUnit je implementován jako třída TestFixture.

by bylo docela dost a čitelnost těchto metod by se výrazně snížila. Kdybychom chtěli implementovat práci se sloupcem, bylo by nutné přidat další větev. Následovalo by přidávání atributů a naše původní třída `Vektor` by se nepříjemně rozrostla. Jak vidíme, takto vytvořený model určitě není indikátorem správného návrhu. Bude proto nutné provést refaktorizaci, protože se tento model vymyká kontrole. Potřebujeme někde uchovat druhou implementaci operací pro vektor. Způsobem, jak to provést, je vytvořit rozhraní[2] obsahující sdílené operace. Toto rozhraní poté použijeme v implemetaci nové třídy, která představuje odlišné chování operací.

Postup bude následující:

1. Nejdříve pouze deklarujeme nové rozhraní. V našem případě bude toto rozhraní představovat abstraktní třídu:

```
class iVektor
{
public:
  iVektor(void);
public:
  virtual ~iVektor(void) = 0;
};
```

2. Existující třída bude implementovat toto rozhraní:

```
class Vektor : public iVektor
{ ... }
```

3. Přidáme do rozhraní nezbytné metody. V našem případě se bude jednat o virtuální metodu pro přístup ke složce vektoru:

```
class iVektor {
public:
    ...
    virtual int & Slozka(int) = 0;
    ...
}

class Vektor :public iVektor {
public:
    ...
    virtual int & Slozka(int);
    ...
}
```

4. Změníme typ deklarací ze třídy na rozhraní. Zatím to v některých případech nebude možné – např. v testu pro velikost vektoru, protože v rozhraní tuto metodu ještě nemáme.

[2] pojem rozhraní je v této pasáži použit pro vysvětlení ilustrované techniky. Jelikož se pohybujeme v C++, bude se jednat o abstraktní třídu

```
void linearniAlgebra_Test::TestPristupu()
{
  iVektor *v = new Vektor(2, poleCisel);
  CPPUNIT_ASSERT_EQUAL( 3, v->Slozka(1) );
  ...
}
```

Takto připravené rozhraní nám umožní vytvořit druhou implementaci metod.

4.2.12 Použití návrhových vzorů

Odlišný pohled TDD oproti klasickým metodologiím vyžaduje i použití návrhových vzorů. Většinou se totiž návrhové vzory(více v [6]) používají pouze ve fázi návrhu aplikace a potlačují se tak jejich možností ve fázi refaktorizace. Nicméně i to je možné a toto refaktorování lze provádět po malých krocích tak, abychom si byli jisti, že provedené změny v návrhu nemají vliv na již fungující testy.

Nyní můžeme vytvořit novou třídu, která bude implementovat operace s řádkem matice a zároveň se bude chovat jako vektor. Bude tedy obsahovat rozhraní definované abstraktní třídou `iVektor`.

Model, který postavíme vytvořením třídy implementující rozhraní `iVektor`, se nápadně podobá návrhovému vzoru *Adapter*. Nová třída `Radek` bude přizpůsobovat chování již existující třídy `Vektor` a komunikovat s objekty třídy `Matice` bez nutnosti měnit tuto třídu. Dá se říct, že objekt třídy `Radek` deleguje zprávy na objekt třídy `Matice`. Jelikož používáme přístup k prvkům matice přes přístupovou metodu `Prvek()`, nemusíme ošetřovat indexaci mimo rozsah.

Při budování třídy `Radek` postupujeme takto:

1. Vytvoříme novou třídu. Bude implementovat rozhraní dané abstraktní třídou `iVektor`:

   ```
   class Radek : public iVektor { ... }
   ```

2. Ze třídy `Vektor` zkopírujeme konstruktor, pracující s odkazem na matici, do třídy `Radek`. Aby se kód zkompiloval, budeme muset přidat i atributy `ukMatice` a `cisloRadku`.

   ```
   class Radek : public iVektor
   {
     Matice *ukMatice;
     int cisloRadku;
   public:
     Radek(Matice *matice, int cisloRadku = 0) : ukMatice(matice),
       cisloRadku(cisloRadku) {}
     ...
   };
   ```

3. Vytvoříme ve třídě `Radek` metodu pro přístup ke složce řádku a tělo if větve metody `Slozka()` třídy `Vektor` tam zkopírujeme.

```
int & Radek::Slozka(int index)
{
  return ukMatice->Prvek(cisloRadku, index);
}
```

4. Upravíme náš test, aby používal naší nově vytvořenou třídu:

```
void linearniAlgebra_Test::TestPraceSRadkemMatice()
{
  int pole[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
  Matice *m = new Matice(3, 3, pole);
  iVektor *radek = new Radek(m, 2);
  CPPUNIT_ASSERT_EQUAL( 7, radek->Slozka(0) );
}
```

5. Nakonec smažeme if větev metody `Slozka()` třídy `Vektor` a následně i konstruktor používající odkaz na matici. Poté můžeme odstranit z této třídy i atributy `ukMatice` a `cisloRadku`.

Pokud testy neselhávají, jsme si jisti, že jsme během postupu neudělali chybu a vše funguje korektně. Nyní můžeme tak pracovat jak se skutečným vektorem, tak s řádkem v matici pomocí rozhraní `iVektor`. Použitím návrhových vzorů jsme odstranili narůstající větvení a nečitelnost třídy `Vektor`.

4.2.13 Odstranění duplicit

Všimněme si, že jsme do rozhraní přidali pouze jednu metodu – metodu pro přístup ke složce, ačkoliv víme, že bude potřeba přidat i další metody, např. `Velikost()`. V TDD implementujeme vždy jen to nejnutnější v závislosti na testech. Další test by mohl vypadat takto:

```
void linearniAlgebra_Test::TestVelikostiRadkuMatice()
{
  int pole[] = {1, 2, 3, 4};
  Matice *m = new Matice(2, 2, pole);
  iVektor *radek = new Radek(m, 1);
  CPPUNIT_ASSERT_EQUAL( 5.0, radek->Velikost() );
}
```

Pro zprovoznění testu musíme rozhraní `iVektor` rozšířit o novou metodu. Ve třídě `Radek` poté implementujeme metodu `Velikost()`:

```
double Radek::Velikost() {
  int suma = 0;
  for (int i = 0; i < ukMatice->pocetSloupcu(); i++)
    suma += ukMatice->Prvek(cisloRadku, i)*ukMatice->Prvek(cisloRadku, i);
  return sqrt(static_cast<double>(suma));
}
```

Vidíme, že implementace této metody ve třídě `Radek` je velice podobná, jak tomu bylo ve třídě `Vektor`. Odlišný je pouze přístup k vlastnostem objektu. Metoda funguje zcela správně, přesto tento způsob zápisu nemusí být tím nejvhodnějším. S pocitem, že již víme jak implementovat další chování třídy `Radek`, bychom přidávali další metody podobné těm ze třídy `Vektor`. Brzy bychom zjistili, že tento způsob přinese duplicitu v kódu a z toho následující nevýhody:

- Při implementaci třídy `Sloupec`, která by reprezentovala sloupec v matici, by bylo nutné opět implementovat metody rozhraní `iVektor` tak, aby bylo možné se sloupcem v matici pracovat jako s vektorem. Množství kódu by se tak opět rapidně zvětšilo.
- Při pozdějším přidání resp. úpravě operace pro vektor ve třídě `Vektor` bychom museli tuto metodu přidat resp. upravit ve všech třídách implementujících rozhraní `iVektor`.
- Bude-li změněn přístup k matici (např. implementace zachytávání výjimek), bude potřeba provést rozsáhlejší úpravu kódu týkající se všech metod přistupujících k matici.

Mohli jsme na to myslet již při vytváření abstraktní třídy `iVektor` anebo možná ještě dříve - při implementaci třídy `Vektor`. V TDD se refaktorizace týká vždy momentálního stavu vývoje aplikace. To znamená, že duplicitu neodstraňujeme před jejím výskytem ale až když ji odhalíme. V našem případě nyní řešíme duplicitu metody `Velikost()` s krátkým výhledem na implementaci dalších metod.

Postup, jak odstraníme duplicitu, spočívá v převedení podobných úseků kódu na stejné:

1. Nejdříve přepíšeme metodu `Velikost()` ve třídě `Vektor` tak, aby nepřistupovala k soukromým atributům své třídy. Na místo přístupu k atributu `slozky` použijeme metodu `Slozka()` implementovanou v dané třídě. Pro zjištění rozměru vektoru použijeme metodu `Rozmer()`:

   ```
   double Vektor::Velikost() {
     int suma = 0;
     for (int i = 0; i < this->Rozmer(); i++)
       suma += this->Slozka(i)*this->Slozka(i);
     return sqrt(static_cast<double>(suma));
   }
   ```

2. Podobně to provedeme s implementací metody ve třídě `Radek`. Pro zjištění rozměru vektoru ale nemáme vytvořenou metodu `Rozmer()`, a tak necháme zjistit rozměr počtem sloupců v matici.

3. Musíme nejřív implementovat metodu `Rozmer()` ve třídě `Radek`. Předtím samozřejmě napíšeme pro tuto metodu test. Pro jednoduchost v tomto případě stačí přidat do testovací metody `TestPraceSRadkemMatice()` aserci. Jakmile jsme udělali vše potřebné pro její úspěšné zprovoznění, můžeme pokračovat dále v odstraňování duplicity tím, že v metodě `Radek::Velikost()` použijeme nově přidanou metodu `Rozmer()`.

4. Nyní máme dvě metody s naprosto stejným tělem. Pokud chceme, aby obě třídy používaly společnou metodu, musíme metodu přesunout výše - do jejich společného předka. V tomto případě se jedná o abstraktní třídu `iVektor`. Nejdříve tedy zkopírujeme implementaci metody do této třídy ale metodu ponecháme virtuální.

```
class iVektor
{ ...
  virtual int & Slozka(int) = 0;
  virtual double Velikost();
  virtual int Rozmer() = 0;
  ... };
```

Poté odebereme metodu jedné ze tříd, např. `Radek`. Pokud se úspěšnost testů nemění, můžeme odebrat metodu i z třídy `Vektor`. Nakonec provedeme změnu deklarace metody z virtuální na obyčejnou:

```
class iVektor
{ ...
  double Velikost();
  ... };
```

Tímto postupem jsme odstranili duplicitu a nevýhody z ní vyplývající. Zároveň jsme nasměrovali další vývoj k lepšímu návrhu. Další operace jako součin vektoru s číslem nebo skalární součin vektorů již můžeme psát s použitím metody `Slozka()` a metody `Rozmer()` do třídy `iVektor` a vyhnout se tak vícenásobné implementaci.

4.2.14 Test pro zhroucení

Když jsme implementovali metodu `Slozka()` ve třídě `Radek`, odmítli jsme ošetřovat přístup k matici s tím, že to již bylo provedeno v metodě `Prvek()` třídy `Matice`. V případě přístupu mimo rozsah matice, bude vyvolána výjimka patřící do množiny výjimek vyvolaných při práci s maticí. Taková výjimka ale nebude nic říkat o tom, že k chybě došlo při použití objektu třídy `Radek`. Proto bychom chtěli vyvolat výjimku popisující chybu při práci s řádkem matice. Zároveň ale nechceme ošetřovat rozsah matice dvakrát - jednou ve třídě `Radek` a podruhé ve třídě `Matice`, což by jednak přineslo duplicitu v kódu, ale také plýtvání strojovým časem.

Při napsání testu bychom chtěli použít *model pro zhroucení*, který spočívá ve vytvoření falešného objektu simulujícího určité chování. Slouží hlavně k testování ošetření chyby, u které je dost nepravděpodobné, že nastane. V mnohých případech je použití falešného objektu jediným způsobem, jak takový případ otestovat – např. zaplnění disku nebo vyčerpání paměti. V našem případě bude sloužit k odklonění od implementace metody `Prvek()`. Test se tak může stát čitelnější. Alternativou by bylo vytvořit skutečnou matici a přistoupit k prvku matice, o kterém víme, že neexistuje. My ale nechceme testovat, zda je ošetřen přístup k prvkům matice. Proto vytvoříme falešný objekt, který simuluje chování potřebné pro test.

Takto by mohl vypadat náš falešný objekt pro matici:

```
class falesnaMatice : public Matice {
public:
  falesnaMatice(): Matice(1,1) {}
  virtual int & Prvek(int, int) { throw MimoRozsahMatice(); }
};
```

Následující test používá námi vytvořený falešný objekt:

```
void linearniAlgebra_Test::TestVyhozeniVyjimkyRadku()
{
  Matice *m = new falesnaMatice();
  Radek *radek = new Radek(m, 0);
  CPPUNIT_ASSERT_THROW(radek->Slozka(0), ChybaRadku);
}
```

Test při spuštění selže. Místo očekávané výjimky `ChybaRadku` je vyvolána výjimka `MimoRozsahMatice`. Upravíme tedy implementaci metody `Slozka()`:

```
int & Radek::Slozka(int index)
{
  try {
    return ukMatice->Prvek(cisloRadku, index);
  }
  catch (MimoRozsahMatice &) {
    throw ChybaRadku("Radek: Pristup k~matici je mimo rozsah!");
  }
}
```

Model pro test zhroucení je technika, která nám pomáhá napodobit chybu, kterou chceme odzkoušet. Toho dosáhneme překrytím metody, jež má vyvolat chybu. V našem příkladu jsme překryli metodu `Prvek()` třídy `Matice`, abychom otestovali zachycení výjimky v metodě `Slozka()` třídy `Radek`. Tímto se náš test stal nezávislý na testu přístupu k matici.

4.3 Shrnutí ukázkové aplikace

V této části se pokusím shrnout demonstrované techniky použité při vývoji ukázkového příkladu knihovny a klasifikovat je do skupin podle jejich cílů. Následuje pak nástin možného pokračování vývoje knihovny pomocí TDD postupů. Tato další rozšíření jsou přítomna v implementaci knihovny.

Dále je diskutována analýza frekvence spouštění testů a tedy důležitost častého spouštění testů.

Nakonec jsou uvedeny poznatky získané během vývoje knihovny charakterizující vliv TDD metodiky.

4.3.1 Použité techniky

Techniky použité při implementaci ukázkové aplikace bychom mohli rozdělit do následujících skupin:

Techniky pro rychlé zprovoznění testu se používají zejména ve třetím kroku cyklu TDD(prvotní úprava kódu). Nefungující test je potřeba co nejdříve spravit i na úkor návrhu. Cílem je provádět co nejkratší kroky.

- **Falešná implementace** (4.2.4) je zřejmě nejznámější metodou, jak rychle zprovoznit test. Charakterizuje ji používání konstant namísto proměnných.
- **Triangulace** (4.2.3) je metodou, kdy tvoříme skutečnou implementaci až tehdy, když máme více testů.

Skutečná implementace. Pokud jsme si jisti správnou abstrakci výpočtu, není důvod „zdržovat se“ falešnou implementací a nic nám nebrání psát kód reprezentující výpočet.

Od jednoho objektu k více (4.2.8) objektům postupujeme, pokud potřebujeme pracovat s kolekcí objektů. Nejdříve otestujeme operaci s jedním objektem.

Techniky pro psaní testů nám pomohou testovat konkrétní chování, kterého chceme docílit.

Testování výjimek (4.2.2) spočívá v zachycení výjimky v testu. V případě, že se nevyvolá, test selže.

Odklonění (4.2.10) je způsob, jak testovat komunikaci s jiným objektem. Nejdříve bude náš testovaný objekt komunikovat s testem, až poté s objektem.

Test pro zhroucení (4.2.14) používá falešný objekt k simulování určitého chování, v tomto případě chyby.

Úvodní test (4.2.1) jako technika nám doporučuje začít těmi nejjednoduššími testy a vybírat vstupní data, u kterých je ihned patrný výsledek.

Návratový test (4.2.5) pomáhá vrátit se zpět k „zelenému ukazateli“ tím, že rozdělíme test na menší části.

Techniky pro refaktorování přichází na řadu často až v posledních fázích vývoje TDD. Jejich cílem je nejen abstrahovat výpočet a vytvořit skutečnou implementaci, ale i zformovat návrh a odstranit duplicitu.

Vyčlenění rozhraní (4.2.11) využijeme, pokud chceme vytvořit druhou implementaci metod s cílem polymorfního chování objektů.

Návrhové vzory (4.2.12) dokážou řešit určité problémy obecným způsobem. Nemusíme se striktně držet „učebnicových“ schémat ale často nás inspirují ke vzniku kvalitnějšího návrhu.

4.3.2 Možné pokračování

Když se v TDD zamyslíme nad pokračováním projektu, většinou nás napadne nějaký další test, který bychom chtěli zprovoznit. A když už máme všechno potřebné chování implementované, stále je na čem pracovat. Můžeme refaktorovat, odstranit překážející duplicitu, pokrýt kód bohatou sadou testů. Následně uvádím stručným způsobem možné pokračování vývoje naší ukázkové aplikace a také použití technik.

- Vytvoření objektu reprezentujícího řádek matice je výhodné nejen pro práci s ním jako s vektorem, ale může sloužit i jako způsob procházení matice po řádcích. Inspirujeme se návrhovým vzorem *Iterator* a ve třídě `Radek` implementujeme metodu, která způsobí, že objekt bude ukazovat na další řádek matice.
- Při implementaci přiřazovacího operátoru a kopírovacího konstruktoru ve třídě `Matice` nám vznikne duplicita v alokaci paměti a inicializaci datové reprezentace matice. Abychom tuto duplicitu odstranili, můžeme vytvořit strukturu zapouzdřující data matice. Objekt třídy `Matice` by uchovával ukazatel na tuto strukturu, která by pak měla na starosti alokaci paměti a inicializaci datových složek.

- Součin matice s vektorem můžeme imlementovat ve třídě `Matice` jako další přetížený operátor. Tělo výpočtu bude ale velice podobné součinu dvou matic. Pokud dokážeme těla těchto dvou přetížených operátorů přiblížit tak, aby se shodovaly, postačí nám implementace jedna. Jedním ze způsobů, jak toho docílit, je použití konverze vektoru na jednosloupcovou matici.

Pokrytí testy v naší ukázkové aplikaci určitě není dostatečné. Testy jsou v TDD nástroj na získání důvery v kód. Poskytují zpětnou vazbu, takže jejich množství závisí na tom, jak moc jsme si jisti svou implementací.

4.3.3 Frekvence spouštění testů

Pro zhodnocení výsledků TDD je frekvence spouštění určitě vypovídající hodnotou. Pro zjištění této hodnoty jsem upravil aplikaci pro vykonávání testů(Test runner) tak, aby při stisknutí tlačítka pro spuštění testů došlo k zaznamenání aktuálního času a výsledku spouštěné sady. Ze zapsaných hodnot lze poté určit intervaly mezi běhy testů.

Z naměřených údajů vyplývá, že během implementace nových metod se interval pohyboval kolem 1-2 minut. To se týká během typického cyklu TDD, kdy spuštění nového testu selhalo, následovalo rychlé zprovoznění testu a postupná refaktorizace na skutečnou implementaci. Interval tak zůstával konstantní. Rozdíl byl patrný zejména při refaktorizaci. Pokud jsem prováděl rozsáhlejší refaktorizaci najednou a interval se prodloužil na 4-6 min, typicky následovalo několik neúspěšných spuštění testů. Zprovoznění obvykle vyžadovalo vrátit se na začátek a provádět refaktorizaci po částech s častějším spouštěním testů, které neselhávají.

Používání TDD tedy skutečně staví na častém spouštění testů a provádění menších kroků při implementaci.

4.3.4 Účinky TDD

Zhodnotit účinky TDD na základě tohoto jednoduchého ukázkového příkladu se nemusí zdát věrohodné. Přesto bych zmínil několik získaných poznatků podle různých hledisek, přičemž všechny tyto poznatky spolu úzce souvisí:

Produktivita Pro kvalitní posouzení z tohoto hlediska by bylo vhodné implementovat stejné zadání klasickou metodikou a porovnat dobu vývoje. Psaní testů může být dosti zpomalující činností. Na druhou stranu má ale svůj opodstatněný smysl. Tím, že píšeme testy dopředu, odhalíme případnou chybu již při jejím prvním vzniku, kdy operujeme v její blízkosti. Pomocí testů jednotek chybu lehce lokalizujeme a můžeme jí ihned opravit. Vyhneme se tak častému používání ladících nástrojů k odhalení skrytých chyb, což vývoj učiní svižnějším.

Kvalita programu Program je pokryt více testy než při klasickém vývoji. Pokud totiž opravdu píšeme testy před každou novou funkčností, nejsme my a ani zákazník překvapeni nežádoucím chováním. Kvalita programu se tak částečně odvíjí od počtu testů, které napíšeme. Kvalita návrhu závisí na míře refaktorování. TDD nám ukazuje způsob, jak provádět rozsáhlejší refaktorizace po malých krocích tím, že budeme změny provádět postupně a zároveň neustále kontrolovat úspěšnost testů. Takový přístup nám znemožní udělat chybu nebo něco opomenout, jak by k tomu došlo, kdybychom chtěli tuto refaktorizaci provést v jednom kroku. Důsledkem tohoto přístupu je také větší důvěra a jistota v prováděné

změny. Když totiž programátor při refaktorizaci přinese do kódu spoustu chyb, často jej to odradí a na refaktorizaci nebude mít příště odvahu.

Pochopení programu Častým psaním testů a refaktorováním pomocí menších kroků se programátor více přiblíží k implementaci. Lépe dokáže odhadnout chování programu a pustit se tak i do odvážnější refaktorizace.

Kapitola 5

Závěr

Záměrem této práce bylo ilustrovat vývoj řízený testy a ukázat tak techniky, kterými TDD dosahuje cíle projektu. V prvních kapitolách seznamuji čtenáře s agilními metodikami, zejména TDD metodikou, jejími principy a používanými nástroji, z nichž nejznámější je rodina testovacích rámců xUnit.

Implementace knihovny pro vektory a matice slouží jako ukázkový příklad TDD v praxi. Zejména je zde demonstrován jeho cyklus, způsoby psaní testů a implementace. V neposlední řadě je na příkladu ukázáno, jakým způsobem se TDD staví k návrhu. Tyto techniky jsou ilustrovány na jednoduchých problémech. Jejich smysl a princip je však nápomocný i v náročnějších projektech.

Vývoj software řízený testy má ale i svá omezení. Nevýhodou používání TDD je problém s testováním samostatných jednotek a automatizovaného testování uživatelského rozhraní. I pro tyto problémy však TDD nabízí řešení uvedena v 2.2.5. Také je obtížné udržet disciplínu při používání TDD. Ze začátku není jednoduché si na cyklus TDD zvyknout a dodržovat jeho pravidla. Zejména jde o psaní testů před samotnou implementací. Programátor je velice zvyklý psát rovnou kód.

Aplikováním testy-řízeného vývoje můžeme dosáhnout předvídatelného chování programu. Používání ladících nástrojů se výrazně minimalizuje. Vývoj řízený testy umožňuje často odhalit chybu již při jejím prvním vzniku. Psychologický efekt na vývojáře hraje také důležitou roli. Častým spouštěním testů se programátor utvrzuje, že program se chová podle jeho představ. To mu dodává odvahu provádět refaktorizaci, která zjednoduší návrh a zároveň odstraní duplicitu.

Podle mého mínění nelze tvrdit, že vývoj řízený testy je univerzální metodologie pro všechny typy projektů. Zvláště pro rozsáhlé projekty může být problémem testování systému před samotnou implementací a jeho restrukturalizace během vývoje. Naproti tomu může být TDD lepším řešením při realizaci menších projektů s měnícími se požadavky. Aplikování vývoje řízeného testy na práci v týmu určitě předpokládá méně početný tým složený ze zkušených vývojářů.

Zajímavým pokračováním projektu by mohla být implementace ukázkové aplikace skupinou vyvojářů, z nichž jedna podmnožina by použila při vývoji klasické metodiky a druhá vývoj řízený testy. Výsledkem by mohla být studie porovnávající vliv TDD na produktivitu vývojářů, kvalitu implementace a další faktory důležité pro úspěch softwarového produktu.

Literatura

[1] Agile manifest. [online], [cit. 2008-03-16].
URL <http://agilemanifesto.org/>

[2] CppUnit - The Unit Testing Library. [online], [cit. 2008-02-27].
URL <http://cppunit.sourceforge.net/cppunit-wiki>

[3] Test Driven Development by Agile Data. [online], [cit. 2008-04-24].
URL <http://www.agiledata.org/essays/tdd.html>

[4] ASTELS, D.: *Test-Driven Development: A Practical Guide*. Pearson Education, 2003.

[5] BECK, K.: *Programování řízené testy*. Grada Publishing, 2004, ISBN 80-247-0901-5, přeložil Slavoj Písek.

[6] DVOŘÁK, M.: Návrhové vzory (design patterns). [online], [cit. 2008-04-15].
URL <http://objekty.vse.cz/Objekty/Vzory>

[7] KADLEC, V.: *Agilní programování*. Computer Press, 2004, ISBN 80-85615-77-0.

[8] LLOPIS, N.: Stepping Through the Looking Glass: Test-Driven Game Development (Part 1). [online], [cit. 2008-03-25], 2005.
URL <http://www.gamesfromwithin.com/articles/0502/000073.html>

[9] MARTIN, R.: *Agile Software Development*. Prentice Hall, 2003, ISBN 80-7302-0007-6.

[10] Microsoft: Microsoft Visual Studio 2005 Team System Web Testing. [online], [cit. 2008-01-20].
URL <http://msdn2.microsoft.com/en-us/library/ms364077.aspx>

[11] Wikipedia: List of Unit Testing frameworks — Wikipedia, The Free Encyclopedia. [online], [cit. 2008-03-31].
URL <http://en.wikipedia.org/w/index.php?title=List_of_unit_testing_frameworks&oldid=208981028>

[12] Wikipedia: Test-driven development — Wikipedia, The Free Encyclopedia. 2008, [online], [cit. 2008-05-10].
URL <http://en.wikipedia.org/w/index.php?title=Test-driven_development&oldid=211459750>