Obrácená metoda k TDD

Matt Stephens, Doug Rosenberg

Testování softwaru

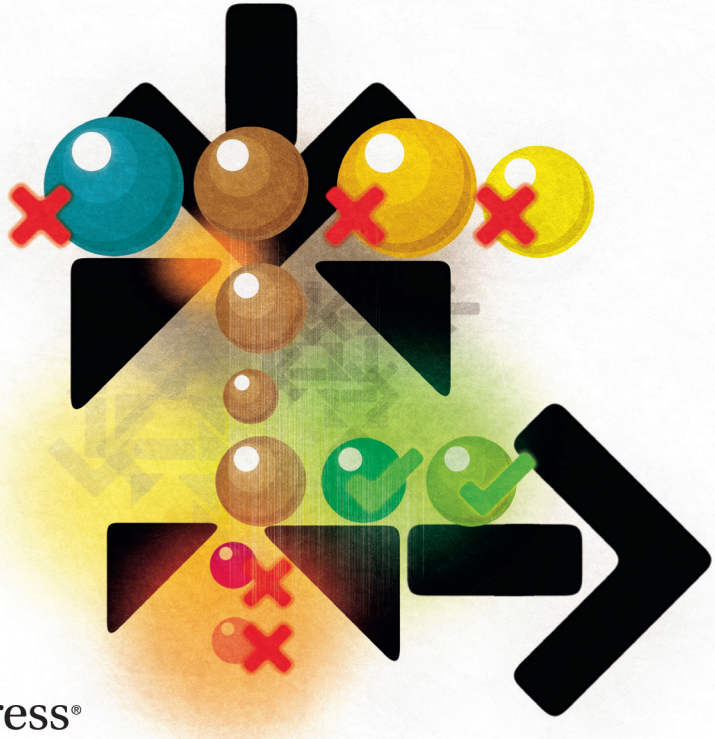
řízené návrhem

Vytvářejte dokonalejší aplikace

Pro každé stadium životního cyklu aplikace

Snadné a efektivní přístupy pomocí UML

Příklady z praxe a využití konkrétních nástrojů



CPRESS Apress®

Matt Stephens, Doug Rosenberg

Testování softwaru řízené návrhem

Computer Press, a. s.
Brno
2011

Testování softwaru řízené návrhem

Matt Stephens, Doug Rosenberg

Computer Press, a. s., 2011. Vydání první.

Překlad: Lukáš Krejčí
Jazyková korektura: Alena Láníčková
Sazba: Dagmar Hajdajová
Rejstřík: Daniel Štreit
Obálka: Martin Sodomka
Komentář na zadní straně obálky: Martin Herodek
Technická spolupráce: Jiří Matoušek, Zuzana Šindlerová, Dagmar Hajdajová
Odpovědný redaktor: Martin Herodek
Technický redaktor: Jiří Matoušek
Produkce: Petr Baláš

Original edition copyright © 2010 by Matt Stephens and Doug Rosenberg. All rights reserved.
Czech edition copyright 2011 by Computer Press. All rights reserved.

Autorizovaný překlad z originálního anglického vydání Design Driven Testing.
Originální copyright: © 2010 by Matt Stephens and Doug Rosenberg. All rights reserved.
Překlad: © Computer Press, a.s., 2011.

Computer Press, a. s.,
Holandská 3, 639 00 Brno

Objednávky knih:
http://knihy.cpress.cz
distribuce@cpress.cz
tel.: 800 555 513

ISBN 978-80-251-3607-2
Prodejní kód: K1950
Vydalo nakladatelství Computer Press, a. s., jako svou 4028. publikaci.

© Computer Press, a. s. Všechna práva vyhrazena. Žádná část této publikace nesmí být kopírována a rozmnožována za účelem rozšiřování v jakékoli formě či jakýmkoli způsobem bez písemného souhlasu vydavatele.

Obsah

Úvod 11
O autorech 13
O odborných korektorech 14
Poděkování 15
Předmluva 16

Část I
DDT vs. TDD

Kapitola 1
Někdo to má pozpátku 19
Problémy, které řeší testování řízené návrhem 20
Vědět, kdy je hotovo, je obtížné 20
Ponechání testování na později je dražší 21
Testování špatně navrženého kódu je těžké 21
Je snadné opomenout testy na úrovni zákazníka 22
Vývojáři bývají samolibí 22
Testy někdy postrádají smysl 22
Rychlý přehled metodiky DDT bez ohledu na nástroje 22
Struktura testování řízeného návrhem 23
Metodika DDT v akci 25
Jak se metodiky TDD a DDT odlišují 26
Ukázkový projekt: představení webové aplikace Mapplet 2.0 28
Shrnutí 29

Kapitola 2
Úvodní příklad s metodikou TDD 33
10 nejdůležitějších vlastností metodiky TDD 34
10. Testy řídí návrh 34

9. Celkový nedostatek dokumentace ... 34
8. Vše je test jednotky ... 35
7. Testy metodiky TDD nejsou tak docela testy jednotek (nebo snad ano?) ... 35
6. Testy přijatelnosti poskytují zpětnou vazbu vůči požadavkům ... 35
5. Metodika TDD propůjčuje důvěru k provádění změn ... 35
4. Návrh se vyvíjí inkrementálním způsobem ... 36
3. Nějaký předběžný návrh je v pořádku ... 36
2. Metodika TDD produkuje velké množství testů ... 36
1. Metodika TDD je nehorázně obtížná ... 37
Přihlašování implementované pomocí metodiky TDD ... 37
Seznámení s požadavkem ... 37
Přemýšlejte o návrhu ... 40
Nejdříve se napíše první test psaný jako první test ... 41
Vytvoření kódu pro kontrolu přihlášení, aby test prošel ... 44
Vytvoření mockobjektu ... 46
Refaktorizace kódu ukazující rozvoj návrhu ... 48
Testy přijatelnosti s metodikou TDD ... 54
Závěr: metodika TDD je opravdu nehorázně obtížná ... 54
Shrnutí ... 56

Kapitola 3
Úvodní příklad s metodikou DDT ... 57
10 nejdůležitějších vlastností metodiky ICONIX/DDT ... 58
10. Metodika DDT zahrnuje testy obchodních požadavků ... 58
9. Metodika DDT zahrnuje testy scénářů ... 58
8. Testy jsou odvozené z návrhu ... 58
7. Metodika DDT zahrnuje testy řadičů ... 59
6. Metodika DDT testuje chytřeji, ne obtížněji ... 59
5. Testy jednotek metodiky DDT jsou „klasickými“ testy jednotek ... 59
4. Testovací případy metodiky DDT lze transformovat do testovacího kódu ... 59
3. Testovací případy metodiky DDT vedou k testovacím plánům ... 60
2. Testy metodiky DDT jsou užitečné pro vývojáře i členy QA týmů ... 60
1. Metodika DDT dokáže eliminovat nadbytečné úsilí ... 60
Přihlašování implementované pomocí metodiky DDT ... 60
Krok 1: Vytvoření diagramu robustnosti ... 62
Krok 2: Vytvoření testovacích případů ... 66
Krok 3: Přidání scénářů ... 68
Krok 4: Transformace testovacích případů testů řadiče do tříd ... 70
Krok 5: Generování testovacího kódu řadičů ... 72
Krok 6: Nakreslení diagramu posloupností ... 75

Krok 7: Vytvoření testovacích případů pro testy jednotek ... 78
Krok 8: Doplnění testovacího kódu ... 82
Shrnutí ... 86

Část II
Metodika DDT v praxi: Mapplet 2.0, web pro cestovatele

Kapitola 4
Seznámení s Mapplet ... 91
10 nejdůležitějších osvědčených postupů metodiky ICONIX Process/DDT ... 92
10. Vytvořte architekturu ... 93
9. Dohodněte se na obchodních požadavcích a testujte vůči nim ... 94
8. Návrh odvoďte z problémové domény ... 96
7. Podle storyboardů uživatelského rozhraní napište případy užití ... 98
6. Pro ověření, že případy užití pracují, napište testy scénářů ... 100
5. Testujte vůči konceptuálnímu a podrobnému návrhu ... 104
4. Model pravidelně aktualizujte ... 104
3. Testy přijatelnosti udržujte synchronizované s požadavky ... 111
2. Mějte automatizované testy stále aktuální ... 112
1. Kandidáta na finální verzi porovnejte s původními případy užití ... 112
Shrnutí ... 116

Kapitola 5
Podrobný návrh a testování jednotek ... 117
10 nejdůležitějších „úkolů“ při testování jednotek ... 118
10. Začněte diagramem posloupností ... 119
9. Na základě návrhu identifikujte testovací případy ... 120
8. Pro každý testovací případ napište scénáře ... 122
7. Testujte chytřeji: nepište překrývající se testy ... 124
6. Testovací případy transformujte do jazyka UML ... 126
5. Napište testy jednotek a doprovodný kód ... 130
4. Napište testy jednotek jako testy „bílé skříňky“ ... 133
3. Použijte framework pro mock objekty ... 137
2. Algoritmickou logiku testujte pomocí testů jednotek ... 140
1. Napište samostatnou sadu integračních testů ... 141
Shrnutí ... 142

Kapitola 6
Konceptuální návrh a testování řadičů 143
10 nejdůležitějších „úkolů“ při testování řadičů 145
10. Začněte diagramem robustnosti 145
9. Z řadičů odvoďte testovací případy 148
8. Pro každý testovací případ definujte jeden či více scénářů 151
7. Vyplňte pole Description, Input a Acceptance Criteria 154
6. Vygenerujte testovací třídy 155
5. Implementujte testy 159
4. Napište kód, který se snadno testuje 160
3. Napište testy řadičů jako testy „šedé skříňky“ 162
2. Zřetězte testy řadičů dohromady 163
1. Napište samostatnou sadu integračních testů 165
Shrnutí 166

Kapitola 7
Testování přijatelnosti: rozvinutí scénářů v případu užití 167
10 nejdůležitějších „úkolů“ při testování scénářů 169
Případy užití aplikace Mapplet 169
10. Začněte popisným případem užití 170
9. Proveďte transformaci na strukturovaný scénář 173
8. Ujistěte se, že všechny cesty mají kroky 174
7. Přidejte předběžné a následné podmínky 175
6. Vygenerujte diagram činností 175
5. Rozviňte „vlákna“ pomocí externích testů 176
4. Umístěte testovací případ do diagramu testovacích případů 178
3. Ponořte se do zobrazení testování nástroje Enterprise Architect 179
2. Do testovacích scénářů přidávejte podrobnosti 179
1. Vygenerujte dokument s plánem testů 180
Morální ponaučení 181
Shrnutí 184

Kapitola 8
Testování přijatelnosti: obchodní požadavky 185
10 nejdůležitějších „úkolů“ při testování požadavků 186
10. Začněte doménovým modelem 187
9. Napište testy obchodních požadavků 189
8. Požadavky vymodelujte a uspořádejte 189
7. Vytvořte testovací případy z požadavků 192

6. Projděte svůj plán se zákazníkem ... 194
5. Napište ruční popisy testů ... 198
4. Napište automatizované testy požadavků ... 198
3. Exportujte testovací případy ... 199
2. Testovací případy náležitě zpřístupněte ... 199
1. Zapojte svůj tým! ... 200
Shrnutí ... 200

Část III
Pokročilá metodika DDT

Kapitola 9
Antivzory při testování jednotek ... 205
Chrám zkázy (neboli kód) ... 206
Celkový pohled ... 207
Třída HotelPriceCalculator ... 208
Podpůrné třídy ... 209
Servisní třídy ... 210
Antivzory ... 212
10. Komplexní konstruktor ... 212
9. Stratosférická hierarchie tříd ... 213
8. Statický spouštěč ... 215
7. Statické metody a proměnné ... 217
6. Návrhový vzor Singleton ... 218
5. Těsně svázaná závislost ... 221
4. Obchodní logika v kódu uživatelského rozhraní ... 223
2. Servisní objekty deklarované jako finální ... 225
1. Nedomyšlené funkce dobromyslného programátora ... 225
Shrnutí ... 225

Kapitola 10
Návrh s ohledem na snazší testování ... 227
Seznam deseti nejdůležitějších úkolů pro „návrh s ohledem na testování“ ... 228
Chrám zkázy – důkladně pročištěný ... 229
Případ užití – pochopení toho, co vlastně chceme dělat ... 230
Identifikování testů řadičů ... 231
Test výpočtu celkové ceny ... 232
Test načtení poslední ceny ... 233

Návrh s ohledem na snazší testování ... 234
10. Inicializační kód udržujte mimo konstruktor ... 235
9. S dědičností šetřete ... 236
8. Vyvarujte se statických inicializačních bloků ... 237
7. Používejte metody a proměnné na úrovni objektu ... 238
6. Nepoužívejte návrhový vzor Singleton ... 238
5. Udržujte své třídy oddělené ... 239
4. Nedávejte obchodní logiku do kódu uživatelského rozhraní ... 241
3. Použijte testování černé skříňky a šedé skříňky ... 246
2. Modifikátor „final“ používejte pro konstanty a obecně jím neoznačujte komplexní typy, jako jsou servisní objekty ... 246
1. Držte se případů užití a návrhu ... 247
Podrobný návrh pro případ užití Quote Hotel Price ... 247
Test řadiče pro výpočet celkové ceny ... 249
Test řadiče pro načtení poslední ceny ... 249
Předělaný návrh a kód ... 250
Shrnutí ... 251

Kapitola 11
Automatizované integrační testování ... 253
Seznam 10 nejdůležitějších „úkolů“ pro integrační testování ... 254
10. Testovací vzory hledejte ve svém konceptuálním návrhu ... 255
9. Nezapomeňte na testy zabezpečení ... 256
8. Rozhodněte, která „úroveň“ integračních testů se má psát ... 258
7. Integrační testy na úrovni řadičů či jednotek odvoďte z konceptuálního návrhu ... 259
6. Testy scénářů odvoďte ze scénářů případů užití ... 262
5. Napište celkové testy scénářů ... 263
4. Použijte testovací framework „přátelský k obchodnímu hledisku“ ... 267
3. Kód grafického uživatelského rozhraní testujte v rámci testů scénářů ... 270
2. Nepodceňujte obtížnost integračního testování ... 270
1. Nepodceňujte hodnotu integračních testů ... 274
Hlavní body při psaní integračních testů ... 274
Shrnutí ... 276

Kapitola 12
Testování algoritmů 277
10 nejdůležitějších „úkolů“ při testování algoritmů 278
10. Začněte řadičem z konceptuálního návrhu 279
9. Rozviňte řadiče do návrhu algoritmu 281
8. Diagram volně propojte s doménovým modelem 282
7. Rozdělte rozhodovací uzly obsahující více než jednu kontrolu 284
6. Pro každý uzel vytvořte testovací případ 284
5. Pro každý testovací případ definujte testovací scénáře 285
4. Vytvořte vstupní data z nejrůznějších zdrojů 288
3. Přiřaďte logický tok do jednotlivých metod a tříd 289
2. Napište testy jednotek jako testy „bílé skříňky“ 293
1. Aplikujte metodiku DDT na další návrhové diagramy 303
Shrnutí 304

Příloha
Alenka v říši případů užití 305
Úvod 306
Část 1 306
Alenka při čtení usnula 307
Příslib vývoje řízeného případy užití 307
Model analýzy spojuje text případů užití s objekty 308
Jednoduché a přímočaré 308
Stereotypy <<includes>> a <<extends>> 308
Máme zpoždění! Musíme začít programovat! 309
Alenka se podivuje, jak se dostat od případů užití ke kódu 309
Abstraktní… podstata 309
Až příliš abstraktní? 310
Teleocentricita… 310
Skutečně je nutné to vše specifikovat pro každý případ užití? 311
Část 2 312
Alenka dostala žízeň 312
Alenka pociťuje mdloby 313
Imagine… (s omluvou Johnu Lennonovi) 313
Párové programování znamená, že se nikdy nezaznamenají požadavky 314
Na zaznamenání požadavků není čas 315
Stejně tak můžete říci, že „kód je návrh“ 315
Koho zajímají případy užití? 316
Projekt C3 ukončen 317

Jen a pouze jedinkrát? ... 318
Alenka odmítá začít programovat bez zapsaných požadavků ... 318
Spáchala jsi VPN… ... 320
Model CMM je mrtvý! Pryč s její hlavou! ... 321
Seriózní refaktorování návrhu ... 321
Část 3 ... 321
Alenka se probouzí ... 322
Uzavření díry mezi „Co“ a „Jak“ ... 323
Statické a dynamické modely jsou spojené dohromady ... 323
Přidělování chování se odehrává na diagramu posloupností ... 323
Ponaučení, které z toho plyne ... 324

Závěr
Byl smažno a lepě svyhlé testy ... 325

Rejstřík ... 331

Úvod

Jim je jako programový manažer pro mapovací softwarový produkt ArcGIS společnosti ESRI odpovědný za každodenní sestavování 20 milionů řádků kódu. Dříve spravovat dopravní a logistické oddělení pro ESRI Professional Services, kde se mu podařilo dokončit bez překročení termínu a rozpočtu řadu softwarových projektů v hodnotě milionů dolarů, přičemž používal techniky návrhu popsané v této knize.

Spousta lidí má řadu vyhraněných názorů na prakticky všechny stránky testování softwaru. Viděl jsem nebo slyšel o různých metodikách, systémech, procedurách, procesech, vzorech, nadějích, modlitbách i prostém štěstí, že se některé cesty kódu nikdy neprovedly. S veškerou touto pomocí musíme přece vymyslet, jak dodávat výjimečný, neprůstřelný a bezchybný software, ne? Přesto s každým novým vydáním další skvělé, nejstabilnější revize našeho softwarového produktu dělají testovací inženýři stále takový ten bolestivý obličej (znáte to) a při otázce „Máš příslušné testy napříč všemi vrstvami softwaru?“ se stahují zpět.

Příčinou bolestivého obličeje je fakt, že pravdivou odpovědí na tuto otázku je téměř vždy: „Myslím, že ano, ale ve skutečnosti neznám všechny oblasti, pro které bych měl mít testy.“ A software se i tak vydá, prevít jeden. To je pak pracovní jistota pro tým technické podpory, protože chyby ve vydaném produktu se vrátí přímo k vašemu týmu pro další servisní balíček nebo nouzovou rychlou opravu. Jde to ale dělat mnohem lépe a tato kniha vám ukáže, jak.

Tato kniha vás provede osvědčeným procesem vývoje softwaru s názvem ICONIX Process a zaměří se na tvorbu a údržbu testů jednotek a testů přijatelnosti založených a řízených návrhem softwaru. Jedná se o testování řízené návrhem (Design-Driven Testing – DDT). Svůj návrh tak využijete pro přesné vymezení míst, kde je nutné založit kritické testy na návrhu a chování objektů. Nejedná se o návrh řízený testy (Test-Driven Design – TDD), kdy se nejdříve píší testy jednotek, a to ještě před dokončením návrhu a zahájením programování. Nevím, jak je to u vás, ale předvídání budoucnosti je myslím docela obtížné, a přimět softwarové inženýry, aby naprogramovali něco, co „pasuje“ do určité sady testů, je dokonce ještě těžší.

Zatímco spousta lidí má na testování různé názory, je tu jedna věc, na které se všichni asi shodneme: testování je často velice obtížné a složité. Jako programový manažer velkého vývojového týmu vím, jak rychle se může testování vymknout z rukou nebo prostě pozastavit celý projekt. Organizace vydávají na testování velmi rozdílné prostředky a naneštěstí existují i velké rozdíly, co se týče návratnosti takto investovaných prostředků. Je možné provádět příliš mnoho testování a investované prostředky tak promarnit. Avšak pravděpodobnější je, že se provede příliš málo testování (samozřejmě s představou, že se udělalo více než dost), v nesprávných oblastech softwaru a s nedostatečnými investicemi. K tomu může dojít proto, že prostě nevíte, kam testy umístit pro vyvážené investice a získat tak správné testovací pokrytí.

Tato kniha vám na příkladu návrhu a vybudování skutečné mapové webové aplikace ukáže, jak docílit této vyváženosti a optimalizovat návratnost investic do testování. Při použití metodik ICONIX Pro-

cess a DDT je naprosto jasné, jaké testy jsou nutné a kde mají být. Ba co víc, řadu z těchto testů za vás automaticky vygenerují používané nástroje (v tomto případě nástroj Enterprise Architect od společnosti Sprax Systems), které (kromě toho, že jde o naprosto skvělé nástroje) jsou pro váš projekt nesmírně cenné. Pokud tedy potřebujete vytvořit skvělý software s použitím agilního procesu, kdy se vám testy vygenerují téměř zadarmo, pak je tato kniha právě pro vás.

Jim McKinney
Programový manažer produktu ArcGIS, společnost ESRI

O autorech

Matt Stephens je softwarový konzultant s finančními organizacemi v Centrálním Londýně a zakladatel nezávislého knižního vydavatelství Fingerpress (*www.fingerpress.co.uk*). Psal pro mnoho časopisů a webů, například pro *The Register* a *Application Development Trends*. Na Internetu jej najdete na webu společnosti Software Reality (*http://articles.softwarereality.com*).

Doug Rosenberg v roce 1984 ve svém obývacím pokoji založil společnost ICONIX (*www.iconixsw.com*). Po několika letech práce na tvorbě CASE nástrojů začal školit společnosti v objektově orientované analýze a návrhu, která se specializuje na školení v oblasti jazyků UML a SysML a nabízí jak místní kurzy, tak i kurzy s otevřeným zápisem. V roce 1993 vyvinul jednotný Booch/Rumbaugh/Jacobson přístup k modelování, což bylo o několik let dříve, než se objevil jazyk UML, a kolem roku 1995 začal psát knihy.

Testování řízené návrhem je šestou knihou z oblasti softwarového inženýrství (a čtvrtou s Mattem Stephensem). Kromě toho vytvořil bezpočet multimediálních tutoriálů, jako je například *Enterprise Architect for Power Users*, a několik elektronických knih, mezi něž patří kupříkladu *Embedded Systems Development with SysML*.

Když nepíše a neučí, tak rád vytváří panoramatické fotografie virtuální reality, které si můžete prohlédnout na jeho webu pro cestovatele *VResorts.com*.

O odborných korektorech

Jeffrey P. Kantor je manažer projektu správy dat teleskopu LSST (Large Synoptic Survey Telescope – velký synoptický přehledový teleskop). V této funkci je odpovědný za implementování výpočetních a komunikačních systémů zajišťujících kalibraci, hodnocení kvality, zpracování, archivaci a přístup koncových uživatelů a externích systémů k astronomickým obrazům a inženýrským datům produkovaným teleskopem LSST.

Po čtyřech letech v Americké armádě, kde působil jako specialista zpravodajské služby na ruský jazyk a signály, začal v roce 1980 jako programátor na základní úrovni, načež zastával pozice na všech úrovních IT organizací v řadě průmyslových odvětví, včetně obrany a letectví, výroby polovodičů, geofyziky, poradenství v oblasti softwarového inženýrství, řízení domácností a budov, zboží dlouhodobé spotřeby, prodeje a elektronické komerce.

Vytvářel, upravoval na míru, aplikoval a auditoval softwarové procesy pro široké spektrum organizací v průmyslu, vládě a školství. U některých z těchto organizací byl odpovědný za dosažení certifikace ISO9000 a úrovně CMM Level 2 institutu SEI. Dále poskytoval poradenství a školení více než 30 společnostem v oblasti objektově orientované analýzy a návrhu, modelování pomocí jazyka UML, testování řízené případy užití a řízení softwarových projektů.

Rád tráví čas se svou rodinou, u fotbalu (jako hráč, sudí a trenér) a jezděním na kole po horách.

David Putnam pracoval posledních 25 let ve vývoji softwaru a od začátku tohoto století je jedním z nejhorlivějších zastánců agilního přístupu v Anglii.

Od začátku bylo jeho zaměření velmi různorodé: vytvářel databázové systémy pro stavební průmysl, pracoval ve fitness odvětví a také přednášel na univerzitě. Během té doby publikoval řadu článků o vývoji softwaru a je známou tváří na konferencích o agilním vývoji softwaru. Nyní pracuje jako nezávislý poradce v oblasti agilního vývoje softwaru a poskytuje neocenitelné rady několika z nejznámějších společností v Anglii.

Poděkování

Autoři by rádi poděkovali:

Týmu společnosti ESRI odpovědnému za projekt Mapplet (Wolfgang Hall, Prakash Darbhamulla, Jim McKinney a Witt Mathot) za jeho práci na ukázkovém projektu v této knize.

Lidem ze společnosti Sparx Systems (Geoff Sparks, Ben Constable, Tom O'Reilly, Aaron Bell, Vimal Kumar a Estelle Gleeson) za vývoj doplňku Agile/ICONIX a editoru strukturovaných scénářů.

Alanahu Stephensovi, který byl naprosto nejlepší „Alenkou", jaká kdy byla, a Michelle Stephensové, za její trpělivost během dalšího knižního projektu.

Jeffu Kantorovi a Robynu Allsmanovi z projektu LSST za jejich pomoci se specifikací expandéru vláken případu užití.

Miku Farnsworthovi a lidem ze společnosti Virginia DMV, kteří se podíleli na předběžném modelování aplikace Mapplet. Barbaře Rosi-Schwartzové za její zpětnou vazbu ohledně metodiky DDT a Jerry Hambymu za sdílení jeho odborných znalosti technologie Flex a zvláště za jeho pomoc se zpětnou analýzou kódu jazyka MXML.

A v neposlední řadě našim redaktorům v nakladatelství Apress: Jonathanu Gennickovi, Anitě Castrové a Mary Ann Fugatové.

Předmluva

Střež se nadšeného povyku kolem agilnosti

Bylo smažno, když se YAGNIdní kód
desetkrát za den sám sestavil.
Indexové kartičky ve své křehkosti,
odklizené, refaktorované návrhy.

Ó synu, střež se nadšeného povyku kolem agilnosti,
více kódu s pachem, více chyb na odchycení.
Refaktorování bylo mnohem zábavnější,
než byly schopnosti tvé na znak poražené.

Své chopil testy jednotek,
proti litým chybám bojoval.
Zmatený přístupem TDD,
zdravý rozum vyhledal.

Ale v jeho dřevěném okně časovém,
určeném hrou kde vládl plán,
ony testy zezelenaly a vše bylo májové,
dokud konečný termín nenastal.

Je půl třetí, máme hotovo,
je čas na svačinku chutnou.
Že testy zčervenaly, pěl jsi, mátoho,
vše rychlým hackem utnou!

A hle, šokující myšlenka jak sen,
ne testy, ale nejdříve návrh!
V této báječný den,
jednodušší proces byl zaveden.

Bylo smažno, když se YAGNIdní kód
desetkrát za den sám sestavil.
Indexové kartičky ve své křehkosti,
odklizené, refaktorované návrhy…

ČÁST I

DDT vs. TDD

„Ať se vývojáři zabývají konceptuálním návrhem," řekl Král, toho dne již asi podvacáté.
„Ne, ne!" oponovala Královna. „Nejdříve testy – potom návrh."
„Hloupost!" zvolala Alenka. „Ten nápad začínat s psaním testů!"

„Mlč!“ řekla Královna, která už začala měnit barvu. „Mimochodem, kolik kódu jsi v poslední době napsala?“ jízlivě se ušklíbla.

„Nebudu mlčet,“ ohradila se odvážně malá Alenka. „Testy by neměly řídit návrh, ale návrh by měl řídit testování. Testy by totiž měly ověřit, že kód pracuje tak, jak je navržený, a také že splňuje požadavky zákazníka,“ dodala, překvapená svým vhledem do podstaty věci. „Když navíc své testy odvodíte od konceptuálního návrhu, nebude testování obtížnější, ale chytřejší.“

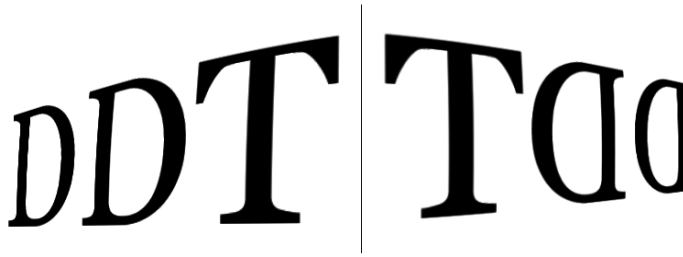
Tato kniha je o testování, a to nejen o vizuální kontrole ve stylu QA, ale také o automatizovaném testování – řízení testů jednotek, testů řadičů a testů scénářů podle vašeho návrhu a požadavků zákazníka. Mezi technikami popsanými v této knize a tím, do čeho se lidé pouštějí při vývoji řízeném testy (TDD – Test-Driven Development), existují určité styčné plochy. V jistých ohledech fungují oba procesy dobře společně, přičemž doufáme, že vývojáři softwaru řízeného testy budou mít užitek z kombinace toho nejlepšího z obou těchto světů. Na druhou stranu existuje několik základních odlišností, a to jak z praktického, tak i z ideového hlediska obou disciplín.

Kapitola 1 nabízí vysokoúrovňový pohled na testování řízené návrhem (DDT). Kromě toho si stručně představíme projekt Mapplet, s jehož pomocí si budeme v pozdější části knihy vysvětlovat, jak z testování řízeného návrhem vytěžit to nejlepší.

Kapitoly 2 a 3 dále srovnávají testování řízené návrhem a vývoj řízený testy. V kapitole 2 si ukážeme, jaké to je, když se k projektu přistupuje pomocí vývoje řízeného testy. Na konci kapitoly snad již budete přesvědčeni o tom, že jistě musí existovat lepší způsob. A ten skutečně existuje! Stejný scénář si projdeme znovu v kapitole 3, avšak tentokrát s použitím testování řízeného návrhem. Výsledky budou mnohem uspokojivější.

KAPITOLA 1

Někdo to má pozpátku



Když jsme poprvé viděli popis vývoje řízeného testy (TDD), okamžitě nás napadlo: „To je jen pozpátku!“ Rozhodli jsme se tento proces podrobit zkoušce, a proto Matt vyrazil vpřed a nasadil vývoj řízený testy do praxe v rámci svých vlastních projektů, navštěvoval semináře, držel krok s nejnovějšími trendy v této oblasti a tak dále. Jenže nepříjemný pocit, že přece jen není úplně správné řídit návrh pomocí testů, ani tak nezmizel. Kromě toho jsme měli pocit, že vývoj řízený testy je velmi pracný, neboť vývojáře vábí do drahého a v důsledku bezvýznamného štvaní se za svatým grálem 100% pokrytí testy. Ne každý kód se vytváří stejně, přičemž některý kód má z pokrytí testy větší užitek nežli jiný.[1] Musí prostě existovat lepší způsob, jak mít z automatizovaného testování užitek.

1 Myslící algoritmy vs. výplňový kód, jako jsou rutiny pro získávání a nastavování vlastností.

Výsledkem bylo testování řízené návrhem (DDT): splynutí přímé analýzy a návrhu s agilním, testy řízeným myšlenkovým přístupem. V mnoha ohledech jde o obrácený přístup stojící v pozadí vývoje řízeného testy, což je také důvod, proč jsme si s tímto názvem užili trošku legrace. Přesto to má však někdo i nadále nepochybně pozpátku. Projděte si několik následujících kapitol a sami se rozhodněte, kdo směřuje kupředu a kdo nazpět.

V této kapitole se podíváme na vysokoúrovňový přehled testování řízeného návrhem a porovnáme hlavní rozdíly mezi testováním řízeným návrhem a vývojem řízeným testováním. Testování řízené návrhem pokrývá také analýzu a testování přijatelnosti, a proto někdy porovnáme testování řízené návrhem s praxí, která se obvykle používá v souvislosti s vývojem řízeným testováním (což je výsadou extrémního programování nebo novějších variant, jako je vývoj řízený testy přijatelnosti – Acceptance TDD).

Poté si v kapitolách 2 a 3 projdeme příklad ve stylu „Ahoj, světe!“, u kterého použijeme nejdříve vývoj řízený testy a poté testování řízené návrhem.

Zbývající část knihy nabídne testování řízené návrhem v akci s uceleným příkladem založeným na skutečném projektu, kterým je aplikace pro vyhledávání hotelů postavená na heterogenním prostředí s technologií Flex na straně klienta a Javou na straně serveru, přičemž mapy se servírují ze serveru ArcGIS a dotazy se provádějí nad hotelovou databází na bázi XML. Jedná se o netriviální projekt se směsicí technologií a jazyků (což je v současnosti stále častější) nabízející řadu výzev, které testování řízené návrhem řádně prověří.

Problémy, které řeší testování řízené návrhem

Testování řízené návrhem (metodika DDT) je v jistých směrech odpovědí na problémy, které se objevily u jiných testovacích metodologií, jako je vývoj řízený testy (metodika TDD). V nemenší míře však jde také o odpověď na mnohem větší problémy, k nimž dochází, když se

- neprovádí žádné testování (nebo se nenapíšou žádné automatizované testy)
- provádí nějaké testování (nebo se napíšou nějaké testy), které je ale celé tak trochu bezcílné a nahodilé
- provádí *příliš mnoho* testování (nebo se napíše příliš mnoho testů, které jsou málo účinné)

Poslední situace může vypadat poněkud zvláštně – přece nemůže existovat něco takového jako příliš mnoho testování? Je-li však testování kontraproduktivní nebo repetitivní, mohl by být dodatečný čas strávený testováním promarněný (zákon klesající návratnosti). Existuje přece tolik způsobů, jak stisknout zvonek u dveří, jenže kolik z nich se bude provádět ve skutečném životě? Je skutečně nezbytné prokázat, že zvonek u dveří bude fungovat i 100 metrů pod vodou? Myšlenka stojící za testováním řízeným návrhem spočívá v tom, že testy, které píšete a provádíte, jsou úzce spjaté s požadavky zákazníka, takže strávíte čas testováním pouze toho, co je nutné testovat.

Podívejme se na některé z problémů, které byste měli být schopni vyřešit pomocí testování řízeného návrhem.

Vědět, kdy je hotovo, je obtížné

Při psaní testů je někdy nejasné, kdy máte „hotovo“. Mohli byste tak donekonečna pokračovat v psaní testů všeho druhu, dokud by vaše kódy nebyly stoprocentně pokryté testy. Proč se zde ale zastavo-

vat? Existují přece neprobádané permutace k otestování, na napsání čekají další testy a tak pořád dál, až zákazník čekání na dodání produktu vzdá a odejde. Při testování řízeném návrhem jsou vaše testy řízené přímo případy užití a konceptuálním návrhem. Vše je docela systematické, takže budete naprosto přesně vědět, kdy máte hotovo.

Poznámka

U klienta, kterého Doug Rosenberg nedávno navštívil, se tým odpovědný za projekt rozhodl, že s testováním přijatelnosti je hotový v okamžiku, kdy jejich timebox signalizoval, že by měli raději začít programovat. Máme podezření, že podobně to funguje docela často.

Pokrytí kódu se stalo synonymem pro „kvalitní práci". Nástroje pro metriky kódu, jako je Clover, poskytují zprávy, které si mohou přehorliví manažeři vytisknout, srolovat a mlátit jimi vývojáře po hlavě, nepíšou-li dostatek testů jednotek. Je-li však 100% pokrytí kódu prakticky nedosažitelné, mohla by vám být prominuta následující otázka: Proč se o to vůbec snažit? Proč předem prohlašovat, že tohoto cíle nelze nikdy dosáhnout? Naproti tomu jsme chtěli, aby testování řízené návrhem poskytovalo jasný, dosažitelný cíl. „Úplnost" v testování řízeném návrhem není o všeobecném pokrytí kódu, ale o zajištění, aby byla adekvátně otestována klíčová místa v kódu (logické softwarové funkce).

Ponechání testování na později je dražší

Stále se můžete setkat se softwarovými procesy, které kladou „testování" do samostatné fáze až za fáze s požadavky, návrhem a programováním. Je všeobecně známo, že ponechání hledání a opravy chyb v projektu na později zvýší dobu a náklady spojené s jejich odstraňováním. Z hlediska intuice je sice zřejmé, že nemůžete testovat něco ještě dříve, než to začne existovat, ale testování řízené návrhem (podobně jako vývoj řízený testováním a další agilní procesy) vstupuje do zákoutí a štěrbin vývoje a nabízí brzkou zpětnou vazbu pro stav vašeho kódu a návrh.

Testování špatně navrženého kódu je těžké

Zní to sice samozřejmě, ale kód, který je špatně navržený, bývá strnulý, obtížně adaptovatelný nebo znovupoužitelný v jiném kontextu a plný vedlejších efektů. Naproti tomu testování řízené návrhem přirozeně podporuje kvalitní návrh a dobře napsaný, snadno testovatelný kód. Pro toto všechno se nesmírně obtížně píšou testy. Ve světě vývoje řízeného testy bude vytvořený kód přirozeně testovatelný, protože jste nejdříve napsali testy. Skončíte však s děsivou hromadou testů jednotek pochybné hodnoty, přičemž bude lákavé přeskočit cenné části myšlenkového procesu analýzy a návrhu, neboť přece „kód je návrh". U testování řízeného návrhem jsme chtěli testovací proces, který přirozeně podporuje kvalitní návrh a dobře napsaný, snadno testovatelný kód.

Poznámka

Každý rozjetý projekt, k němuž se Matt Stephens po čase připojil, byl bez výjimky napsaný tak, že testování kódu bylo velice obtížné (nebo prakticky nemožné). Programátoři se často snaží přidávat testy do svého kódu, načež to rychle vzdají s tvrzením, že testování jednotek je příliš obtížné. Jedná se o obecně rozšířený problém, takže kapitolu 9 věnujeme problému obtížně testovatelného kódu a podíváme se, proč určité styly programování a návrhové vzory činí testování jednotek obtížné.

Je snadné opomenout testy na úrovni zákazníka

Vývoj řízený testy je již ze své podstaty celý o testování na úrovni podrobného návrhu. Neradi to říkáme, ale vývoj řízený testy ztratil při svém oddělení od extrémního programování cenného souputníka: testy přijatelnosti. Knihy o vývoji řízeném testy tuto zásadní stránku automatizovaného testování zcela opomíjejí a místo toho hovoří o výběru uživatelského příběhu (neboli požadavku) a okamžitém napsání testu jednotek pro něj.

Testování řízené návrhem podporuje psaní testů přijatelnosti i testů jednotek, avšak v jeho jádru jsou **testy řadičů**, které stojí někde v půli. Testy řadičů jsou „vývojářské testy", které sice vypadají jako testy jednotek, pracují však na úrovni konceptuálního návrhu (neboli „logických softwarových funkcí"). Poskytují velmi užitečné pojivo mezi analýzou (prostor problému) a návrhem (prostor řešení).

Vývojáři bývají samolibí

Není nijak neobvyklé, že když vývojáři napíšou pár testů, zjistí, že nedosáhli žádných hmatatelných výsledků, a vrátí se zpět k chrlení neotestovaného kódu. Z naší zkušenosti vyplývá, že zvláště „svatý grál" 100% pokrytí kódu může u vývojářů vést k pěstování samolibosti. Je-li 100% pokrytí nemožné nebo nepraktické, je 90% v pořádku? A co 80%? Pro tyto třídy jsem testy sice nenapsal, ale vesmír se (zatím) nezhroutil, takže proč se tím vůbec znepokojovat? Je-li cíl nastavený vaším testovacím procesem snadno a jasně dosažitelný, měli byste zjistit, že vývojáři ve vašem týmu se jej snaží docílit s větším smyslem pro účelnost. Tím se dostáváme k poslednímu problému, s nímž si testování řízené návrhem dokáže poradit.

Testy někdy postrádají smysl

Bezúčelné testování je někdy horší než žádné testování, protože dává iluzi bezpečnosti. To platí jak pro ruční testování (kdy tester postupuje podle popisu testu nebo jen klepe na uživatelské rozhraní a posuzuje produktu k vydání), tak i pro psaní automatizovaných testů (kdy vývojáři nesystematickým způsobem napíšou hromadu testů).

Bezúčelné testování jednotek je také problém, protože testy jednotek znamenají více kódu pro údržbu a mohou ztížit úpravu stávajícího kódu, aniž by došlo k narušení testů, které se opírají o příliš mnoho předpokladů o útrobách daného kódu. Psaní samotných testů nadto spotřebovává cenný čas.

Znalost toho, proč testujete a proč píšete konkrétní test jednotky (schopnost stručně vyjádřit, co daný test prokazuje), zajistí, že každý test musí unést svou vlastní váhu. Je nutné ospravedlnit jeho existenci a čas strávený jeho psaním. Účel testů při testování řízeném návrhem je jednoduchý: systematicky prokázat, že návrh splňuje požadavky a že kód odpovídá návrhu.

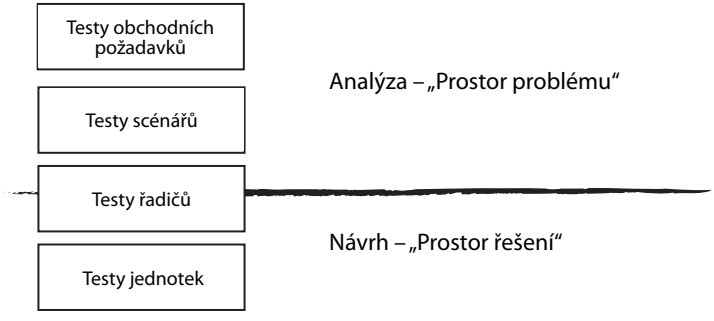
Rychlý přehled metodiky DDT bez ohledu na nástroje

Tato část nabízí bleskovou prohlídku procesu testování řízeného návrhem, zhuštěného až na úroveň jednotlivých kroků. Testování řízené návrhem lze sice adaptovat do libovolného procesu OOAD, původně byl ale navržený pro použití s metodikou ICONIX Process (agilní proces OOAD, který

používá základní podmnožinu prvků jazyka UML).[2] V této části si ukážeme každý krok v metodice ICONIX Process s odpovídajícím krokem testování řízeného návrhem. Jde o to, že testování řízené návrhem nabízí okamžitou odezvu, při čemž se ověřuje každý krok v procesu analýzy či návrhu.

Struktura testování řízeného návrhem

Obrázek 1.1 znázorňuje čtyři hlavní artefakty testování: testy jednotek, testy řadičů, testy scénářů a testy obchodních požadavků. Jak je z tohoto obrázku patrné, testy jednotek jsou v podstatě zakořeněné v prostoru návrhu/řešení/implementace. Programátoři je píšou a „vlastní". Nad nimi jsou mezi prostorem analýzy a prostorem návrhu vložené testy řadičů, které pomáhají poskytovat přemostění mezi oběma prostory. Testy scénářů náleží do prostoru analýzy. Jedná se o ručně specifikované testy představující rozvinutí optimistických/pesimistických permutací daného případu užití a obsahující pokyny ve formě jednotlivých kroků, podle kterých mají testeři postupovat. Jakmile se s procesem seznámíte a organizace bude této myšlence více nakloněna, vřele doporučujeme založit celkové integrační testy na specifikacích testů scénářů. A nakonec, testy obchodních požadavků mají téměř vždy podobu ručních specifikací testů. Usnadňují „kontrolu zdravým rozumem" před zapečetěním produktu pro vypuštění do divočiny.



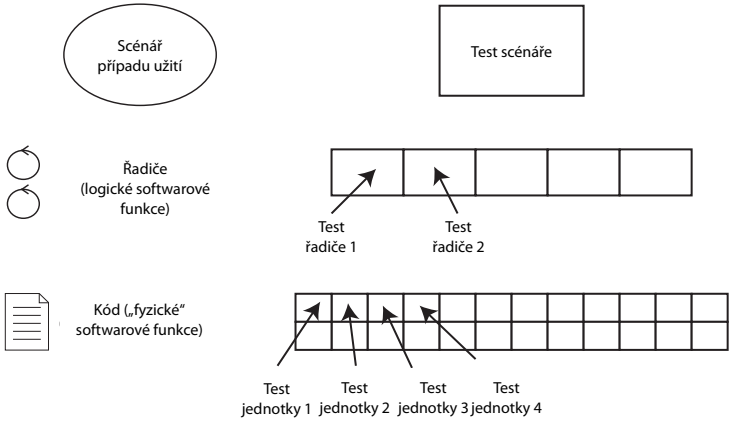
Obrázek 1.1: Čtyři hlavní příchutě testování v DDT

Testy se liší granularitou (tj. množstvím kódu, který každý test ověřuje) a počtem automatizovaných testů, které jsou ve skutečnosti napsané. Toto ukazuje obrázek 1.2.

Z obrázku 1.2 je rovněž patrné to (pokud jsou vaše popisy testů scénářů implementované také jako automatizované integrační testy), že každý případ užití pokrývá právě jedna testovací třída (ve skutečnosti se jedná o jeden testovací případ pro každý scénář případu užití; více o tom ale až později). Při použití metodiky ICONIX Process se případy užití rozdělí do řadičů, přičemž každý test řadiče (dle očekávání) pokrývá právě jeden řadič. Řadiče se pak rozdělí na samotné funkce či metody kódu s tím, že významnější metody získají jednu či více metod testu jednotky.[3]

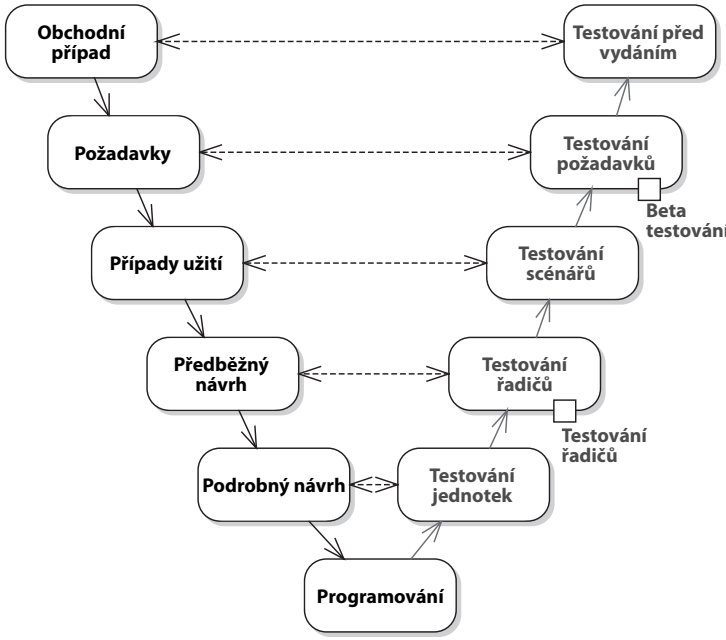
2 V této knize je obsažen dostatek informací o metodice ICONIX Process k tomu, abyste mohli rozjet testování řízené návrhem ve vlastních projektech. Pokud se však chcete o metodice ICONIX Process dozvědět více, sáhněte po doprovodné knize s původním názvem *Use Case Driven Object Modeling with UML: Theory and Practice*.

3 Možná vás zajímá, jak se pozná, zda danou metodu pokrýt testem jednotky, nebo testem řadiče? Jednoduše: nejdříve implementujte testy řadiče, a zůstane-li nějaká „významná" část kódu nepokrytá, tak zvažte, zda jí nepřidělit test jednotky. Více se o těchto místech rozhodování dozvíte v kapitolách 5 a 6.



Obrázek 1.2: Granularita testů: čím blíže se dostáváte k návrhu/kódu, tím více testů se píše, ovšem s tím, že každý test má menší rozsah působnosti

Pravděpodobně jste se již setkali s tradičním modelem vývoje softwaru znázorněným ve tvaru písmene „V“, kdy jsou činnosti spojené s analýzou, návrhem a vývojem na levé straně a činnosti spojené s testováním na straně pravé. S troškou snahy zapadá testování řízené návrhem do tohoto modelu docela dobře (viz obrázek 1.3).



Obrázek 1.3: Model ve tvaru písmene „V“ pro vývoj přizpůsobený testování řízenému návrhem

Každý krok na levé straně je krokem v metodice ICONIX Process, kterou doplňuje testování řízené návrhem, jehož kroky jsou uvedené na pravé straně. Při vytváření požadavků vytváříte specifikace testů požadavků poskytující zpětnou vazbu, abyste tak na všeobecné úrovni věděli, kdy je projekt

hotový. Skutečné jádro testování řízeného návrhem (tedy alespoň pro programátory) se točí kolem testování řadičů, které poskytuje zpětnou vazbu pro krok konceptuálního návrhu a systematickou metodu testování chování softwaru. Hlavní část testování řízeného návrhem se z hlediska analytiků a testerů QA točí kolem testování scénářů.

Metodika DDT v akci

Nyní si postupně projdeme jednotlivé kroky metodiky ICONIX/DDT. Budeme vycházet z diagramu ve tvaru písmene „V" na obrázku 1.3. Zcela vědomě zde nezapojujeme žádné nástroje. Klidně si ale nalistujte místo v kapitole 3, kde si ukazujeme příklad „Ahoj, světě!" (ve skutečnosti jde o případ užití pro přihlášení) pro testování řízené návrhem v akci s použitím nástroje Enterprise Architect. Pojďme si tedy projít jednotlivé kroky.

1. Metodika ICONIX → Podrobně prostudujte obchodní požadavky (tj. funkční a nefunkční požadavky). Promluvte si s příslušnými lidmi – zákazníky, analytiky, koncovými uživateli a tak dále. Nakreslete storyboardy uživatelského rozhraní (náčrtky) a vytvořte sadu požadavků na vysoké úrovni neboli uživatelské příběhy.

   ← *Metodika DDT*: Z požadavků vytvořte testovací případy. Mělo by jít o kritéria přijatelnosti, která si na širší úrovni můžete jedno po druhém odškrtnout a ověřit si tak, že projekt je hotový.
2. Metodika ICONIX → Vytvořte doménový model a napište několik případů užití. Případ užití si můžete představit jako popis jednotlivých kroků interakce uživatele/systému: „Uživatel stiskne tlačítko; systém provedete nějakou věc a poté zobrazí výsledky. Uživatel udělá něco jiného… atd." Každý případ užití rozdělte na jeho Základní průběh („optimistický scénář") a tolik Alternativních průběhů („pesimistický scénář"), kolik si dokážete vymyslet.

   ← *Metodika DDT*: Rozviňte vlákna případů užití do testů scénářů. Jedná se o specifikace testů, které předáte testovacímu týmu a které mohou být rovněž automatizovanými, celkovými integračními testy.[4] Všimněte si, že samotné popisy případů užití jsou velmi užitečné jako specifikace testů, neboť (napsáno způsobem metodiky ICONIX) se čtou jako průvodce jednotlivými kroky pro použití systému a pro prozkoumání nejrůznějších přístupů a režimů úspěchu/neúspěchu.
3. Metodika ICONIX → Pro každý případ užití začněte studiem návrhu na předběžné úrovni. Při použití metodiky ICONIX Process se konceptuální návrh vytvoří pomocí analýzy robustnosti. Jedná se o efektivní techniku, která slouží jako „kontrola realitou" pro popisy vašich případů užití. Kromě toho vám pomůže identifikovat chování (slovesa, činnosti či „řadiče") ve vašem návrhu. Řadič si můžete představit jako „logickou softwarovou funkci", která se v kódu může proměnit do několika „skutečných" funkcí.

   ← *Metodika DDT*: Z diagramů robustnosti systematicky vytvářejte testy řadičů, a to následujícím způsobem:

   - Pro každý řadič vytvořte testovací případ. Tyto testovací případy ověřují kritické chování vašeho návrhu řízené řadiči identifikovanými ve vašich případech užití během analýzy robustnosti (odtud název „test řadiče"). Každý test řadiče se vytváří jako metoda vámi zvoleného frameworku pro testování jednotek (JUnit, FlexUnit, NUnit, atd.). Vzhledem k tomu, že každý test pokrývá „logickou" softwarovou funkci (tj. skupinu „skutečných" funkcí s jed-

4 Integrační testy mají svoji vlastní sadu problémů, které – v závislosti na faktorech, jako je kultura organizace a dostupnost aktuálních, izolovaných schémat testovacích databází – nelze vždy tak snadno překonat. O implementaci testů scénářů jako automatizovaných integračních testů se dozvíte více ve třetí části.

ním souhrnným výstupem), vystačí s menším počtem testů řadičů, než kolik byste napsali testů jednotek v metodice TDD.

- U každého testovacího případu řadiče si promyslete očekávané kritérium přijatelnosti – co představuje úspěšný běh této části případu užití? (Užitečný seznam poskytují dříve vytvořené alternativní průběhy daného případu užití.)

4. Metodika ICONIX → U každého případu užití jděte až k podrobnému návrhu. Je načase „vstoupit do reality“ a pečlivě promyslet každý detail toho, jak bude daný systém implementován. Pro každý případ užití vytvořte diagram posloupností.

← *Metodika DDT*: Z návrhu systematicky vytvářejte testy jednotek.

Pokud znáte vývoj řízený testy, pak vám nejspíše neunikne, že tento proces je zcela odlišný. Ve skutečnosti existuje řada způsobů, ve kterých jsou si tyto dva procesy podobné (co se mechaniky i výchozích cílů týče). Nicméně jsou zde praktické i filozofické rozdíly. Hlavním odlišnostem se budeme věnovat v následující části.

Jak se metodiky TDD a DDT odlišují

Techniky, které popisuje tato kniha, nejsou povětšinou nekompatibilní s vývojem řízeným testy. Ve skutečnosti doufáme, že ti, kteří používají vývoj řízený testy, budou moci tyto principy a techniky vzít a úspěšně je aplikovat ve svých projektech. Jak ukazuje tabulka 1.1, mezi naší vůdčí filozofií a filozofií původních duchovních otců metodiky TDD však existuje několik zásadních rozdílů. Uvedené komentáře si blíže vysvětlíme na začátku kapitol 2 a 3.

Tabulka 1.1: Rozdíly mezi metodikami TDD a ICONIX/DDT

Metodika TDD	Metodika ICONIX/DDT
Testy se používají pro řízení návrhu aplikace.	U testování řízeného návrhem je tomu právě naopak: testy jsou řízené návrhem, a proto je hlavním účelem testů ověřit návrh. Na druhou stranu je mezi oběma metodikami více společného, než si možná myslíte. Spoustu „návrhového čeření“ (tj. refaktorování) vyskytujícího se v projektech na bázi metodiky TDD lze zklidnit a dát mu určitou stabilitu, k čemuž stačí ze všeho nejdříve aplikovat techniky návrhu a testování popsané v této knize.
Kód je návrh a testy jsou dokumentace.	Návrh je návrh, kód je kód a testy jsou testy. U testování řízeného návrhem se pro synchronizaci zdokumentovaného návrhového modelu s kódem používají moderní vývojové nástroje.
Při postupu dle metodiky TDD můžete skončit s množstvím testů (skutečně obrovským množstvím testů).	Metodika DDT se ubírá směrem, který lze charakterizovat jako „testuj chytřeji, ne obtížněji“, což znamená, že testy se více soustředí na „žhavá místa“ kódu.

Metodika TDD	Metodika ICONIX/DDT
Testy metodiky TDD mají svůj vlastní účel. Proto u projektu, v němž se skutečně nejdříve tvoří testy, budou testy vypadat malinko odlišně od „klasických" jemných testů jednotek. Test jednotek metodiky TDD může testovat více než jedinou metodu současně.	V metodice DDT má test jednotky ověřit jedinou metodu. Testy jednotek metodiky DDT jsou blíže „skutečným" testům jednotek. Při psaní každého testu se podíváte na podrobný návrh, vezmete další zprávu předávanou mezi objekty a napíšete pro ni testovací případ. Metodika DDT má také testy řadičů, jejichž obor platnosti je širší.[5] Z hlediska oboru platnosti se tedy testy metodiky TDD nacházejí někde mezi testy jednotek a testy řadičů.
Metodika TDD nemá testy přijatelnosti, dokud do ní nepřimícháte část jiného procesu. Důraz se obvykle klade na automatizované testy přijatelnosti (např. v extrémním programování): pokud „specifikaci spustitelného prvku" (tj. testy přijatelnosti) nelze automatizovat, pak tento proces selhává. Jak si ukážeme ve třetí části, psaní a údržba automatizovaných testů přijatelnosti může být velice obtížná.	„Testy přijatelnosti" metodiky DDT (jež doprovázejí testy scénářů a testy obchodních požadavků) jsou „ruční" specifikace testů určené pro člověka. Testy scénářů je sice možné automatizovat (), ale tato metodika na tom nezávisí.
Co se týče návrhu, je metodika TDD mnohem jemnější.[6] U přístupu, kdy se nejdříve vytvářejí testy, si ze stěny vezmete kartičku uživatelského příběhu, využijete její zadní stranu k prodiskutování kritéria úspěchu s testerem anebo zástupcem zákazníka, napíšete první (selhávající) test, napíšete nějaký kód zajišťující, aby test prošel, napíšete další test, a tak dále, dokud se daná kartička uživatelského příběhu neimplementuje. Poté zkontrolujete návrh a dle potřeby jej refaktorujete (tj. návrh „až poté").	Z našeho pohledu je metodika DDT docela jemná: svůj prvotní návrh můžete postavit třeba na sadě případů užití. Podle výsledného návrhového modelu identifikujete své testy a třídy a naprogramujete je. Při psaní kódu pak spouštíte testy.
Zelený proužek v metodice TDD znamená: „Žádný z testů, které jsem dosud napsal, neselhává."	Zelený proužek v metodice DDT znamená: „Všechny dosud implementované kritické situace návrhu, logické softwarové funkce a interakce uživatele/systému procházejí dle návrhu." Sami víme, díky kterému výsledku máme větší důvěru ve vlastní kód…
Po provedení testovacího průchodu zhodnotíte návrh a v případě potřeby refaktorujete kód.	U metodiky DDT je „návrhové čeření" minimální, protože návrh je promyšlený s přihlédnutím k širší množině funkčnosti. Nijak nepředstíráme, že při zahájení programování nebudou žádné změny v návrhu nebo že se požadavky nikdy nezmění, avšak díky této metodice budou tyto změny minimální. Kromě toho tato metodika provádění změn nijak nebrání (viz kapitola 4).

5 Pokud nějaký kód již pokrývá test řadiče, pak nemusíte psát duplicitní testy jednotek pro pokrytí téhož kódu, nejedná-li se ovšem o algoritmický nebo kritický kód, což je oblast kódu, pro kterou budou dodatečné testy jen přínosem. Testování algoritmů řízené návrhem se budeme věnovat v kapitole 12.

6 Kent Beck to popsal jako „vodopád procházející mixérem".

Metodika TDD	Metodika ICONIX/DDT
Neodmyslitelnou součástí metodiky TDD je to, že se nejdříve napíše test a poté se napíše kód.	U testování řízeného návrhem nic neurčujeme: není vůbec žádný problém, pokud vám vyhovuje psát testy před doprovodným kódem. Klidně to můžete udělat a nadále zůstat „věrní" metodice DDT.
Pokud se u metodiky TDD chcete předběžnému návrhu věnovat více, než kolik považují za rozumné vaši kolegové, tak klidně můžete. Můžete tak učinit a nadále zůstat „věrní" metodice TDD. (Provedení velkého množství předběžného návrhu a napsání všech těch myriád testů by představovalo spoustu duplicitně prováděného úsilí.)	Základní součástí metodiky DDT je nejdříve vytvořit návrh a poté napsat testy a kód. Budete tak v důsledku psát méně testů než u metodiky TDD, protože testy, z jejichž napsání budete mít největší užitek, přesně vymezíte během procesu analýzy a návrhu.

U metodiky DDT tedy nevytváříte návrh podle testů jednotek. To ovšem neznamená, že na návrh v projektu dle metodiky DDT nemají testy žádný vliv. Nakonec totiž budete budovat návrh kolem testovatelnosti. Jak si ukážeme v kapitole 3, kód, který nebyl napsaný s ohledem na testovatelnost, je z hlediska testů naprosto špatný. Proto je velice důležité zapracovat testovatelnost do návrhů od co možná nejranější fáze. Není náhoda, že kód, který se snadno testuje, bývá obyčejně také dobře navržený. Řečeno jinak, vlastnosti návrhu kódu, díky nimž se snadno provádí testování jednotky, jsou stejné jako ty vlastnosti, které činí kód čistý, udržovatelný, flexibilní, tvárný, dobře rozvržený a vysoce modulární. Navíc není náhoda, že metodiky ICONIX Process a DDT kladou hlavní důraz na tvorbu návrhů právě s těmito vlastnostmi.

Ukázkový projekt: představení webové aplikace Mapplet 2.0

Hlavní příklad, který budeme v rámci celé této knihy používat, nese název Mapplet 2.0 a jedná se o skutečnou webovou aplikaci pro vyhledávání hotelů, která je umístěná na serveru *vresorts.com* (což je web věnovaný cestování vlastněný jedním ze spoluautorů této knihy). Máte tak možnost porovnat si případy užití v této knize s hotovým produktem na uvedeném webu.

Mapplet 2.0 je novou generací ukázkového projektu pro naši dřívější knihu s původním názvem *Agile Development with ICONIX Process*, kterou v průběhu několika posledních let s úspěchem používala společnost ICONIX jako učební příklad v kurzech s otevřeným zápisem. Původní aplikace „Mapplet 1.0" byla tenkým klientem napsaným v jazycích HTML a JavaScript, přičemž na straně serveru se využíval jazyk C#, kdežto u nové verze se na straně klienta používá technologie Flex a komponenty na straně serveru jsou napsané v Javě. V současném světě heterogenních podnikových technologií je směsice jazyků a technologií docela častá, takže se nebudeme vůbec vyhýbat vyhlídce na testování aplikace, která je souhrnně napsaná ve více než jednom jazyku. Z toho vyplývá, že se setkáte s testy napsanými pro Flex i pro Javu. Ukážeme si, jak psát testy, které jasně ověří návrh a požadavky, přičemž budeme pro otestování jednotlivých obchodních záležitostí přecházet mezi vrstvami architektury.

Obrázky 1.4 a 1.5 zachycují původní aplikaci Mapplet 1.0 postavenou na jazyku HTML v akci.

Obrázek 1.4: Aplikace Mapplet 1.0 na bázi HTML/Ajax sledující případ užití „Mapa USA“: začněte s mapou USA a přibližujte zobrazení do oblasti svého zájmu

Naproti tomu obrázek 1.6 přeskočil k hotovému produktu, kterým je na aplikace Mapplet 2.0 založená na technologii Flash.

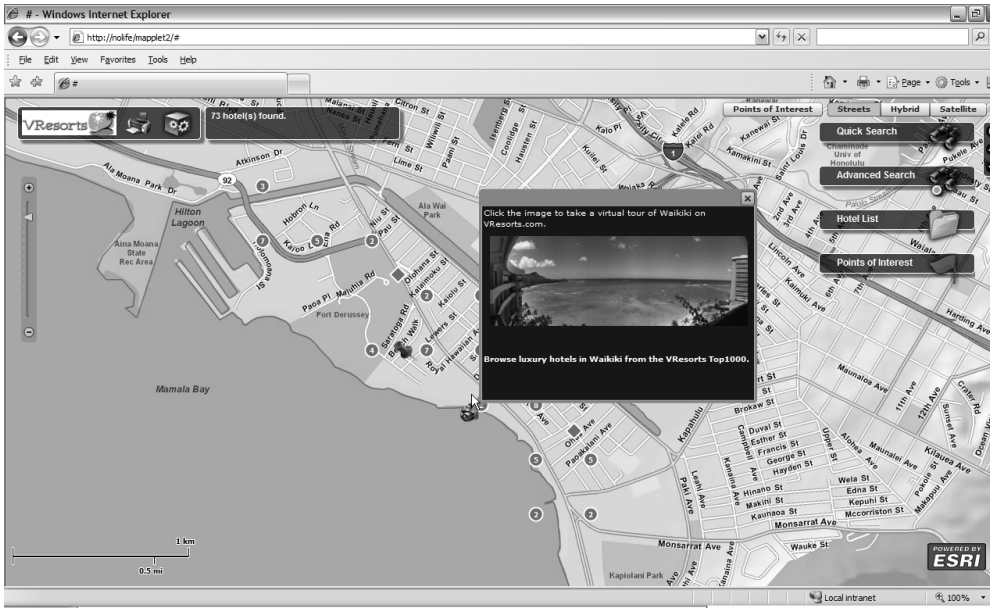
Shrnutí

V této kapitole jste se seznámili s hlavními principy, které budete studovat v této knize. Získali jste velmi zhuštěný přehled o metodice DDT a jejím porovnání s vývojem řízeným testy, což se v řadě organizací stává stále častěji synonymem pro „napsání nějakých testů jednotek“. Kromě toho jste se seznámili s hlavním projektem, na němž budeme v této knize pracovat a kterým je skutečná aplikace sloužící k vyhledávání hotelů určená pro web věnovaný cestování.

I když tato kniha není vysloveně o vývoji řízeném testy, porovnání s touto metodikou je velice užitečné, neboť metodika TDD je v povědomí vývojářů velice zakořeněná. V následující kapitole si projdeme příklad „Ahoj, světe!“, kterým je ve skutečnosti případ užití pro přihlášení, a to tak, že jej budeme nejdříve vyvíjet pomocí metodiky TDD a poté (v kapitole 3) pomocí metodiky DDT.

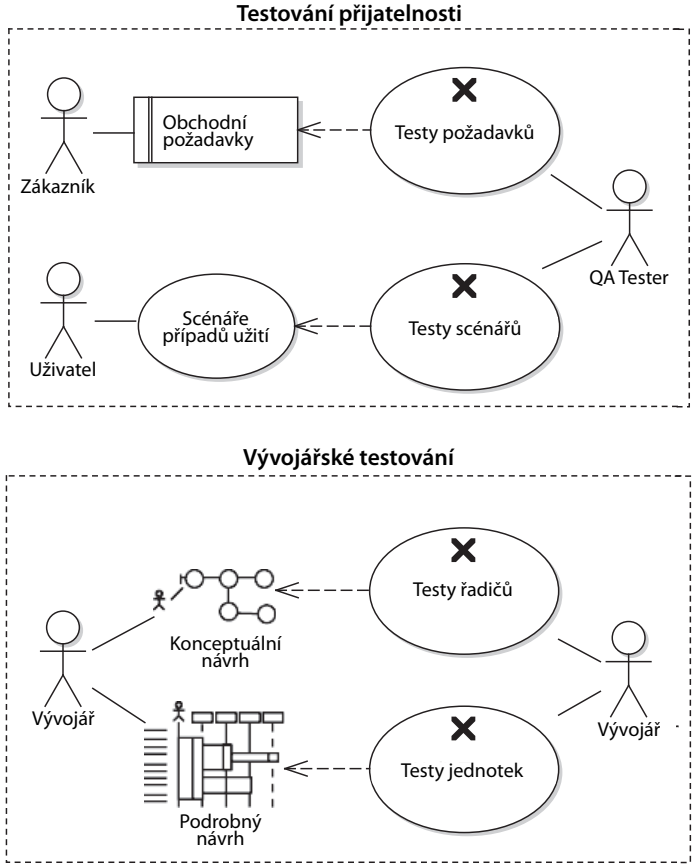
V části 2 začneme studovat vývoj aplikace Mapplet 2.0 – pozpátku. Jinými slovy, v kapitole 4 začneme u hotového produktu a v kapitolách 5–8 půjdeme postupně zpět v čase přes podrobný návrh, konceptuální návrh, případy užití až nakonec (nebo na začátek, podle toho, z jaké strany se na to podíváme) k prvotní sadě obchodních požadavků.

Obrázek 1.5: Vyhledání vhodného hotelu pomocí aplikace Mapplet 1.0



Obrázek 1.6: Kompletní potomek aplikace Mapplet i s fotografií dané lokace a virtuálními prohlídkami

Tuto kapitolu zakončíme diagramem procesu DDT (viz obrázek 1.7). Každá kapitola v části 2 začíná tímto diagramem společně s jakýmsi kruhem „nacházíte se zde“, který vymazuje látku probíranou v dané kapitole. Jak můžete vidět, metodika DDT rozlišuje mezi „zónou vývojářů“ s automatizovanými testy, které poskytují zpětnou vazbu pro návrh, a „zónou zákazníka“ s popisy testů (zákaznické testy lze rovněž automatizovat, jedná se však o pokročilou záležitost – viz kapitola 11).



Obrázek 1.7: Anatomie metodiky DDT: čtyři typy testů, co každý z nich řídí a kdo který test „vlastní“

Pokud váš tým diskutuje o tom, zda místo metodiky TDD použít DDT, pak můžete poukázat na to, že testy řadičů jsou v podstatě náhradou za testy jednotek ve stylu metodiky TDD, kterých se navíc nemusí psát tolik!

KAPITOLA 2

Úvodní příklad s metodikou TDD

V této kapitole budeme pokračovat v našem porovnání metodik TDD a DDT s tím, že realizujeme jednoduchý příklad ve stylu „Ahoj, světe!“ (ve skutečnosti jde o přihlašovací systém) pomocí vývoje řízeného testy. Přihlašování je pro úvodní příklad zvláště vhodné, neboť: (a) jde o jednoduchou koncepci a můžeme se s ním setkat v prakticky každé seriózní aplikaci v oblasti IT, a (b) systém řekne uživateli „Ahoj“ (a uživatel odpoví).

Poznámka pro neznalé metodiky TDD

Tuto kapitolu můžete klidně jen prolistovat, abyste viděli, jak se při vývoji řízeném testy programuje, a poté se v kapitole 3 nechat okouzlit tím, jak to lze dělat způsobem, který nabízí metodika DDT.

Poznámka pro neznalé metodiky TDD

Chápeme vaše strasti a pokusíme se v této kapitole objasnit, jak je vaše práce obtížná. V další kapitole vám pak ukážeme to, o čem si myslíme, že představuje snadnější způsob.

Ještě dříve, než se ponoříme do králičí nory vývoje řízeného testy a začneme pracovat na našem příkladu, podívejme se na 10 nejdůležitějších vlastností metodiky TDD (z hlediska metodiky ICONIX/DDT).

10 nejdůležitějších vlastností metodiky TDD

Tato část rozvíjí text v levém sloupci tabulky 1.1 z předchozí kapitoly, která uvádí hlavní rozdíly mezi metodikami TDD a DDT.

10. Testy řídí návrh

U vývoje řízeného testy mají testy v zásadě tři úkoly:

- řídit návrh,
- dokumentovat návrh,
- fungovat jako regresní testy.

První položka (testy řídí během programování návrh) není ani tak hlavním myšlenkovým a modelovacím procesem řídícím návrh, ale spíše charakteristickou vlastností metodiky TDD. Právě díky ní je metodika TDD tak revoluční (což ovšem není totéž, jako „dobrá“). Metodika TDD řídí návrh s malým dosahem, na taktické úrovni, přičemž počítá s tím, že se strategičtější rozhodnutí zpracují jinde. Představte si slepého muže s hůlkou, který je schopen určit, co se nachází bezprostředně před ním, a porovnejte to s možností vidět, že se přímo na vás řítí zatím vzdálené nákladní auto.

9. Celkový nedostatek dokumentace

Dokumentace není nedílnou součástí procesu TDD.

Výsledek: žádná dokumentace. Heslo „kód je návrh“ znamená, že stačí psát jen velmi malé množství dokumentace, což nově příchozím ztěžuje osvojení podrobností související s daným projektem. Při požádání o předložení celkového pohledu na systém nebo o stanovení, která třída je hlavní pro určitou obrazovku nebo obchodní funkci, pak lovení v testech jednotek (neboť „testy jsou návrh“) nebude příliš užitečné. Podnikaví jedinci si mohou říci, že pro projekt zřídí stránky wikiwebu s několika stránkami o různých aspektech návrhu, což si ale jiní neřeknou. Tvorba dokumentace návrhu není nedílnou součástí procesu, takže není pravděpodobné, že by se prováděla.

8. Vše je test jednotky

Pokud to není v testovacím frameworku JUnit, tak to neexistuje…

Myšlenkový přístup u metodiky, kdy se nejdříve testuje, spočívá dle očekávání v tom, že se jako první napíše test. Potom se přidá něco nového, aby test prošel, díky čemuž lze „prokázat“, že práce je hotová (příznivci Gödelovy věty o neúplnosti by se nyní měli dívat jinam). To naneštěstí může znamenat, že k prokázání stačí implementovat jen „optimistický scénář“. Veškeré neočekávané záležitosti (různé způsoby, jak může uživatel procházet uživatelské rozhraní, částečná selhání systému nebo vstupy a odpovědi mimo povolený rozsah) zůstávající neočekávané, protože nikdo nevěnoval čas na jejich strukturované a systematické promyšlení.

Nicméně…

7. Testy metodiky TDD nejsou tak docela testy jednotek (nebo snad ano?)

Ve světě TDD probíhala jistá diskuze o skutečné povaze testů metodiky TDD,[1] která se točila především kolem otázky: „Jsou testy metodiky TDD skutečnými testy jednotek?“ Rychlou a jednoduchou odpovědí je: „Ano. Ne. Ne tak docela.“ Testy metodiky TDD (někdy označované též jako *testy programátorů*, obvykle pro jejich spárování se *zákaznickými testy* neboli testy přijatelnosti) mají svůj vlastní účel. Proto budou v projektu, kde se skutečně nejdříve jako první píšou testy, vypadat testy malinko jinak než „klasické“ jemné testy jednotek. Test jednotky metodiky TDD může naráz otestovat víc než jedinou jednotku kódu. Martin Fowler, zastánce metodiky TDD a extrémního programování, napsal následující:

Testování jednotek v extrémním programování je často jiné než klasické testování jednotek, protože v extrémním programování se obvykle netestuje každá jednotka izolovaně. Testuje se každá třída a její bezprostřední spojení s jejími sousedy.[2]

Tím se ve skutečnosti dostává testování metodiky TDD někam mezi klasické testy jednotek (které používá metodika DDT) a vlastní testy řadičů metodiky DDT (viz kapitola 6). Nicméně v této knize budeme i nadále označovat testy metodiky DDT jako „testy jednotek“, poněvadž právě tak je obvykle označuje zbytek světa.

6. Testy přijatelnosti poskytují zpětnou vazbu vůči požadavkům

Testy přijatelnosti jsou rovněž součástí specifikace, nebo by byly, pokud bychom testy přijatelnosti měli (a pokud bychom měli specifikaci). Dokud totiž nad metodiku TDD neumístíte další proces (jako je extrémní programování nebo vývoj řízený testy přijatelnosti), budou testy jednotek vycházet přímo z požadavků/příběhů.

5. Metodika TDD propůjčuje důvěru k provádění změn

Důvěra k provádění ustavičného proudu změn je vlastně „návrh dle refaktoringu“. Není ale tato důvěra nemístná?

1 Viz *http://stephenwalther.com/blog/archive/2009/04/11/tdd-tests-are-not-unit-tests.aspx*

2 Viz *www.artima.com/intv/testdriven4.html*

Nadpis lze přeformulovat tak, že zelený proužek znamená, že „všechny testy, které jsem dosud napsal, neselhávají“. Pokud při spouštění testů jednotek přes spouštěč testů, jako je ten, který je vestavěný například v prostředí NetBeans, Eclipse, Flash Builder nebo IntelliJ, všechny vaše testy projdou, uvidíte napříč oknem zelený proužek, což je jakási menší odměna (), která může vést k příjemnému pocitu, že vše na světě je v pořádku – nebo alespoň vše u daného projektu.

Při pokrytí kódu tenkou vrstvou testů jednotek by vám měla představa, že „vše je v pořádku“, dát jistotu, která je nezbytná k nepřetržitému refaktorování kódu bez neúmyslného zanesení chyb. Přečtěte si nicméně článek „Green Bar of Shangri-La“[3] (nápověda: jak víte, zda jste nějaký test nevynechali?).

4. Návrh se vyvíjí inkrementálním způsobem

Nejdříve napíšete test a poté napíšete nějaký kód, aby test prošel. Pak kód refaktorujete pro zlepšení návrhu, aniž byste porušili jakýkoli test, který byl dosud napsán. Návrh se tedy vyvíjí s tím, jak inkrementálně zvětšujete velikost kódu prostřednictvím cyklu test/kód/refaktorizace. Pro vývojáře, kteří nepoužívají metodiku TDD, je to zhruba stejné, jako když boucháte čelem do zdi ve snaze ujistit se, že jste schopni cítit bolest, a to ještě před tím, než začnete stavět tuto zeď, jejíž existenci pak ověříte tak, že do ní budete bouchat čelem. Samozřejmě potřebujete „atrapu představující zeď“, na kterou můžete bouchat čelem, poněvadž skutečná zeď ještě neexistuje. Takový postup rádi označujeme jako „konstantní refaktorování až po programování“ (v původním změní „Constant Refactoring After Programming“ neboli „CRAP“).

3. Nějaký předběžný návrh je v pořádku

Strávit čas předběžným uvažováním o návrhu, nebo dokonce kreslením diagramů UML je zcela ve shodě s metodikou TDD, i když v ideálním případě by se tak mělo dít v kolaborativním prostředí – tedy například skupina vývojářů stojící u kreslicí tabule. (Věnování velkého množství času předběžnému návrhu, psaní všech těch myriád testů a průběžného refaktorování návrhu by tedy znamenalo spoustu duplicitního úsilí.)

Z teoretického hlediska to znamená, že předběžný návrh bude sice hotový, v praxi však vývojáři postupující dle metodiky TDD () zjistí, že předběžný návrh není na seznamu výsledků pro aktuální běh.

2. Metodika TDD produkuje velké množství testů

Filozofie „testy jako první“ stojící v pozadí metodiky TDD spočívá v tom, že ještě před tím, než napíšete jakýkoliv kód, napíšete nejdříve test, který selže. Potom napíšete kód, díky němuž daný test projde. Čistým výsledkem je pak to, že agresivní refaktorování (kterého bude zapotřebí opravdu velké množství, budete-li k návrhu přistupovat tímto inkrementálním způsobem) se zabezpečí masivním počtem testů kryjících kódovou bázi. Testy se tedy zdvojnásobí, neboť slouží jako nástroj pro návrh a těžce se na ně spoléhá jako na regresní testy.

3 Viz *www.theregister.co.uk/2007/04/25/unit_test_code_coverage/*

Metodika TDD navíc ve skutečnosti nerozlišuje mezi testy na „úrovni návrhu“ (nebo v prostoru řešení) a testy na „úrovni analýzy“ (nebo v prostoru problému).[4]

1. Metodika TDD je nehorázně obtížná

Čistý efekt postupování podle metodiky TDD spočívá v tom, že vše dostane vlastní test jednotky. Teoreticky to sice zní skvěle, ale prakticky budete mít hrozně moc redundantních testů.[5] Metodika TDD má image „odlehčeného“ nebo též agilního procesu, protože se vyhýbá představě „velkého návrhu na začátku“, což může nalákat k dříve zahájenému programování. Jenže tato iluze rychlého úspěchu je brzy odsunuta těžkou prací ve formě refaktorování až k hotovému produktu, v jejímž průběhu se přepisuje kód i testy.

Přihlašování implementované pomocí metodiky TDD

Bylo by dobré ukázat si plnohodnotný příklad použití metodiky TDD již v počáteční části knihy; a právě k tomuto účelu slouží tato kapitola. Hlavní myšlenka stojící v pozadí metodiky TDD spočívá v tom, že si postupně berete jednotlivé položky ze seznamu požadavků (nebo uživatelských příběhů) a pro každou z nich implementujete jen tolik, aby byl splněný daný požadavek. Po řádném seznámení se s požadavkem věnujete určitý čas přemýšlení o návrhu. Pak napíšete test. Ten by měl nejprve selhat (ve skutečnosti by se neměl ani zkompilovat), protože jste dosud nenapsali kód zajišťující, aby prošel. Potom napíšete dostatek kódu k tomu, aby tento test prošel, a revidujete návrh s cílem zajistit, aby byl „těsný“.[6] Znovu spustíte testy, přidáte další test pro další element kódu a tak dále. To vše opakujete, dokud je daný požadavek implementovaný.

Seznámení s požadavkem

Uvažme jednoduchý uživatelský příběh pro přihlášení:

Jako koncový uživatel chci být schopen přihlásit se na web.

Vypadá to, jako by zde chyběla nějaká drobnost, takže s kolegyní Lenkou, s níž programujete ve dvojici, vyhledáte Tomáše, který u vás zastupuje zákazníka.

„To je trošku komplikovanější,“ vysvětluje Tomáš. „Nejde jen o jednoduché přihlášení uživatele. Musíte se na to podívat z pohledu koncového uživatele a z pohledu vlastníka webu. Jakožto vlastník webu chci mít jistotu, že neplatící uživatelé nemohou přistupovat k placeným funkcím, že budou směřováni skrze cestu maximalizující příjem a že web nebude umožňovat pokusy o prolomení hesel uživatelů hrubou silou.“

4 To lze očekávat u zákaznických testů přijatelnosti ve stylu extrémního programování, přestože nejsou oficiální součástí metodiky TDD. Chcete-li do metodiky TDD přidat testy přijatelnosti, musíte se podívat mimo hlavní proces po nějaké jiné metodice, jako je BDD, ATDD (Acceptance Test-Driven Development – vývoj řízený testy přijatelnosti) nebo samotné extrémní programování. Další možností je samozřejmě metodika ICONIX/DDT.

5 Naše představa „redundantního testu“ může být jiná než představa vývojáře postupujícího podle metodiky TDD. O eliminaci redundantních testů (tím, že místo nich napíšete „hrubší“ testy řadičů) se více dozvíte v kapitole 6.

6 Jinak řečeno, návrh by měl pokrývat jen to, co bylo dosud napsáno, a to co nejefektivnějším způsobem, bez zbytečného kódu pro „možné budoucí požadavky“, k nimž nemusí nikdy dojít.

„A jistě byste chtěl, aby se uživatelé cítili bezpečně," dodáte a Tomáš přikývne.

Jdete tedy s Lenkou zpět do programátorského doupěte a tento uživatelský příběh rozvinete do podrobnější formy:

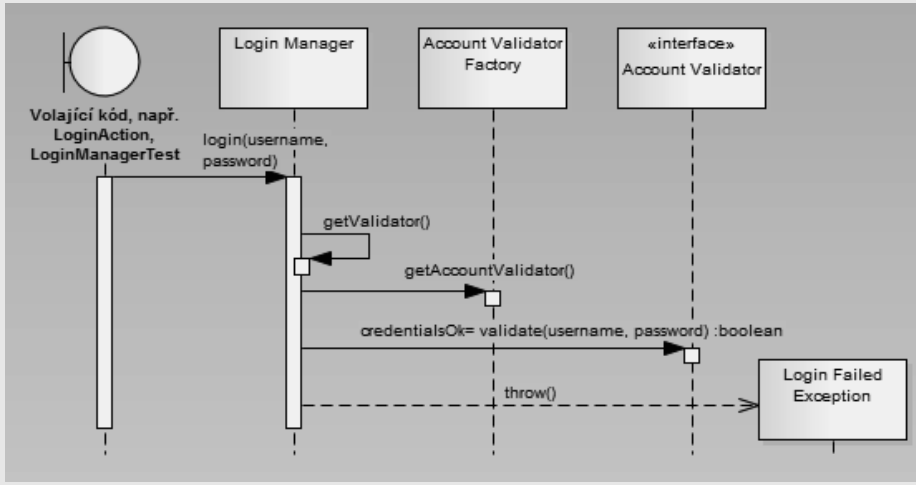
1. a. Jako vlastník webu chci poskytovat možnost bezpečného přihlášení, abych mohl získat příjem zpoplatněním prémiového obsahu.

   b. Jako uživatel webu chci být schopen bezpečně se přihlásit, abych mohl přistupovat k prémiovému obsahu.
2. Jako vlastník webu chci, aby systém uzamknul účet uživatele po 3 nezdařených pokusech o přihlášení.
3. Jako uživatel webu chci zadávání hesla skrýt, aby jej náhodou někdo nezahlédl.

Krátce se přete o to, zda jde skutečně o jeden uživatelský příběh s přidanými podrobnostmi nebo o dva (či tři) samostatné příběhy. Potom se dohodnete, že budete jednoduše pokračovat dál a začnete programovat.

Nakouknutí na odpověď

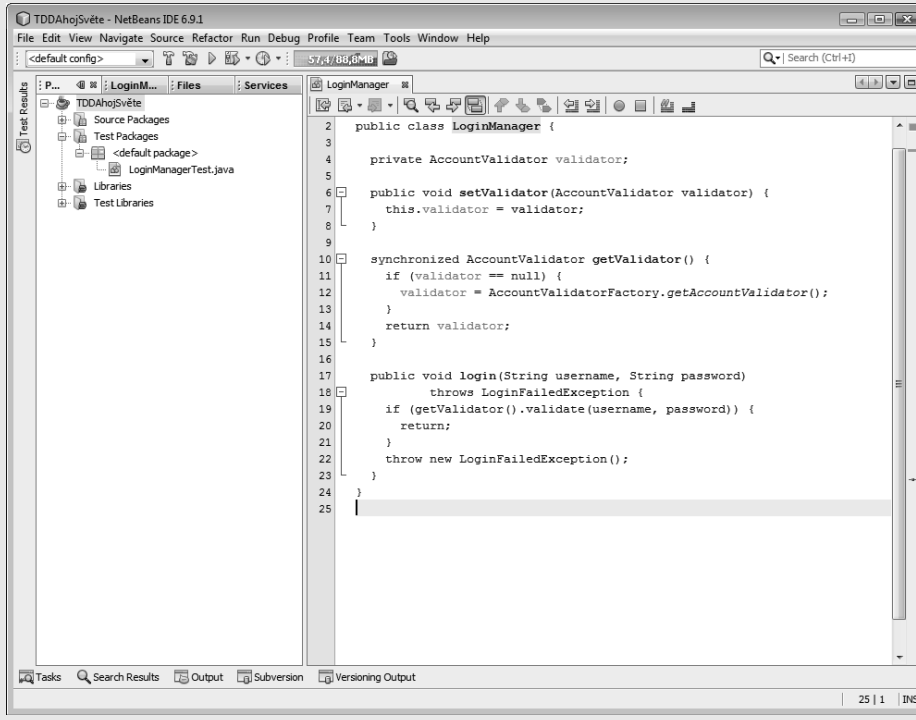
Obrázek 2.1 ukazuje pomocí diagramu posloupností, jak bychom běžně navrhli něco takového, jako je přihlašování na web (návrh dle metodiky ICONIX Process by byl více „doménově řízený", jak si ukážeme v následující kapitole):



Obrázek 2.1: Diagram posloupností pro proces přihlášení na web

Návrh na obrázku 2.1 je opravdu pěkně jednoduchý. Možná si myslíte, že navržení systému, jako je tento, nebude příliš obtížné, zvláště pak s tak populární metodikou, jako je vývoj řízený testy. Jak se ale sami přesvědčíte ve zbývající části této kapitoly, budeme-li postupovat dle metodiky TDD (), zabere refaktorování pro získání výsledného kódu zachyceného na obrázku 2.2 celou kapitolu.

Jinými slovy, ve zbývající části této kapitoly strávíme obrovskou spoustu času prací, na jejímž konci budeme mít jen o něco více než 20 řádků kódu. Při programování to vše „působí" tak nějak



Obrázek 2.2: Výsledný kód pro řešení přihlašování na web

korektně, neboť pozornost je zaměřena na detekci špatného návrhu a jeho inkrementální zlepšování, přičemž se neustále rozsvěcuje zelený proužek – jako nějaká odměna, která se dává křečkovi za napití ze správného dávkovače. Příjemný pocit narůstá a projekt se valí dál, tedy až na to, že na sklonku dne, po celé té námaze, bylo napsáno pouze 20 řádků kódu.

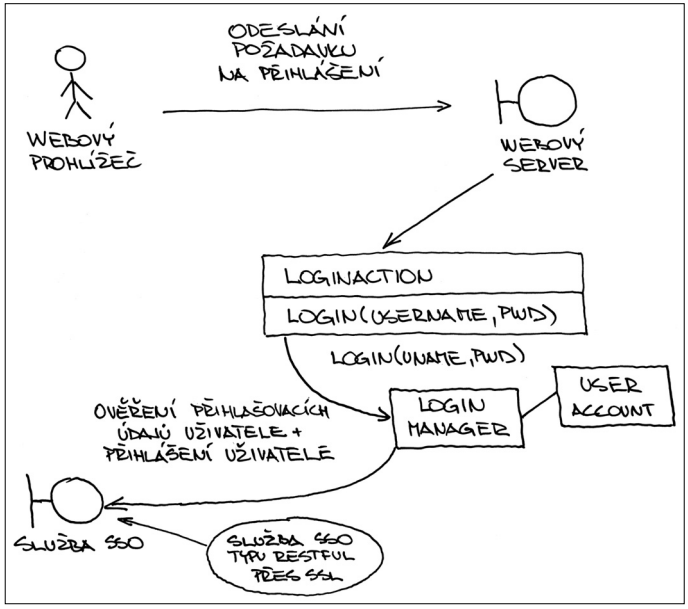
Pokud znáte metodiku ICONIX Process nebo případy užití obecně, pak jste si možná přirovnali tyto příběhy k požadavkům na vysoké úrovni. Přirozeným krokem, který by měl následovat, by bylo jejich rozvinutí do případů užití, neboť samotné příběhy ve skutečnosti neřeší řadu otázek, které vypučí při implementaci systému: Co by se mělo stát při zadání nesprávného hesla (zobrazí se jiná stránka s odkazy „zapomenuté heslo“ a „vytvořit nový účet“)? Na kterou stránku se uživatel po přihlášení dostane? Co se stane, pokud se uživatel pokusí zobrazit stránku bez přihlášení? (Patrně by měl být přesměrovaný na přihlašovací stránku, je to však s určitostí to, co si přeje zákazník?) A tak dále. Toto je druh otázek, k jejichž brzkému promyšlení vás metodika ICONIX Process (a tudíž i metodika DDT) povzbuzuje, díky čemuž je můžete zapracovat do návrhu.

Více si ale necháme na později. Prozatím se raději připoutejte, neboť v této kapitole se ponoříte králičí norou do „pohádkové země testů jednotek“. Vraťme se tedy zpět k naší neohrožené dvojici vývojářů používajících metodiku TDD, kteří jsou nyní připraveni pustit se do programování.

Přemýšlejte o návrhu

Ještě před tím, než se pustíte do programování, je dobré trošku se zamyslet nad návrhem. Chcete, aby systém přijímal požadavek na přihlášení, což patrně znamená, že uživatel zadá uživatelské jméno a heslo. Možná budete muset poslat Lenku zpátky pro získání dalších údajů od zástupce zákazníka. Co ale potom? Jak se bude heslo ověřovat?

Spolupráce u kreslicí tabule vedla k vytvoření náčrtku zachyceného na obrázku 2.3.



Obrázek 2.3: Prvotní náčrt návrhu pro příběh přihlášení

> **Poznámka**
>
> Obrázek 2.3 zachycuje směsici notačních stylů. Podstatná zde ale není správnost jazyka UML, ale promyšlení návrhu a představení celkového pohledu na plán útoku vývojářů.

`LoginAction` je generická třída frameworku Spring představující akci uživatelského rozhraní pro obsluhu příchozích webových požadavků, v tomto případě požadavku na přihlášení. Přijímá dvě vstupní hodnoty, uživatelské jméno a heslo, které jednoduše předá třídě vhodnější pro obsluhu požadavku na přihlášení.

Objekt typu `LoginManager` přijímá uživatelské jméno a heslo, pro jejichž ověření musí provést externí volání do služby typu RESTful. Pokud ověření selže, vyvolá se výjimka typu `ValidationException`, kterou zachytí objekt typu `LoginAction`, jenž uživateli odešle odpověď „CHYBA“. Dále máme dojem, že v budoucnu bude potřeba třída `UserAccount`, což snad s jistotou zjistíme, jakmile začneme programovat.

Nejdříve se napíše první test psaný jako první test

Pojďme se v rychlosti podívat na strukturu projektu. V prostředí NetBeans jsme vytvořili projekt s názvem „TDDAhojSvěte“, jehož struktura vypadá takto:

```
TDDAhojSvěte
  |__ src
  |__ test
  |__ bin
```

Veškerý produkční kód bude v balíčcích v adresáři `src`, všechny testy jednotek budou v adresáři `test` a zkompilovaný kód bude v adresáři `bin`.

Vyjdeme-li z nákresu na obrázku 2.1, pak není nijak překvapivé, že se soustředíme nejdříve na třídu `LoginManager`. Začneme tedy vytvořením testovací třídy pro třídu `LoginManager`:

```
import org.junit.Test;

public class LoginManagerTest extends junit.framework.TestCase {

  @Test
  public void testLogin() throws Exception {
  }

}
```

Zatím je vše v pořádku. Vytvořili jsme testovací kostru pro metodu `testLogin` třídy `LoginManager`, kterou jsme identifikovali během krátké porady u kreslicí tabule. Nyní přidáme do tohoto testu nějaký kód pro vytvoření instance třídy `LoginManager` a vyzkoušení přihlášení:

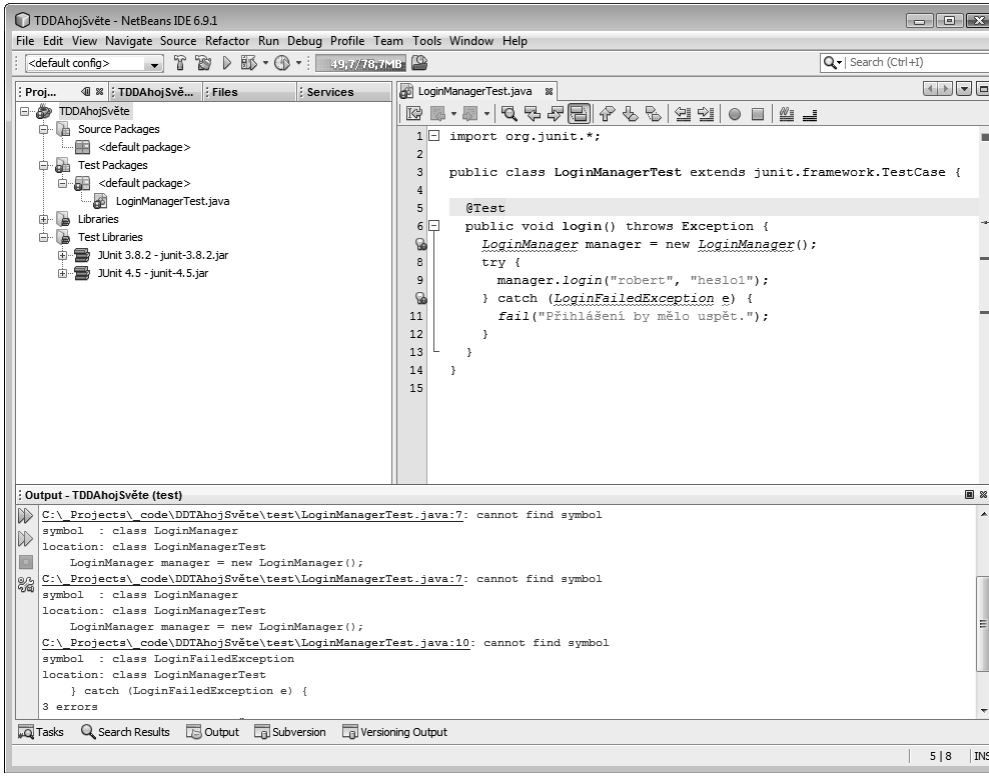
```
@Test
public void testLogin() throws Exception {
  LoginManager manager = new LoginManager();
  try {
    manager.login("robert", "heslo1");
  } catch (LoginFailedException e) {
    fail("Přihlášení by mělo uspět.");
  }
}
```

V této fázi je editor NetBeans posetý červenými vlnovkami, takže kompilace zcela jistě selže (viz obrázek 2.4).

Selhání kompilace je platnou fází v procesu TDD: chyby při kompilování testu nám říkají, že je nutné napsat nějaký kód, aby daný test prošel. Nyní to tedy uděláme s použitím dvou nových tříd:

```
public class LoginManager {
  public void login(String username, String password)
  throws LoginFailedException {
  }
}

public class LoginFailedException extends Exception {
}
```



Obrázek 2.4: Tento test potřebuje nějaký kód, nad nímž může běžet

Poznámka od našeho redaktora

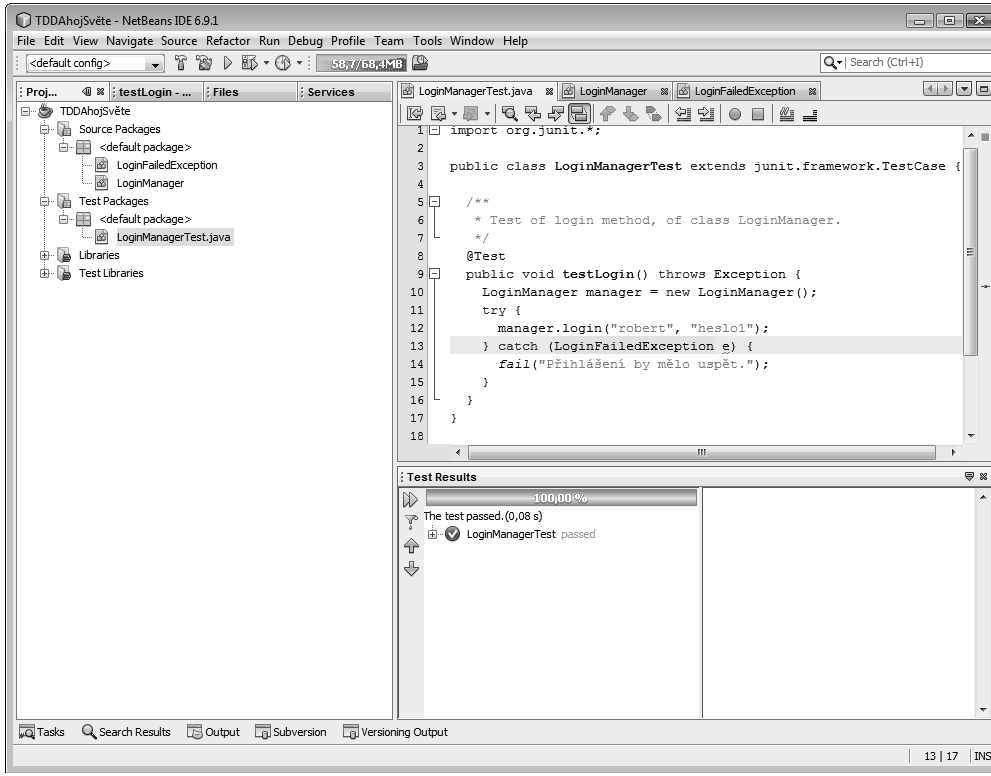
Náš redaktor Jonathan Gennick nám poslal tento komentář, který zde uvádíme, neboť velmi pěkně shrnuje naše vlastní pocity o metodice TDD:

Minulé léto mi vyměňovali okna v jídelně a obývacím pokoji. Měl jsem trvat na tom, aby se stavební dělník pokusil zavřít okno (čímž by před usazením oken do zdi provedl na prázdném otvoru test „zavření okna"). Do otvorů by měl vsadit nová okna až po provedení testu se zavřením okna a jeho selháním.

Další na řadě by byl test „jestlipak okno vypadne, narazí na podlahu a rozbije se". Ten by selhal, což by signalizovalo nutnost použít šroubky pro upevnění oken ve zdech. Samozřejmě bychom již měli rozbitá okna s dvojsklem s kryptonovým plynem uvnitř (nebo co tam vlastně je) za několik stovek dolarů.

Stavební dělník by mě kvůli takovému postupu měl jistě za hlupáka. Je překvapivé, že tolik vývojářů bylo svedeno v podstatě po stejné cestě.

Kód se nyní zkompiluje, takže rychle spustíme náš nový test jednotky, přičemž očekáváme (a doufáme), že se objeví červený proužek, který signalizuje, že test selhal. Jaké překvapení! Podívejte se na obrázek 2.5. U našeho testu nedošlo k úspěšnému selhání, ale k neúspěšnému úspěchu.



Obrázek 2.5: Zelený proužek – život je sladký. Až na... co je to za dotěrný pocit?

Test prošel v okamžiku, kdy měl selhat! Někdy by procházející test měl být tak znepokojující jako selhávající test. Jedná se o ukázku toho, kdy kód produktu poskytuje zpětnou vazbu testům, stejně jako když testy poskytují zpětnou vazbu kódu produktu. Jde o symbiózu, která odpovídá na otázku, co testuje testy? (Což je jen jiná varianta otázky „kdo hlídá hlídače?“)

Dle přesného postupu dle metodiky TDD nyní do kódu přidáme řádek, díky kterému test selže:

```
public void login(String username, String password)
throws LoginFailedException {
  throw new LoginFailedException();
}
```

Metoda `login()` jednoduše vyvolá výjimku typu `LoginFailedException`, která signalizuje, že všechny pokusy o přihlášení v současnosti selžou. Systém je nyní docela pěkně zamknutý: nikdo se nemůže přihlásit, dokud nenapíšeme kód, který to umožní. První test bychom mohli změnit také na `@Test loginFails()` a použít neplatné uživatelské jméno a heslo. Pak bychom ale nejdříve neimplementovali základní průchod uživatelským příběhem (tj. uživatel se úspěšně přihlásí). V každém případě jsme nyní ověřili, že skutečně dokážeme cítit bolest v čele, když s ním bouchneme o zeď!

Nyní přidáme další kód, aby test prošel:

```
public void login(String username, String password)
throws LoginFailedException {
```

```
  if ("robert".equals(username) && "heslo1".equals(password)) {
    return;
  }
  throw new LoginFailedException();
}
```

Pokud test nyní znovu spustíme, uvidíme zelený proužek signalizující, že test prošel. Vypadá to snad, že podvádíme? Inu, jedná se o nejjednodušší kód, který zajistí, že test projde, a proto je platný, alespoň tedy do chvíle, než přidáme další test, který zajistí, aby byly požadavky přísnější:

```
@Test
public void testAnotherLogin() throws Exception {
  LoginManager manager = new LoginManager();
  try {
    manager.login("marie", "heslo2");
  } catch (LoginFailedException e) {
    fail("Přihlášení by mělo uspět.");
  }
}
```

Nyní máme dva testovací případy: jeden, v němž se snaží přihlásit platný uživatel Robert, a druhý, v němž se snaží přihlásit platný uživatel Marie. Když ale testovací třídu znovu spustíme, dojde k selhání:

```
junit.framework.AssertionFailedError: Přihlášení by mělo uspět.
 at com.softwarereality.login.LoginManagerTest.testAnotherLogin
 (LoginManagerTest.java:24)
```

Nyní zjevně nastal čas, abychom se vrátili do reality a přidali nějaký kód, který doopravdy provede kontrolu přihlášení.

Vytvoření kódu pro kontrolu přihlášení, aby test prošel

Skutečný kód musí provádět volání služby typu RESTful, předat jí uživatelské jméno a heslo a obdržet zprávu „přihlášení prošlo“ nebo „přihlášení selhalo“. Rychlá zpráva příslušnému týmu odpovědnému za middleware/SSO (Single-Sign-On – jednotné přihlašování) vede k vytvoření šikovného souboru typu JAR, který zapouzdřuje detaily volání REST a místo nich exponuje toto užitečné rozhraní.

```
package com.mymiddlewareservice.sso;

public interface AccountValidator {
  public boolean validate(String username, String password);
  public void startSession(String username);
  public void endSession(String username);
}
```

Tato knihovna rovněž obsahuje třídu `AccountValidatorFactory` ve formě černé skříňky, kterou můžeme použít pro přístup ke konkrétní instanci implementující rozhraní `AccountValidator`:

```
public class AccountValidatorFactory {
  public static AccountValidator getAccountValidator() {...}
}
```

Tento soubor typu JAR můžeme jednoduše umístit do našeho projektu a pro ověření uživatele a ustavení pro něj relace SSO zavolat službu middlewaru. Při využití této pohodlné knihovny vypadá třída `LoginManager` takto:

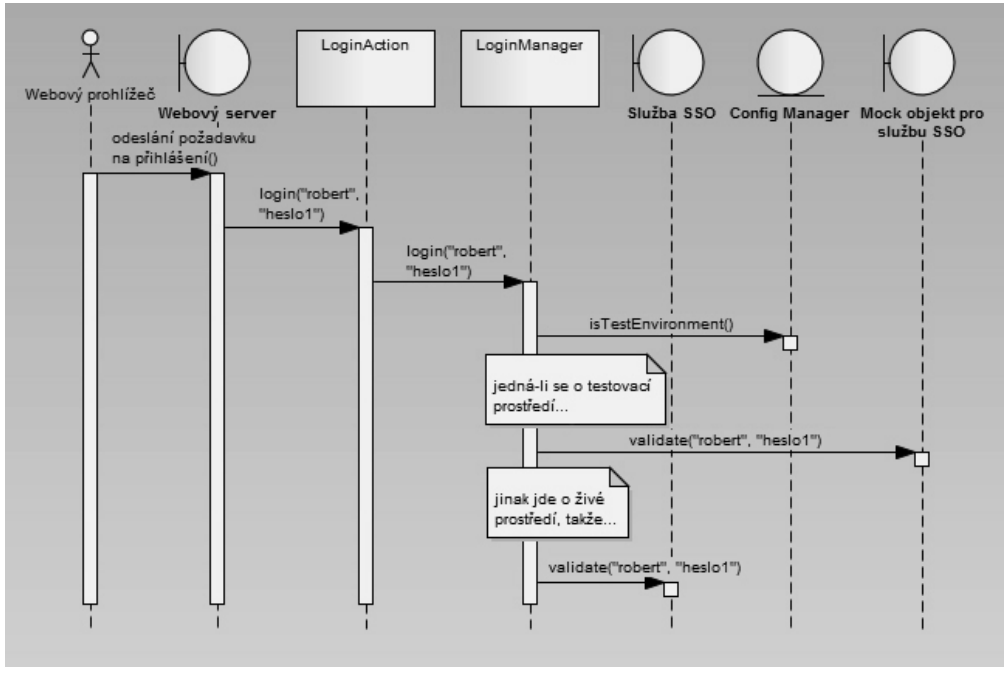
```
public class LoginManager {
  public void login(String username, String password)
```

```
  throws LoginFailedException {
    AccountValidator validator =
      AccountValidatorFactory.getAccountValidator();
    if (validator.validate(username, password)) {
      return;
    }
    throw new LoginFailedException();
  }
}
```

Pokud bychom nyní znovu spustili testy, volání objektu typu `AccountValidator` by provedlo síťové volání vzdálené služby SSO a ověření uživatelského jména a hesla, takže test by měl bez problémů projít.

Tím se ale dostáváme k důležitému problému s obvyklým testováním jednotek: skutečně není dobré, aby kód prováděl při testování externí volání. Vaše testovací sada se bude provádět během procesu sestavení, takže by sestavení bylo při spoléhání na externí volání tak nějak křehčí a najednou by záviselo na dostupnosti sítě, fungování serverů a podobně. Zpráva „služba není dostupná“ by jistě neměla být chybou při sestavování.

Z tohoto důvodu obvykle jdeme do velkých délek, abychom udrželi kód, na něj se aplikuje test jednotky, izolovaný. Obrázek 2.6 ukazuje jeden ze způsobů, jak by se to dalo provést. V diagramu posloupností kontroluje objekt typu `LoginManager`, zda běží v „živém“ prostředí nebo v prostředí s testem jednotky/mock objektem. V prvním případě zavolá skutečnou službu SSO, ve druhém verzi mock objektu.



Obrázek 2.6: Běžně používaný antivzor: kód se větví podle toho, ve kterém prostředí se nachází

Z toho by se ale rychle stala noční můra s hromadou příkazů „if-else if“, které ověřují, zda jde o živé nebo testovací prostředí. Produkční kód by byl mnohem složitější, neboť by se zaneřádil těmito kontrolami pokaždé, když by bylo nutné provést externí volání, a to nepočítáme speciální kód pro vracení „falešných“ objektů pro testovací účely. Do produkčního kódu je skutečně dobré dát co nejméně omezení a výhrad (tj. hacků či alternativních řešení), přičemž jistě nechcete kazit návrh jen kvůli tomu, aby bylo možné provádět testování jednotek.

Vytvoření mockobjektu

Naštěstí existuje další možnost, která navíc náhodou podporuje kvalitní návrh. Frameworky pro práci s tzv. mock objekty, jako je JMock, EasyMock, and Mockito[7], používají speciální čarování (reflexi Javy a dynamické proxy objekty) pro vytvoření vhodných, prázdných/nefunkčních verzí tříd a rozhraní. V tomto příkladu použijeme framework JMock (v pozdější části knihy použijeme také framework Mockito).

Vrátíme-li se k naší třídě `LoginManagerTest`, můžeme ji připravit pro použití ve frameworku JMock pomocí anotace a kontextu:

```
@RunWith(JMock.class)
public class LoginManagerTest {
  Mockery context = new JUnit4Mockery();
```

Po opětovném spuštění testu (a obdržení stejné chyby jako dříve) si můžete prohlédnutím zásobníku volání pro výjimku typu `AssertionFailedError` ověřit, že test nyní prochází přes framework JMock:

```
junit.framework.AssertionFailedError: Přihlášení by mělo uspět.
 at com.softwarereality.login.LoginManagerTest.testAnotherLogin
 (LoginManagerTest.java:32)
 at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
 at sun.reflect.NativeMethodAccessorImpl.invoke
 (NativeMethodAccessorImpl.java:39)
 at sun.reflect.DelegatingMethodAccessorImpl.invoke
 (DelegatingMethodAccessorImpl.java:25)
 at org.junit.internal.runners.TestMethod.invoke(TestMethod.java:66)
 at org.jmock.integration.junit4.JMock$1.invoke(JMock.java:37)
```

Vytvoření mock instance třídy `AccountValidator` je docela přímočaré:

```
validator = context.mock(AccountValidator.class);
```

Další problém tkví v tom, jak zařídit, aby třída `LoginManager` použila tuto mock instanci místo produkční verze objektu typu `AccountValidator` vráceného třídou `AccountValidatorFactory`. V kódu třídy `LoginManager`, který jsme právě napsali, volá metoda `login()` přímo metodu `AccountValidatorFactory.getAccountValidator()`. Nemáme žádný „záchytný bod“, pomocí neboť bychom jí řekli, aby použila naši verzi. Nyní takový záchytný bod přidáme:

```
public class LoginManager {

  private AccountValidator validator;

  public void setValidator(AccountValidator validator) {
```

7 Viz *www.jmock.org*, *http://mockito.org* a *http://easymock.org*. Frameworky pro práci s mock objekty jsou k dispozici i pro jiné jazyky, např. Mock4AS pro FlexUnit. Framework NUnit (pro platformu .Net) má již podporu pro dynamické mock objekty zabudovanou.

```
    this.validator = validator;
  }

  synchronized AccountValidator getValidator() {
    if (validator == null) {
      validator = AccountValidatorFactory.getAccountValidator();
    }
    return validator;
  }

  public void login(String username, String password)
  throws LoginFailedException {
    if (getValidator().validate(username, password)) {
      return;
    }
    throw new LoginFailedException();
  }
}
```

Tato verze používá injektáž závislostí (Dependency Injection), která umožňuje vkládat námi zvolené objekty implementující rozhraní `AccountValidator`. Nyní můžeme před zavoláním metody `login()` nastavit alternativní mock objekt typu `AccountValidator`. Jedinou změnou uvnitř metody `login()` bude to, že místo přímého použití člena validátoru se zavolá metoda `getValidator()`, která vytvoří „skutečnou“ verzi objektu typu `AccountValidator`, pokud jsme již neprovedli injektáž jiné verze.

Kompletní testovací metoda `testLogin()` nyní vypadá takto:

```
@Test
public void testLogin() throws Exception {
  final AccountValidator validator = context.mock(AccountValidator.class);
  LoginManager manager = new LoginManager();
  manager.setValidator(validator);
  try {
    manager.login("robert", "heslo1");
  } catch (LoginFailedException e) {
    fail("Přihlášení by mělo uspět.");
  }
}
```

Nicméně při spuštění testu se i nyní objeví červený proužek. To je dáno tím, že nová metoda mock objektu `AccountValidator.validate()` vrací ve výchozím stavu hodnotu `false`. K tomu, aby vrátila hodnotu `true`, je nutné říct frameworku JMock, co čekáte, že se má stát. To lze provést předáním jednoho z vhodně pojmenovaných objektů typu `Expectations` do daného kontextu:

```
context.checking(new Expectations() {{
  oneOf(validator).validate("robert", "heslo1");
  will(returnValue(true));
}});
```

Poznámka

Na první pohled to možná nevypadá jako skutečně platný kód jazyka Java. Návrháři frameworku JMock ve skutečnosti usilují o takový přístup k syntaxi, který by se co nejblíže přibližoval k „přirozenému jazyku". K tomu využívá tzv. plynulé rozhraní[8], díky němuž lze kód číst jako jednoduchý text v angličtině. Zda se jim podařilo dosáhnout čistšího, pro člověka čitelného aplikačního rozhraní, nebo něčeho mnohem hloupějšího, je nicméně otázka do diskuze![9]

Tento úryvek kódu říká: během testu očekávám jedno volání metody `validate()` s argumenty „robert" a „heslo1" s tím, že pro toto volání chci, aby se vrátila hodnota `true`. To je ve skutečnosti jedna z největších předností frameworku JMock. Umožňuje totiž přesně stanovit, kolikrát očekáváte, že se daná metoda zavolá, a nechat test automaticky selhat, pokud se daná metoda vůbec nezavolá, zavolá se v nesprávném počtu nebo se zavolá s nesprávnou sadou hodnot.

Pro naše bezprostřední účely je to nicméně jen příjemný bonus: v této fázi nás totiž zajímá pouze to, aby mock objekt vrátil hodnotu, kterou očekáváme.

Výsledkem opětovného spuštění kódu s naším novým objektem typu `Expectations` je zelený proužek. Nyní musíme provést to samé i pro ostatní testovací případy, kde předáváme „marie" a „heslo2". Jenže pouhé opakování této techniky povede k duplikovanému kódu, přičemž se jedná o „podpůrný" kód, neboť má jen velmi málo společného se samotným testovacím případem. Nastal tedy čas provést refaktorovat testovací třídy do něčeho hubenějšího.

Refaktorizace kódu ukazující rozvoj návrhu

Začneme refaktorizací testovacího kódu, neboť ten nás bezprostředně zajímá. Nicméně skutečným smyslem tohoto cvičení je vidět, jak se návrhu kódu produktu rozvíjí při přidávání dalších testů a psaní kódu, aby tyto testy prošly.

Zde je refaktorovaný testovací kód:

```
@RunWith(JMock.class)
public class LoginManagerTest extends junit.framework.TestCase {

  Mockery context = new JUnit4Mockery();
  LoginManager manager;
  AccountValidator validator;

  @Before
  @Override
  public void setUp() throws Exception {
    validator = context.mock(AccountValidator.class);
    manager = new LoginManager();
    manager.setValidator(validator);
  }

  @After
```

8 Viz *www.martinfowler.com/bliki/FluentInterface.html*

9 Tento dokument od tvůrců frameworku JMock popisuje vývoj jazyka EDSL (Embedded Domain-Specific Language – vkládaný jazyk specifický pro určitou doménu), přičemž používá framework JMock jako příklad: *www.mockobjects.com/files/evolving_an_edsl.ooplsa2006.pdf*.

```
  @Override
  public void tearDown() throws Exception {
    manager.setValidator(null);
    manager = null;
    validator = null;
  }

  @Test
  public void testLogin() throws Exception {
    context.checking(new Expectations() {{
      oneOf(validator).validate("robert", "heslo1");
      will(returnValue(true));
    }});

    try {
      manager.login("robert", "heslo1");
    } catch (LoginFailedException e) {
      fail("Přihlášení by mělo uspět.");
    }
  }

  @Test
  public void testAnotherLogin() throws Exception {
    context.checking(new Expectations() {{
      oneOf(validator).validate("marie", "heslo2");
      will(returnValue(true));
    }});

    try {
      manager.login("marie", "heslo2");
    } catch (LoginFailedException e) {
      fail("Přihlášení by mělo uspět.");
    }
  }
}
```

Vytvořili jsme testovací přípravky `@Before` a `@After`, které se spouštějí před a po každém testovacím případu za účelem vytvoření validátoru mock objektu a objektu typu `LoginManager`, což je naše testovaná třída. Díky tomu není nutné pokaždé opakovat tento přípravný kód. Návrh této testovací třídy lze i nadále vylepšovat, k tomu se ale vrátíme později. Další rychlé spuštění přes spouštěč testů způsobí zobrazení zeleného proužku, což vypadá dobře. Jenže až dosud jsme netestovali nic jiného než úspěšné přihlášení. Měli bychom rovněž použít nějaké neplatné přihlašovací údaje a zajistit, aby se správně zpracovaly. Přidejme tedy nový testovací případ:

```
@Test( expected = LoginFailedException.class )
public void testInvalidLogin() throws Exception {
  context.checking(new Expectations() {{
    oneOf(validator).validate("vilnius", "neznamy1");
    will(returnValue(false));
  }});

  manager.login("vilnius", "neznamy1");
}
```

Tentokrát se snaží starý podvodník Vilnius získat přístup do systému. Daleko se ale nedostane – ne snad proto, že bychom měli neprodyšně navrženou vzdálenou službu SSO běžící na šifrovaném pro-