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.

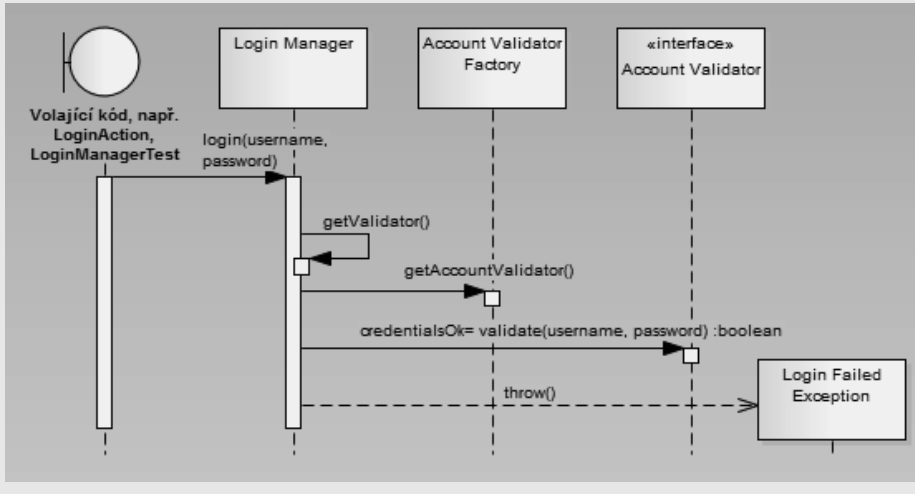
Jdete tedy s Lenkou zpět do programátorského doupěte a tento uživatelský příběh rozvinete do podrobnější formy:

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

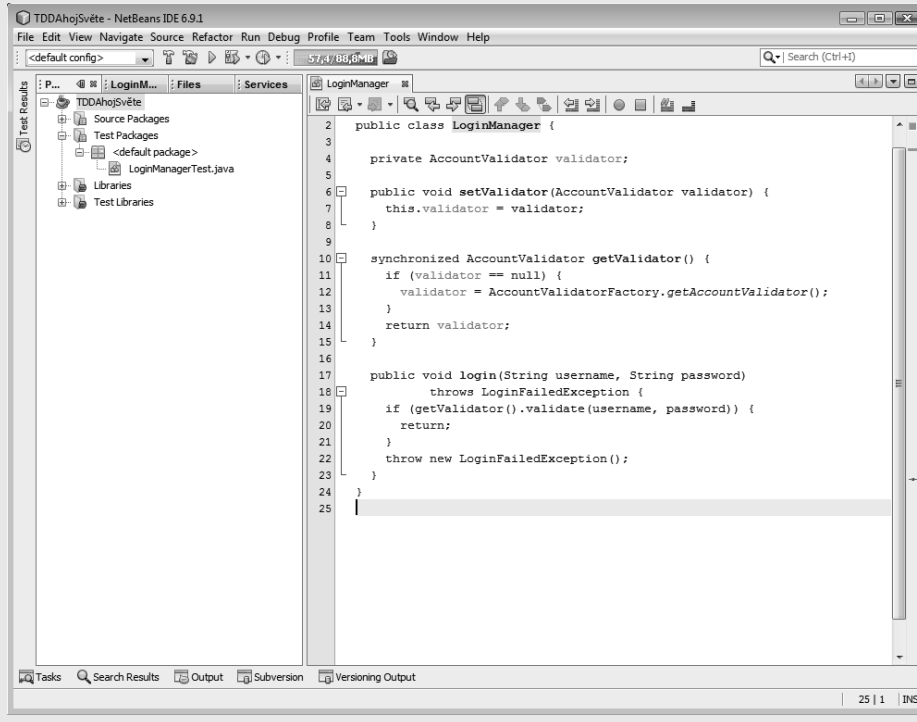
Obrázek 2.1 ukazuje pomocí diagramu posloupností, jak bychom běžně navrhli něco takového, jako je přihlašování na web (návrh dle metodiky ICONIX Process by byl více „doménově řízený", jak si ukážeme v následující kapitole):



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.

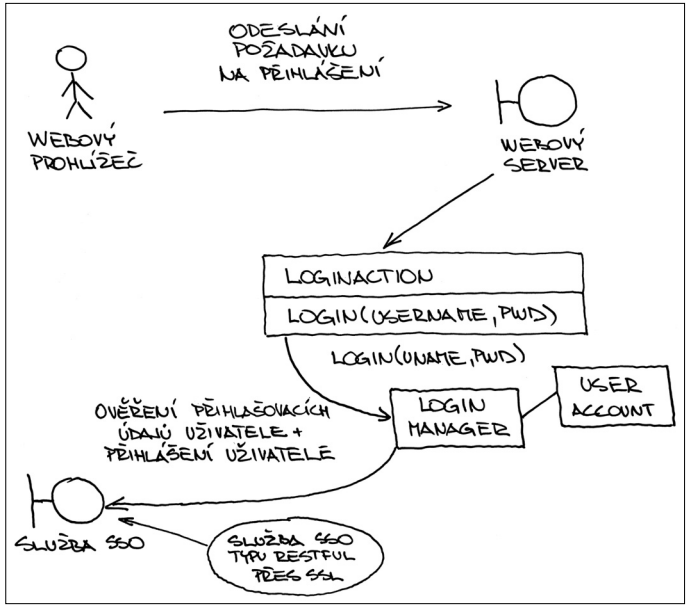
Pokud znáte metodiku ICONIX Process nebo případy užití obecně, pak jste si možná přirovnali tyto příběhy k požadavkům na vysoké úrovni. Přirozeným krokem, který by měl následovat, by bylo jejich rozvinutí do případů užití, neboť samotné příběhy ve skutečnosti neřeší řadu otázek, které vypučí při implementaci 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:

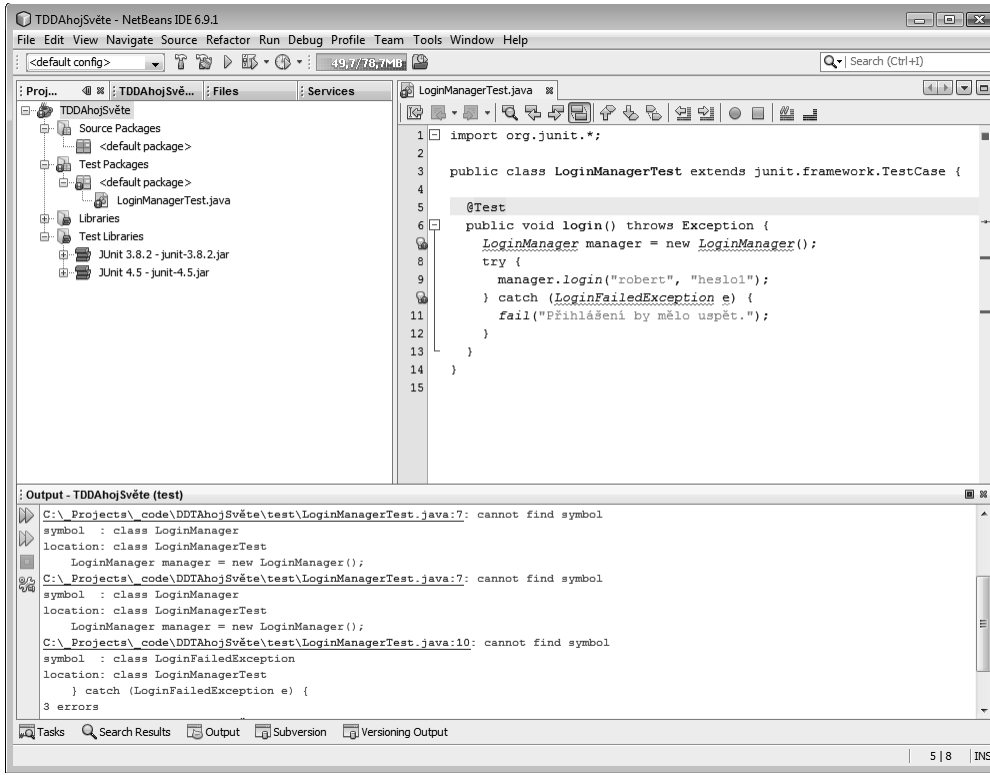
```
@Test
public void testLogin() throws Exception {
  LoginManager manager = new LoginManager();
  try {
    manager.login("robert", "heslo1");
  } catch (LoginFailedException e) {
    fail("Přihlášení by mělo uspět.");
  }
}
```

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

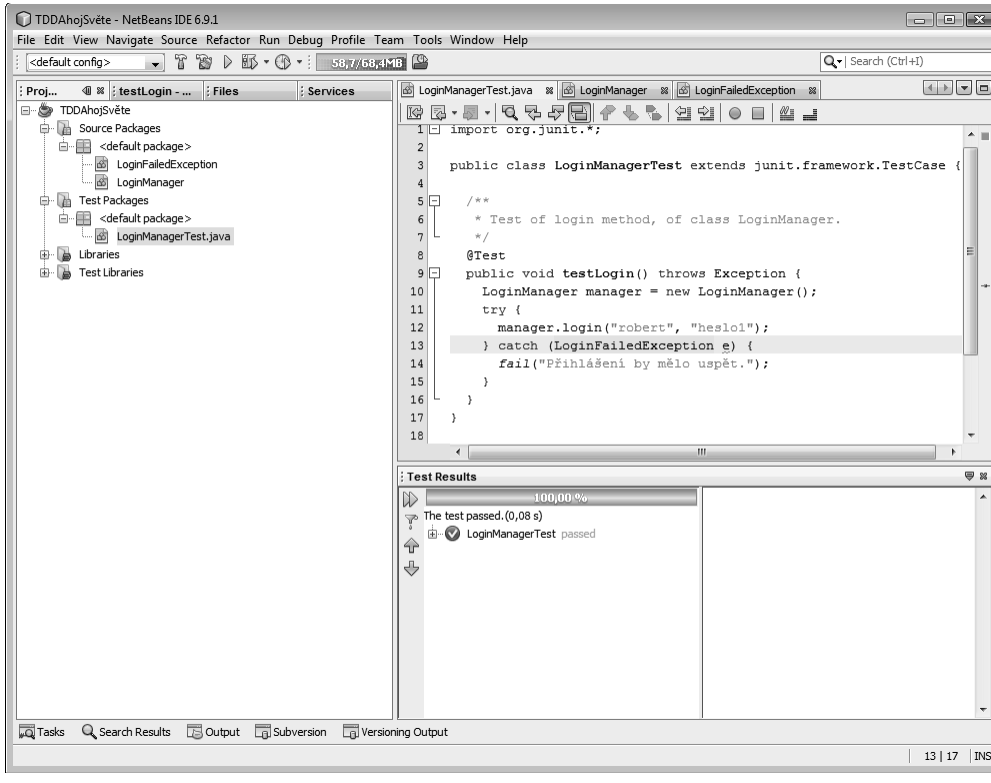
Náš redaktor Jonathan Gennick nám poslal tento komentář, který zde uvádíme, neboť velmi pěkně shrnuje naše vlastní pocity o metodice TDD:

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:

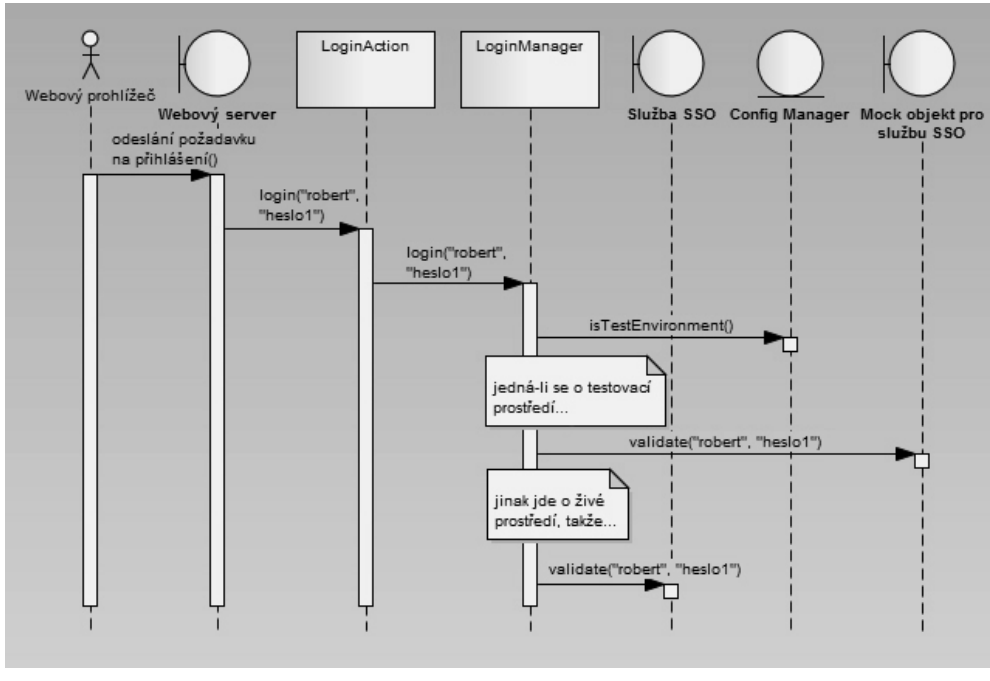
```
public class LoginManager {
  public void login(String username, String password)
```

```
  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-

tokolu SSL, ale protože náš mock objekt je nastavený tak, aby vrátil hodnotu `false`. To se bude divit! První řádek kódu používá anotaci frameworku JUnit 4, stejně jako u ostatních testovacích metod. Zde jsme však navíc stanovili, že očekáváme vyvolání výjimky typu `LoginFailedException`. Její vyvolání čekáme z toho důvodu, že mock objekt vrátí hodnotu `false`, což signalizuje selhání ze služby SSO za mock objektem. Kódem, který se ve skutečnosti testuje, je metoda `login()` ve třídě `LoginManager`. Tento test ukazuje, že dělá to, co očekáváme.

Refaktorová párty šíleného kloboučníka[10]

Alenka, která nevěděla co si počít a cítila se poněkud otřesená po té zprávě o návrzích, které krystalizují samy ze sebe, se rozhodla jít na chvíli s králíkem. Jak tak šli, králík se vždy po několika krocích zastavil a několikrát si v kruhu zaiteroval. Nakonec vyrazil nejvyšší rychlostí vpřed a dostal se tak daleko, že Alenka jej již nemohla vidět.

Alenka šla dál a po chvíli došla na mýtinku, kde zahlédla králíka společně s velkou myší a malým človíčkem s velkým kloboukem na hlavě, jak zuřivě pracují se stolem a několika židlemi. Když Alenka vstoupila, všechny nohy od stolu a židlí ležely na velké hromadě na zemi. Alenka sledovala, jak Kloboučník, králík a plch vzali čtyři nohy z hromady a šroubovali je k židli. Problém byl v tom, že nohy měly různou délku, a Alenka hleděla s úžasem, jak všichni dokončili montáž židle, otočili ji a postavili na zem. Židle se často převrhly, což se u židle s nohama různých délek dá čekat, a když se tak stalo, všichni jednohlasně zaječeli, „test jednotky se nezdařil", otočili židli vzhůru nohama, odšroubovali nohy a hodili je zpět na hromadu.

„Jakou hru to hrajete?" zeptala se Alenka.

„To není hra, refaktorujeme nábytek," odvětil Kloboučník.

„Proč si nepřečtete dokumentaci pro montáž židlí?" podivila se Alenka. „Jistě uvádí, které nohy patří ke kterým židlím."

„Jedná se o ústní dokumentaci," odpověděl Kloboučník.

10 Výňatek z proslovu, který byl s původním názvem „Alice in Use Case Land" (Alenka v říši případů užití) Douga Rosenberga uveden jako programové prohlášení na konferenci UML World v roce 2001. Celý přepis najdete na adrese *www.iconixsw.com/aliceinusecaseland.html* a také v příloze.

„Aha. Takže to znamená, že žádnou nemáte."

„Ne," řekl Kloboučník. „Psaná dokumentace je pro zbabělce. My umíme refaktorovat velice rychle, takže si můžeme dovolit být tak odvážní, že necháme návrh, aby sám vykrystalizoval," dodal.

„Ach tak. Králík řekl, že bych se tě na to měla zeptat, " řekla Alice. „Jak může návrh sám vykrystalizovat z kódu?"

„Ubohé, nevědomé dítě," řekl Kloboučník tichým blahosklonným tónem. „Copak nevíš, že kód je návrh? Třeba si to necháš lépe vysvětlit od Vévodkyně." Vrátil se k refaktorování židlí.

Při pohledu na poslední tři testovací metody byste měli zahlédnout, jak se navzdory našemu poslednímu refaktorování vynořuje vzor opakujícího se kódu. Bylo by rozumné vytvořit „pracovní" metodu a přesunout do ní náš podpůrný kód (nastavující očekávání testovacího případu a volající objekt typu `LoginManager`). Jedná se sice jen o přesun kódu, jak ale uvidíte, má výrazný vliv na čitelnost testovací třídy. Ve skutečnosti budeme mít dvě nové metody: `expectLoginSuccess()` a `expectLoginFailure()`.

Zde jsou naše dvě nové metody:

```
void expectLoginSuccess(final String username, final String password) {
  context.checking(new Expectations() {{
    oneOf(validator).validate(username, password);
    will(returnValue(true));
  }});

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

void expectLoginFailure(final String username, final String password) throws
LoginFailedException {
  context.checking(new Expectations() {{
    oneOf(validator).validate(username, password);
    will(returnValue(false));
  }});

  manager.login(username, password);
}
```

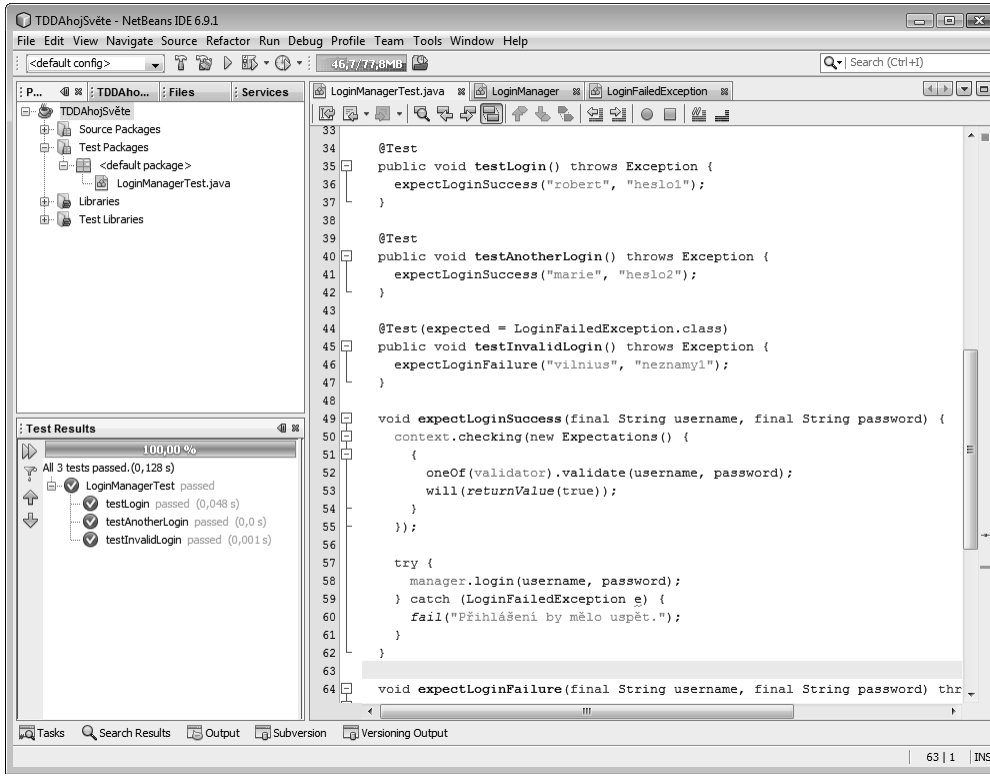
A takto nyní vypadají tři refaktorované testovací metody:

```
@Test
public void testLogin() throws Exception {
  expectLoginSuccess("robert", "heslo1");
}

@Test
public void testAnotherLogin() throws Exception {
  expectLoginSuccess("marie", "heslo2");
}
```

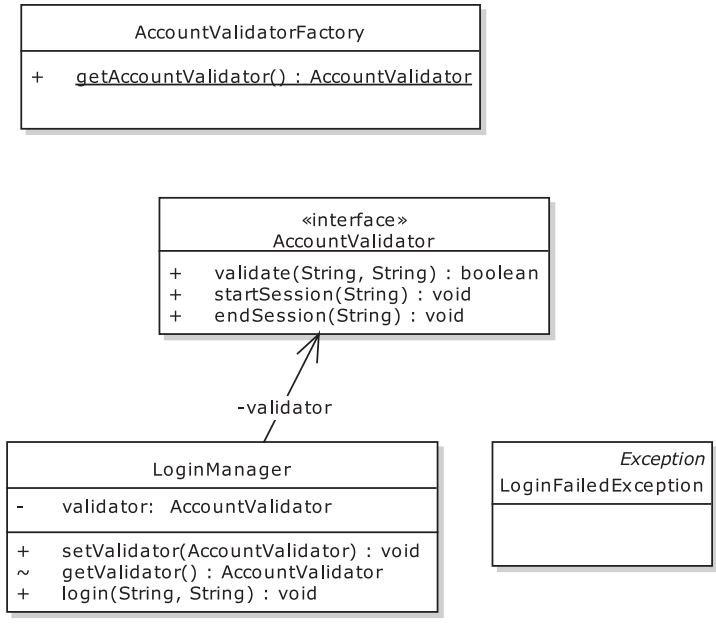
```
@Test( expected = LoginFailedException.class )
public void testInvalidLogin() throws Exception {
  expectLoginFailure("vilnius", "neznamy1");
}
```

Znenadání to vypadá, že každý testovací případ je zaměřený čistě na testovací hodnoty, bez podpůrného kódu skrývajícího účel nebo očekávání testovacího případu. Výsledkem dalšího rychlého spuštění testů je zelený proužek (viz obrázek 2.7), takže to vypadá, že tato změna nic nenarušila.

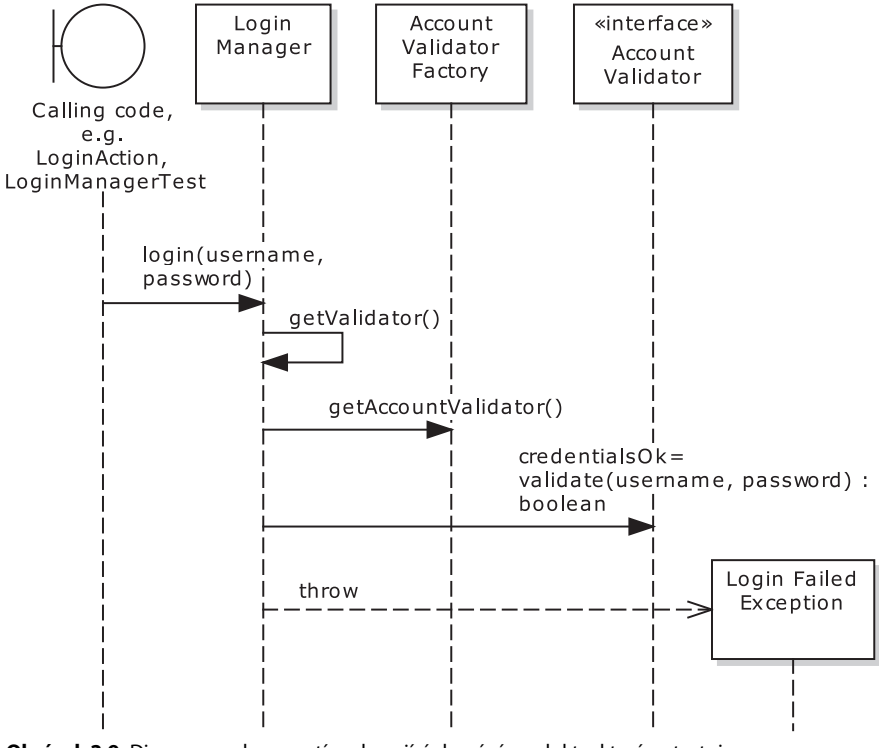


Obrázek 2.7: Kód jsme (znovu) refaktorovali, takže jsme znovu spustili testy, abychom měli jistotu, že se nic nenarušilo

Přestože vývojáři používající metodiku TDD mají někdy k modelování v jazyku UML určitý odpor (ač jde o zmenšující se menšinu, jak doufáme), využili jsme schopnosti editoru Enterprise Architect provádět zpětnou analýzu pro vytvoření modelu tříd podle nového zdrojového kódu (viz obrázek 2.8). Dali jsme dohromady také stručný diagram posloupností zachycující interakce kódu (viz obrázek 2.9). Všimněte si, že u reálného projektu dle metodiky TDD tyto diagramy mohou, ale také nemusejí být vytvořené. Domníváme se, že nejspíše nebudou, zvláště pak diagram posloupností. Kdo koneckonců potřebuje dokumentaci, když existuje kód pro framework JUnit?



Obrázek 2.8: Kód produktu po zpětné analýze v editoru Enterprise Architect



Obrázek 2.9: Diagram posloupností zachycující chování produktu, které se testuje

Všimněte si, že v rámci našeho programování jsme se nepokusili napsat žádný test pro vzdálenou službu SSO, kterou bude náš kód při ostrém provozu používat. To je dáno tím (tedy alespoň v této fázi), že služba SSO není to, co vyvíjíme. Pokud by se ukázalo, že s kódem služby SSO je nějaký problém, měli bychom dostatečnou důvěru v náš vlastní kód (neboť je pokrytý testy jednotek), abychom mohli bezpečně říci, že problém musí být uvnitř služby, kterou bereme jako „černou skříňku". Nejde ani tak o to, zda je to „jejich problém, a ne náš", ale spíše o pomoc při hledání zdroje problému a jeho co nejrychlejším vyřešení. To můžeme udělat, protože jsme vytvořili kvalitně rozvržený kód s dobrým pokrytím jednotkami testů.

V určité fázi tedy budeme chtít prokázat, že náš kód funguje společně s touto službou SSO. Kromě starého dobrého ručního testování navíc napíšeme test přijatelnosti, který otestuje systém jako celek.

Testy přijatelnosti s metodikou TDD

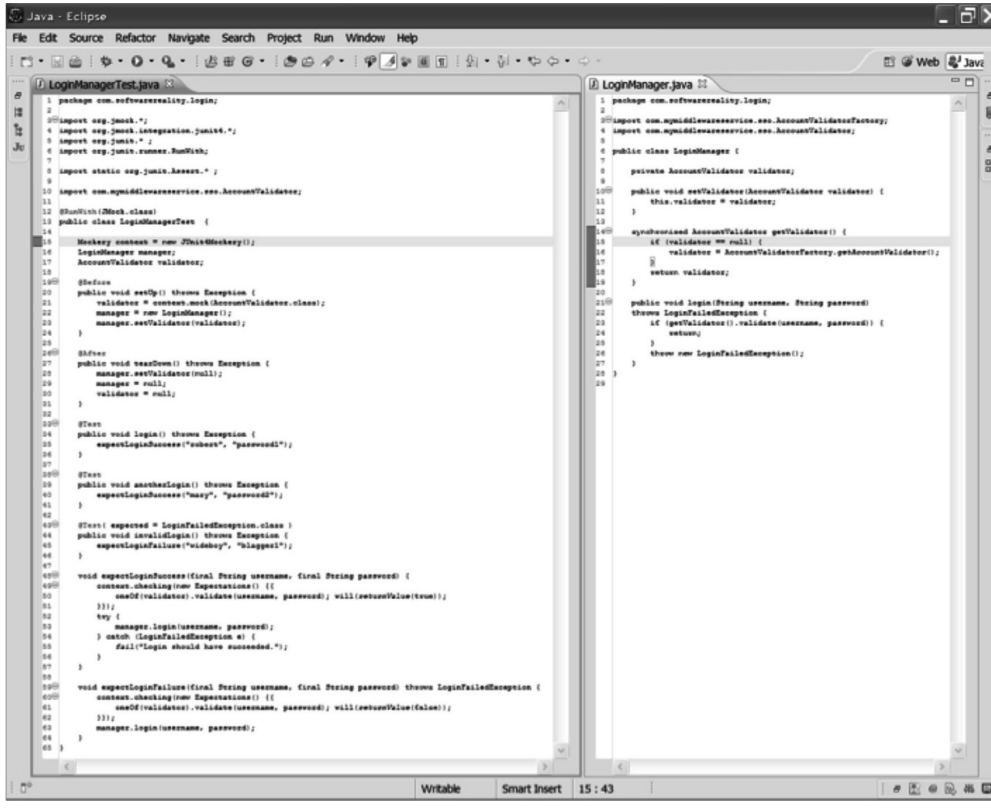
Uživatelský příběh u metodiky TDD vesměs začíná a končí testováním jednotek. Po zadání uživatelského příběhu vytvoříte testovací přípravky a od testu jednotky a kód produktu řídíte těmito testy. Je zde ovšem jeden zásadní aspekt testování, který často přehlížejí týmy, jež adoptují metodiku TDD: zákaznické testy (neboli testy přijatelnosti).

U testů přijatelnosti píšete testy, které „pokrývají" požadavky, zatímco u testů jednotek píšete testy, jež „pokrývají" návrh a kód. Můžete se na to podívat také tak, že testy přijatelnosti zajišťují, že programujete to, co máte, kdežto testy jednotek zajišťují, že to programujete správně.

Zatímco principy testování jednotek jsou jasné, srozumitelné a obecně snadno uchopitelné, rady ohledně testování přijatelnosti jsou vzácné a nejasné. Frameworky pro testování přijatelnosti, jako je Fitnesse, poskytují jen o málo více než prostor na papíře pro tabulky hodnot, přičemž obtížnou část spočívající v začlenění testů do kódové báze ponechávají jako cvičení pro čtenáře. Ve skutečnosti jsme si všimli, že řada vývojářů používajících metodu TDD o testech přijatelnosti prostě neuvažuje. Vypadá to, že v okamžiku, kdy byla metodika TDD vysvobozena z pout extrémního programování a odplula za horizont, zapomněla si vzít s sebou svoji naprosto základní část – zákaznické testy přijatelnosti. Téměř hypnotizující soustředění na kód není překvapivé, neboť obvykle jsou to programátoři, kteří do organizací metodiku TDD zavádějí.

Závěr: metodika TDD je opravdu nehorázně obtížná

Nyní, když jsme si prošli vyčerpávajícím cvičením řízení návrhu z testů jednotek, podívejme se znovu na to nepatrné množství kódu, které jsme tímto hrdinským úsilím vytvořili. Obrázek 2.10 ukazuje celkové množství kódu, který jsme testovali (na pravé straně, celkem 28 řádků ve třídě `LoginManager`), a kódu pro test jednotky, na jehož základě jsme se dostali až sem (65 řádků ve třídě `LoginManagerTest` na levé straně).



Obrázek 2.10: K vytvoření tohoto množství produkčního kódu (na pravé straně) bylo zapotřebí tohoto množství testovacího kódu (na levé straně) a to nepočítáme refaktorování

Na první pohled to skutečně nevypadá tak nesmyslně. 65 řádků kódu pro test jednotky k vytvoření 25 řádků produkčního kódu (něco přes 2:1) není tak nepřiměřená cena za kvalitní sadu regresních testů. Opravdový problém ovšem tkví v tom, kolik úsilí nás stálo vytvoření těch 65 řádků kódu pro framework JUnit.

Když uvážíme, že pro vývoj našich 65 řádků výsledného testovacího kódu bylo zapotřebí zhruba osm refaktorování, můžeme odhadnout s přibližnou úrovní úsilí při psaní 500 řádků testovacího kódu pro vytvoření 25 řádku produkčního kódu. Díváme se tedy na „násobič úsilí" něčeho v řádu 20 řádků testovacího kódu pro každý řádek produkčního kódu.

Po všem tom čeření kol, skřípání zubů a napodobování objektů mock objekty jsme ve skutečnosti otestovali pouze kód pro „ověření hesla". Veškeré další požadavky (zamčení účtu po třech nezdařených pokusech o přihlášení, uživatelské jméno není na hlavním seznamu účtů atd.) jsme řešit ani nezačali. Verze přihlašování (se všemi požadavky) postavená na metodice TDD by tedy naplnila kapitolu s 60 až 70 stranami.

Pokud vám to připadá nehorázně obtížné, pak si rozhodně užijete následující kapitolu, v níž uvidíte jednodušší způsob.

Shrnutí

V této kapitole jsme pokračovali v našem srovnávání metodik TDD a DDT, k čemuž jsme využili praktický příklad požadavku na implementaci přihlašování pomocí metodiky TDD. Začíná být zřejmé, jak se z kódu a testů klube návrh. Možná jste si ale všimli, že jsme strávili více času refaktorováním testovacího kódu než „skutečného" produkčního kódu (přestože to bylo z větší části přehnané jednoduchostí tohoto příkladu – ve složitějším příkladu bude mnohem více refaktorování produkčního kódu). Proces je navíc velmi nízkoúrovňový, mohli bychom říci krátkozraký, přičemž se od rané fáze zaměřuje na jednotlivé řádky kódu. Zatímco jedním z hlavních cílů metodiky TDD je zlepšit kvalitu kódu, v žádném případě nejde o metodiku poskytující „celkový pohled". Pro získání celkového pohledu na návrh systému, zvláště pak u rozsáhlejších projektů, je nutné zapojit kromě metodiky TDD ještě nějaký další návrhový proces.

V následující kapitole se do příkladu „Ahoj, světe!" pustíme znovu a tentokrát jej použijeme pro vysvětlení, jak funguje testování řízené návrhem (Design-Driven Testing – DDT).