VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

TESTOVÁNÍ SOFTWARE PŘI UPLATNĚNÍ VÝVOJE ŘÍZENÉHO DOMÉNOU

SOFTWARE TESTING IN DOMAIN DRIVEN DEVELOPMENT

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **TOMÁŠ POLEŠOVSKÝ**
AUTHOR

VEDOUCÍ PRÁCE — **RNDr. MAREK RYCHLÝ, Ph.D.**
SUPERVISOR

BRNO 2017

Zadání bakalářské práce/19695/2016/xpoles05

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2016/2017

Zadání bakalářské práce

Řešitel: **Polešovský Tomáš**
Obor: Informační technologie
Téma: **Testování software při uplatnění vývoje řízeného doménou**
Software Testing in Domain Driven Development
Kategorie: Softwarové inženýrství

Pokyny:

1. Seznamte se přístupem vývoje a návrhu řízeného doménou (Domain Driven Development/Design, DDD). Prozkoumejte důsledky uplatnění tohoto přístupu na testování software (zejména možnosti automatického generování testů). Analyzujte a porovnejte softwarové rámce s podporou DDD (např. Apache Isis).
2. Zvolte vhodný softwarový rámec s podporou DDD a pro tento navrhněte řešení pro automatické generování testů a jejich uplatnění ve vývojovém procesu (regresní a akceptační testy, atp.).
3. Po konzultaci s vedoucím implementujte nástroj pro podporu automatického testování v DDD.
4. Zdokumentujte výsledky a zveřejněte projekt pod svobodnou licencí jako open-source.

Literatura:

- Apache Isis. [https://isis.apache.org/]
- K. Klíč. *Vývoj řízený doménou*. Diplomová práce, FI MU, 2009. [https://is.muni.cz/th/99315/fi_m/]
- D. Ševčík. *Domain-Driven Design*. Bakalářská práce, FI MU, 2009. [https://is.muni.cz/th/173376/fi_b/]
- E. C. S. Santos, D. M. Beder, R. A. D. Penteado. *A Study of Test Techniques for Integration with Domain Driven Design*. In: 12th International Conference on Information Technology - New Generations (ITNG), Las Vegas, 2015, pp. 373-378. [http://dx.doi.org/10.1109/ITNG.2015.66]

Pro udělení zápočtu za první semestr je požadováno:

- Body 1, 2 a započatá práce na řešení bodu 3.

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

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

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

Vedoucí: **Rychlý Marek, RNDr., Ph.D.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 17. května 2017

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

L.S.

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

Tato práce se zabývá možnostmi automatizace testování software se zaměřením na aplikace vyvíjené pomocí techniky Domain-driven design (DDD). Na tomto teoretickém základě byl poté vytvořen generátor automatických testů v programovacím jazyce Java. Celé řešení demonstruje informační systém půjčovny aut, který ke své činnosti využívá framework Apache ISIS. Vygenerované testy spadají do agilní metodiky Behaviour-driven design (BDD).

Abstract

This thesis deals with possibilities of automation in software testing with focusing on applications developed using Domain-driven design (DDD). The automatic test generator was created in Java programming language and demonstrate the solution in car rental information system, which uses the Apache ISIS framework. The resulting generated tests are driven by behaviour (BDD).

Klíčová slova

testování software, automatizace, generátor testů, vývoj řízený doménou, programování řízené chováním

Keywords

software testing, automation, test generator, Domain-driven design, Behaviour-driven development

Citace

POLEŠOVSKÝ, Tomáš. *Testování software při uplatnění vývoje řízeného doménou*. Brno, 2017. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce RNDr. Marek Rychlý, Ph.D.

Testování software při uplatnění vývoje řízeného doménou

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana RNDr. Marka Rychlého, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Tomáš Polešovský
15. května 2017

Poděkování

Chtěl bych poděkovat svému vedoucímu bakalářské práce RNDr. Marku Rychlému, Ph.D za odborné vedení, za pomoc a rady při zpracování této práce.

Obsah

Kapitola 1

Úvod

Nezbytnou součásti každého procesu vývoje software je ověření všech uživatelských a technických požadavků. V případě, že aplikace některé z nich nesplňuje, má to přímý vliv na kvalitu výsledného produktu. Aby bylo možné všechny tyto požadavky ověřit, je nutné do jednotlivých fází vývojového procesu zařadit pravidelné testování. Existuje mnoho způsobů, jak toho docílit a není vždy nutné všechny testy provádět ručně.

Automatizace v dnešní době představuje efektivní prostředek, jak redukovat náklady, a tak není divu, že pronikla i do této oblasti. Výhod pro užití automatizovaných testů je hned několik. První z nich je snížení počtu lidských zdrojů, tedy pracovníků, kteří by za normálních okolností prováděli testy ručně. Další benefit spočívá v možnosti jejich spuštění kdykoliv a kdekoliv na světě. V neposlední řadě rovněž efektivně zabraňují znovuzavedení již opravených chyb.

Vytváření automatických testů však s sebou nenese jen samé výhody, tím nejzásadnějším negativem je nutnost udržovat testy vždy aktuální a určitý čas zabere také jejich prvotní implementace. Lze si však nějak tyto činnosti zjednodušit? Právě těmito metodami se zabývá následující práce, a to konkrétně u aplikací, které jsou vyvíjené pomocí přístupu Domain-driven design. Výstupem je pak generátor, který v rámci frameworku Apache ISIS vytváří automaticky dílčí sady testů na základě analýzy modelu aplikace.

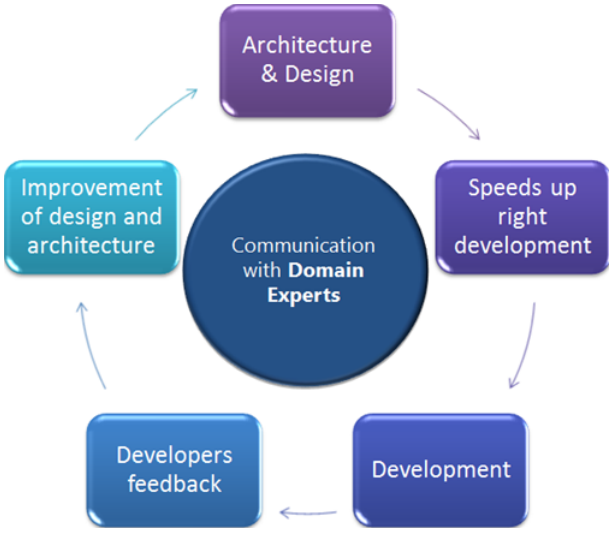
V jednotlivých kapitolách jsou postupně představeny způsoby testování software v jeho vývojovém procesu, dále je zaveden již zmíněný pojem Domain-driven design (DDD) a dojde také na porovnání přístupu vývoje řízeného testy (TDD) a chováním řízeného vývoje (BDD). Druhá polovina práce se již věnuje samotnému řešení generátoru, jeho architektuře, principu fungování a také frameworku pro demonstraci analýzy modelu. Úplný závěr pak slouží ke shrnutí dosažených cílů a zhodnocení výsledků práce.

Kapitola 2

Návrh a vývoj řízený doménou

Jeden z problémů, který se vyskytuje při vývoji software, je špatná analýza domény (tedy oblasti, kterou se software zabývá např. bankovnictví) a následná nevhodná abstrakce do doménového modelu. Následkem toho vznikají nepřehledné vazby mezi entitami modelu a programový kód se stává neudržitelný.

Snahu o zlepšení procesu návrhu doménové logiky projevil v roce 2003 Eric Evans, který ve své knize *Domain-Driven Design: Tackling Complexity in the Heart of Software* zavádí pojem Domain-driven design (DDD). Ten je definován jako přístup k vývoji software, který umožňuje vývojářům efektivně řešit návrh a údržbu softwarových projektů i se složitými doménovými problémy [8].



Obrázek 2.1: Životní cyklus vývoje aplikace [21].

Nutno podotknout, že DDD se neváže k žádné konkrétní metodice, používá se však společně s iterativním přístupem vývoje software a nejčastěji také s agilními metodikami, jako je například SCRUM či Extrémní programování (XP). Jak takový životní cyklus vývoje může vypadat, je možné si prohlédnout na obr. 2.1.

Kromě nutnosti modelovat objekty tak, aby co možná nejvíce odpovídaly skutečnosti, zdůrazňuje DDD rovněž nezbytnost komunikace a zavádí tzv. sdílený slovník (angl. *ubiquitous language*). To znamená, že všechny termíny, které si vyměňují jednotliví členové týmu

(doménoví experti, systémoví analytici, vývojáři), musí být zachyceny ve slovníku (v ústní či písemné podobě) [23][19].

Samotné DDD také nedefinuje žádné konkrétní návrhové vzory, ale pouze základní strukturu kódu. Jedná se tedy pouze o rozšířené dělení tříd na datové objekty a služby. Stejně jako v případě metodik se však užívá společně s dalšími vzory v rámci architektury podnikových aplikací.

2.1 Architektura aplikace

Typická podniková aplikace je rozdělena do čtyř následujících vrstev, kdy každá z nich řeší svůj specifický problém.

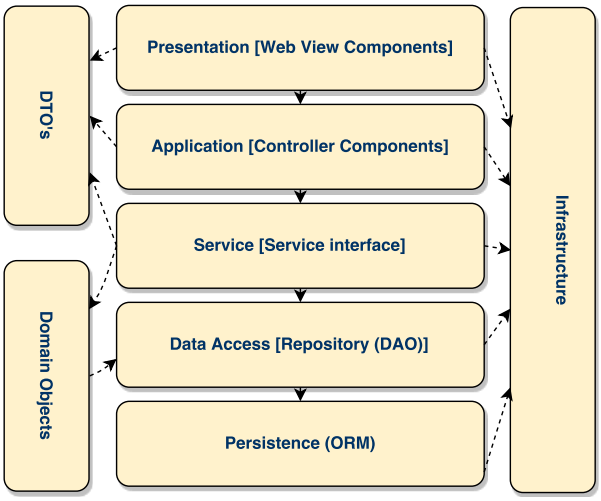
1. **Prezentační vrstva**
   Umožňuje interakci s uživatelem, prezentuje informace a interpretuje uživatelské příkazy.

2. **Aplikační vrstva**
   Koordinuje činnost aplikace, neobsahuje žádnou „business“ logiku.

3. **Doménová vrstva**
   Obsahuje doménové objekty a jejich stav. Persistence objektů je delegována na vrstvu infrastruktury.

4. **Vrstva infrastruktury**
   Slouží k podpoře všech ostatních vrstev, součásti jsou komunikační prostředky, knihovny apod.

Jak lze vidět z výše uvedeného rozdělení, srdce aplikace tvoří doménová vrstva, která obsahuje veškerou logiku aplikace zapouzdřenou do datových objektů. Součásti jsou také služby (angl. *services*) vymezující jejich chování. Důležité je, že zatímco datové objekty jsou stavové, služby pouze poskytují skrze své rozhraní možnost provádět operace v dané doméně [19][13].



Obrázek 2.2: Architektura aplikace diagram [13].

Celá vrstva musí být rovněž striktně izolována od ostatních části aplikace, a také by neměla být závislá na užitých frameworcích. Obrázek 2.2 ukazuje, jakým způsobem mohou být provázané různé vrstvy aplikace společně s DDD [13].

2.2 Základní doménové objekty

Primární úlohou při vývoji software je zvládnout převést komplexní celek na jednotlivé elementy. Tyto elementy, jak už bylo řečeno výše, lze rozdělit na datové objekty a služby, souhrnně se pak jedná o doménové objekty, které jsou provázané různými vztahy.

- **Entity**
  Entita je základní datový objekt, který často vychází z reálného světa a lze jej jednoznačně identifikovat. Skládá se z atributů, které ho popisují a metod pro získání, ukládání a úpravu těchto informací. V praxi se může jednat například o entitu Auto, která se skládá z atributů, jako je typ, značka nebo barva a je jednoznačně identifikována pomocí sériového čísla [8].

- **Objekty reprezentující hodnotu**
  Na rozdíl od entit, objekty reprezentující hodnotu nejsou založené na identitě. Z tohoto důvodu je jejich vzájemné porovnávání proveditelné pouze na základě atributů. Obecně se jedná o malé objekty s hodnotami inicializovanými na počátku, které se už v průběhu většinou nemění. Jako typickou ukázku lze uvést objekt Peníze, u kterého není potřeba vědět, o jakou bankovku se přesně jedná, ale důležitá je pouze jeho hodnota. [19].

- **Agregáty**
  Agregáty vytváří logickou hranici mezi doménovými objekty a zbytkem aplikace. Jejich cílem je snížit počet vzájemných vazeb a ustanovit jedno rozhraní pro společné spojení s okolím. To je možné prostřednictvím jedné či více tříd, kdy jedna z nich je označena jako kořenová a přes kterou probíhá veškerá komunikace. Struktura doménových objektů je skryta [8].

- **Služby**
  Pojem služba je poměrně obecné označení a jeho význam závisí na uváděném kontextu. V DDD se jedná o rozhraní, které poskytuje operace. Tyto operace jsou definovány ve sdíleném slovníku a souvisí s konkrétní oblasti SW. Základem však je, aby stavy byly delegovány na ostatní doménové objekty a služby zůstaly bezstavové [19].

- **Repozitáře**
  Pro pohodlné spojení doménového modelu s perzistentním uložištěm je nutné vytvořit vhodnou mezivrstvu. K tomuto účelu slouží repozitář reprezentující rozhraní pro práci s entitami. Zdroj datového uložiště může být libovolný, nejčastěji se však jedná o relační databázi. Výhoda tohoto řešení tak spočívá ve schopnosti obsluhy různých zdrojů na základě infrastruktury [19].

- **Továrny**
  Továrny, jak už vyplývá z názvu, vytvářejí různé konstrukce objektu na základě vkládaných parametrů. Instance koncové třídy vzniká a je navrácena ve veřejné metodě, typicky se jedná o metodu `create()`. Továrny ulehčuji práci zejména při tvorbě komplexních objektů. Spolu s technikou Dependency injection lze navíc snadným způsobem dosazovat libovolné registrované komponenty [19].

Kapitola 3

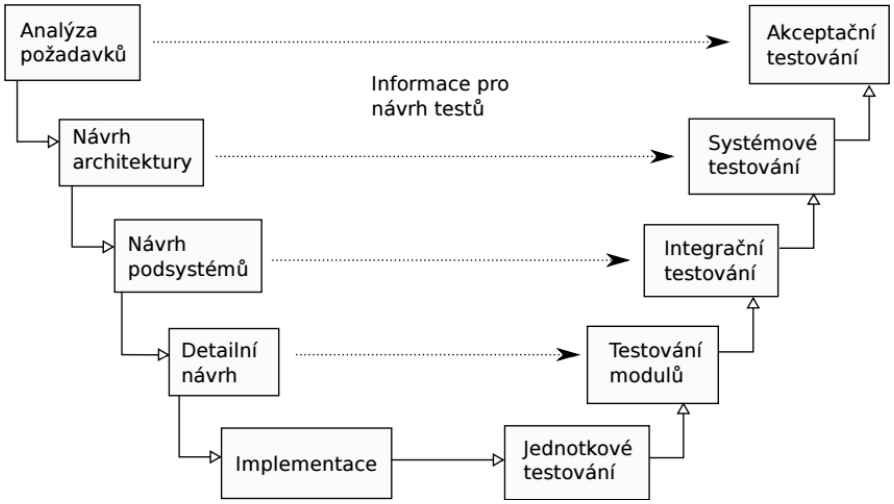
Testování software

Již od dob, kdy se začal vyvíjet první software, doprovázela vývojáře nepříjemnost v podobě chyb, které se v něm vyskytovaly. Někdy se jednalo pouze o drobnosti v uživatelském rozhraní, které znepříjemňovaly práci, jindy měly za následek škody v řádech milionů dolarů. Takovým příkladem je zkáza sondy Mars Climate Orbiter v roce 1999, která shořela v atmosféře Rudé planety v důsledku mylné manipulace s fyzikálními jednotkami [12].

Testování software je nicméně tak obsáhlé téma, že cílem této práce není popsat všechny známé techniky, ale pouze poskytnout určitý náhled do této problematiky, a to obzvlášť se zaměřením na automatizované testy a jejich integraci do etap vývojového cyklu.

3.1 Fáze testování

Vytváření aplikace lze rozdělit do několika základních etap, a to nezávisle na zvoleném modelu životního cyklu. S těmito fázemi souvisí také nutnost vhodně navrhnout jednotlivé formy testování. Bylo by samozřejmě možné provést všechny testy až po dokončení vývoje. Čím dříve je však chyba nalezena, tím menší jsou náklady na její opravení. Stanovení této strategie má proto přímý vliv na kvalitu konečného produktu. Obrázek 3.1 znázorňuje pět základních úrovní dle tzv. V-modelu (platí pro vodopádový i spirálový model) [24][18].



Obrázek 3.1: Fáze testování aplikace [18].

Akceptační testy

V rámci první fáze jsou zjišťovány požadavky klienta s účelem stanovit konkrétní specifikaci výsledného programu. Výstup tvoří nejčastěji dokumentace, která obsahuje různé UML diagramy (např. use case diagram) společně s textovým popisem.

Akceptační testování se pak zaměřuje na to, zdali tyto požadavky byly skutečně splněny. Realizace testů probíhá na straně klienta před definitivním převzetím projektu často na základě předpřipravených scénářů. V případě, že se vyskytnou nějaké nesrovnalosti mezi specifikací a zhotovenou aplikací či případné chyby, dochází k jejich oznámení vývojářskému týmu a dodavatel je v dohodnuté lhůtě odstraní [4].

Systémové testy

Po dokončení první etapy následuje tvorba architektonického návrhu. Ten definuje základní jednotky systému a způsob jejich vzájemné komunikace.

Pro ověření výsledku slouží systémové testy, které testují vytvářený systém jako funkční celek. Tuto činnost, na rozdíl od akceptačních testů, má na starost oddělená skupina programátorů případně samostatný tým testerů. Zjišťuje se, zda systém splňuje všechny požadavky, validují se výstupy a kontroluje se celková správnost návrhu. Z tohoto důvodu je vhodné tyto testy provádět nejen před finálním dokončením projektu, ale také již ve druhé části vývojového cyklu při samém návrhu. Architekturu software může být totiž obtížné v jeho pozdějších fázích změnit [24][18].

Integrační testy

Každá základní jednotka systému obsahuje další podsystémy (moduly). U jednodušších aplikací se jedná již o specifické komponenty. Ty mohou být vyvíjeny nezávisle na zbytku programu. Poté dochází k integraci těchto dvou dílčích části. Jelikož mezi těmito moduly probíhá komunikace, je nutné její bezchybnost ověřit pomocí integračních testů.

V praxi to znamená zejména testování správnosti rozhraní jednotlivých modulů a předpokladů pro vzájemnou komunikaci. U menších projektů se často tento typ testů vypouští – systém není dostatečně komplexní, a tak pro kontrolu funkčnosti celku plnohodnotně postačují systémové testy [4].

Testy modulů

Modul je izolovaná kolekce jednotek, které jsou v závislosti na zvoleném programovacím jazyku umístěné v jednom souboru (C), třídě (Java), balíčku (Python) či rozšíření (PHP). Předpoklad při tvorbě testů modulů představuje splnění základního chování. Modul zkrátka musí fungovat nezávisle na svém okolí a korektně reagovat na zaslané zprávy skrze jeho rozhraní. Tyto testy, na rozdíl od těch předchozích, už provádí většinou programátor daného subsystému [18].

Jednotkové testy

Těsně po vytvoření programového kódu a jeho kontroly vývojářem, přicházejí na řadu jednotkové testy. V případě objektově orientovaného programování se provádí u jednotlivých tříd a jejich metod, u procedurálního paradigmatu se pak jedná o testování jiné ohraničené části zdrojového kódu (např. procedury nebo funkce) [4][18].

Komplikace však mohou nastat při nesprávném návrhu aplikace, především pokud jsou na dané jednotce závislé další části systému. V takovém případě je často nutný rozsáhlý refaktoring kódu s užitím techniky mockování (odstínění závislosti). Proto je dobré myslet na tyto testy již při návrhu aplikace a vyhnout se tak později těmto nepříjemnostem [4].

Velká výhoda jednotkových testů spočívá také v jejich snadné automatizaci. K tomuto účelu slouží nejčastěji nějaký framework, v jazyku Java se používá kupříkladu JUnit.

3.2 Základní druhy a typy testování

Druhů, typů a metod provádění testů lze nalézt nepřeberné množství. Kromě níže uvedených se provádějí kupříkladu ještě Smoke testy – rychlé ověření, zdali je daná aplikace připravena na další fázi testování nebo třeba pozitivní/negativní testy [4]. V rámci této práce však budou stěžejní následující tři.

Regresní testování

Při každé úpravě aplikace, ať už se jedná o implementaci nových funkcionalit či změny stávajících, se mohou vyskytnout chyby. Přirozeně se tak děje například při refaktoringu jádra systému, kdy i na první pohled neviditelné závislosti mohou způsobit nepříjemné důsledky. Z tohoto důvodu je potřeba, aby ještě před uvedením do produkce celou aplikaci překontrolovali vývojáři. Pro zjednodušení tohoto procesu se využívají právě regresní testy, které jsou úzce spjaty s automatickým testováním.

Regresní testy jsou tak velice rozšířené, používají se takřka u většiny projektů a ve všech etapách vývojového cyklu. Jejich princip je prostý. V případě objevení nové chyby by měl být v ideálním případě nejdříve vytvořen test, který ji po spuštění vyvolá a teprve poté by mělo dojít k její opravě. Výhoda spočívá nepochybně v jistotě, že chyba je opravena korektně a v případě užití vhodného nástroje také v možnosti spustit tyto testy při budoucích úpravách [18].

Jako vhodnou ukázku lze uvést sadu nástrojů Selenium pro testování uživatelského rozhraní webové aplikace. Po přidání scénáře, napsaného v jednom z podporovaných programovacích jazyků, umožňuje Selenium jejich spuštění nekonečně mnohokrát.

Vedle regresního testování existuje také progresní testování, které se však moc nevyužívá. Za úkol má kontrolu nových funkcí implementovaných v aplikaci.

Statické a dynamické testování

Při provádění statických testů nedochází ke spuštění aplikace, analyzován je pouze zdrojový kód nebo specifikace programu. Této techniky je využíváno zejména v ranné fázi projektu [5].

Naproti tomu dynamické testy se užívají až v pozdějších fázích vývoje a již je nutné spuštění aplikace. Zkoumají se výstupní hodnoty při provádění zadaných testů. Tento proces může být buď automatizovaný nebo se provádí ručně [5][24].

Testování bílé a černé skříňky

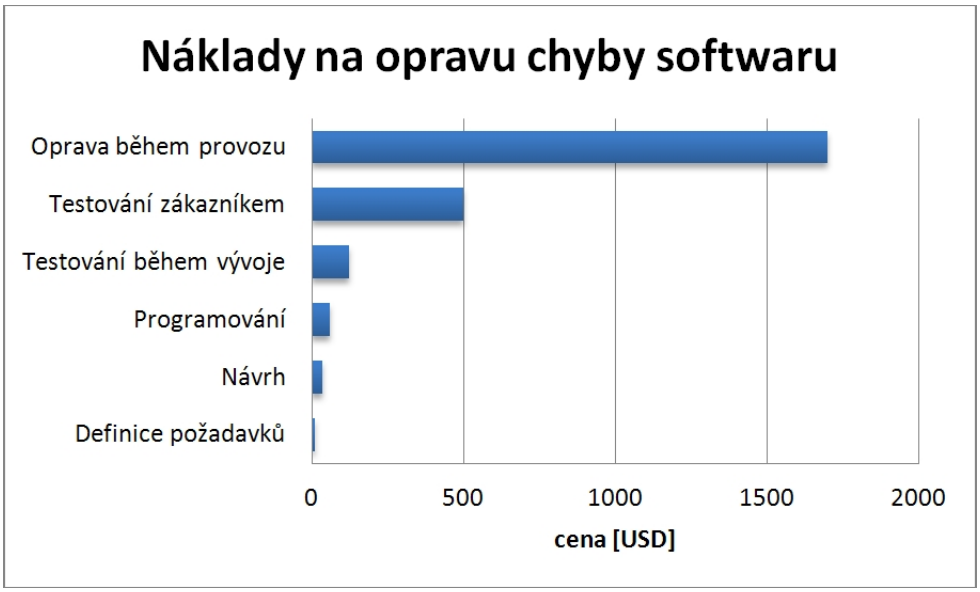
Další rozdělení dle přístupu k testování je analogie černé a bílé skříňky, kdy daný objekt představuje samotný software. Rozdíl mezi těmito přístupy spočívá ve viditelnosti vnitřní struktury. Při testování metodou černé skříňky není podstatná vnitřní konstrukce objektu ani jeho zdrojový kód. V případě zadání údajů na vstupu se kontrolují pouze hodnoty, které

se objeví na výstupu. Tento přístup umožňuje částečné odstínění od technické části a bližší pohled z hlediska zákazníka [12].

Oproti tomu v případě bílé skříňky se pozornost přesouvá ke zdrojovému kódu a mechanismu fungování dílčích algoritmů. Z tohoto důvodu je možné snadno nalézt chyby na konkrétním řádku programu a testy vytvořit na míru. Nevýhoda tohoto přístupu se však projevuje v nutnosti psát testy objektivně, a ne pouze popsat zdrojový kód (nedošlo by tak k maximálnímu pokrytí aplikace) [12].

3.3 Automatické testování

V předchozích dvou kapitolách padly již některé zmínky o automatických testech, nyní je třeba všechny tyto informace ucelit. Testování se v dnešní době považuje za plnohodnotnou disciplínu při vývoji software a počet členů v testovacích týmech může převyšovat i počet samotných programátorů (např. v kosmonautice) [6].



Obrázek 3.2: Náklady na opravu chyb [6].

Tyto náklady představují značnou finanční a časovou zátěž při budování projektu, a to zejména u oprav chyb během provozu (viz obr. 3.2). Přirozená snaha o jejich snížení vede jednoznačně k automatizaci. Tento důvod však není jediný. Další výhoda spočívá v omezení rutinní činnosti testerů, kteří pak věnuji svou pozornost důležitějším věcem. V neposlední řadě lze také snadno provést návrat k poslední stabilní verzi software. Kromě těchto přínosů se bohužel objevují v některých případech i nevýhody.

Ty plynou hlavně z nutnosti všechny testy udržovat. V praxi to znamená, že při každé změně aplikace musí dojít k jejich aktualizaci. V případě že se tak nestane, dochází buď k selhání testů při spuštění nebo zavlečení možných chyb, které nejsou pokryté a mohou se projevit později. Pro spouštění testů je nutné také přizpůsobit infrastrukturu (např. vytvořit testovací server), což může zvýšit časové nároky.

Přes všechny tyto nevýhody se vyplatí automatické testy ve většině projektu využít, i když je třeba zvážit všechny důsledky z nich vyplývající. Nejčastějším důvodem jejich

problematického nasazení se stává špatně zvolená oblast. Čím náročnější je jejich údržba, o to více je praktičtější zvolit ruční přístup k testování. Proto by se v praxi měly nejvíce vyskytovat automatizované jednotkové testy a integrační testy (popř. testy API, komponent). Vysoké pokrytí kupříkladu webového rozhraní (např. GUI) není z důvodu častých změn výhodné [7].

Realizace automatických testů

Pro zavedení automatických testů je nutné splnit několik podmínek. V případě již existující aplikace se situace stává komplikovanější, protože architektura často neodpovídá požadavkům a jsou potřeba její úpravy. Pokud je již v pořádku následuje výběr vhodných nástrojů.

Jejich výběr závisí jak na daném projektu, tak na naších preferencích. Kromě základních vlastností, jako je vytváření testovacích skriptů, často obsahují i další funkce např. správu chyb. Může se tak jednat o pouhý doplněk internetového prohlížeče (Selenium plugin) nebo celé firemní řešení (Jira).

Jestliže projekt, u kterého dochází k zavedení automatických testů, závisí na okolí nebo využívá nějaké další služby, vyplatí se technika tzv. mockingu nebo simulační nástroje. Cílem je vytvořit podobné podmínky jako v reálném prostředí. Typicky se jedná o závislosti na databázi (mockování objektů) nebo simulaci webových služeb (např. SOAP/Rest API).

Další krok spočívá v návrhu a následné kontrole testů. Po vytvoření jejich specifikace a implementaci je nutné stanovit jejich priority a určit, které z nich budou kritické. Tyto testy pak budou tvořit základní kostru u většiny testovacích plánů [24]. Kromě toho bývá v tomto kroku také vhodné zvolit metodiku životního cyklu (viz kapitola 3.4).

Integrace automatických testů

Poslední krok souvisí se spouštěním a vyhodnocováním automatických testů. Velké množství vývojářských týmů tento úkol podceňuje. Své testy spouští ručně, a to s větší či menší četností. Výsledkem pak mohou být například nefunkční části aplikace v repozitáři. Proto se vyplatí investovat do automatizace i v tomto směru [7].

Standardní proces pro vytváření buildů, provádění testů a dalších praktik se nazývá průběžná integrace (angl. *Continous integration*). V rámci něho jsou při každém vložení nového příspěvku do repozitáře spuštěny automaticky jednotkové testy a při vytvoření nové verze software také další typy testů. Průběžná integrace tak umožňuje lepší dohled nad softwarovými projekty, které jsou pak ve výsledku kvalitnější (obsahují menší počet chyb) [7].

3.4 Metodiky vytváření testů

Při moderním vývoji pomocí agilních metodik hledají vývojáři často způsob, jak optimálně pokrýt jejich kód testy (u jednotkových testů se jedná o pokrytí přibližně od 75 % do 85 %). Po několika experimentech s různými styly zvítězila technika programování Test-first. Ta zahrnuje psaní automatizovaných testů ještě předtím, než je vytvořena funkcionalita, kterou mají za úkol ověřit [22].

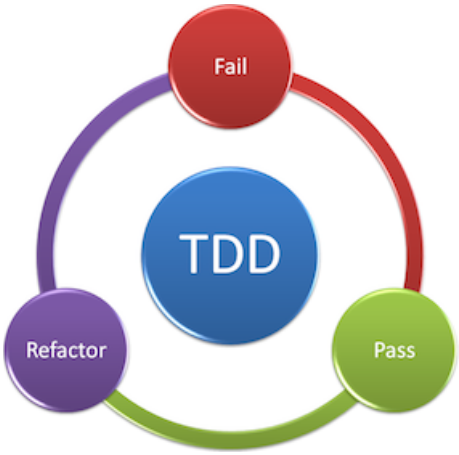
Mezi hlavní představitelé se řadí programování řízené testy (angl. *Test-driven development*) a programování řízené chováním (angl. *Behaviour-driven development*). Obě tyto metodiky techniky Test-First využívají.

3.4.1 Programování řízené testy

Programování řízené testy (zkráceně TDD) byla původně technika v rámci metodiky Extrémního programování (XP). Postupem času se z ní stal obecný životní cyklus, jakým lze vytvářet automatizované testy u agilního vývoje [1]. Ve vývoji řízeném testy platí vždy dvě pravidla [2]:

1. Nový kód je psán pouze tehdy, když automatizovaný test selže.
2. Je třeba eliminovat duplikace.

Tyto dvě základní pravidla jsou jednoduchá a snadno zapamatovatelná. Vychází z nich však mnohem více technických důsledků, které závisí na chování skupin a jednotlivců. U prvního z nich je potřeba brát ohled na již fungující kód k získání zpětné vazby. Kromě toho z nich plyne také nezbytnost psát vlastní testy, a nikoliv spoléhat na někoho jiného, než je napíše on sám. V neposlední řadě je potřeba takové vývojové prostředí, které pružně reaguje na malé změny a celá architektura se musí skládat z volně propojených komponent (jinak by testy nebylo možné vytvářet) [2].



Obrázek 3.3: TDD vývojový cyklus [14].

Jak už bylo řečeno v úvodu, v případě užití metodiky TDD by všechny funkce, které jsou stanovené projektovým vedoucím, měly být nejprve pokryté testy. Jak takový proces vypadá, lze vidět na obrázku 3.3. Základní trojici tvoří stavy – červená, zelená a refaktor, které reprezentují selhaný test, úspěšný test a proces refaktoringu.

Pro praktické použití TDD je možné vývojový cyklus rozdělit do pěti následujících kroků [1].

1. **Přidání nového testu**
   Na základě techniky test-first dochází v prvním kroku nejdříve k vytvoření sady testů k dané funkcionalitě. Ta je většinou popsána v dokumentaci či například za pomocí use case diagramu. Test ve své podstatě programově přesně definuje její specifikaci.

2. **Spuštění všech testů a jejich selhání**
   Možná se může zdát zvláštní spouštět testy ihned po jejich sestavení. V současném stavu přece musí zákonitě dojít k jejich selhání. I tak je vhodné tuto akci provést,

protože pokud by k jejich selhání nedošlo, bude to poukazovat na nějakou chybu v procesu (např. nedodržení TDD, neplatný test).

3. **Napsaní vlastního kódu dané funkcionality**
   Ve třetím bodu už konečně přichází na řadu psáni kódu specifikovaných vlastností. Úkolem však není vytvořit zdrojový kód efektivní a elegantní, ale pouze funkční, a to tak ať projde všemi testy. Dodatečné úpravy budou předmětem posledního kroku.

4. **Úspěšné spuštění všech testů**
   Nyní je již vlastní funkcionalita vytvořena a všechny testy by měly úspěšně projít. Pokud se tak nestane, dochází k opětovnému návratu na bod 3.

5. **Refaktorace**
   Po úspěšném dokončení funkční části kódu dochází k jeho dalším úpravám, které byly nastíněny v kroku 2. Jedná se tak zejména o odstranění duplicit, zlepšení čitelnosti kódu anebo rychlosti provádění. Výhodou je v tomto případě možnost opětovného spouštění testů.

Jelikož se všechny stavy životního cyklu opakují, celý proces probíhá, dokud projekt není dokončen. To umožňuje příjemné řízení vývoje i v rámci dalších agilních metodik jako je například SCRUM.

K hlavním benefitům TDD patří vysoké pokrytí aplikace testy a určitá garance, že všechny provedené změny budou řádně otestovány. Díky tomu v kódu nevznikají skryté chyby, ani nepředvídatelné vazby mezi jednotlivými komponentami. Další přínos spočívá v pravidelném refaktoringu u posledního kroku. Vývojáři jsou tak doslova tlačeni k tomu, aby jejich výstup zůstal čistý. Kromě toho je většina chyb odhalena mnohem dříve, což spolu s rychlejším debugováním umožňuje jejich méně nákladnou eliminaci.

Jako nevýhoda se naopak uvádí špatná testovatelnost některých částí aplikace (např. závislosti na externích zdrojích) a zprvu také větší časová náročnost vývoje. Nepříjemností se mohou stát i chyby, které se objeví v samotných testech. Přes všechna tato omezení je vhodné metodiku TDD využívat, a to obzvlášť u středních a větších projektů.

3.4.2 Programování řízené chováním

Během zavádění metodiky TDD v různých projektech docházelo často k nedorozuměním mezi vývojáři. Ti přicházeli s otázkami typu, kde začít testovat, co testovat, jak nazvat jednotlivé testy apod [10]. V závislosti na tyto otázky definoval Dan North novou agilní metodiku zvanou programování řízené chováním (BDD). Ta se zaměřuje primárně na způsoby vyřízení definovaného požadavku a jeho pokrytí testy [16]. Mezi její hlavní principy se řadí [9]:

- **Dost je dost** (angl. *enough is enough*) – pod tímto slovním spojením se skrývá jednoduchá myšlenka. Při samotné tvorbě je třeba se zaměřit pouze na to, co je důležité a co požadují zúčastněné strany. Veškeré další úsilí se může projevit jako zbytečné.

- **Dodání hodnoty pro všechny zúčastněné strany** (angl. *deliver stakeholder value*) – znamená snahu uspokojit všechny požadavky zúčastněných stran. Veškerá práce musí směrovat k něčemu, kde lze jednoznačně demonstrovat přidanou hodnotu.

- **Vše je chování** (angl. *it's all behaviour*) – popis aplikace musí být na všech úrovních stejný, tedy pohled na systém se musí u jednotlivých zúčastněných stran shodovat.

BDD se někdy označuje jako agilní metodika druhé generace, a to proto, že přímo vychází z TDD. Kromě něj kombinuje také v první kapitole popsaný DDD. Z těchto důvodů se nabízejí dva pohledy. První z nich je analytický a tyká se více dané domény a specifikace. Druhý pohled se zabývá čistě implementačními detaily testů [16].

Stejně jako v případě DDD dochází u analytické úrovně k vyzdvižení osob, které se vývoje účastní, nejčastěji se jedná o role klient, tester a vývojář. Aby se jejich popis chování systému nelišil, využívá se také sdíleného slovníku. Na této úrovni se jedná o psaní tzv. příběhů, které se skládají z jednoho či více scénářů. Ty mají následující tvar [15]:

As a **role**
I want a **feature**
So that **benefit**

V příběhu jsou vždy popsány tři aspekty. Role určuje, jakou pozici zaujímá zúčastněna strana, vlastnost požadovanou funkci a benefit její přínos. Každý tento příběh formuluje komponentu, která se bude vytvářet a rovněž specifikuje výstup iterace. Výhodou je jednoznačné vymezení hodnoty (angl. *business value*). Konkrétní příklad může vypadat následovně [15]:

Jakožto **uživatel kalkulačky**
Chci **sečíst dvě čísla**
Abych **se vyhnul matematické nepřesnosti**

Pro realizaci daného příběhu se využívají scénáře. Ty se skládají ze sekvence kroků, jejichž základní trojici tvoří pokud-když-pak (angl. given-when-then) s volitelným rozšířením a (angl. and). Zde se projevuje největší výhoda BDD, a to v čitelnosti scénářů pro všechny zúčastněné strany. Stanovení specifikace tak již není pouze doménou dokumentace [15].

- **Given** – slouží k nastavení kontextu u daného scénáře a výstavbě jeho pozadí. Popisuje současný stav situace a veškeré počáteční podmínky. Z tohoto důvodu se využívá vždy na začátku scénáře jako úvodní krok.
- **When** – definuje činnost, která jakmile se stane, tak přepíše výchozí podmínky stanovené na začátku a změní stav systému. Obsah představuje uživatelovu práci se systémem, a nikoliv komunikaci mezi jeho komponenty. Pro větší názornost je vhodné respektovat doménu, pro kterou je software vyvíjen.
- **Then** – tento krok vymezuje důsledky po reakci aplikace na předchozí činnost. Musí tedy dojít k ověření, zdali všechny původní předpoklady odpovídají skutečnému výsledku.

Každý řádek scénáře začíná jedním z výše uvedených klíčových slov s velkým počátečním písmenem. Přípustný je jejich libovolný počet za podmínky dodržení pořadí. Jednotlivé kroky by měly také obsahovat pokaždé pouze jednu skutečnost, a to z důvodu snadnější implementace. Konkrétní příklad reprezentuje následující úryvek [15]:

Scenario: Přidání zboží do košíku
Given nákupní košík je prázdný
When přidám nové zboží s cenou 25 Kč
When přidám nové zboží s cenou 30 Kč
Then košík obsahuje 2 ks zboží s cenou 55 Kč

Tento scénář se ukládá do klasického textového souboru. Pro automatizované spuštění je tak ještě potřeba vytvořit implementaci dílčích kroků. K tomu se využívají specializované nástroje (frameworky). Jako příklad lze uvést Cucumber v programovacím jazyku Java. Každý unikátní řádek je pak reprezentován příslušnou metodou.

```
@Given("^nákupní košík je prázdný$")
public function basketIsEmpty();

@When("^přidám nové zboží s cenou (\\d+) Kč$)
public function addNewProduct(int price);

@Then(^košík obsahuje (\\d+) ks zboží s cenou (\\d+) Kč$)
public function basketProducts(int number, int totalPrice);
```

Kapitola 4

Tvorba generátoru automatických testů

V první kapitole byla představena technika DDD. Nyní je třeba prozkoumat, jakou míru automatizace lze při jejím použití uplatnit k tvorbě testů. Ideální stav by nastal v případě, kdy by po návrhu doménové logiky dokázala aplikace v podobě generátoru vytvořit kompletní sadu testů pouze na základě kontextu. To však v současné době bohužel není možné. I přesto nicméně existuje alespoň několik dílčích způsobů, které umožňují zvýšit úroveň automatizace.

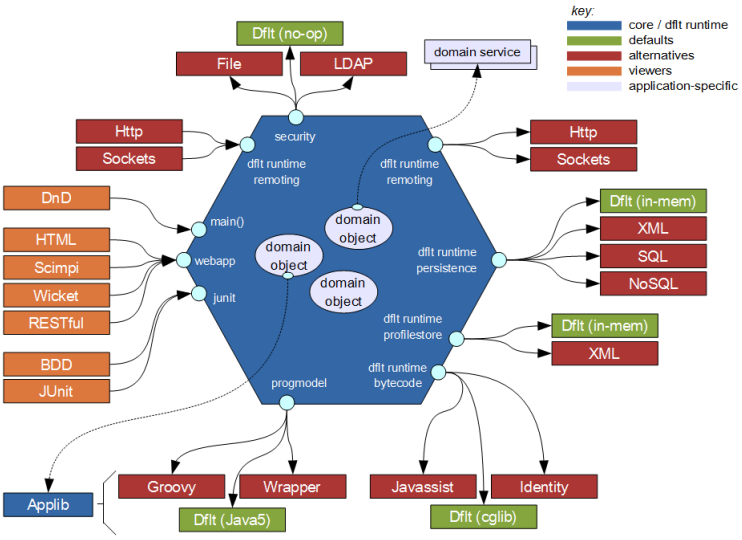
Roku 2015 na konferenci zvané „*Information Technology - New Generations (ITNG)*“ byla v příspěvku „*A study of test techniques for integration with Domain Driven Design*“ teoreticky nastíněna jedna z těchto metod [17]. Cílem této práce je tak navázat na zmíněnou publikaci a v praxi demonstrovat konkrétní implementační řešení.

Pro realizaci generátoru automatických testů byl použit programovací jazyk Java spolu s frameworkem Apache ISIS a nástrojem Cucumber. Ten zajišťuje za pomocí agilní metodiky BDD (popsané v kapitole 3.4.2) integrační testy celé aplikace.

4.1 Framework Apache ISIS

Apache ISIS je framework pro rychlý vývoj webových aplikací (RAD) na bázi Domain-driven design. Jeho výhody se projevují zejména při prototypování aplikací, lze jej však využít i pro produkční verzi software. Struktura projektu není většinou založena na standardním MVC přístupu, ale je reprezentována hexagonálním architektonickým vzorem (viz obr. 4.1) [3].

Z tohoto důvodu stačí, aby se vývojář soustředil pouze na návrh doménového modelu, který tvoří srdce aplikace. Zbytek dokáže Apache ISIS vygenerovat sám. Typicky se jedná o grafické prostředí nebo RESTful API.



Obrázek 4.1: Hexagonální architektonický vzor [3].

Valná většina dnešních frameworků obsahuje pro rychlý start (angl. *quickstart*) základní strukturu projektu. Ne jinak je tomu i u Apache ISIS. Zde se architektura pro lepší přehled dělí na pět modulů, které spolu vzájemně spolupracují [20].

- **Application** – předmětem tohoto modulu je volitelný aplikační manifest a libovolné podpůrné služby.
- **DOM** – pro vývojáře nejdůležitější část systému, obsahuje doménový model, který se skládá z jeho typických tříd označenými anotacemi.
- **Fixtures** – zahrnuje pomocné třídy, které slouží pro inicializaci systému v případě demo aplikace nebo automatických testů. Umožňují vygenerovat například doménové objekty s náhodnými hodnotami atributů.
- **WebApp** – využívá se ke spuštění webové aplikace (Wicket viewer), která vizuálně prezentuje doménový model pro koncové uživatele.
- **Integrations tests** – tento modul spouští v rámci aplikace integrační testy na bázi scénářů, které jsou jeho obsahem.

Aby bylo možné automaticky spustit integrační testy, využívá ke své práci Apache ISIS framework Cucumber. Ten zpracovává scénáře (přípona feature) napsané v jazyku Gherkin ve standardním formátu given/when/then. Dílčí kroky popsané pomocí regulárních výrazů v anotaci příslušných metod jsou pak součásti potomků třídy `CukeGlueAbstract`. Integraci zajišťuje JUnit Runner, který umožňuje spuštění testů v rámci CI nebo manuálně skrze příkazový řádek [20].

4.1.1 Konkurenční technologie

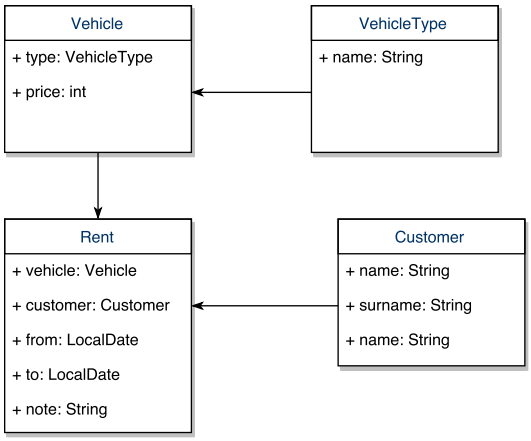
Mezi další frameworky, které podporují techniku DDD se řadí OpenXava (`www.openxava.com`) nebo JDon (`www.en.jdon.com`).

První z nich je plnohodnotně srovnatelný s Apache ISIS. Na základě vytvořeného doménového modelu umožňuje taktéž vygenerovat grafické webové rozhraní. Jeho součásti je poměrně dobře napsaná dokumentace a aktivní komunita vývojářů. Kromě toho nabízí také placenou verzi XavaPro, která rozšiřuje původní framework o další vlastnosti, jako jsou uživatelé, jejich role, správa oprávnění nebo mobilní verze. Jako nedostatek však lze hodnotit absenci podpory agilní metodiky BDD a ze subjektivního hlediska menší přehlednost uživatelského rozhraní.

JDon naproti tomu není zástupcem RAD přístupu, proto funkce na generování dalších částí aplikace zcela chybí. Hlavní přednost totiž spočívá v zaměření na reaktivní paradigma programování. Framework tak kombinuje architekturu založenou na událostech (EDA) s DDD. Výhody takového řešení vyplývají z pružnosti systému na asynchronní dotazy, snadné rozšiřitelnosti nebo menší provázanosti komponent. K podstatným negativům JDonu nicméně patří nízká podpora ze strany vývojářů a slabší dokumentace.

4.2 Informační systém půjčovny aut

Jako příklad aplikace vyvíjené ve frameworku Apache ISIS poslouží jednoduchý informační systém půjčovny aut. Ten bude využit také k demonstraci generátoru automatických testů. Obrázek 4.2 znázorňuje UML diagram jeho doménového modelu, konkrétně závislosti základních stavebních kamenů – entit.



Obrázek 4.2: Doménový model IS půjčovny aut.

Kromě nich používá systém ke své práci také další objekty ze sady DDD, jako jsou například objekty reprezentující hodnotu nebo agregáty a rovněž speciální třídy UI vrstvy. Z důvodu jejich rozlišení se využívají anotace, které definuje framework. Základní rozdělení rozlišuje stavové doménové objekty (`@DomainObject`) a služby (`@DomainService`). Níže jsou uvedené dvě ukázky doprovodných tříd pro entitu `Vehicle`.

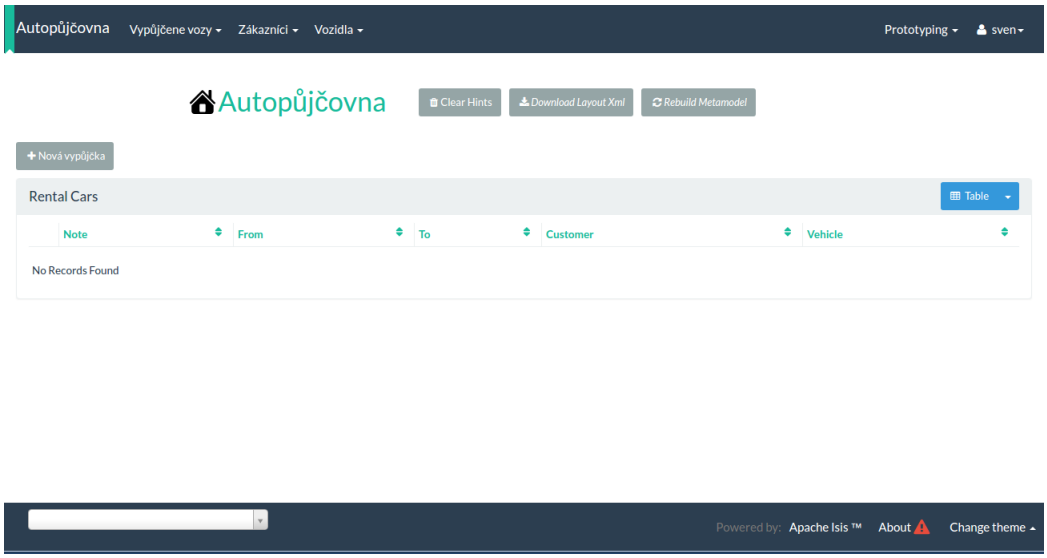
- **VehicleRepository** – obsahuje metody pro práci s perzistentním úložištěm např. specializované dotazy za pomocí JDOQL (rozšíření JDO) nebo ukládání nových objektu.
- **VehicleMenu** – jedná se o službu označenou jako `VIEW_MENU_ONLY`, což znamená, že slouží pro vymezení akcí a položek v menu.

Velké pozitivum přináší fakt, že stačí v zásadě pouze tyto tři objekty a základní rozhraní je hotové (např. seznam vozidel nebo jejich úpravy). Ke konfiguraci jednotlivých parametrů poté znovu poslouží anotace. Ty se tykají třeba pojmenování položek v menu, určení jejich pozic nebo definice vlastních formulářů.

Mimo to se vzhledem souvisí i layout stránky, tedy rozložení dílčích prvků. Aby bylo možné provádět jakékoliv jejich úpravy, existuje k tomuto účelu XML soubor, který nese počáteční název jako stránka, se kterou se pojí – v tomto případě `Vehicle.layout.xml`. V devadesáti procentech případů tak stačí změny realizovat pouze zde a není potřeba zasahovat do žádných jiných html, css nebo javascript souborů.

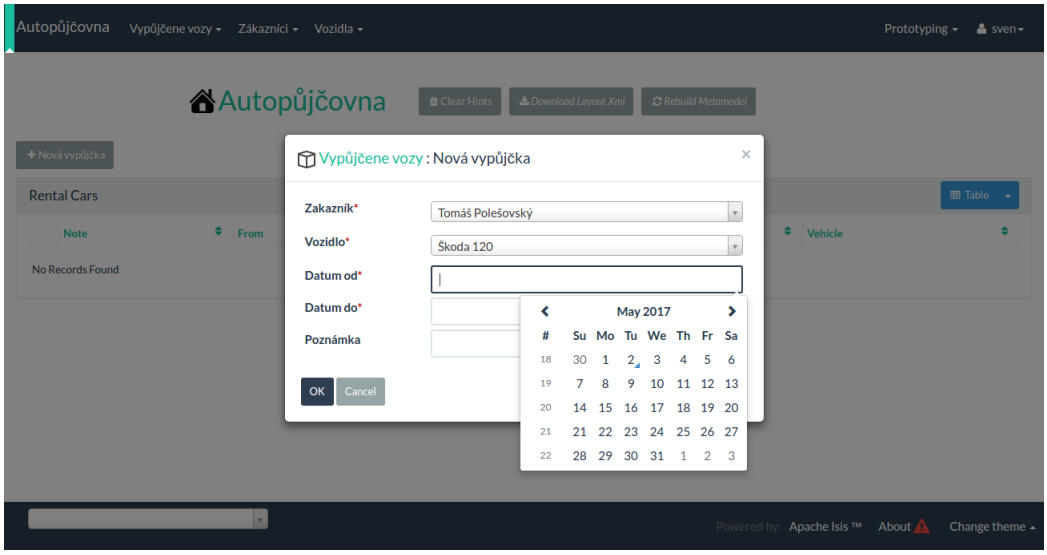
Apache ISIS pro svůj chod ve výchozím stavu využívá aplikační server Jetty ve formě Maven pluginu. Spuštění systému je tak otázkou jednoho příkazu `mvn jetty:run` ve složce s projektem (konkrétně webapp). Více informací o zavedení aplikace lze nalézt v dokumentaci frameworku.

Po jejím zapnutí se na příslušné adrese a portu nachází úvodní obrazovka s přístupem do UI rozhraní pro koncové uživatele (Wicket Viewer) a vygenerované RESTful API (Swagger-UI). Při vstupu do webové aplikace je nutné se přihlásit pomocí uživatelského jména a hesla. V případě ponechání počátečního nastavení v soboru `shiro.ini` se jedná o sven/pass. Jakmile je autorizace dokončena, uživatel může zahájit práci se systémem (viz obr. 4.3).



Obrázek 4.3: Úvodní obrazovka IS půjčovny aut.

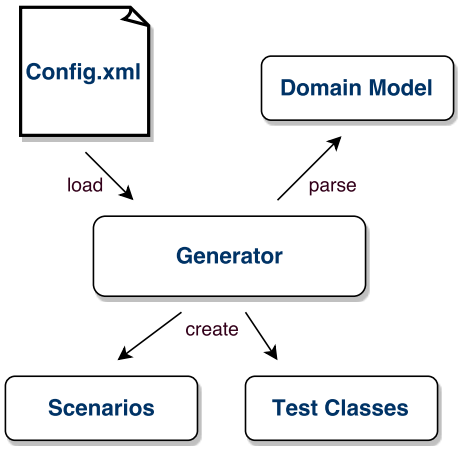
Aplikace zahrnuje seznam aut společně s jejich výpůjčkami a klienty. Všechny položky je možné libovolně editovat či přidávat. Validaci provádí specializované metody `validateXXX()`, které budou mimo jiné předmětem pro generovaní automatizovaných testů. Kromě nich se používají i jiné metody, jako je například `autoCompleteXXX()` pro tvorbu rozbalovacích nabídek nebo `removeFromXXX` za účelem odstranění požadovaného objektu z kolekce. Obrázek 4.4 znázorňuje jeden z těchto formulářů.



Obrázek 4.4: Příklad formuláře IS půjčovny aut.

4.3 Analýza a návrh generátoru

Jak již bylo naznačeno v předchozích kapitolách, základní koncept generátoru automatických testů vychází z analýzy doménového modelu dané aplikace. Po jejím dokončení dochází k vytvoření několika sad scénářů společně s kroky implementovanými v testovacích třídách. Přehledně tento proces zachycuje obr. 4.5.



Obrázek 4.5: Koncept generátoru automatických testů.

Jelikož existuje hned několik možností, jak by mohl výsledný generátor automatických testů vypadat, je nutné nejdříve definovat základní požadavky. Celkem se k nim řadí pět následujících podmínek.

1. Úloha scénářů vychází z potřeby základního otestování aplikace před jejím spuštěním. To znamená, že se bohužel neváže na doménu software. Takové řešení by totiž

vyžadovalo mnohem složitější analýzu. Místo toho se však budou scénáře skládat z jednoduchých kroků, které ověří správnost implementace.

2. Po vygenerování sady testů již stačí provést pouze minimální změny (např. doplnit konkrétní hodnoty). Nemělo by se tak stát, že úsilí vývojáře bude větší než kdyby si testy vytvářel sám.

3. Další podmínku představuje nezbytnost snadné integrace aplikace do vývojového procesu a její pravidelné automatické spouštění. Doplňkem pak může být rovněž manuální generování testů pomocí CLI Runneru.

4. Návrh architektury aplikace též počítá s tvorbou vlastních scénářů na bázi infrastruktury generátoru.

5. Součástí by také měly být již předpřipravené testovací sady, které lze v případě potřeby využít nebo rozšířit.

4.3.1 Architektura

S ohledem na poslední dva body požadavků musí být architektura generátoru navržena dostatečně flexibilně. Jednotlivé součásti pak budou využity při tvorbě vlastních testovacích sad. Z tohoto důvodu celkem vzniklo přibližně pět prototypů návrhu, z nichž byl vybrán nejlepší. Ten se dělí na několik následujících částí.

Vyhledání vhodných tříd

Za výběr potřebných tříd zodpovídá tzv. Scanner. Ten postupně prochází iterativně stromovou strukturu souborového systému a při nalezení souboru s příponou `.java` vytvoří instanci třídy `ClassFile`. Jejím obsahem je název příslušného souboru, jeho absolutní/relativní cesta a jmenný prostor. Z důvodu větší efektivity neobsahuje Scanner žádnou vnitřní paměť, ale pouze pomocný zásobník. Řízení vyhledávání přenechává na vnějším okolí (metody `next()` a `hasNext()`).

Analýza modelu

Způsob řešení analyzátoru doménového modelu blíže popisuje až kapitola implementace. Co se však týče jeho funkce, tak ta spočívá především v rozboru dílčích tříd. Z hlediska architektury je pak rozdělen na více částí s přesně definovanou úlohou (např. analýza metod), které spojuje agregátor. K ukládání získaných informací slouží hašovací tabulka, která pojímá všechny instance třídy `MetaClass` s požadovanými údaji.

Generování scénářů a testovacích tříd

Přímo na analyzátor modelu navazuje generátor testů. Jeho návrh je obecný a dělí se na generátor scénářů a testovacích tříd. Součást prvního z nich tvoří návrhový vzor stavitel (angl. *Builder*), který umožňuje definovat jednotlivé kroky scénáře a také implementace rozhraní `IScenarioGenerator` samotného generátoru. Pro vytváření testovacích tříd pak existuje obdobná forma řešení za pomocí `ITestSetGenerator`.

Všechny výše uvedené třídy využívají benefitů analyzátoru modelu, konkrétně jeho hašovací tabulku. Na základě těchto informací dochází k automatickému doplnění hodnot

proměnných a zápis do příslušných souborů. Přesný návod na tvorbu testovacích sad shrnuje kapitola 5.5.

Konfigurace

Nastavení aplikace má na starost konfigurační soubor `config.xml`, který je umístěn v kořenovém adresáři projektu. Jeho obsah zahrnuje vstupní a výstupní cesty k požadovanému informačnímu systému a další doplňkové parametry, jako je např. uvedení jmenného prostoru. Mezi další užitečné vlastnosti se řadí seznam aktivních testovacích sad. Na základě něj jsou pak automaticky spouštěny příslušné generátory.

4.4 Implementace generátoru

Jedna z hlavních otázek, které bylo třeba při implementaci řešit, se týkala způsobu analýzy doménového modelu. V zásadě existují dvě metody přístupu k tomuto problému, přičemž každá má své přednosti a nedostatky.

První z nich vychází ze struktury klasického kompilátoru pro překlad programovacích jazyků. V případě jejího užití by se architektura skládala z lexikálního analyzátoru společně s prostředkem na zpracování příslušných informací a jejich uložení do vnitřní reprezentace programu. Princip činnosti spočívá ve vyhodnocení těchto jazykových konstrukcí a vygenerování odpovídajících testů. Mezi výhody patří jednoznačně větší hloubka analýzy a možnost rozboru libovolných souborů. Podstatný handicap však tkví v rozsáhlosti a rozmanitosti syntaxe zdrojového kódu Javy. Proto tato technika není kvůli její složitosti úplně vhodná.

Druhá metoda je o poznání jednoduší. Ke svému základnímu fungování využívá tzv. reflexi. Tato schopnost umožňuje při běhu aplikace zkoumat nebo měnit její chování [11]. Z tohoto důvodu je získání potřebných dat mnohem snazší a efektivnější. I zde se však objevuje několik nepříjemných komplikací. Jedná se zejména o nutnost všechny nezbytné třídy správně načíst a také zajistit jejich závislosti. Druhý problém pak souvisí s vnitřní reprezentací běžícího programu, která nemusí obsahovat zcela úplné údaje o jeho struktuře. To se týká například parametrů metod nebo konstruktorů, kde jsou jejich názvy nahrazeny náhodným řetězcem. Přes veškeré tyto nesnáze se nicméně jeví tento způsob jako nejlepší.

Generátor automatických testů je vytvořen stejně jako informační systém půjčovny aut v programovacím jazyku Java verze 8. Tato volba nebyla učiněna náhodou, ale z důvodu schopnosti aplikace načíst požadované třídy. S jiným programovacím jazykem či jeho verzí by řešení za pomocí reflexe nebylo možné. Pro správu závislostí znovu poslouží nástroj Maven a využity jsou také dvě externí knihovny. Za účelem urychlení vývoje se jedná o Projectlombok a API Mavenu s cílem usnadnění práce abstrahuje Naether.

Java reflection API

Konkrétní implementaci reflexe v Jave zajišťuje její rozhraní zvané Java reflection API. K nejvýznamnějším typům patří `Method`, `Constructor` a především `Class`. Pomocí nich lze například vytvářet instance neznámých tříd v době komplikace nebo volat jejich metody [11]. Při zkoumání vnitřní struktury nicméně tyto informace musí být v době běhu programu vždy uloženy. Pokud třeba není dostupný anotační typ rozhraní, tak v rámci reflexe jsou všechny anotace neviditelné.

Dynamické načítání tříd

V případě, že generátor automatických testů není integrovaný v informačním systému, není možné využít reflexi, aniž by třídy nebyly načtené. Vzniká tak poměrně složitý problém, jehož řešení vyžaduje jejich dynamické načítání za běhu včetně všech potřebných knihoven. V Jave se k tomuto účelu používá tzv. `ClassLoader`, což je komponenta JRE. Během přidávání třídy do JVM musí být z důvodu stability a bezpečnosti veškeré její závislosti dostupné. Tyto dvě operace nelze od sebe oddělit.

Správa závislosti

V souvislosti s výše popsaným problémem se nabízí otázka, pomocí jaké metody lze tyto závislosti získat. Jelikož deklarace jednotlivých balíčků u informačního systému probíhá v Mavenu, nastává zde možnost zpracování jeho konfiguračního souboru `pom.xml`. Základní formát pro připojení externí knihovny je pak následující.

```
<dependency>
    <groupId>id skupiny - org.projectlombok</groupId>
    <artifactId>id artefaktu - </artifactId>
    <version>verze artefaktu</version>
</dependency>
```

Maven kromě tohoto stylu zápisu poskytuje i některé další, které však nejsou v rámci dané aplikace podporovány, jelikož svým rozsahem již přesahuji hranice této práce. Jedná se kupříkladu o různé podpůrné pluginy webových služeb. Proto je pro korektní práci generátoru třeba mít výše představený formát na paměti.

Původní záměr předpokládal analyzovat konfigurační soubor tradiční cestou díky rozhraní `DocumentBuilder`. Existuje nicméně schůdnější prostředek a tím se stává API Mavenu, konkrétně modelová komponenta `MavenXpp3Reader`. Její součást tvoří metoda na získání závislosti ve formě seznamu tříd typu `Dependency`.

Nevýhoda bohužel spočívá ve faktu, že na tyto knihovny mohou navazovat další balíčky. Celkově tak vzniká stromová struktura. Aby proto tento postup byl vůbec možný, přichází na řadu rozšíření nazvané Naether. To dokáže automaticky vyhledat celý výčet závislostí ve tvaru odkazů do lokálního repozitáře v Mavenu.

Poté, co jsou požadované balíčky úspěšně přidány, dochází rovněž k postupnému načítání tříd. Za situace, kdy je knihovna využívána nestandardním způsobem, umožňuje konfigurační soubor generátoru definovat její absolutní cestu.

Definice výchozích hodnot pomocí anotací

Další nesnáze, která se projevuje při užívání reflexe, tkví v nemožnosti zjistit názvy parametrů (případně dalších hodnot) v době běhu programu. K vyřešení této záležitost lze použit dva různé postupy. První z nich je jednoduše doplnění těchto hodnot do scénářů po vygenerování testů. V případě integrace do vývojového cyklu se však jedná o krajně nepohodlnou možnost.

Proto výhodnější řešení zahrnuje určení výchozích hodnot pomocí anotací již v samotném informačním systému. Jejich deklarace se provádí před parametrem metody. Pokud u nějakého z nich tato anotace chybí, bude automaticky doplněn náhodný řetězec/číslo (v závislosti na datovém typu) do vygenerovaných scénářů.

Vytváření nových objektů

Jednotlivé metody u testovacích tříd, které mapují kroky scénáře, často musí pracovat s novou instanci libovolné doménové třídy. Za těchto okolností nastává situace, kdy její konstruktor vyžaduje parametry, ať už ve formě primitivních typů nebo dalších objektů. Znovu se tak rodí stromová struktura, u níž může být obtížné správně inicializovat její potomky.

Východiskem tohoto stavu jsou speciální skripty (angl. *fixtures*), které umožňují definovat libovolnou akci po zavolání jejich metody `execute()`. Vhodné tak budou i v případě vytváření nových objektů. Pokud generátor nenalezl alespoň jednu instanci příslušné třídy, pokusí se vyhledat dotyčný skript ve tvaru `{NazevObjektu}Create` a následně jej spustit. Jejich implementace se pak provádí ve jmenném prostoru `projekt.dom.generator.fixture` modulu Dom.

Tento způsob však není úplně ideální zejména z nutnosti manuálních zásahů. Vyplatila by se zde proto hlubší analýza doménové vrstvy a automatizovaná tvorba podobných objektů na základě kontextu nebo případných předdefinovaných hodnot.

Přepisování scénářů a testovacích tříd

Pokud dochází k ručním úpravám scénářů nebo testovacích tříd, nastává potřeba tyto změny trvale uložit. Kdyby se tak nestalo, veškeré provedené úpravy by během dalšího spuštění generátoru byly ztraceny. Tuto komplikaci poměrně snadným postupem znovu řeší anotace u testovacích tříd a komentáře ve scénářích.

V prvním případě se anotace zapisuje ve formátu `@TestChanged(number=pocet_radku)` před danou metodou a jelikož její zpracování neprobíhá skrze reflexi, ale pouze mechanickým čtením souboru, musí být umístěna na samostatném řádku. U scénářů se technicky jedná o ten samý způsob, liší se akorát formou zápisu ve tvaru komentáře `#@changed`. Umístění se provádí před každým krokem, který je potřeba zachovat.

Kapitola 5

Demonstrace generátoru automatických testů

V předchozích kapitolách byl předveden návrh aplikace generátoru automatických testů a jeho implementace. Nyní nastává čas na spuštění a demonstraci celého projektu. Ta bude provedena na již popsaném informačním systému půjčovny aut. Aplikace rovněž obsahuje tři předpřipravené testovací sady, nicméně její smysl spočívá také v poskytnutí platformy pro tvorbu vlastních souborů automatických testů. To je díky flexibilní architektuře na základě dostupných komponent systému poměrně jednoduché a podrobně se těmto principům věnuje závěrečná kapitola.

5.1 Konfigurace informačního systému a generátoru

Ještě před samotným spuštěním generátoru musí být správně nastavený projekt informačního systému, respektive jeho konfigurační soubor `pom.xml`. Největší komplikace, která již byla popsána v kapitole 4.4, tkví ve složitosti balíčkovacího systému Maven. Jelikož u standardních knihoven Apache ISIS nejsou definovány verze v přímém zápisu u konfiguračního souboru, musí se zapsat ručně. V praxi tak stačí přidat vždy tag `<version>` u jednotlivých závislostí. Níže uvedený příklad reprezentuje pro názornost zápis knihovny `org.apache.isis.core`.

```
<dependency>
    <groupId>org.apache.isis.core</groupId>
    <artifactId>isis-core-applib</artifactId>
    <version>1.13.1</version>
</dependency>
```

Kromě toho je nutné obdobným způsobem rovněž přidat balíček `org.datanucleus`, ten se totiž standardně nahrává za pomocí pluginu pro správu závislostí.

```
<dependency>
    <groupId>org.datanucleus</groupId>
    <artifactId>datanucleus-core</artifactId>
    <version>4.1.7</version>
</dependency>
```

Tímto krokem je v zásadě celý projekt informačního systému správně nastaven. Jako doplněk lze pak samozřejmě využít některé z již zmíněných anotací dle aplikované testovací

sady. K tomuto účelu je třeba přidat obvyklým postupem knihovnu `com.bc.test`. Nejedná se však o povinnou součást konfigurace.

```
<dependency>
    <groupId>com.bc</groupId>
    <artifactId>test</artifactId>
    <version>1.0</version>
</dependency>
```

Běh systému standardně zajišťuje webový server Jetty, který se běžně používá v integraci s Maven pluginem. Díky němu lze tak projekt snadno spustit z příkazového řádku bez nutnosti nějakého dalšího nastavení.

```
$ mvn -pl webapp jetty:run
```

Co se týče generátoru automatických testů, tak jeho nastavení upravuje soubor `config.xml`, který je umístěn v kořenovém adresáři této aplikace a obsahuje v rámci elementu `<config>` následující sekce.

- `<path>` - zahrnuje absolutní cesty k informačnímu systému. Součástí jsou prvky `<pom>` pro definici souboru `pom.xml`, `<input>` / `<output>` k určení modulů DOM / Integtests a `<target>` vymezující adresář cílových tříd buildu.
- `<package>` - formuluje názvy jmenných prostorů `<input>` / `<output>` u modulů Dom / Integtests.
- `<tests>` - obsahuje seznam aktuálně používaných testovacích sad ve formátu `<set> jmenný prostor generátoru</set>`.
- `<dependencies>` - volitelná položka, slouží k případnému doplnění dalších závislostí, které není možné z nějakého důvodu zapsat v souboru `pom.xml` projektu informačního systému.

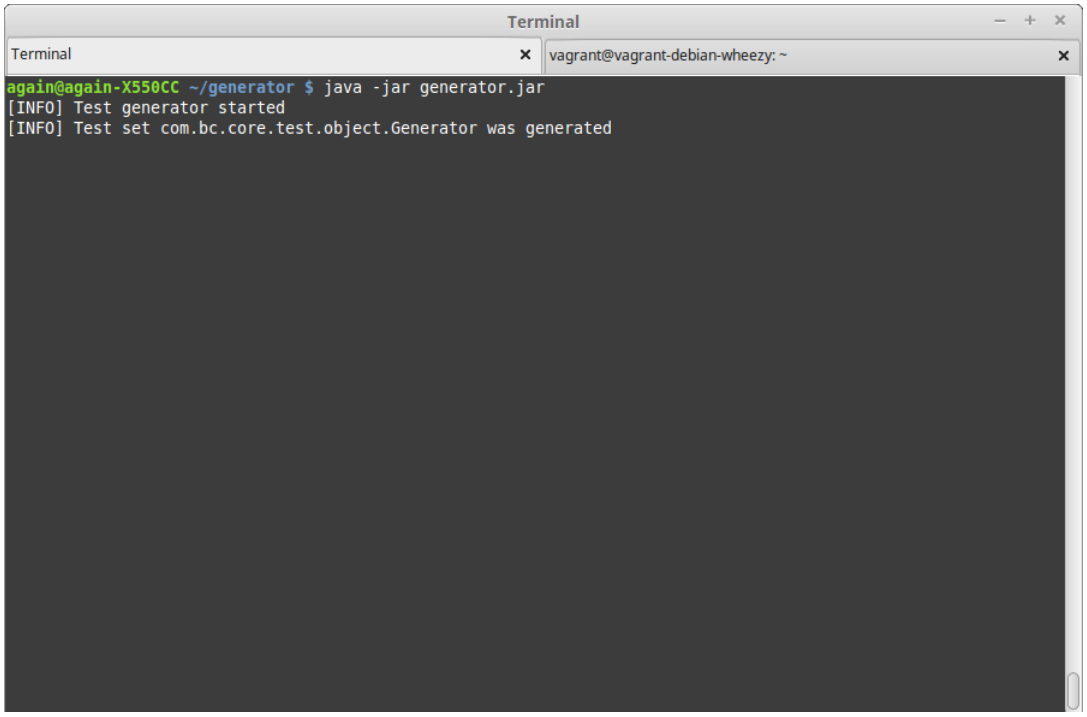
5.2 Manuální spouštění generátoru

Po konfiguraci aplikace lze přistoupit k jejímu spuštění. To může být buď manuální nebo automatické v závislosti na zvoleném přístupu. V případě prvního způsobu se k tomuto účelu využívá třída `Runner`, která představuje vstupní bod do aplikace. Musí však být splněna podmínka správně zadané cesty k výslednému buildu informačního systému. Ten je tak nutné provést ještě před vygenerováním automatických testů.

Ruční spouštění aplikace se provádí z konzole tak, jak je znázorněno na obrázku 5.1 pomocí následujícího příkazu.

```
$ java -jar generator
```

Poté dochází ke generování automatických testů na základě definovaných sad v konfiguračním souboru. U daného systému půjčovny aut se jedná o scénáře umístěné ve jmenném prostoru `domainapp.integtests.specs.modules` a také testovací třídy spadající do `domainapp.integtests.specglue.modules` a `domainapp.integtests.specglue.modules`.



Obrázek 5.1: Spuštění generátoru z příkazového řádku.

Konkrétní zařazení určují cesty zdrojových tříd. Pokud existují nějaké ruční úpravy těchto souborů a jsou označené příslušnou anotací, budou tyto změny automaticky přeneseny.

Dalším typickým požadavkem po úspěšném vytvoření testů je jejich provedení. To zajišťuje plugin Mavenu `maven-surefire-plugin`, který zavádí životní cyklus testování společně s pluginem Apache ISIS `isis-maven-plugin` pro zařazení integračních testů a případných dalších jejich typů. K vykonání testů je nutné přemístění do odpovídající složky s projektem informačního systému a zadání uvedeného příkazu.

```
$ mvn test
```

Ten spustí postupně veškeré nalezené testy a jejich případné neúspěchy oznámí příslušným chybovým hlášením na obrazovku příkazové řádky.

5.3 Integrace generátoru

Manuální spouštění aplikace generátoru testů je sice funkční, ale nepříliš pohodlné řešení. Výhodnější by proto bylo, kdyby se tato činnost prováděla automaticky v rámci životního cyklu. Z tohoto důvodu se zavádí integrace této aplikace do prostředí informačního systému. Nejjednodušším způsobem se pak stává implementace speciálního testovacího případu za podpory knihovny JUnit. Ten slouží čistě ke konfiguračním účelům a jeho umístění spadá do modulu doménového modelu.

V první řadě je zapotřebí načíst generátor do zmíněného modulu. Tuto akci lze provést opět díky souboru `pom.xml` obvyklou deklaraci nové závislosti.

```
<dependency>
    <groupId>com.bc</groupId>
    <artifactId>generator</artifactId>
    <version>1.0</version>
</dependency>
```

Poté již stačí vytvořit novou třídu s názvem povinně končicím na Test, a to kvůli požadavku na automatické nalezení a zpracování. Její jmenný prostor je možné pojmenovat například `setup`. V této konkrétní situaci se tak jedná o soubor `setup/genTest.java`. Třída `genTest` zahrnuje metodu `setUp()` označenou anotací `@Before`, která udává vykonání metody při inicializaci testu.

Její obsah tvoří objekt `Runner` znázorňující vstupní bod do aplikace generující automatické testy a volání metody `run()` pro zahájení této akce. V praxi se může tento zdrojový kód podobat níže uvedenému příkladu.

```
@Before
public void setUp() {
    try {
        (new Runner()).run();
    } catch (Exception e) {
        System.out.println("[WARNING]
            Test generator was not correctly loaded");
        System.err.println("[WARNING] Exception: "
            + e.getMessage());
    }
}
```

Jelikož při spuštění testovacích tříd dochází k invokaci metod s anotací `@Test`, musí být rovněž definována alespoň jedna z nich. V tomto případě však stačí ponechat její tělo prázdné. Jako ukázka integrace generátoru slouží demonstrační informační systém půjčovny aut. Po vykonání příkazu `mvn test` jsou nejprve vytvořeny jednotlivé testovací sady a následně dochází k jejich spuštění během provádění integračních testů.

Další způsob, jakým lze tuto integraci realizovat, spočívá v přidání speciální třídy do životního cyklu Maven pluginu `maven-surefire-plugin`. Jelikož se však informační systém skládá z více modulů, jednalo by se o obtížnější řešení. Původní rovněž zamyšlena alternativa počítala s využitím třídy `RunSpecs` pro spouštění nástroje Cucumber. Ta však bohužel neumožňuje učinit příslušné akce ještě před samotnými testy.

5.4 Předpřipravené testovací sady

V rámci generátoru automatických testů již existují tři testovací sady, které pokrývají základní funkčnost napříč běžnými aplikacemi. Pokud však v některém ze svých aspektů nevyhovují, je možné provést jednoduše jejich úpravu.

5.4.1 Testovací sada pro ověřování objektů

První z předpřipravených testovacích sad slouží k ověření správnosti vytvářených objektů na základě zadaných hodnot ve scénáři. Pokud během tohoto procesu dojde k nějakému chybovému hlášení, test není úspěšný. Na konci rovněž dochází ke kontrole počtu instancí

nových objektů. Níže uvedený příklad zachycuje scénář u konkrétního doménového objektu `Vehicle`.

```
Feature: List and Create new Vehicle Objects
    @integration
        Scenario: Existing Vehicle objects can be listed and
        new ones created
        Given I have 0 Vehicle objects
        When I create a Vehicle object with
        type "Skoda 120" and price "10000"
        When I create a Vehicle object with
        type "Skoda Fabia" and price "50000"
        Then I have 2 Vehicle objects
```

Názvy argumentů a výchozích hodnot vkládaných do konstruktoru při tvorbě nové instance doménové třídy, jsou standardně generovány náhodně. U hodnot pak závisí tento řetězec na svém datovém typu. Alternativu pro tuto techniku představuje uvedení anotací u parametrů metod informačního systému, a to ve znázorněném formátu.

```
@TestParam(name="název", def={"pole počátečních hodnot"})
```

Pokud dotyčný objekt obsahuje parametr potomka typu `Object`, potom se ve scénáři neuvádí. Místo toho je uplatněn princip popsaný v kapitole 4.4. Ve stručnosti řečeno se vyhledá instance patřičné třídy a jestliže není nalezena, aplikace zavolá metodu `execute()` příslušného skriptu `{NazevObjektu}Create`. Tyto třídy musí být uveřejněné ve jmenném prostoru `projekt.dom.generator.fixture` modulu Dom.

5.4.2 Testovací sada pro ověřování metod `validate()`

Apache ISIS zahrnuje několik vyhrazených metod pro různé akce s doménovými objekty. Ty poskytují pravidla ke kontrole uživatelských akcí při interakci se systémem. Jedna z těchto metod disponuje prefixem `validateXXX()` a umožňuje validaci atribut třídy (např. u políček formuláře). Vygenerované scénáře pak ověřují jejich správnou funkčnost.

```
Feature: Validate the price of the Vehicle
    @integration
        Scenario: Validate the price and receive error message
        Given I have the value "-500" for a price
        When I validate the price of the Vehicle
        Then I receive the error message
```

Na rozdíl od předešlé testovací sady, jsou nyní anotace uváděné v informačním systému povinné. Deklarují se vždy nad daným atributem třídy, ke kterému se váže příslušná metoda `validate()`. Tento princip se uplatňuje zejména z důvodu, že nelze jednoduchým způsobem odhadnout hodnoty ve scénářích. Proto pokud není anotace uvedena, generátor tuto metodu mezi výsledné testy nezahrne.

```
@TestVar(name="název", def={"pole počátečních hodnot"})
```

5.4.3 Testovací sada pro ověřování metod `title()`

Každý objekt je v uživatelském rozhraní reprezentován za pomocí titulku (angl. *title*. Ten se využívá v hlavičce samotné webové stránky a u hypertextových odkazů vedoucích právě na tento objekt. Hlavní cíl testovací sady tedy spočívá v ověření metody `title()`, která slouží k vytvoření požadovaného řetězce. Vygenerovaný scénář má v tomto případě poměrně prostý tvar.

```
Feature: Set the title of the Vehicle
    @integration
        Scenario: Set the title and receive string
        Given I have the object Vehicle
        When I set the title of the Vehicle
        Then I receive the string title
```

Aby bylo možné zjistit u posledního kroku scénáře finální titulek, musí být jeho podoba zapsána v anotaci před metodou `title()`.

```
@TestMethod(name="nazev", def={"pole počátečních hodnot"})
```

5.5 Tvorba vlastní testovací sady

Jednu z výhod generátoru automatických testů představuje robustní architektura, díky niž je velice jednoduché vytvořit si vlastní testovací sadu. Ta se v základním uspořádaní skládá ze čtyř pilířů, které ve většině případů reprezentují stejnojmenné třídy.

ScenarioFactory

Pro generování scénářů se používá továrnička `ScenarioFactory`, jejíž součástí je metoda `createScenario()` s návratovým typem `ScenarioTemplate`. Samotné zhotovení scénáře zajišťuje třída `ScenarioBuilder` vystavěná na architektonickém vzoru Stavitel, konkrétně její metoda `build()`. Ta vyžaduje argument typu `GeneratorTable` k překladu parametrů v šabloně. Jejich převedení na metody spočívá v postupném odstraňování oddělovače (.) a úpravách do požadovaného formátu.

```
scenarioBuilder scenario = new ScenarioBuilder("listAllAndCreate");

scenario.setFeature("List and Create new {class.name} Objects");
scenario.setScenario("Existing {class.name} objects can be listed
                      and new ones created");

scenario.addGiven("I have 0 {class.name} objects");
scenario.addWhen("I create a {class.name}
                  object {class.construct.params}")
        .addWhen("I create a {class.name}
                  object {class.construct.params}");
scenario.addThen("I have 2 {class.name} objects");

scenario.build(this.generator); // vysledna sablona scenare
```

Uvedený příklad znázorňuje formu první předpřipravené testovací sady. Výhoda takového řešení je mimo jiné i ve znovupoužitelnosti u ostatních případů.

ScenarioGenerator

Kromě šablony scénáře nastává nutnost stanovit podmínky vymezující užité třídy či jejich metody v informačním systému. K tomuto účelu se provádí implementace rozhraní `IscenarioGenerator`. Obsah hlavní metody `generateScenarios()` pak tvoří cyklus zahrnující libovolnou logiku společně se zápisem do konkrétního souboru za pomocí třídy `ScenarioWriter`.

TestSetGenerator

Mezi další část generátoru, kterou nelze opomenout, se řadí testovací třídy. Jejich náplň pojímá metody, které představují jednotlivé kroky ve scénáři. Ty lze generovat v cyklu prostřednictvím metody `generateTestSets()` třídy `TestSetGenerátor`. Veškerá programová logika je pak popořadě přidána díky třídě `TestSet`.

Generator

Poslední důležitý díl testovací sady reprezentuje třída `Generator`. Její hlavní funkce tkví v agregaci předchozích částí skrze metodu `generate()`. Ta umožňuje vygenerování příslušných scénářů a také testovacích tříd.

Vyjma těchto čtyř skupin umožňuje architektura systému libovolné rozšíření některé z bázových tříd generátoru. V praxi se to týká zejména analyzátoru nebo třídy `ParserTable`.

Kapitola 6

Závěr

Předmětem této práce bylo prozkoumat možnosti automatizace testování software s primárním zaměřením na techniku Domain-driven design (DDD). V rámci teoretické části textu je proveden komplexní náhled do této problematiky společně s porovnáním dílčích metodik. Konkrétně se jedná o přístupy Test-driven development a Behaviour-driven development. Na základě posledního z nich byl následně vytvořen generátor automatických testů zveřejněný na adrese `https://github.com/tomPolesovsky/test-generator` podléhající licenci MIT.

Tato aplikace umožňuje generování jednotlivých scénářů a testovacích tříd díky analýze doménového modelu informačního systému. V tomto případě se jednalo o demonstrační projekt půjčovny aut. Ten ke své činnosti využívá framework Apache ISIS, který slouží pro rychlý vývoj aplikací s orientací právě na DDD. Součástí generátoru automatických testů jsou také tři předpřipravené testovací sady pokrývající běžnou funkčnost systému. Provedení jejich libovolných úprav nebo tvorba nové sady je rovněž velice jednoduchá.

V průběhu implementace generátoru nastalo několik komplikací. Zásadní rozhodnutí spočívalo ve způsobu, jakým se bude provádět analýza doménového modelu. Nakonec v tomto směru zvítězila technika reflexe užívaná ke zkoumání chování aplikace při jejím běhu. Tato forma řešení však s sebou i přesto nese určité problémy, které byly z velké části úspěšně překonány. Další nesnáze se pak naskytly zejména u závislosti objektů ve vnitřní struktuře systému. Celkový výsledek je nicméně dostatečně kvalitní a použitelný v praktickém nasazení.

Co se týče dalšího rozšíření aplikace, největší prospěch by přinesla zřejmě hlubší analýza informačního systému. Její součástí může být také rozpoznání kontextu u dílčích objektů. Přínos takového řešení spočívá ve snazším doplňování argumentů při vytváření nových instancí tříd. Pro snadnější konfiguraci projektu je pak vhodné rozšířit zpracování příslušného souboru i o pluginy nástroje Maven. Největší výzvu však představuje zobecnění těchto principů i pro ostatní frameworky či programovací jazyky.

Literatura

[1] Beck, K.: *Test Driven Development: By Example*. Addison Wesley Professional, 2003, ISBN 0321146530.

[2] Beck, K.: *Programování řízené testy*. Praha: Grada, 2004, ISBN 80-247-0901-5.

[3] Bienvenido, D.: *Apache Isis: Java Framework for Domain-Driven Design*. 2013, [Online; navštíveno 28.04.2017].
URL https://www.infoq.com/news/2013/01/apache-isis-java-domain-driven

[4] Hlava, T.: *Fáze a úrovně provádění testů*. 2011, [Online; navštíveno 23.04.2017].
URL http://testovanisoftwaru.cz/metodika-testovani/druhy-typy-a-kategorie-testu/faze-testu/

[5] Hlava, T.: *Statické a dynamické testy*. 2011, [Online; navštíveno 23.04.2017].
URL http://testovanisoftwaru.cz/metodika-testovani/druhy-typy-a-kategorie-testu/staticke-a-dynamicke-testy/

[6] Ing. Michal Pavlík, P.: *Důležitost testování*. [Online; navštíveno 23.04.2017].
URL http://www.umel.feec.vutbr.cz/bdts/index.php/diagnosticke-testy/dulezitost-testovani

[7] Kitner, I. R.: *3 časté chyby při automatizaci testů*. 2017, [Online; navštíveno 23.04.2017].
URL http://kitner.cz/blog/jak-na-automatizovane-testovani-softwaru/

[8] Millett, S.; Tune, N.: *Patterns, Principles, and Practices of Domain-Driven Design*. Wrox Press, 2015, ISBN 1118714709.

[9] Naik, H.: *Behavior Driven Development: An Effective Technical Practice to Develop Good Software*. International Journal of Computer Applications 149(5):23-27, Září 2016, [Online; navštíveno 23.04.2017].
URL
http://www.ijcaonline.org/archives/volume149/number5/25993-2016911400

[10] North, D.: *Introducing BDD*. 2006, [Online; navštíveno 23.04.2017].
URL https://dannorth.net/introducing-bdd

[11] Parlog, N.: *Understanding Java's Reflection API in Five Minutes*. 2017, [Online; navštíveno 28.04.2017].
URL https://www.sitepoint.com/java-reflection-api-tutorial/

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

[13] Penchikala, S.: *Domain Driven Design and Development In Practice*. [Online; navštíveno 23.04.2017].
URL https://www.infoq.com/articles/ddd-in-practice

[14] Perforce: *Test-Driven Development*. 2013, [Online; navštíveno 23.04.2017].
URL https://www.perforce.com/blog/130315/test-driven-development

[15] Pánek, R. D.: *Behavior Driven Development (zápisky)*. 2015, [Online; navštíveno 23.04.2017].
URL https://medium.com/@RDPanek/behavior-driven-development-z%C3%A1pisky-6214ffa5c234

[16] Rice, B.; Jones, R.; Engel, J.: *Behavior Driven Development*. [Online; navštíveno 23.04.2017].
URL https://pythonhosted.org/behave/philosophy.html

[17] Santos, E.; Beder, D.; Penteado, R.: *A Study of Test Techniques for Integration with Domain Driven Design*. Information Technology - New Generations (ITNG), 2015 12th International Conference on Information Technology - New Generations, 2015, [Online; navštíveno 28.04.2017].
URL http://ieeexplore.ieee.org/document/7113501/

[18] Smrčka, A.: *Testování a dynamická analýza - Úvod a pojmy v testování*. FIT VUT v Brně, 2016, [Online; navštíveno 23.04.2017].
URL https://wis.fit.vutbr.cz/FIT/st/course-files-st.php.cs?file=%2Fcourse%2FITS-IT%2Flectures%2F1-uvod.pdf

[19] Terz, M.: *Domain Driven Design*. [Online; navštíveno 23.04.2017].
URL http://php.websites.cz/cms/domain-driven-design

[20] The Apache Software Foundation: *Apache ISIS User Guide*. [Online; navštíveno 28.04.2017].
URL https://isis.apache.org/guides/ugfun/ugfun.html#3.2.7.-app-structure

[21] de la Torre, C.: *Domain Driven Design (DDD) & Visual Studio 11 Beta ALM, great fit!* 2012, [Online; navštíveno 23.04.2017].
URL https://blogs.msdn.microsoft.com/cesardelatorre/2012/04/06/domain-driven-design-ddd-visual-studio-11-beta-alm-great-fit/

[22] VersionOne: *Test-First Programming*. [Online; navštíveno 23.04.2017].
URL https://www.versionone.com/agile-101/agile-software-programming-best-practices/test-first-programming/

[23] Ševčík, D.: *Domain-Driven Design*. Bakalářská práce, FI MU, 2009, [Online; navštíveno 23.04.2017].
URL https://is.muni.cz/th/173376/fi_b

[24] Šmerda, B. J.: *Zdokonalení procesu automatického testování softwarových aplikací*. Diplomová práce, FIT VUT v Brně, 2014, [Online; navštíveno 23.04.2017].
URL https://www.vutbr.cz/www_base/zav_prace_soubor_verejne.php?file_id=119332

Přílohy

Příloha A

Obsah CD

- Informační systém půjčovny aut
- Generátor automatických testů
- Návod k obsluze – soubor README
- Bakalářská práce ve formátu .pdf a soubory .tex