ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název: Testování software v agilních vývojových metodikách
Student: Yulia Boronenko
Vedoucí: Ing. Martin Ledvinka
Studijní program: Informatika
Studijní obor: Webové a softwarové inženýrství
Katedra: Katedra softwarového inženýrství
Platnost zadání: Do konce letního semestru 2019/20

Pokyny pro vypracování

1. Proveďte podrobnou rešerši technik testování software. Zaměřte se především na techniky využívající automatizaci.
2. Diskutujte vhodnost popsaných technik v podmínkách agilního vývoje webových aplikací.
3. Zvolte vhodnou kombinaci technik a vytvořte testovací plán pro vybranou webovou aplikaci vyvíjenou pomocí metodiky SCRUM.
4. S použitím vámi zvolených technik vytvořte testy pro ukázkovou aplikaci.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 30. ledna 2019

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Testování software v agilních vývojových metodikách

Yulia Boronenko

Katedra softwarového inženýrství
Vedoucí práce: Ing. Martin Ledvinka

16. května 2019

Poděkování

Chtěla bych poděkovat vedoucímu své práci Ing. Martinu Ledvinkovi za cenné rady a pomoc při tvorbě daného dokumentu, Marku Kodrovi, bez jehož morální podpory by se tento dokument nikdy nezrodil nebo nedokončil, Báře Spudilové za trpělivost při provedení gramatické korektury a svým milovaným rodičům za nekonečnou podporu a poskytnutou možnost studovat vysokou školu.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracovala samostatně a že jsem uvedla veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 16. května 2019 …………………

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2019 Yulia Boronenko. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.

Odkaz na tuto práci

Boronenko, Yulia. *Testování software v agilních vývojových metodikách.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2019.

Abstrakt

Táto práce se věnuje detailnímu souhrnu odborných informací na téma testování softwaru se zaměřením na webové aplikace a může sloužit jako návod k zaučení na pozici Junior testera v libovolné české firmě propagující agilní metodiky vývoje softwaru. Klíčovou roli v práci představuje podrobný výzkum vhodného použití různých technik testování v přislušných modelech s cílovým zaměřením na model agilní. Díky čemuž bude mít čtenář představu, jak dané rady aplikovat na své vlastní webové projekty. Dále jsou v práci uvedené příklady automatizace vytvoření různých druhů testů pomocí víceúčelového testovacího frameworku pro PHP Codeception a propojení daného frameworku se Zephyrem pro trekovácí systém Jira.

Klíčová slova agilní testovácí metodiky, testování webových aplikací, testovácí techniky, automatizace testování, SCRUM, Codeception, Zephyr

Abstract

This work is devoted to a detailed summary of professional information about software testing with a focus on web applications and can be used as a tutorial for teaching a junior tester in any Czech firm, which is promoting agile software

development methodologies. Main role of the work is detailed research on the appropriate use of different testing techniques in relevant modalities (mostly agile description), giving the reader an idea of how to apply the advice to their own web projects. Also the work contains examples of automation of creating different types of tests using the multi-featured testing framework for PHP Codeception and connect it with Zephyr for the Jira tracking system.

Keywords agile testing methods, web aplication testing, testing techniques, testing automation, SCRUM, Codeception, Zephyr

Obsah

1 Úvod **1**
- 1.1 Cíl práce . . . 1

2 Úvod do testování **3**
- 2.1 Pohledy na testování softwaru . . . 3
- 2.2 Účel testování . . . 4
- 2.3 Kapesní kniha testerů . . . 5
  - 2.3.1 Slovník pojmů . . . 5
  - 2.3.2 Úrovně testování . . . 8
- 2.4 Testerská role . . . 9
  - 2.4.1 Hierarchie . . . 9
  - 2.4.2 Praxe . . . 10

3 Techniky a rozdělení testů **15**
- 3.1 Statické techniky . . . 15
  - 3.1.1 Neformální recenze . . . 16
  - 3.1.2 Návody . . . 17
  - 3.1.3 Technická kontrola . . . 17
  - 3.1.4 Statická analýza . . . 17
- 3.2 Dynamické techniky . . . 18
  - 3.2.1 White box testování . . . 18
    - 3.2.1.1 Testování řídících struktur (control flow testing) 18
    - 3.2.1.2 Testování datového toku (Data flow testing) . 19
  - 3.2.2 Black box testování . . . 19
    - 3.2.2.1 Rozdělení na třídy ekvivalence (Equivalence Partitioning) . . . 20
    - 3.2.2.2 Analýza hraničních hodnot . . . 20
    - 3.2.2.3 Testování přechodů stavů (state transition) . . 21
    - 3.2.2.4 Rozhodovací tabulka (decision table) . . . 23

3.2.2.5 Testování případů užití (use case testing) . . . 24
3.2.3 Testování na základě zkušeností 25
3.2.3.1 Exploratory testing 25
3.2.3.2 Předpoklady existence softwarové chyby 27

4 Modely softwarového procesu a metodiky vývoje 29
4.1 Vodopádový model 29
4.2 Spirálový model 31
4.3 Iterativní model 32
4.4 Agilní model 34
4.4.1 SCRUM 36
4.4.2 Extrémní programování (XP) 39
4.4.3 Kanban a Lean 40
4.4.4 Dynamic Systems Development Method (DSDM) 41
4.4.5 Feature-Driven Development (FDD) 42

5 Testovaní webových aplikací v agilních metodikách a automatizace procesů 45
5.1 Aplikace testovacích technik v agilních metodikách vývoje . . . 46
5.1.1 Programování řízené testy (TDD) 47
5.1.2 Programování řízené akceptačními testy (ATDD) 47
5.1.3 Vývoj řízený požadavky na chování (BDD) 48
5.1.4 End-to-end testování (E2E) 49
5.1.5 Exploratory testování 49
5.2 Automatizace testovacích procesů 49

6 Kombinace testovacích technik na projektu vyvinutém ve SCRUMu 53
6.1 Seznámení s projektem 53
6.1.1 Funkční požadavky 54
6.1.2 Nefunkční požadavky 55
6.2 Vytváření testovacího plánu regresních testů webové aplikace . 56
6.3 Automatizace regresních testů 57
6.3.1 Využití Codeception 59
6.3.2 Propojení testů s testovacím nástrojem Zephyr pomocí ZAPI 63

Závěr 65

Literatura 67

A Slovník pojmů 71

B Seznam použitých zkratek 73

C Obsah přiloženého CD **75**

Seznam obrázků

2.1 Cena opravy defektů 5
2.2 Struktura testovácího plánu 8

3.1 Testovací techniky 16
3.2 Stavový diagram objednávky 23

4.1 Vodopádový model vývoje softwaru 30
4.2 Iterativní model vývoje softwaru 33
4.3 Procesy ve SCRUMu 37

5.1 Kvadrant agilního testování [14] 50
5.2 Pyramida automatizace testů 51

Seznam tabulek

3.1 Třídy ekvivalence – datum narození cestujícího 20
3.2 Tabulka přechodů stavů objednávky 22
3.3 Rozhodovací tabulka – provedení online objednávky 24
3.4 Případ užití – scénář provedení objednávky zájezdu 26
3.5 Případ užití – metadata objednávky zájezdu 27

5.1 Použití „Given-When-Then“ šablony pro psaní testovacích případů. 48

6.1 Tabulka jazykových verzí základního průchodu aplikace a specifikace počtu cestujících. 57
6.2 Případ užití – scénář provedení objednávky hotelu 58
6.3 Případ užití – metadata objednávky hotelu 59

Kapitola 1

Úvod

Testování softwaru je poměrně mladým jevem v moderním světě, který navzdory své krátké historii prošel výrazně rychlým a rozmanitým průběhem změn, reorganizací a přestavbami svých základních stavebních kamenů, aby se k nám dostal v té podobě, v níž ji zná každý člověk pohybující se v IT. Nahlédneme do dějin testování a seznámíme se s nimi blíž. Začínající jako synonymum ladění aplikací, pojem testování neměl žádnou vnitřní strukturu, dokud se na něj v roce 1957 nezačali nahlížet jako na činnost, která by měla prokázat, že software splňuje definované požadavky. Období mezi rokem 1979 a 1982 je obdobím, kdy testování získalo nový smysl, konkretně – záměrné hledání chyb, díky čemuž bylo později asociováno s jinými činnostmi detekce různorodých defektů [12]. S příchodem nové doby zaměřené na vyhodnocení softwaru přichází měření kvality produktů. Od roku 1970 už nikdo nepovažuje testování za laickou činnost, zvedá se úroveň profesionalismu v daném oboru, vznikají první národní standardy, a čím dál více se publikují odborné články na dané téma. Od roku 1988 je testování považované za kombinaci tří odlišných činností: kontrolu specifikací, detekování a předcházení chyb. [12] Cesta vývoje procesu testování byla komplikovaná z hlediska toho, že měla reflektovat potřeby doby velkých inovací a celosvětové modernizace. Vznik nových technik a metodik kontroly kvality, zrod různotvárných modelů vývoje SW, pokusy o automatizaci – všechno je výsledkem dlouhodobého vývoje softwarových procesů a přímo či nepřímo souvisí s testováním. Proto je softwarové inženýrství oborem, jež si nelze bez testování představit.

1.1 Cíl práce

Účelem této práce je provedení rešerše různorodých technik testování softwaru se zaměřením na webové aplikace v agilních metodikách, kde se větší důraz klade na využití automatizaci. Zároveň tento odborný dokument může sloužit jako krátký výukový materiál pro seznámení začínajícího testera s proble-

matikou testování ze všeobecného pohledu. Práce nejenom popisuje základní pojmy, ale stejně jako každý návod, krok po kroku provede čtenáře většinou nejpoužívanějších modelů softwarového procesu, odkáže na důvody jejich vzniku a testerskou roli v každém z nich. Velká část práce se věnuje agilním metodikám vzhledem k jejích aktuálnímu využití a popularitě ve většině firem, v rámci čehož také dojde k porovnání a nalezení kombinací nejvhodnějších již zmíněných technik. Kapitola popisující agilní přístup k problematice automatizace testování zaujme zkušenější testery, kteří mají znalosti procedurálního a objektově orientovaného programování. Výstupem implementační části bakalářské práce je testovací plán anonymizované webové aplikace existující cestovní agentury, jejíž celý vývoj byl postaven na metodice SCRUM. Pro dosažení definovaného cíle se použily UI a unit testy realizované v Codeception frameworku pro PHP, propojené se Zephyrem (nástrojem pro správu testů) pro Jiru pomocí ZAPI. Výsledný testovací plán zahrnuje spojení několika předem popsaných technik, což může posloužit jako pevný základ pro čtenáře, jejich účelem je nalezení optimální cesty pro sestavení testovacích plánů pro webové aplikace a automatizace většiny procesů.

Kapitola 2

Úvod do testování

„A project is like a road trip. Some projects are simple and routine, like driving to the store in broad daylight. But most projects worth doing are more like driving a truck off-road in the mountains, at night.“ [2]

2.1 Pohledy na testování softwaru

Testování softwaru. Jedná se o všeobecně známý pojem, se kterým se čtenář potýká každý den svého života, přestože si to dost často ani neuvědomuje. Po několika hodinách brouzdání na internetu, čtení odborných článků či knih, zjistíme, že testování je koncept mnohotvárný a může se kompletně lišit v závislosti na použitých metodikách a přístupech při vývoji webových, mobilních nebo desktopových aplikací. Každopádně po delší chvilce přijdeme na to, že i v tom velkém množství rozdílů dokážeme mluvit o testování softwaru jako o jevu obecném, který má mnoho společných rysů mezi všemi svými odvětvími.

Testování softwaru je „provozování systému nebo komponenty za specifických podmínek, pozorování nebo zaznamenávání výsledků a vyhodnocování některých aspektů systému nebo komponenty.“[14] Občas narazíme na diskutabilní názor, že testování produktů je proces hledání chyb nebo defektů programu. Je a zároveň není, vzhledem k tomu, že popsané aktivity jsou jen nepatrným množstvím povinností testera. Ale zásadně bychom měli vyloučit kritické názory menší skupiny vývojářů, kteří se vztahují k testování jako k procesu, jehož účelem je poukázat na jejích vlastní neschopnost psát program bez vad. Greg Fournier na začatku své knihy „Essential Software Testing“ odkazuje na deprimující postoje kolegů vůči jeho pracovní pozici testera, které pak shrnuje do několika důležitých bodů [4]:

- Testeři jsou rigidní.

- Testeři se víc soustředí na selhání softwaru, než na jeho funkčnost jako celku.
- Testeři čekají do poslední minuty, aby objevili problémy způsobující odložení projektů, a aby projektový tým nevypadal dobře.

Taková stanoviska jsou nerelevantní z hlediska profesionality a správného řízení vztahů v týmech, ale i přesto jsou bohužel dosti častými jevy v IT.

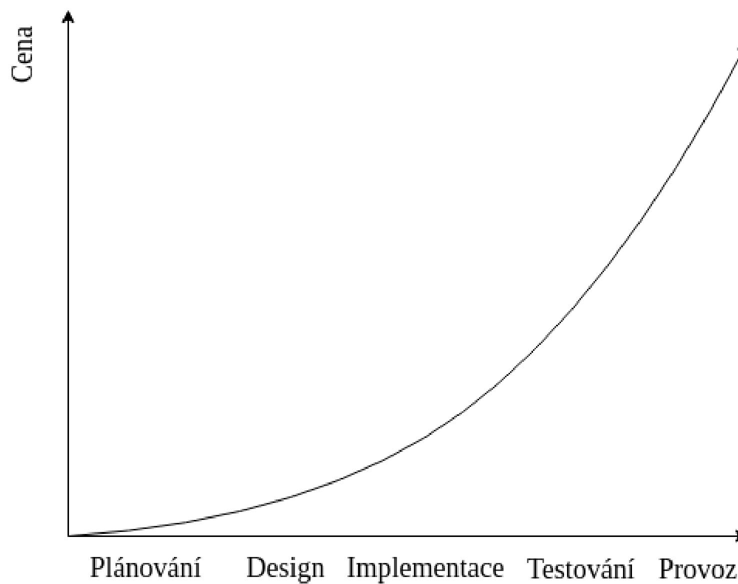
2.2 Účel testování

Existence procesu testování je založená především na tom, že jsou aplikace vyvíjeny obyčejnými lidmi, ne bezchybnými a dokonalými bytostmi či roboty. Účelem testování není záměrné hledání vad, chyb či odchylek, je to proces a společná snaha celého vývojového týmu předložit klientovi kvalitní, pomocí metody FURPS[1] ověřený software, odpovídající představám zainteresovaných osob (stakeholders).

Shrneme a zapíšeme důvody testování do několika bodů:

- Ověřit, zda výsledný software funguje podle návrhu, splňuje nefunkční požadavky a dosahuje akceptovatelných výsledků.
- Zjistit, zda se software neodchyluje od návrhu, a odhalit chyby dokumentace.
- Zkontrolovat, zda aplikace nabízí funkce, které po ní vyžadují uživatelé.
- Zajistit efektivní výkon softwarové aplikace nebo produktu.[20]
- Náklady spojené s opravou neotestovaného softwaru mohou být kolosální (viz obr. 2.1). Řádné testování zajišťuje, že chyby a problémy jsou detekovány na počátku životního cyklu produktu nebo aplikace. [20]
- Testování je jedním z mnoha měřítek kvality softwaru.
- Následky neprovedeného nebo nedostatečného testování mohou být kritické. V závislosti na druhu vyráběného softwaru se může jednat buď „jenom“ o zmeškaný let nebo o automobilovou katastrofu, jenž si vyžádá několik lidských životů. Rizika jsou nepředvídatelná. Jako příklad uvedeme případ nekvalitně otestovaného nástroje radioterapie Therac25, jehož dávky radiace byly několikanásobně větší, než bylo nastaveno programem. Výsledkem je několik desítek smrtí.

[1] Jedna z metod ověření kvality softwaru. Její hlavní měřítka jsou: funkčnost, vhodnost k použití, spolehlivost, výkonnost a schopnost podpory.



Obrázek 2.1: Cena opravy defektů

2.3 Kapesní kniha testerů

Záměrem této podkapitoly je uvedení čtenáře do problematiky testování ze všeobecného pohledu, shrnutí nejdůležitějších pojmů (testovací základny), se kterými se každý tester bez výjimky denně potýká.

2.3.1 Slovník pojmů

Softwarový proces/životní cyklus softwaru (Software Development Lifecycle neboli SDLC) – sekvence fází vývoje produktu. Takové rozdělení na stadia má za důsledek zlepšení kvality softwaru. Rozlišuje se několik následujících fází životního cyklu:

- plánování (sběr požadavků a jejich důsledná analýza),
- design (zpracování požadavků a tvorba návrhu),
- implementace (realizace stanoveného návrhu),
- testování,
- provoz (nasazení projektu do produkce, částečná údržba),
- údržba a rozvoj produktu (ve většině případů se jedná o déle trvající proces, než v případě předchozí fáze životního cyklu).[2]

[2] V některých zdrojích je popsaná fáze neoddělitelnou částí etapy provozu.

Testovací data (test data) – souhrn dat, který byl vytvořen za účelem použití v konkretním testu. V případě automatizací bychom se měli snažit generovat kvalitní data pro usnadnění testovacích procesů a snížení jejích ceny.

Testovací případ (test case) je sekvencí kroků, generovaných testovacích dat, podmínek provedení a očekávaných výsledků, které by měly být splněny a dosaženy během testování produktu. Na základě popsané množiny se dá usoudit, zda software funguje korektně. [15]

Testovací skript (test script) – dokument, jež představuje logický, neoddělitelný celek, zahrnující do sebe skupinu testovácích případů. Zároveň tvoří množinu kroků, jejíž splňení prokazuje správnost testované funkcionality vůči daným vstupům. Každý krok navazuje na výstup předešlého, a celý testovácí script je obrovskou propojenou strukturou.

Testovací scénář (test scenario) – množina testovacích skriptů, řídící celý testovací proces. Tento dokument popisuje postup provedení různých testů, a to především na jejich rozsah, pokrytí, primární funkcionality pro otestování a managerské záležitosti, např. rozdělení testovácích úloh v závislosti na rolích členů týmu, použití testovácích prostředí apod. Vystupuje jako nejdůležitější souhrn testovacích instrukcí v rámci jednoho projektu.

Test suite (validation suite) je kolekcí testovacích případů, která funguje jako menší kontejner. Laicky řečeno, test suite plní funkci logicky rozdělené seskupovácí složky několika testovacích případů. Daná struktura je podmnožinou testovacího plánu[3]. (viz obr. 2.2)

Testovací souhrnný report (test summary report) je dokument, který sjednocuje výsledky testování. Jedná se o plnohodnotné hlášení zpřehledňující provedenou testerskou práci. Účelem podobného dokumentu není jenom předložení podrobného výpisu provedených testů managerům či klientům, ale zároveň zdokumentování aktuálního stavu softwaru.

Testovací plán – dokument popisující celkový proces testování, včetně vybrané testovací strategie, objektů, kriterií hodnocení, specifických znalostí a pravděpodobných rizik. [21]

Pojem **koncový uživatel** (end user) se používá k rozlišení skupiny následných uživatelů od testerů, vývojářů, managerů a klientů – lidí, kteří se potykají s aplikací v rámci jejího vývoje.

[3] V praxi se můžeme potkat s názory, že neexistuji rozdíly mezi testovacím skriptem a testovacím případem vzhledem k jejich obrovskému množství shodných vlastností, jako je například vzájemné propojení kroků. Proto se často první pojem používá při automatizaci procesů a druhý v případě manuálního testování.

Způsoby realizace testovacího procesu se kardinálně liší. Ve světě softwarových technologií se vyznačují 3 druhy:

- **Manuální testy** – ručně se provádí člověkem na základě předem popsaného případu/skriptu/scénaře/plánu.

- **Automatizované testy** – jeden ze způsobů ulehčení a zefektivnění práce testera pomocí exekuce softwaru jiným softwarem. Proces automatizace přináší spousty výhod i nevýhod, většina kterých bude popsána v dalších kapitolách.

- **Exploratory testy** – patří do testovacích technik, ale v seznamu se objevily kvůli tomu, že se svou podstatou vykonávání liší od manuálních a automatických. Hlavní rozdíl spočívá v tom, že testování neprobíhá podle předem výtvořeného plánu, tester náhodně zkoumá fungování aplikace jako celku.

Pozitivní test – druh testu, jenž je vykonáván na softwaru s použitím validních dat. Jeho účelem je ověřit, zda aplikace funguje podle všech předpisů a specifikací.

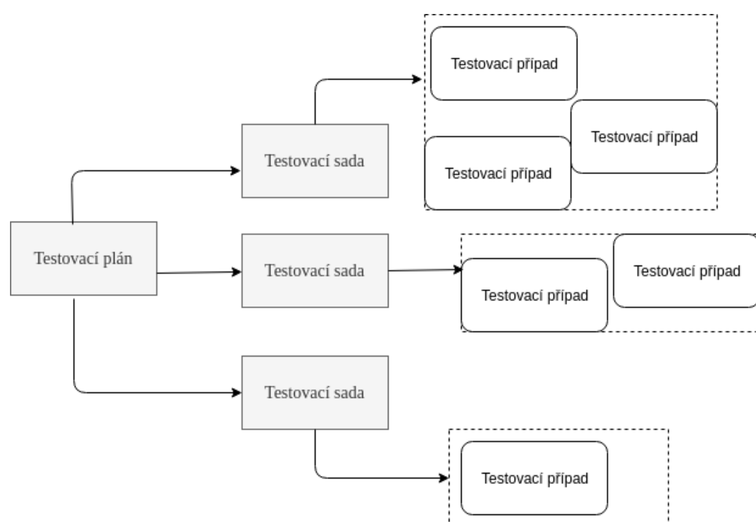
Negativní test – druh testu opačný k pozitivnímu testu, jelikož se používají nevalidní data. Během jejích spuštění se prověřuje chování systému, jestli se neprovede akce, která by se provést neměla.

Regresní testování – specifický druh testování softwaru, jenž kontroluje, zda nové změny či opravy kódu neovlivnily existující funkcionality. Regresní testy by měly validovat správnost chování klíčových funkcionalit softwaru. Jejich obsah je tvořen často opakovanými testovacími scénáři.

Progresní testování – je druh testování, jehož primárním účelem je ověřování správného a předem definovaného chování nově vyvinuté funkcionality produktu na základě existující dokumentace.

Smoke testování – druh testování, jehož klíčovým účelem je kontrola základních funkcionalit softwaru. Nejedná se o hlubší kontrolu. Negativní dopad vykonání podobných testů (odchylky od popisu z dokumentace, chyby) znamená, že by software neměl být kontrolován dále. Jedním z příkladů smoke testů muže být kontrola připojení k databázi.

Zajištování kvality (Quality Assurance nebo QA) – je činnost, jejíž primární záměr je kontrola a zlepšování vývojových a testovacích procesů (tvořících podmnožinu QA) ve všech fázích životního cyklu softwaru [11].



Obrázek 2.2: Struktura testovácího plánu

2.3.2 Úrovně testování

Testovací úrovně jsou skupiny testovacích aktivit, které jsou organizovány a řízeny společně. [25] Každá úroveň testování sjednocuje množinu procesů aplikovatelných v konkretní fázi vývoje – od jednotek zdrojového kódu [4] do celého systému. Zdroje určují 5 různých úrovní:

- **Jednotkové testy** (Unit testing) – testování zdrojového kódu softwaru, jež umožňuje odhalení chyb návrhu funkcionality či implementace ještě na raných, počátečních fázích vývoje projektu. Dané testy poskytují programátorovi možnost přesněji definovat a ujasnit si požadavky na software jak pro sebe, tak i pro celý tým. Hlavní myšlenka spočívá v izolaci jednotek kódu od „zbytku“ systému pro kontrolu jejích funkcionality. Psání jednotkových testů nespadá do povinností testera na projektu, jelikož vyžaduje detailní znalosti kódu, a v některých vývojových modelech dokonce řídí samotný vývoj softwaru, s čímž se potýká převážně jen vývojářský tým.

- **Komponentové testy** (Component testing) – používají stejný princip izolace kódu jako unit testy, ale v tomto případě operují ne s programovými jednotkami, ale s celostními moduly, aspekty, jež sjednocují prvky dohromady vzhledem k jejich logické propojeností a funkcionalitě, nebo programy. Komponentové testy jsou druhým stupněm realizace testování na projektu (po jednotkovém) a jsou rovněž zodpovědností programátorského týmu. Testování komponent poskytuje možnost snížení rizika

[4] Např. proměnné, funkce, metody, třídy nebo objekty, v případě, že se jedná o objektově orientované programování (OOP).

„probublání“ chyb do vyšších úrovní, a tím pádem zvýšení nákladů na jejich opravu pro všechny zainteresované strány.

- **Integrační testy** (Integration testing) – nadúroveň komponentových testů, na které se provádí kontrola interakcí integrovaných komponent, podsystémů, mikroservis, rozhraní, API[5] apod. Většina testovacích scénářů dané úrovně by měla projít procesem automatizace, čímž by se zvýšila hodnotu projektu vzhledem k „okamžitému“ objevení problému v systému. Existuje definovaný seznam chyb, jenž bychom dokázali odhalit pomocí testování na dané úrovni, do kterého patří nekonzistentní struktury zpráv mezí systémy, nesprávná či chybějící data, selhání komunikace mezi systémy, nedodržení závažných bezpečnostních předpisů atd. [25] Popsaná úroveň a další nadúrovně testování jsou zcela zodpovědností testera.
- **Systémové testování** (System testing) – jedna z nejvyšších úrovní testování, která by měla zjistit pomocí nahlížení na produkt jako na celek, zda software splňuje funkční a nefunkční požadavky. Kontrola by měla probíhat v prostředí co nejvíce podobnému produkčnímu, se kterým se běžně potýkají koncoví uživatelé.
- **Akceptační testy** (Acceptance testing) – účelem akceptační úrovně je se definitivně vyjádřit k tomu, zdali hotový produkt muže být nasazen a poskytnut koncovým uživatelům, či jestli porušuje nebo nesplňuje stanovené specifikace a smlouvy a není kompletní.

2.4 Testerská role

2.4.1 Hierarchie

Každá odborná práce a každá softwarová firma definuje svoji vlastní hierarchii testerských rolí a rozdělení zodpovědností. Z velkého seznamu, který najdeme v inzerátech s poptávkámi o nové členy softwarového týmu, bychom označili dvě základní úrovně:

- Test manager/lídr, jehož zodpovědností je:
  - Budování, kontrola a organizace testovácího procesu.
  - Pravidelná interakce se zákazníkem.
  - Zajištění viditelnosti, sledovatelnosti a kontroly testovácího procesu pro poskytování vysoce kvalitního softwaru.
  - Aplikace nejvhodnějších testovacích metrik v produktu či testovacím týmu.

[5] API (Application Programming Interface) – rozhraní pro programování aplikací.

  - Správa a údržba zdrojů pro testování.
  - Zaručení kvality softwaru, jeho dostatečnosti, efektivnosti a výkonosti.

- Test inženýr/QA testeři/QC testeři, jejichž zodpovědností je [9]
  - Tvorba různých drůhu testů (funkční, nefunkční, regresní, automatické, stress testy apod.)
  - Být součásti procesu sběru informací (zjištění důvodů vzniku problémů, seznamení s novými technologiemi, reporty o provedení testování, neustalé vyptávání na fungování softwaru atd.)
  - Psaní a vykonávání testovácích scénářů, vytvaření plánů (může patřit do seznamu zodpovědností test managera), provádění monitoringu, odhalení chyb návrhu, údržba testovacího prostředí, provedení automatizace procesů nebo cokoliv, co souvisí s testy.
  - Organizace analýzy (sestávení reportů, pochopení zákaznických požadavků a jejich aplikace ve vývoji, zkoumání logů atd.)
  - Zajištění (částečně) kvality a zlepšení softwaru (výroba uživatelských příruček, kontrola kódu, psání dokumentace a hlubší zkoumání kódu z důvodu pochopení vzniku chyby)

2.4.2 Praxe

Vymezení na 2 hlavní kategorie, podrobný výpis povinnosti každé z pozicí nám nabízí jenom abstraktní přestavu o testerské roli v softwarovém týmu. Abychom předešli obrovské kritice testerské práce jako práce „klíkácí opičky, která nerozumí tomu, co dělá“, vyvrátili názory popsané v kapitole 2.1 a přiblížili se více k praxi, vytkneme několik zásadních vlastností a funkci, jež by měl splňovat tester na projektu dle C. Kanera, J. Bacha, B. Pettichorda a M. Eldridge [2]:

- Tester je reflektorem projektu. Důraz se klade na pochopení faktu, že testování je procesem sběru informací, „zmapování okolí“ a detailního reportování aktuálního stavu vývoje.

- Mise testera řídí absolutně všechno, co dělá. Organizace testovácího procesu, volba, kombinace a provedení specifických druhů testů je zavislá na konkretní misi či poslání testera. Zaměření se na to, co je podstatné. Zkuste odpovědět na dvě důležité zavádějící otázky týkající se vaší práce a jejího celkového přínosu pro projekt. Co byste měli dělat v případě, že zrovna nemáte nic na práci? Co byste měli dělat, když přesně víte, co konkretně dělat máte?

- Softwarový tester obsluhuje spousty klientů. Smiřte se s rolí obsluhy. Vaším zákazníkem není jenom skupina stakeholdrů, ale zároveň project manager, vývojářský tým, technická podpora, marketing a hlavně konečný uživatel produktu.

- Tester odhalujete chyby lidí, na jejichž názoru mu záleží. Povinností testera je reportování jakékoliv odchylky, která by dokázala snižit hodnotu výsledného produktu. Je potřeba být statečným a odvažným, aby poukázal stakeholderům na jejich chyby a možné negativní dopady jejich požádavků. Je podstatné zdůraznit, že je ve společném zájmu testerů a klientů dbat na vysokou hodnotu softwaru.

- Najde důležité chyby rychle. Rychlost a kvalita jsou dvě stěžejní kritéria, kterým se vyznačuje protřelý Senior tester mezi stovkami začátečníků. Na otázku, jak se dostat do té vyšší ligy, se nám vždy dostane tytéž odpovědi: „Postupem času a praxí". Požadavky na znalosti a dovednosti se liší mezi firmami, ale bude mezi ně patřit i to nezbytné – rychlost nalezení chyby. Návod k „akceleraci" testovacího procesu nám znovu laskově nabídli pánové Cem Kaner, James Bach a Bret Pettichord ve své knize „Lessons learned in software testing: a context-driven approach".

  – Otestujte funkcionality, které se změnily. Opravy, aktualizace, zlepšení a vývoj nových částí systému jsou jako hrom, jež naznačuje příchod bouřky ve formě rizika, chyb a defektů.

  – Testování jádra systému je přednostnější, než zkoumání kvality dodatečných funkcí. Zaměřte se na podstatu softwaru. Například stránka cestovní kanceláře může mít špičkově vyladěné uživatelské rozhraní, ale když si zákazníci nebudou schopni objednat žádný zájezd, tak logicky byla udělána chyba v přístupu k testování a testovácím plánu.

  – Schopnosti před spolehlivostí. Krátce a zjednodušeně prosřednictvím jedné věty to lze popsát následujícím způsobem „Než se vrhneme na detailnější testování funkcí, zjistíme, jestli jsou vůbec funkčí."

  – Důraz bychom měli klást na běžné, všední použití softwaru, než procházet nerealistické scenáře. Dané doporučení se vztahuje i na hrozby.

  – Přednostně otestujte problémy, které mají větší dopad na aplikace. Například pro každého zákazníka České Spořitelny nebo jiné banky nabízející zákazníkům možnost použití internetového bankovnictví, přetékající přes okraj políčka příjmení bude sice nepříjemný postřeh, ale nehrající tak velkou roli, jako neschopnost systému provést transakci.

  – Soustřeďte se na žádanější a zásadnější části vašeho systému.

- Funguje spolu s programátory. Jedná se o utopický předpoklad společného fungování testerského a vývojářského týmu. I přesto, že občas se zdá, že podobná vize je nepředstavitelná, je klíčovou částí testerské mise. Kompletní, stručné, přehledné, rychlé reporty a zpětná vazba na aktuálně se vyvijející části projektu jsou programátorskými požadavky na testera, které skutečně dávají smysl a ulehčí spolupráci.

- Dejte si pozor na „Není moje práce“ teorii. Brzo či později se projeví i sebemenší chyba a bude mít dopad na aplikaci. Vzhledem k dobré testerské znalosti softwaru jako celku, libovolná předtucha, podezření nebo představa o zvlaštním chování softwaru, náznaku nekonzistence může pomoci najít chybu co nejdříve. I občas nesmyslné požadavky od stakeholderů by měly být zvážené z vaší strany.

- Ptejte se na všechno ale ne nutně nahlas. Testování je možné provést bez ptání se na otázky, ale jaká je pravděpodobnost, že proces proběhne úspěšně a kvalitně? Přiznejme si, že vichr otázek na jednoho či dva programátory, kteří vždy mají něco na práci a přepínájí mezi projekty, může být poměrně otravným a nesnesitelným. Měli bychom si uvědomit, že libovolná otázka, která vás napadne, může pomoci odhalit nejzákeřnější vady softwaru.

- Soustřeďte se na selhání, aby se váš klient mohl soustředit na úspěch. Testeři jsou negativní, a je to součástí jejich práce. Vývojářský tým nemůže vnímat testování zaměřené na zajištění kvality a nalezení bugů jako pozitivní věc. Avšak lepší výsledný produkt s dobrou podporou, který bude pravděpodobně úspěšnější na trhu, je dobrou odměnou za včas projevený skepticismus a kritický testerský pohled.

- Nikdy nezajistíte kvalitu software (QA) skrze testování. Jedná se o podobné pojmy, ale nejsou ekvivalentní. Jak bylo již řečeno, testování je podmnožinou QA. Ve velice malém množství případů tester zasahuje do procesu návrhu a vývoje softwaru.

- Nenajdete všechny chyby. Nejedná se o podceňování zkušeností a znalostí testera, ale o realistický pohled na problematiku testování. „20 % vašeho času produkuje 80 % vašich výsledků a naopak“ - Paretův Princip, který se zaměřuje především na pozitivní testovací případy. [19] Z tohoto výroku můžeme odvodit, že negativní testovací případy zaberou 80 % času a měly by simulovat nejméně pravděpodobné scénáře, které 20 % konečných uživatelů je schopno reprodukovat. Vzhledem k tomu, že proces zajištění kvality může zastavit omezené financování ze strany klienta nebo nedostatek času a pracovních jednotek, je vždycky potřeba najít optimální řešení sestavení testovacího plánu. Paretův Princip může napomoci vyřešit nastalý problém a zvýšit produktivitu.

- Nečekejte od nikoho, že porozumí testování, nebo tomu, co je potřeba, abyste odvedli dobře svoji práci. Počítejte s tím, že manageři a programátoři nejsou schopni číst vaše myšlenky a nejsou ve většině případů seznámení s průběhem všech procesů, které musí tester provést. Vaší nejdůležitější zodpovědností je připomínání a permanentní vysvětlování zákazníkům, proč a jakým způsobem probíhá proces testování aplikace.

Testování se určitě řadí mezi dobré praktiky vývoje softwaru (best practices).[10]

Kapitola **3**

Techniky a rozdělení testů

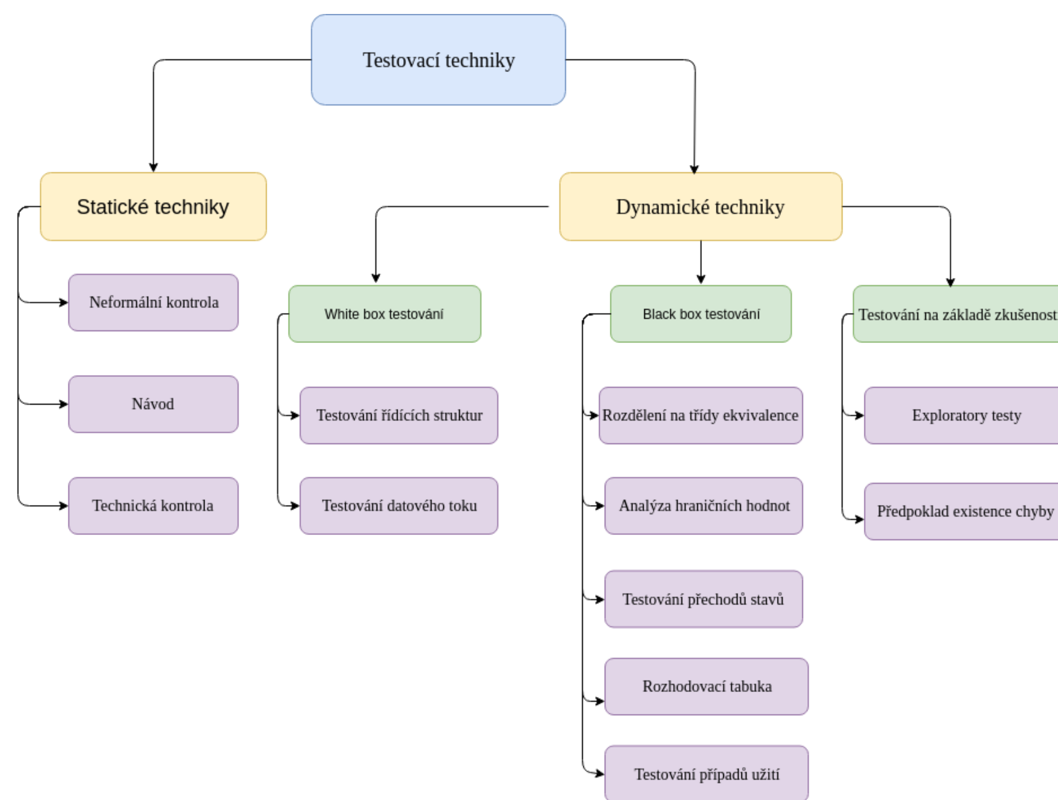
Stejně jako v procesu programování se vývojářský tým snaží dodržovat osvědčené postupy ve své práci, aby se vyhnul tvorbě nepochopitelného, spletitého, neokomentovaného, nestrukturovaného, nekonzistentního, ... kódu, testeři mají svoji sadu osvědčených a spolehlivých testovacích technik, jejichž primárním účelem je „pomoci identifikovat podmínky, testovací případy a testovací data.“ [25] V závislosti na životním cyklu softwaru, úrovních testování, pomocném testovacím nastroji, kterým tester disponuje, provedené analýze, účelu testování, se používá vybraná množina z velké taxonomie testovacích technik. Rovněž rozdělujeme techniky na 2 základní kategorie:

- Statické, kde neexistuje potřeba spouštět výslednou aplikaci.
- Dynamické, vyžadující provedení v běhovém prostředí.

Oba druhy se člení na další kategorie a podkategorie (viz obr. 3.1). V rámci dané kapitoly provedeme podrobnou analýzu každé z technik, uvedeme jejich aplikaci v reálném životě na skutečných projektech a prozkoumáme použitelnost v různých úrovních testování aplikací.

3.1 Statické techniky

Jak bylo řečeno, statické testování nevyžaduje spuštění aplikace. Jako názorný příklad podobné kontroly může posloužit kontrola návrhu, designu, uživatelských příruček, testovacích plánů, různorodých scénářů, sestavených případů, sepsaných specifikací a požadavků, což jsou příklady dokumentů vytvořených v přirozeném jazyce. Hlavní výhodou podobného testování spočívá ve zjištění a nalezení chyb a odchylek ještě v počáteční, „embryonální“ fázi životního cyklu softwaru. Toto zajistí snížení nákladů firmy na opravu chyb (mezi nalezením chyb v různých fázích vývoje a náklady s tím spojenými existuje kladná korelace), lepší interakci členů týmu a z toho vyplývající zefektivnění probíhajících procesů. Nesmíme se zabývat pouze produkty napsanými v přirozeném



Obrázek 3.1: Testovací techniky

jazyce, poněvadž se statické testování rovněž zaměřuje na kód, jeho kvalitu (mrtvé části[6], znovupoužitelnost, složitost, syntaxe atd.), bezpečnostní chyby. Seznámíme se blíž s odlišnými základními a všeobecně známými technikami statického testování.

3.1.1 Neformální recenze

Je druhem zkoumání, analýzy a kontroly patřičného dokumentu mezi dvěma nebo několika členy týmu, za účelem zlepšení jeho kvality, konzultace, hledání řešení vzniklých problémů a případného nalezení chyb. Je v podstatě neformálním druhem schůzky, jejíž výsledky mohou, ale nemusí, být zaznamenány. Charakter podobného setkání má, jak je patrné z názvu „nefomální charakter a není založen na zdokumentovaném procesu." [25]

[6] Dead code – kód, který se už v aplikaci nepoužívá.

3.1.2 Návody

Návody, známé v angličtině jako walkthrough, mají jako primární účel zlepšení kvality výsledného produktu (stejně jako neformální recenze). Jsou formálnějším druhem schůzky, jejíž cílem je záměrné seznámení posluchačů s funkcionalitami softwaru v podobě procházení softwarového produktu a dokumentace krok za krokem (v mnoha případech účelem může být hodnocení publika a zpětná vazba či diskuze změn, implementace a požadavků položených na aplikaci), čímž se odhalí velký počet chyb návrhu produktu. Několik zdrojů včetně ISTQB[7] zmiňují, že druhotným záměrem návodů může být koučování a zaučování účastníků.[25] Výstupem srazu může být sepsaný dokument nalezených defektů a klíčových aspektů schůzky.

3.1.3 Technická kontrola

Technická kontrola (technical review) je druh neformálního srazu (nemusí se pokaždé jednat o osobní setkání, ale o libovolný způsob vzájemné interakce) technického charakteru (v praxi se formalita schůzky liší vzhledem k nastaveným vztahům mezi účastníky review), v jehož rámci se provádí kontrola technického obsahu, korigování prvotní strategie vývoje, projednávání způsobů zlepšení existujících a budoucích produktů. Účastníci se vyznačují svojí profesionalitou a odborným pohledem na proces vývoje (techničtí architekti, experti a designeři produktů). Se zřetelem na zkušenosti účastníků a způsobu provedení kontroly, jenž vyžaduje dlouhodobou přípravu, takovýto neformální sraz dokáže motivovat ke generování nových užitečných nápadů, o kterých se dá následně uvažovat jako o jiných možných implementacích.

3.1.4 Statická analýza

Jedním z nejpopulárnějších projevů statické analýzy v praxi může být všeobecné známé a nejpoužívanější manuální testování ve formě code review (méně známý pojem v praxi – posouzení kódu), během kterého proběhne proces posouzení kvality kódu. Existují speciální automatické nástroje pro různé programátorské jazyky, které usnadňují provedení statické analýzy. Jako příklad můžeme uvést RIPS pro PHP, což je open source nástroj, bezpečnostní analyzátor zdrojového kódu. Rovněž do statické analýzy můžeme zahrnout také softwarové metriky (produktu, běhu procesu a údržby) a reverzní inženýrství – „proces, ve kterém se ze vzorového předmětu snažíme získat co nejvíce informací o jeho vlastnostech, postupu výroby či funkcích, neboť máme většinou jen holý předmět bez (…) dalších výrobních podkladů.“ [17]

[7] International software testing qualifications board je mezinárodně uznávanou organizací pro certifikování softwarových testerů.

3.2 Dynamické techniky

Dynamické techniky jsou techniky vyžadující spuštění testovaného softwaru nebo jeho části. Jsou rozdělené na 3 skupiny na základě přístupu k vnitřní struktuře produktu (znalosti kódu, spolupráce subsystémů, struktury databáze apod.).

3.2.1 White box testování

Techniky white boxu se dají použít na všech testovacích úrovních, ale jejich největší uplatnění najdeme při implementaci komponentových testů[25]. Jejich záměrem je zkoumání interního stavu softwaru s účelem zvyšení jeho kvality a redukování počtu chyb. Klíčová vlastnost daného druhu dynamických technik spočívá v tom, že je od testera vyžadována znalost kódu a vnitřního stavu aplikace. Spolu s danými techniky přichází nový pojem – **pokrytí kódu (code coverage)**, což je podíl počtu otestovaných řádků ku všem, vyjadřený primárně v procentech. Zároveň by tester měl déle uvažovat, jestli zavedení a realizace white box technik na projektu náhodou nezkomplikuje testovací proces, který se může rozrůstat. Procházení existujícími cestami kódu není jednoduchá záležitost, a vzhledem k tomu, že se ve většině projektů počítá s rozšiřitelností a změnou logiky aplikace, takovéto testování může stát více úsílí, energie a investicí, než přinese užitku. Měli bychom počítat s tím, že použití podmínkových výrazů *if* dokáže zdvojnásobit počet unikátních cest, zatímco *for*, *while* nebo *do while* cykly je zvětší o počet iterací.

3.2.1.1 Testování řídících struktur (control flow testing)

Testování řídících struktur je ve většině případů reprezentováno pomocí stejnojmenných grafů, které zobrazují procházení všech cest programem během jeho spuštění. **Rozhodnutí (decision points)** jsou speciálními uzly takového grafu, kde se provede větvení programu. Typickými příkazy, které „způsobují" tato větvení jsou if-else nebo case bloky. **Spojovací bod (junction point)** je bodem, v němž probíhá sjednocování různých odvětví do jedné cesty.

Testování postupů definuje několik různých úrovní pokrytí kódu: [24]

- Úroveň 0 – testování konkretních částí kódu, pokrytí je menší než 100%. Nezodpovědný a neprofesionální přístup, jak vůči klientovi, tak i vůči zákazníkům.
- Nejnižší úroveň pokrýtí je „100 % pokrýtí příkazů" [24], všeobecně známá jako **statement coverage**, je těžce simulovatelná, poněvadž vyžaduje provedení všech příkazů programu alespoň jednou.
- **Branch/decision coverage**. Každé rozhodnutí, které má hodnotu true nebo false, je hodnocené (alespoň jendou) a je zahrnuté do testovacích

případů. Tato úroveň pokryje všechny příkazy, ale ne všechny existující cesty.

- 100 % **condition coverage**. V tomto pokrytí jsou uvažovány jenom výrazy s logickými operandy (OR, AND, XOR).
- 100 % **path coverage** – pokrytí všech možných cest v programu. Čím více cyklů a podmínek se v kódu vyskytne, tím větší (enormní) počet cest vznikne, proto zkoumání každé z nich je nepředstavitelnou činností pro testování moderního softwaru.

Použití daných technik pro manuální testování softwaru není správnou logickou volbou z důvodu svého obrovského rozsahu a opakovatelnosti. Proto vyjmenujeme sadu nástrojů pro automatizaci jejich realizace: Coco (C, C++, C#), Parasoft Jtest (Java), DevPartner (Java), Emma, JTest, Sonar. [3]

3.2.1.2 Testování datového toku (Data flow testing)

Testování datového toku je jedna z nejčastěji používaných technik mezi vývojáři. I přestože je zařazena do kategorie dynamických testů, exituje statická varianta v podobě procházení a kontroly kódu. Problémy, jež by daná technika měla odhalit, spočívají v rozsahu proměnných v programu, jejichž životní cyklus by měl vždy obsahovat 3 různé fáze: vznik, inicializaci a zánik. [24] Ve statické verzi testování datového toku by se mělo provést pozorování správnosti plynulých změn tří etap vůči flow bez spuštění aplikace. Dynamická podoba by zas měla představovat kontrolu průběžných výsledků po spuštění softwaru pomocí pozorování inicializovaných proměnných a jejích použítí v aplikaci. Stejně jako v případě testování řídicích struktur se může lišit zvolená úroveň pokrytí kódu. [13] Jednou velkou nevýhodou popsané techniky, stejně jako v případě testování řídících struktur, je vyžadování znalostí programování od testera. Je povinností zmínit, že existuje poměrně velké množství nástrojů a IDE[8], podporujících analýzu toku dat. Jako příklad bychom mohli uvést IntelliJ IDEA nebo PHPStorm od JetBrains.

3.2.2 Black box testování

Black box testování je souhrnem postupů, strategií, které se zaměřují na kontrolu softwarových specifikací, požadavků, množství případů užití[9], a nevyžaduje znalosti vnitřního systému, interakcí mezi moduly, použitých struktur a probíhajících procesů. Cílem je stanovit, zda software splňuje požadavky či ne. Black box jako kategorie dynamických testovacích technik vyniká díky svému hlavnímu konceptu: tester přistupuje k aplikaci z pohledu koncového uživatele,

[8] Integrated Development Environment – vývojové prostředí

[9] Use Case nebo případ užití je popis možných sekvencí interakcí mezi diskutovaným systémem a jeho externími aktéry, které se týkají konkrétního cíle.[22]

Nevalidní vstup	Validní vstup	Nevalidní vstup
před 01.01.1900	01.01.1900 – 20.04.2001	po 20.04.2001

Tabulka 3.1: Třídy ekvivalence – datum narození cestujícího

kde na rozdíl od něj dokáže posoudit na základě dokumentace, jestli výstupem odvedeného testování je správná kombinace výstupů. Techniky block box přístupu jsou aplikovatelné na všech testovacích úrovních.

3.2.2.1 Rozdělení na třídy ekvivalence (Equivalence Partitioning)

Jedna z malého množství technik, kterou dokáže použít začínající tester intuitivně i bez znalostí teoretických základů. Podstata vybrané techniky spočívá v tom, že uživatel rozdělí množinu vstupů do několika tříd, v rámci kterých by aplikace měla použít stejnou logiku pro zpracování údajů. Každá testováná hodnota by měla patřit pouze do jedné označené třídy.

Zvažme následující situaci: Dospělý zákazník cestovní agentury si vybral zájezd pro jednoho cestujícího a přešel na stránku s formulářem, kde musí uvést své osobní údaje. Otestujeme jeden z inputů[10], konkretně – datum narození. Organizační pravidla a specifikace, které doložil zákazník softwaru jsou:

- Rok narození cestujícího by měl být větší nebo se rovnat 1900.
- Za dospělého je považován cestující starší 18 let (v den realizace zájezdu.).

Představme si, že datum odletu cestujího je 20. dubna 2019. Vytvoříme tabulku intervalů možných vstupů pro přehlednost použití dané techniky v popsaném příkladě (viz tabulka 3.1).

Skupina testů tvoří jednu třídu ekvivalence, pokud se domníváme, že [23]

- Všechny testy ve třídě kontrolují totéž.
- Pokud jeden test odchytí chybu, ostatní pravděpodobně udělají totéž.
- Pokud jeden test odhalí chybu, ostatní je pravděpodobně naleznou.

3.2.2.2 Analýza hraničních hodnot

Důležitá technika, která nám přináší nový pohled na validace vstupních hodnot. Na základě vlastních zkušeností s programováním webových aplikací můžu potvrdit, že hraniční hodnoty ve všech případech jsou těmi „nejslabšími místy“ programu, jež mohou způsobit nepředvídatelné chování aplikace.

[10] Název HTML prvku.

Strategie aplikace dané techniky pro testování případů užití je poměrně jednoduchá. Po sestavení a rozdělení vstupních hodnot na třídy ekvivalence nalezneme hraniční hodnoty.

Tak v případě dříve uvedeného příkladu políčka dne narození cestujícího bychom měli mít 2 hraniční hodnoty: 01.01.1900 a 20.04.2001. Popsaná technika zkoumá chování aplikace vždy ve 3 různých bodech z definované hranice: v nejbližším nad, pod a prověřovanou hodnotu. Po provedení dalších analýz zjistíme, že ty nejkritičtější a nejdůležitější body pro zkoumání chování SUT[11] jsou následující trojice: {31.12.1899, 01.01.1900, 02.01.1900} a {19.04.2001, 20.04.2001, 21.04.2001}. V dané technice se dají použit i jiné vstupy, nejenom číselné (např. $HASHTAG, -18.01.1992, regulární výrazy apod.).

3.2.2.3 Testování přechodů stavů (state transition)

Tato technika se používá v systémech, jež mají objekty měnící stav během své životnosti. Diagramy stavů zobrazující chování systému v reakci na konající se události zpřehledňují všechny možné stavy, ve kterých se systém či objekt může nacházet. Zároveň přesně definují množinu platných přechodů, jež jsou výstupem konkretní události, a vylučují nevalidní přechody mezi nimi. Existuje i jiný nástroj zachycující chování systému – tabulka přechodů, jež zpřehledňuje všechny možné kombinace přechodů mezi stavy (existují omezení, jelikož zvětšení počtu stavů má přesně opačný efekt), čímž eliminuje počet chyb nebo nejasnosností při provedení procesu testování. Vzhledem ke své praktické stránce, se oba nástroje dají využít pro účely dokumentace projektu. Výhodou použití tabulky přechodů oproti stavovému diagramu je rychlější odhalení implementačních chyb.

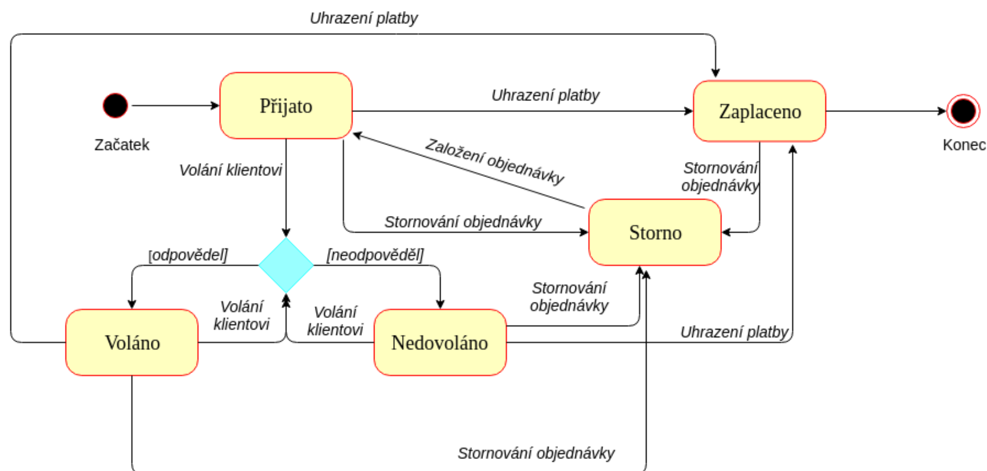
Pro přehlednější reprezentaci uvedených nástrojů znovu použijeme znamý příklad cestovní agentury a tentokát se zaměříme ma zkoumání objednávky. Dokumentace uvádí, že se libovolná zákazka může nacházet v jednom z následujících stavů: přijato, voláno, nedovoláno, zaplaceno, storno. Představíme UML diagram stavů objednávky, [12] který transformuje diagram do grafové reprezentace, kde uzly tvoří stavy objektu, a přechody, označené šipkami (mohou být jednosměrné či obousměrné), – spouštění události. (viz obr. 3.2)

Pro vytvoření tabulky přechodů (viz tabulka 3.2) se inspirujeme příkladem z knihy „A practitioner's guide to software test design", napsanou panem Lee Copelandem, ve které uvedená tabulka přechodů obsahuje základní 4 sloupce: aktuální stav objednávky, možná konající událost, provedená akce a následující stav objednávky. Za null považujeme neexistující stav, který vznikl jako výstup neplatné operace. [24]

[11] System under test – systém, který je testován na správnou funkčnost.
[12] Unified Modeling Language – notační jazyk použitý při vývoji systémů.

Aktuální stav	Událost	Akce	Následující stav
null	Uhrazení platby	–	null
null	Stornování objednávky	–	null
null	Úspěšné volání klientovi	–	null
null	Neúspěšné volání klientovi	–	null
null	Založení nové objednávky	Nová objednávka	Přijato
Přijato	Uhrazení platby	Konfirace objednávky	Zaplaceno
Přijato	Stornování objednávky	Storno objednávky	Storno
Přijato	Úspěšné volání klientovi	–	Voláno
Přijato	Neúspěšné volání klientovi	–	Nedovoláno
Přijato	Založení nové objednávky	–	null
Voláno	Uhrazení platby	Konfirace objednávky	Zaplaceno
Voláno	Stornování objednávky	Storno objednávky	Storno
Voláno	Úspěšné volání klientovi	–	Voláno
Voláno	Neúspěšné volání klientovi	–	Nedovoláno
Voláno	Založení nové objednávky	–	null
Nedovoláno	Uhrazení platby	Konfirace objednávky	Zaplaceno
Nedovoláno	Stornování objednávky	Storno objednávky	Storno
Nedovoláno	Úspěšné volání klientovi	–	Voláno
Nedovoláno	Neúspěšné volání klientovi	–	Nedovoláno
Nedovoláno	Založení nové objednávky	–	null
Zaplaceno	Uhrazení platby	–	null
Zaplaceno	Stornování objednávky	Storno objednávky	Storno
Zaplaceno	Úspěšné volání klientovi	–	null
Zaplaceno	Neúspěšné volání klientovi	–	null
Zaplaceno	Založení nové objednávky	–	null
Strono	Uhrazení platby	–	null
Storno	Stornování objednávky	–	null
Storno	Úspěšné volání klientovi	–	null
Storno	Neúspěšné volání klientovi	–	null
Storno	Založení nové objednávky	Nová objednávky	Přijato

Tabulka 3.2: Tabulka přechodů stavů objednávky



Obrázek 3.2: Stavový diagram objednávky

3.2.2.4 Rozhodovací tabulka (decision table)

Rozhodovací tabulka se používá pro přehlednější reprezentaci „aplikační vrstvy"[13] softwaru, poněvadž dokáže popsat komplikované nebo nezřejmé chování systému. Základem dané kombinatorické techniky je „implementace systémových požadavků, které specifikují, jak různé kombinace podmínek vedou k jiným výsledkům." [25] Rozhodovácí tabulka je jedním z nejdůležitějších nástrojů v práci softwarového testera, který později dokáže sloužit jako návod pro psání testovacích případů (test case).

Řádky rozhodovací tabulky představují seznam různorodných podmínek a na ně navazujících akcí, které mohou či nemohou nastat. Každý sloupec je unikátní a tvoří tzv. rozhodovací pravidlo, jež je kombinací několika podmínek, jejichž splňení (ne)zahajují akce, vztahující se k danému konkretnímu pravidlu (sloupci). Hodnoty podmínek a akcí můžou být binární (Y, 1, N, 0 – mezinárodní verze; A či N – česká verze) nebo diskretní (červená, zelená, žlutá, 1, 0, -15, 1–100 atd.).

Znázorníme použití dané techniky na znamém příkladě cestovní agentury. Popsané specifikace požadavků klienta tvrdí, že pokud si zákazník zvolí zájezd, jehož cena je menší než 40 000 Kč pro jednoho člověka, a zároveň každý spolucestující je plnoletý, tak se podaří provést online objednávku, umožňující rovnou zaplatit zákazku kartou (viz tabulka 3.3). Vzhledem k tomu, že

[13] Business logic

Podmínky	Pravidlo 1	Pravidlo 2	Pravidlo 3	Pravidlo 4
Cena zájezdu je >= 40 000 Kč	A	A	N	N
Spolucestující není plnoletý	A	N	A	N
Akce				
Online objednávka	N	N	N	A

Tabulka 3.3: Rozhodovací tabulka – provedení online objednávky

hodnoty v uvedené tabulce jsou booleovské, počet vzniklých pravidel se rovná 2^n, kde n jsou všechny existující podmínky. Zjednodušeně řečeno, rozhodovací tabulka popisuje exekuce různých akcí v případě splnění kombinace specifických podmínek, a může sloužit jako kompaktní forma dokumentace logiky a zpracování dat softwarem.

3.2.2.5 Testování případů užití (use case testing)

Testy mohou být navrženy na základě případů užití, které jsou postupným popisem funkcí libovolné transakce. Takový dokument popisuje interakci uživatelů, systémů nebo komponentů s jiným systémem či komponentem, nikoliv vnitřní chování aplikace během podobné „komunikace". I přestože případy užití mají dost často větší význam pro programátory, pro něž se zároveň vytváří z důvodu shrnutí funkčních systemových požadavků, jsou „informačním střediskem" pro softwarové testery.

Výhody přicházející spolu s testováním případů užití, se dají shrnout do několika bodů: [24]

- Zachycení funkčních požadavků z pohledu uživatele bez ohledu na paradigma vývoje.
- Lze použít k aktivnímu zapojení uživatelů do procesu shromažďování a definování požadavků.
- Poskytování základů pro identifikaci klíčových vnitřních součástí systému, struktur, databází a vztahů.
- Slouží jako základ pro vývoj testovacích případů na systémní a akceptační úrovni.

Šablonu pro vytváření testovacích případů uvádí pan Cocburn ve své knize „Writing effective use cases" [22]. Daný vzor do sebe zahrnuje následující položky: název, účel, rozsah, úroveň, hlavní aktér[14], stav systému před exekucí

[14]Systém, podsystém, komponenta, obyčejný člověk.

případu užití a po ní (v případě úspěchu a neúspěchu), stav systému v případě chyby, spouštějící akce, hlavní úspěšný scénář, výjimky, popis priority, frekvence použití, dobu odezvy systému, další aktéři, datum splatnosti, úroveň úplnosti[15] testovacího případu. [22]

Příkladem tentokrát poslouží provedení objednávky na stránkách anonymní cestovní agentury. Interakce proběhne mezi systémem (označení S) a zákazníkem (Z).

Uvedený případ použití je jedním z mnoha způsobů interakce webové aplikace s obyčejným uživatelem, jenž bude použit v implementační části této práce (viz tabulky 3.4 a 3.5).

3.2.3 Testování na základě zkušeností

Testování na základě zkušeností je nadskupinou několika druhů testovacích technik, jež vynikají na pozadí black a white box testování tím, že míra jejich úspěšnosti je přímo závislá na zkušenostech testera s podobnými či stejnými nástroji a aplikacemi. Přestože jeho dovednosti hrají poměrně důležitou roli v absolutně každé technice, v testování na základě zkušeností mají klíčové postavení a jsou fakticky jediným nástrojem, který může být použit v testovacím procesu. Na rozdíl od popsaných dynamických technik, tester nemá k dispozici předem sepsané příručky nebo šablony, jež ulehčují proces validace systému. Zároveň se mu nabízí velký prostor k tzv. „improvizaci".

3.2.3.1 Exploratory testing

Exploratory testing není jenom bezcílné procházení aplikací s účelem narazit na náhodnou chybu, která se najednou zjeví před očima testera, aby splnila účel dlouhodobého brouzdání v dobře známé aplikaci. Nesmíme podobné testování podceňovat, poněvadž splnění jeho cílů dokáže odhalit ty nejzákeřnější chyby aplikace. Do seznamu účelů exploratory testování patří: [5]

- Získat pochopení toho, jak aplikace funguje, jak její rozhraní vypadá a jakou funkčnost implementuje. Může sloužit jako seznamovácí prostředek pro testery, kteří pouze poznavají procesy fungování softwaru. Zároveň může být použité při psaní nových testovacích plánů, aby identifikovalo vstupní body testů.

- Přinutit software, aby projevil své funkce. Nejedná se o testování aplikací podle předem vytvořeného plánu. Hlavní idea je klást na produkt složité na zpracování požadavky a pozorovat jejích exekuci. Výsledkem nemusí být nelezení řady chyb či odchylek, ale opakovaná kontrola funkcionalit specifikovaných v dokumentaci.

[15] Alistair Cocburn úvadí 4 různé úrovně: základní identifikace případů užití (0.1), hlavní scénář je definován (0.5), vzmezeny všechna rozšíření (0.8), všechna pole jsou vyplněna (1.0).

Hlavní úspěšný scénář	0. Z: Přejít do zvolené jazykové verze aplikace. 1. Z: Změnit vyhledávací parametr „délka pobytu“. 2. S: Provést změnu parametru. 3. Z: Změnit vyhledávací parametr „počet dni“. 4. S: Provést změnu parametru. 5. Z: Změnit vyhledávací parametr „destinace“. 6. S: Provést změnu parametru. 7. Z: Kliknout na tlačítko „Hledat“. 8. S: Zobrazit seznam hotelů. 9. Z: Vybrat libovolný hotel. 10. S: Přejít do detailu hotelu. 11. Z: Zvolit vhodný termín. 10. S: Přejít do checkoutu. 11. Z: Vyplnit cestovní formulář. 12. Z: Skrze tlačítko „Objednat“ provést objednávku. 13. S: Vytvořit novou objednávku.
Výjimky	8a. Chyba poskytovatele dat. Nejsou k dispozici žádné hotely. S: Zobrazí seznam hotelů neodpovídající parametrům vyhledávání.
	10a. Chyba poskytovatele dat. Nejsou k dispozici žádné termíny zájezdů. S: Zobrazí seznám zájezdů neodpovídající uvedené předem délce pobytu nebo termínu.
	13a. Proběhla změna ceny. S: Objednávka je uložena ale není odeslána.
	13b. Double booking. S: Objednávka je uložena ale není odeslána.
	13c. Chyba poskytovatele. S: Objednávka je uložena ale není odeslána.

Tabulka 3.4: Případ užití – scénář provedení objednávky zájezdu

Název	Rezervace zájezdu
Účel	Založení nové objednávky
Úroveň	Základní úloha
Hlavní aktér	Plnoletý zákazník
Stav před exekucí	Není definováno
Stav po exekuci (úspěch)	Plnoletý zákazník si úspěšně objednal zájezd
Stav po exekuci (neúspěch)	Zákazníkovi se nepodařilo založit novou objednávku
Spouštějící akce	Vyhledávání zájezdu nebo pobytu
Priorita	Kritická
Frekvence použití	∼ 1000 zákazníků týdně
Doba odezvy	10 sekund nebo méně
Jiní uživatelé	Žádné
Datum splatnosti	Není určen
Úroveň úplnosti	1.0 – všechna pole jsou vyplněna
Otevřené úlohy	Žádné

Tabulka 3.5: Případ užití – metadata objednávky zájezdu

- Najít chyby, čímž myslíme objevení zastaralých a historických vad softwaru, pomocí jeho účelově hlubšího průzkumu. Zaměření na ty části produktu, které pod sebou skrývají složitou a spletitou logiku, několikanásobně zefektivní exploratory testování.

3.2.3.2 Předpoklady existence softwarové chyby

Podstata dané techniky spočívá v neustálém skeptickém testerském vnímání hotového softwaru nebo aplikace, s představou, že produkt nemůže být bezchybný, nezavisle na tom, jakým odborníkem je programátor. Takový koncept se zakladá na testerských znalostech systému, celého procesu jeho vývoje, chyb, kterých se vývojáři dopustili dříve nebo je často opakovali, informacích ohledně fungování aplikace v minulosti. [25] Výstupem aplikace popsané techniky by měl být dokument shrnující všechny potenciální různorodé chyby, způsobující provedení nevalidních operací a pád programu.

Kapitola **4**

Modely softwarového procesu a metodiky vývoje

Metodiky vývoje softwaru je souhrn postupů a přístupů k problematice tvorby softwarových produktů ověřených časem a dlouhodobou praxí . [16] Rovněž vystupují jako předem navržené struktury (frameworky) a plány, kterými se řídí většina procesů týkajících se vývoje softwaru od návrhu do managmentu, jež nepopisují žádné technické aspekty.

Popsaný v podkapitole 2.3.1 model životního cyklu nebo systems development life cycle (vznik kolem roku 1960) je příkladem podobné metodiky. SDLC jako proces nasměrovaný na efektivní tvorbu a úpravu softwaru za minimální ztráty vystupuje větší měrou jako všeobecný pojem, který do sebe zahrnuje několik znamých přístupů, např. vodopádový nebo spirálový. Společně s vývojem nástrojů, zrodem nových jazyků a jiných programovacích paradigmat, procesem modernizace a technologickým pokrokem vznikaly nové modely vývoje v rámci SDLC odpovídající současným klientským požadavkům měnicích se v reakci na probíhající transformace.

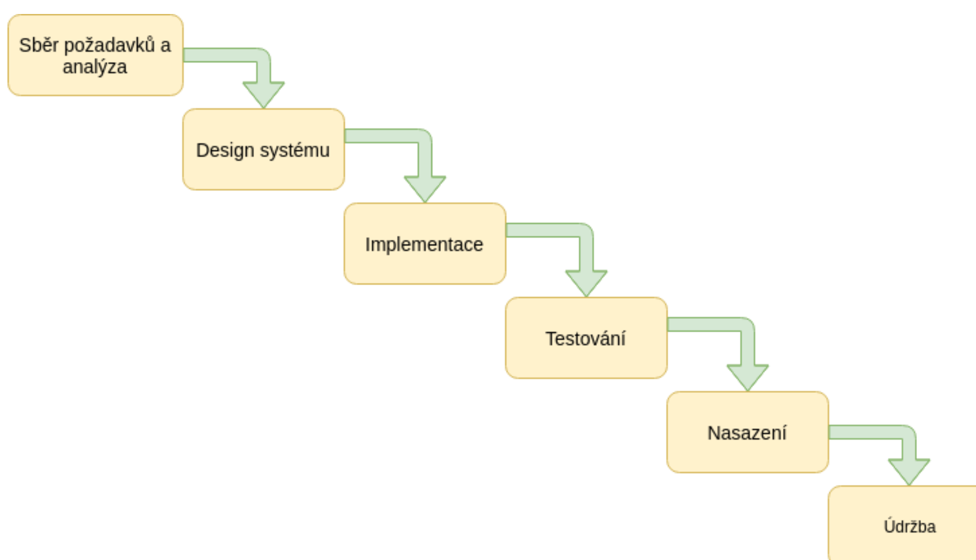
Účelem této kapitoly je stručný popis různorodých modelů životního cyklu vývoje softwaru, jejich vzájemné porovnání, historickou navázanost a zdůraznění příčin vzniku. Velká pozornost je věnována rozmanitosti agilních metodik a vhodnosti jejich použití vůči projektům s odlišným rozsahem.

4.1 Vodopádový model

Vodopádový přístup vývoje zachycuje a přesně definuje pořadí fází tvorby softwaru. Takový postup precizně vymezuje každou etapu, čímž poukazuje na to, že se během vývoje produktu nesmí spustit následující fáze, dokud nebude dokončena předchozí fáze. Tímto postupem se zaručila snadná koordinace práce všech členů vývojového týmu, jež postupovali podle předem

[16] V několika odborných článcích se vyskytuje název „metodologie vývoje softwaru“.

stanoveného plánu. Takový model zachycuje 6 fází vývoje: sběr požadavků a provedení analýzy, design systému, implementace, proces testování, nasazení projektu a údržba. Každá z nich se opakuje jenom jednou během životnosti aplikace (viz obr. 4.1). Etapa testování následuje jako čtvrtá, až po dokončení implementace, kvůli čemu neumožňuje dřívější ovlivnění testerským týmem procesů vývoje projektu a odhalení kritických návrhových, designových a implementáčních závad. Popsaný model vývoje není schopný rychle reagovat na změny množiny požadavků na software, vyznačuje se nedostatkem „pružnosti". I přestože daný přístup je jednoduchý při definicí úloh a zádání, a rovněž dokáže předpovědět práci a strávený čas v každé fázi, nic nezaručí přesnost takovýchto odhadů (zvyšuje se riziko odložení deploye, zpoždění termínu dodání) a úspešnost výsledného projektu.



Obrázek 4.1: Vodopádový model vývoje softwaru

Zmíněné nevýhody by měly mít stěžejní roli v rozhodování týmu při hledání správného modelu vývoje. Navzdory prvotnímu pocitu zastaralosti popsaných přístupů se vodopádový model aktivně používá v moderním světě. Navíc neexistuje žádný náznak toho, že by úplně zmizel z důvodu své nepoužitelnosti a neaplikovatelnosti.

Existuje několik podnětů, proč bychom se mohli rozhodnout pro použití vodopádového modelu při tvorbě projektu [26]:

- Požadavky jsou velmi dobře zdokumentované, jasné a fixní.
- Definice produktu je stabilní.
- Technologie je srozumitelná a není dynamická.

- Neexistují žádné nejednoznačné požadavky.
- K podpoře produktu je k dispozici dostatek zdrojů s požadovanou úrovní odborností.

Jako příklad logického využití uvedené metodiky může posloužit vývoj interního informačního systému letišť, vzhledem k tomu, že splňuje všechny předchozí body. Naopak se důrazně nedoporučuje aplikovat vodopádový model při tvorbě webových aplikací nedisponujících neměnnými požadavky a statickými technologiemi (samy o sobě pojmy statická, neměnná technologie a web jsou antonyma).

4.2 Spirálový model

Zřetelné nedostatky vodopádového modelu způsobily neustálé hledání nového modelu zkušenými IT specialisty, modelu schopnému napravit nevýhody klasického vývoje a umožňujícího aplikaci svých postupů na komplikované, rozsáhlé a nákladné projekty. V roce 1988 byl panem Parry Boenhmem definován spirálový přístup [15], jenž byl považován za naději, která dokáže splnit veškerá očekavání, jak ze strany vývojového týmu, tak i ze strany zainteresovaných subjektů ve výsledném softwaru.

Hlavní myšlenka popsaného přístupu spočívá ve vývoji a řádném testování prototypů konečného softwaru, jež jsou konzultovány se zákazníkem, na základě čehož se jejich množství může zvýšit (prototyp neodpovídá veškerým specifikacím zájemců, proto by se měl vytvořit nový) nebo se použije pro přípravu finální verze produktu. Rovněž se model vyznačuje rozdělením vývoje na několik menších části a provedením detailnější analýzy rizik. Jedna z úspěšných vlastností tohoto modelu je zrychlení komunikace s klientem, který bude mít k dispozici několik prototypů požadované aplikace.

Druhá až pátá fáze spirálového modelu jsou opakovatelné, jelikož proběhne provedení několika iterací etap, dokud zainteresovaná strana nepodtvrdí, že prototyp splňuje náležité požadavky a specifikace.

Fáze spirálového modelu jsou:

1. Sestavení balíčku požadavků a jeho zdokumentování pomocí provedení dlouhodobých diskuzí se zákazníky (workshopů).
2. Shromáždění požadavků pro konkretní část spirály (v závislosti na poloze ve vývojové spirále se jejich druh může lišit – obchodní, systémové atd.).
3. Design – vytvoření počátečního návrhu.
4. Tvorba a sestavení funkčních prototypů produktu. Konstrukce do sebe zahrnuje řadu prototypů, provedení testování.

5. Konzultace se zainteresovanými osobami, shrnutí možných rizik projektu (včetně zvýšení nákladů, nerealistických odhadů), čekání na zpětnou zákaznickou vazbu.

6. Finální proces testování, včasně realizované akceptační testy před předáním hotového konečného produktu klientovi.

7. Případná vyžadovaná údržba a s ní spojené minimální testování.

I přes výrazné zlepšení sady vývojových přístupů (rychlejší testování, větší důraz na analýzu a řízení rizik, předložení klientovi několik prototypů produktu a zajistění lepší reakce na změny), spirálový model obnáší i poměrně velké nevýhody:

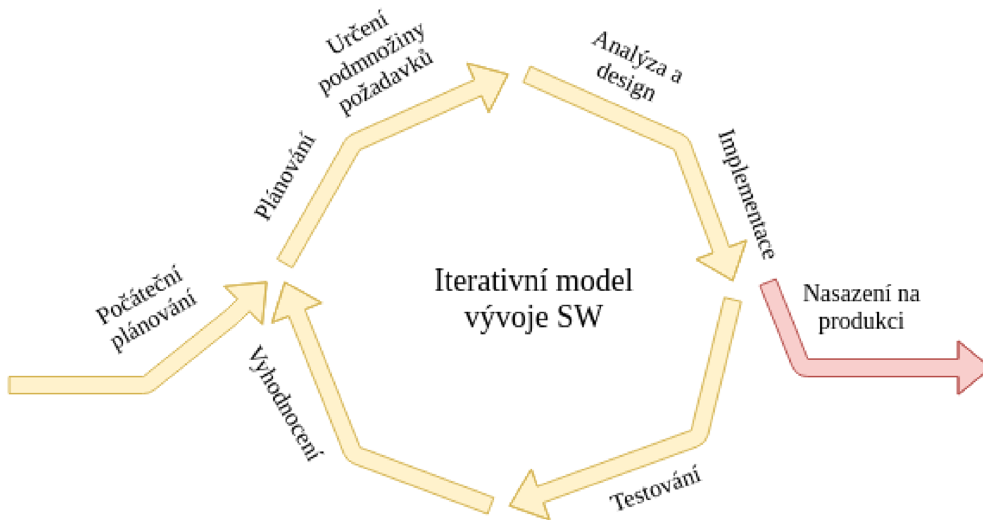
- Pokrok ve vývoji aplikace obnáší zvětšení nákladů. Permanentní růst počtu prototypů nebo nespokojenosti klienta zvyšuje riziko toho, že se projekt nedočká své finální verze. Problém spojený s nerealistickými odhady na realizaci a dodání, špatně vytvořené časové diagramy.

- Komplikovaný management projektu.

4.3 Iterativní model

Historický vznik iterativního modelu vývoje byl reakcí na otázky a problémy objevené ve dvou již dříve existujících modelech – vodopádovém a spirálovém. Nalezeným východiskem a hlavním objevem daného přístupu bylo „rozdělení velkých monolitických vývojových projektů do menších, snadněji řízených iterací.“ [15] V porovnání s vodopádovým modelem, v iterativním modelu neexistuje snaha shromáždit a zanalyzovat všechny existující požadavky na software v počáteční fázi, ale jenom ty, jež jsou potřeba během konkrétní iterace, na jejímž konci se vždy provede vypuštění (release) verze projektu. Proces vývoje aplikací a systémů je důkladný a postupný, má inkrementální charakter, jelikož každá nová verze softwaru oproti té předchozí do sebe zahrnuje implementace dalších funkcionalit a úpravy těch minulých. Proces se nezastaví, dokud nebude vydána finální verze splňující veškeré požadavky klientů.

Každá verze produktu by měla projít několika fázemi vývoje: sběr požadavků (určení realizovatelné podmnožiny z celé množiny požadavků), design a implementace, testování a vyhodnocení. (viz obr. 4.2) Díky tomu, že všechny iterace iterativního přístupu představují minivodopád, model si zachoval výhody klasického vývoje: přesný, pochopitelný plán a snadnou koordinaci práce všech členů týmu. Zároveň byl doladěn přístup klientů k aplikaci díky vydání několika verzí produktu. Je povinností zmínit, že rychlost reakcí na změny požadavků či hlašené chyby je zřetelně pomalá, nalezneme zde viditelné zlepšení v porovnání s klasickým vývojem, ale i přesto takovéto tempo není dost často vyhovující.



Obrázek 4.2: Iterativní model vývoje softwaru

Hlavní vlastnosti projektů, ve kterých se doporučuje použítí iteračního modelu vývoje: [26]

- Většina požadavků musí být definována; některé funkce nebo požadovaná vylepšení se však mohou časem vyvíjet.
- Je čas na omezení trhu.
- Používají se nové technologie, vývojový tým se učí při práci na projektu.
- Zdroje s potřebnými zkušenostmi nejsou k dispozici a plánují se použít na základě smlouvy pro konkrétní iterace.
- Existují některé vysoce rizikové rysy a cíle, které se mohou v budoucnu změnit.

Díky podobnému přístupu má proces testování několik výhod oproti dřívějším modelům:

- Je snadnější, pochopitelnější, má přesně definující rozsah testovaných funkcionalit.
- Nový testovací přístup je schopný za kratší dobu nalézt chyby aplikace, jež budou včas opravené.
- Použití případů užití zachycující interakci mezi aktérem/aktéry a systémem, zpřehledňují konající se procesy, čímž napomáhají testerskému týmu k jejich pochopení a k jednoduššímu psání testovacích případů a scénářů.

Proces testování je pružnější než byl předtím, psání testů začalo být důkladně promyšlenou činnosti kvůli novému požadavku na rozšiřitelnost a rozvoj.

4.4 Agilní model

„76 % všech softwarových projektů se nedostaví včas kvůli rozpočtu nebo nespokojenosti zákazníků.“ [15] Takováto otřesná statistika vyvolává poměrně velké množství otázek týkajících se předpokladu expertů, že žádné existující metodiky vývoje nejsou relevantní a nedokážou poskytnout jistotu úspěšnosti převážně menších projektů. Z toho důvodu během poslední dekády 20. století na scéně vznikají nové metodiky schopné se rychleji přizpůsobit požadavkům projektů díky svému pružnému a flexibilnímu přístupu k vývoji. Model a jeho principy, na jejichž základě se vytvořily dané metodiky, byl přesně definován až v roce 2001, a díky své povaze dostal název agilní. Taková struktura by měla ztělesňovat nejlepší vlastnosti iterativního a inkrementálního vývojového přístupu a zároveň provést několik zásadních změn, aby se agilní model podařilo aplikovat na menší projekty.

Hlavní koncepty agilního vývoje jsou definované v agilním manifestu, jenž je rozdělen do následujích bodů: [27]

- **Jednotlivci a interakce** před procesy a nástroji.
- **Fungující software** před vyčerpávající dokumentací.
- **Spolupráce se zákazníkem** před vyjednáváním o smlouvě.
- **Reagování na změny** před dodržováním plánu.

„Jakkoliv jsou body napravo hodnotné, bodů nalevo si ceníme více.“ [27]

Rovněž bylo zformulováno 12 principů, stojících za agilním manifestem, jež přináší nový pohled nejenom na proces vývoje produktů, ale i na roli každého členu týmu, spolupráci se zainteresovanými osobami, zvýšení pracovní motivace, rychlé řízení změn a dokumentaci. Je potřeba poznamenat, že velký důraz se klade na schopnost týmu k samoorganizaci.

Inovační pohled přinesený agilním modelem spočívá ve vývoji softwaru v kratkých časových intervalech, iteracích (od 1 do 4 týdnů), výstupem každé z nich by měla být inkrementační část nové funkcionality, jež i přesto, že ne každá verze je určena pro nasazení do produkčního prostředí, by měla být předložena zainteresovaným osobám. Iterace v agilním modelu zahrnuje do sebe několik fází vývoje:

1. plánování
2. analýza požadavků
3. design

4. implementace
5. testování
6. nasazení
7. hodnocení

Spolu s transformací podstaty několika procesů, agilní metodiky zasahují do týmové reprezentace, vznikají nové role.

Týmový lídr[17] – zajišťuje projektové řízení, komunikaci na projektu. Popsaná pozice vyžaduje především managerské dovednosti, týmový lídr nemusí mít technické znalosti.

Člen týmu – neoddělitelná součastka celku, ve kterém účastníci spolupracují z důvodu existence společného cíle – tvorby kvalitního softwaru splňujícího klientské požadavky. Členové týmu jsou zodpovědní za technickou část projektu, od modelování, návrhu a implementace, k testování a nasazení. Prvotní návrh agilního modelu úplně nerozděloval členy na vývojáře a testery, poněvadž v idealistických představách jeho tvůrců byly tyto role zaměnitelné. Popsaná praktika se v reálném životě neosvědčila, proto ve většině agilních týmů pozorujeme striktní rozdělení mezi danými pozicemi.

Projektový manager – provádí komunikaci se zákazníkem, je jeho zástupcem v týmu. Jednou z jeho zodpovědností je údržba balíčku, seznamu prioritních úloh. Hraje roli zdroje informací na projektu, v případě potřeby pomáhá s rychlým rozhodováním.

(Stakeholders) – zainteresované osoby (přímý či nepřímý uživatel, investoři apod.)

Výbuch popularity různorodých agilních metodik, jež později podrobněji popíšeme v rámci této kapitoly, neměl hranice. Dodnes i přes velkou vlnu skeptických názorů lidí blízce s nimi seznamenými, narazíme na touhy poměrně velkého množství firem zavést daný model do svých projektů. Agilní model přináší naprostou změnu myšlení a spolu s ní hodně výhod, jako je například význačná rychlost oproti ostatním modelům a rychlá reakce na změny díky svým kratkým iteracím a vnímání zákazníků jako součásti týmu. Rovněž přichází s novou praktikou – kontinuální integrací (Continious integration), zaměřenou na integraci vývojáři provedených změn do sdíleného úložiště s účelem jeho nasazení a následujícího testování (rychlejší proces testování => rychlejší nalezení existujících chyb rychlejší oprava).

Tester nebo člen týmu věnující se testování softwaru by si měl uvědomovat, že jeho pozice v rámci agilního vývoje obnáší: [4]

- pojímání toho, co by se mělo udělat (mít představu o provedených změnách, přesně definovat prioritní úlohy a nastavit hranice jejích rozsahu)

[17] SCRUM mater nebo vedoucí projektu, pojmenování záleží na konkretní metodice.

- být seznámen s testovacím prostředím (každé testovací prostředí má svoje specifické vlastnosti)
- věnovat většinu svého času komunikaci
- být připraven se vyjadřit
- být minimalistou (správný management a neustálé přemýšlení o tom, jak minimalizovat práci pro dosažení největších výsledků)

Stejně jako každý model, agilní obnáší i sadu nevýhod, se kterými by měl počítat každý tým rozhodující, zdali zavést pružný model vývoje do svých projektů.

Vyjmenujeme pár z nich:

- Vytvoření minimální dokumentace produkuje vysokou závislost dalšího vývoje na konkretním členu týmu (je potřeba počítat s bus faktorem)[18].
- Je nutností mít silného vlastníka produktu.
- Interakce se zákazníkem řídí další vývoj, který po něm vyžaduje nejenom finanční ale i časové investice. Zainteresované osoby by měly mít přesně zformulované požadavky a vize.
- Náročná integrace nových členů týmu do projektu kvůli nedostačujícím dokumentacím.
- Agile není univerzálním řešením všech projektových problémů.
- Agile v porovnání s vodopádovým řešením ztrácí predikovatelnost času a rozsahu práce.

Vytkneme vlastnosti nejpopulárnějších agilních metodik.

4.4.1 SCRUM

Jedna z nejpopulárnějších a světově rozšířená agilní metodika založená na přírůstkovém a iterativním přístupu vývoje softwaru, jejíž název a základní koncept vznikl v jiném odvětví, jež na první pohled nemá nic společného se softwarovým vývojem, dokonce ani na druhý ani na třetí, jelikož ho přebírá z britské sportovní hry – ragby, kde taktika SCRUM hraje jednou z klíčových rolí. Stejně jako ve hře, softwarová SCRUM metodika používá týmový přístup k vývoji, zakládá se na flexibilitě a respektuje seznam připravených hodnot.

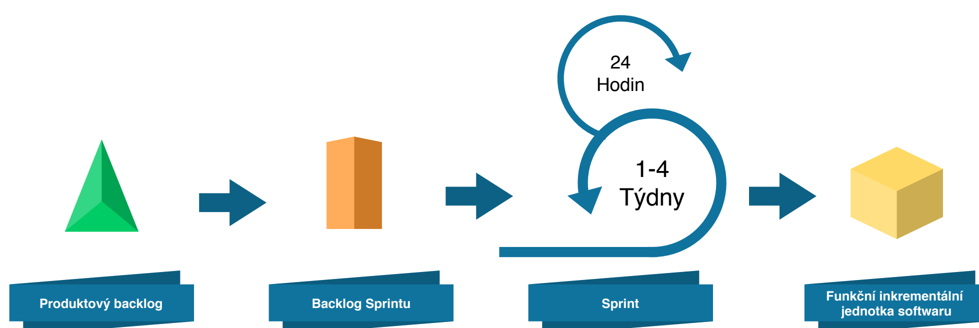
Tato metodika stanoví 3 hlavní role v celém SCRUM týmu (minimální množství, které dokáže zajistit optimální interakci):

[18] Bus faktor – počet vývojářů, jejichž náhlý odchod způsobí krizi na projektu (zvýšení nákladů, úplné zastavení)

- SCRUM master nebo kouč a trenér týmu. Jeho hlavní úlohou je sjednocovat všechny členy skupiny a dohlížet na dodržování stanovených hodnot. I přesto, že se setkáme s obrovským množstvím stížností a kritiky vyplývající z nepochopení role „mistra“, je v našem zájmu si uvědomit, že jeho práce není vždy pozorovatelná, jelikož se nejedná o přímý vliv na vývoj (SCRUM master není ve styku se softwarem celodenně), ale o zdánlivé nepatrnosti (odstranění přepážek ve vývoji, zajištění vysoké motivace týmu k dosažení pokroku), jež mohou ovšem mít velký impakt na výsledek projektu.

- Vlastník produktu (product owner), je název pozice projektového managera v rámci SCRUMu. V případě této metodiky klíčová část jeho práce tkví v sestavení *User Story*[19] vytvořených na základě představ a požadavků klientů.

- Vyvojářský tým.



Obrázek 4.3: Procesy ve SCRUMu

Všimneme si, že ve SCRUMu neexistuje nadřízená nebo podřízená osoba, poněvadž záměr každého z účastníků procesu vývoje spočívá v úspěšném dokončení projektu. Koncept týmového přístupu, kde nikdo nepreceňuje svoji důležitost a práce každého jednotlivce se vysoce cení, zaručuje produktivitu takovéhoto vývoje a sjednocuje jeho účastníky. Ze všeobecného hlediska metodika SCRUM může připomínat spíše psychologickou doktrínu, což má velké pozitivní následky, neboť se počítá s tím, že jednotlivci pracující na softwaru jsou pouze obyčejní lidé.

Je důležité zdůraznit, v jakých časových intervalech probíhá příprava a dodání nových potenciálně softwarových použitelných inkrementů, což vyjadřuje pojem **Sprint**, rovněž převzatý z ragby. Je stanoveno, že Sprint trvá od týdnu do měsice (viz obr. 4.3). Všeobecnou praktikou je zavedení dvoutýdenního Sprintu. Rovněž se ve SCRUMu setkáme i s dvěma novými pojmy:

[19] Popis funkcionalit systému z hlediska koncového uživatele v přirozeném jazyce.

produktový backlog nebo sprint backlog. **Produktový backlog** je množina, seznam požadavků na projekt, jenž není roztříděn a úspořadán na základě priorit. Vybraná část požadavků, která by měla být implementována v budoucím „inkrementu“ softwaru (**backlog sprintu**) bude předložena skupině zainteresovaných osob. Backlog sprintu, produktový backlog a produktový inkrement tvoří množinu artefaktů SCRUMu. **Cíl sprintu (sprint goal)** je výsledkem společného jednání vývojového týmu a vlastníka produktu, jehož naplňění by mělo být důležitější, než realizace úloh z backlogu sprintu.

SCRUM předpokládá a předepisuje 3 základní události, jejichž zavedení a dodržování pomáhá týmu v údržování vývojového procesu v rámci metodiky:

- Planování sprintu (doba jednání záleží na délce jedné iterace) – počáteční fáze SCRUMu. Jedná se o jednání všech členů týmu z důvodu definice rozsahu práce, určení priorit, hlavního cíle sprintu a časové náročnosti úkolů v novém sprintu. Hodnocení jednotlivé úlohy probíhá v tzv. „člověkodnech“[20] nebo ve speciálních příběhových bodech (story points).[21] Je potřeba zmínit, že by se stanovený backlog sprintu podle základů SCRUMu neměl měnit.

- Každodenní SCRUM, kterému se říká v rámci dané metodiky stand up – schůzky o délce maximálně 15 minut, jehož účelem je sdílení informací mezi jednotlivci o stavu řešených úkolů a dosažených výsledků, popis vzniklých problémů a předložení seznamu úkolů, které plánují vyřešit během jednoho dne.

- Sprint review – je úmálost ukončující iteraci a zároveň druh schůzky, jejímž primárním účelem je prezentace dosažených výsledků zákazníkovi s cílem obdržení zpětné vazby. Zuzana Šochová, autorka knihy „The Great ScrumMaster“ tvrdí [34], že v utopickém agilním světě místo prezentujícího může zaujmout libovolný člen týmu, není to povinností vlastníka produktu. V praxi se seznamíme s přístupem, že funkcionality prezentují členové týmu, jejichž podíl na vývojí byl největší.

Dost často se opomíjejí dva zbylé druhy schůzek z důvodů malé vyznamnosti vůči rozsahu projektu (v případě **Sprint Dema**) nebo z důvodu splynutí se zmíněnými údálostmi (Retrospektiva sprintu – jednání o povedených a neúspěšných fázích vývoje, diskutuje se o možných řešeních naskytnutých problémů – v některých případech se koná společně se sprint review).

Každá úloha ve SCRUMu by měla projít od plánování přes design, implementaci, testování až ke schválení. V některých sférách je možné narazit na kritiku této metodiky z důvodu podpory multitaskingu, který neomezuje počet aktuálně zpracovávaných nebo nových zádání. Proto dochází k tomu,

[20]Man-day – jednotka ukazující, kolik práce vykoná jeden člověk za jeden den.

[21]Story point je relativní druh odhadů a způsob hodnocení obtížnosti uživatelského příběhu týmem.

že se v mnoha případech nestihne zpracovat během sprintu většina z těchto úloh.

4.4.2 Extrémní programování (XP)

Extremní programování (XP) je agilní metodikou vývoje softwaru zaměřenou na zvýšení jeho kvalitz a zlepšení života vyvojového týmu.

Důvody způsobující takovouto úspěšnost agilní metodiky vývoje popsal Don Wells:

- Zdůraznění spokojenosti zákazníků.
- Zdůraznění týmové práce.
- Zlepšuje softwarový proces pěti různými způsoby: pomocí komunikace, svou jednoduchostí, zpětnou vazbou, respektem a odvahou.

Rovněž je podstatné poznamenat specifické principy a „zákony“ tvořící kostru extremního programování. Jednou z nejdůležitějších charakteristik XP je rychlá a neodložená reakce na změny požadavků. Totéž bychom mohli říct o nepřetržité snaze zachovat jednoduchost návrhu (vyvarujeme se zbytečností a zaměříme se pouze na nezbytné funkcionality), jelikož chyby s ním spojené patří k těm nejnákladnějším z celého rozsahu.

Pojem *odvaha* vznikl v terminologii softwarového inženýrství konkretně s příchodem XP. Pro stručnou a zjednodušenou představu uvedeme, že pouze statečný programátor v zájmu celého týmu dokáže zahodit svoji dlouhodobou práci, jež podle jeho představ nesplňuje postatu úlohy či laicky řečeno nefunguje.

Vzájemný respekt kolegů v týmu je jádrem dobré komunikace zvyšující produktivitu, efektivitu, kvalitu kódu atd.

Extremní programování prosazuje sadu postupů vývoje softwaru, díky kterým je tato metodika taková svérázná:

- Párové programování – úplně odlišná a z některých pohledů „divoká“ agilní technika, kde se psání aplikace realizuje dvěma programátory, jež se permanentně střídají ve psání a pozorování, na jednom počítači. Takový příklad spolupráce nejenom okamžitě zlepší kvalitu kódu, ale napomůže vzájemné vyměně zkušeností, a hlavně redukuje případy prokrastinace, čímž zefektivní pracovní přístup a morálku.
- Všichni členové týmu tvoří jediný celek.
- Jeden ze způsobu dokumentace projektu je použití uživatelských příběhů (vytvořené zákazníkem) popisujících chování aplikace z pohledu koncových uživatelů.
- Týdenní cyklus (weekly cycle) je název pro iteraci v dané metodice.

- Použití statické analýzy ve formě code review a refaktorování.[22]

- Programování řízené testy (TDD) – nová inovační praktika, která stanovuje nový podklad k vývoji funkcionalit pro aplikace. Psaní jednotkových testů jakožto počáteční fáze vývoje napomůže programátorovi předejít dualitě nebo špatnému porozumnění zadání. Daný přístup bude lépe posaný v následující kapitole.

4.4.3 Kanban a Lean

Celá metodika **Kanban** vznikla díky velkému přínosu japonské továrny Toyota do výrobního procesu díky modernímu konceptu a pricipům. Termín „kanban“ vznikl jako spojení dvou slov „kan“ – v překladu znamená viditelný, a „ban“, což je označení v japonském jazyce pro tabuli. Takováto metodika se používá k redukci počtu nedodělané práce. Jak naznačuje název, kanban se zaměřuje na vizualizaci práce s tabulkami (buď fyzickými nebo elektronickými). Každý úkol prochází několika fázemi nový/přiřazený/vrácený, design, probíhající práce (WIP – work in progress), testování, otestováno, schváleno. Stavy vymezuje pracovní postup (workflow), který definuje pro svoje potřeby každá firma zvlašť. Každá fáze, v níž se může nachazet úloha v takových tabulkách, má svůj sloupec.

Vzhledem k tomu, že kanban splňuje základní principy agilního modelu, omezíme se pouze na popis odlišností a specifik, jež ho odlišují na pozadí ostatních metodik:

- Kanban nemá předem určenou dobu iterace, ta může být proměnlivá, stejně jako i seznam požadavků, jenž se tým zavazuje zpracovat v rámci iterace a jenž se během iterace mění. Díky této vlastnosti a schopnosti jednoduše měnit priority úloh, je kanban mnohem pružnější než populární SCRUM. Je potřeba zmínit, že tým samostatně stanoví, kdy je potřeba aktuální iteraci ukončit a začít novou.

- Neexistuje žádné hodnocení úlohy podobné tomu, se kterým jsme se potkali v rámci SCRUMu. Jedinou zásadní metrikou je **délka cyklu** (cycle time), což je měření času, rozdíl mezi časem ukončení a začatkem práce nad úlohou. Účelem kanbanu není dokončení sprintu, ale vyřešení zadání.

- Inovační princip omezení nových nebo rozpracovaných (WIP) úloh přiřazených členovi týmu, napomáhá rychlému procházení zadání všemi fázemi označenými na kanbanské tabuli. V praxi se za optimální volbu omezení považují 3 úlohy.

[22] Zlepšení kvality kódu neovlivňující chování aplikace ale podporující jeho udržitelnost. Jedná se o proces dodržování nejlepších programovacích praktik a zavedených standardů.

Metodika **Lean** se stejně jako Kanban zrodila v prostorách tovaren patřících koncernu Toyota v Japonsku. „Lean je způsob uvažování – mentální model, popisující, jak funguje svět".[28] Daná metodika se více používá v průmyslových odvětvích, než ve vývoji softwarých produktů.

Hlavní postuláty můžeme shrnout do sedmi stěžejních bodů: [29]

- Odstranění „odpadů" – vádné, nepoužitelné části kódu apod.
- Budování kvality.
- Vytváření znalostí – vytváření dokumentace.
- Rozšíření práv a možností členů týmu.
- Odkladání povinností – nalezení optimálního času sběrem potřebných informací pro správné rozhodnutí.
- Rychle poskytování.
- Respektování lidí.
- Optimalizace celku.

Hlavní princip leanu je snaha o nekonečné zdokonalení v detailech a zvýšení produktivity. Tým, který zavedl lean metodiku vývoje projektu, se vždy orientuje na provedení velkého množství testů. Existuje několik názorů, že lean je odlehčenou formou agilního modelu, která přináší svoje vlastní nástroje jako VSM (Value Stream Map).[23]

4.4.4 Dynamic Systems Development Method (DSDM)

Metodika vývoje založená na principech inkrementálního a iterativního vývoje, kde se největší důraz klade na účast v procesu vývoje uživatele/spotřebitele. „DSDM vyjadřuje kompatibilitu s méně agilními průmyslovými konstrukcemi",[31] má široký a různorodý seznam aktivit a velkou sadu artefaktů. DSDM je založen na 9 stěžejních principech: „včasně dodat, zaměřit se na obchodní potřeby, nehledat kompromis s kvalitou, sestavovat přírůstkově od firemních zákaladů, vyvíjet iterativně, průběžně a jasně komunikovat, demonstrovat kontrolu."

DSDM zavádí nové praktiky, klíčové popišeme podrobněji:

- Timeboxing – pevně stanovené časové období, délka kterého trvá od dvou do čtyřech týdnů. Na konci vždy proběhne hodnocení úspěchu produktu na základě stavu dokončení softwaru jako celku ale ne množiny vykonaných úloh.

  Libovolný timeboxing se skládá ze třech stěžejních etap a dvou schůzek na začátku a konci časového intervalu:

[23] Proces vizualizace kroků vývoje produktu pro jeho předání zainteresovaným osobám.

1. Kick-Off – společné jednání všech členů týmu s účelem porozumění cílu.
2. Investigation – proces výzkumu a analýzy stanovených požadavků, jež odhadem trvá 10-20% celého timeboxingu. [31]
3. Refinement – proces zahrnující do sebe vývoj a testování, jež se dokončí opakovanou kontrolou. (60-80% času)
4. Consolidation – provedení kontroly implementováným a vyvinutých na základě dohodnutých požadavků pomocí akceptáčních kriterií.
5. Close-Out – záverečná schůzka, předání výsledků práce.

- MoSCoW je způsob definice a pochopení projektových priorit a požadavků, rozdělení do 4 skupin:
  - **M**ust Have
  - **S**hould Have
  - **C**ould Have
  - **W**on't have

4.4.5 Feature-Driven Development (FDD)

Vývoj řízený užitnými vlastnostmi daného softwaru (FDD) je druhem agilního modelu, jehož vznik přisuzujeme roku 1999, neboť byl poprvé popsán v knize „Java Modeling In Color with UML". Ze stejných důvodů jako v případě SCRUMu se používájí uživatelské příběhy napsané vlastníkem projektu, FDD se řídí projednanými a stanovenými požadavky. Vývojáři se dělí na dvě odlišné skupiny podle jejich postavení a pracovního zaměření: „vlastníci tříd" a „hlavní programátoři", což danou metodiku výrazně odlišuje od XP. Jedná se o modelový přístup, jenž se dělí na 5 základních fází (první dvě fáze se konájí pouze na počátku projektu, během 0-té iterace): [30]

1. Tvorba doménového modelu, jehož podstatou je pochopení nosných pilířů budoucí aplikace/systému. Výstupem je objektový model systému a seznam zápisků či poznámek.

2. Definice množiny požadavků a funkcionalit, jejich logické roztřídění.

3. Plánování na základě funkcionality, během kterého se zformuluje pořadí vykonání procesů vývoje, přiřadí se sada funkcionalit nebo tříd konkretním programátorům (převezmou roli vlastníků tříd).

4. Design podle funkcionalit. Pro každou funkcionalitu je vytvořen designový balíček (design package). Vznikají sekvenční diagramy, jež doplňují „mezery" objektového modelu. Společně s hlavními programátory se řeší, jaká podmnožina funkcionalit by se měla implementovat v rámci jedné iterace.

5. Sestavení podle funkcionalit. Po provedení testování se každá úspěšně vyvinutá funkcionalita dostane do centrálního úložiště.

Stejně jako každá metodika FDD počítá s tím, že se seznam požadavků průběžně mění. Modely se stejně jako aplikace vyvíjí s každou novou iterací. Účast a vliv zainteresovaných osob na vývoji softwaru je nezbytný. FDD defenuje 6 různých rolí: hlavní architekt, doménový expert, hlavní vývojář, tester, vlastník třídy a vedoucí vývoje. Měli bychom podotknout, že každý jednotlivec může plnit několik funkcí najednou.

Kapitola 5

Testovaní webových aplikací v agilních metodikách a automatizace procesů

Vzhledem k zásadním odlišnostem mezi agilními metodikami a vodopádovým, spirálovým nebo jiným modelem, je potřeba zdůraznit, že testování v rámci agilního vývoje má svá specifika, která ho činí unikátním. Přechod z tradičních hodnot a pochopení nových principů testování je dosti obtížná a komplikovaná činnost, jíž není každá firma schopna zorganizovat, proto s každým rokem získávají větší popularitu externí agilní koučové.

V závislosti na rozsahu budoucí webové aplikace a požadavcích, jež klade tým na zvolený vývojový framework, se používají následující agilní metodiky: SCRUM, XP, DSDM, FDD a lean, přičemž dle [33] se pro vývoj kvalitního produktu v případě malých softwarových firem doporučuje používat XP a SCRUM.

Jednou z nejdůležitějších vlastností testování v agilním týmu je vnímání libovolného problému jako překážky pro každého jednotlivce nezávisle na jeho specializaci. Koncept zainteresovanosti ve vyřešení problémů jedním společným celkem (týmem) je ta nejpodstatnější myšlenka, jež by měla utkvět v hlavě každého agilního testera. Máte stejnou hodnotu v týmu jako vývojář či architekt a jste jedním z týmu, jenž zaručuje kvalitu vyvíjené aplikace.

Nové funkcionality, libovolné frontendové či backendové změny mohou být okamžitě otestovány pomocí akceptačních testů po jejich implementaci a přidání do centrálního úložiště zdrojového kódu. Není potřeba čekat na dokončení celého projektu, pokud existuje možnost nabídnout testerovi částečně hotovou práci a dostat rychlou zpětnou vazbu. Tvorba libovolného testu by měla začínat nejenom čtením odpovídající dokumentace (uživatelských příběhů, požadavků apod. v závislosti na použité agilní metodice), ale společným jednáním s ostatními členy týmu o sestavení nejlepšího plánu a přístupu. Doporučuje se

použití speciálních nástrojů pro správu a organizaci testovacích procesů jako TestLink[24], testRail[25] nebo Zephyr[26].

Definice hotového stavu (Definition of Done) patří výhradně do terminologie SCRUMu. Vzhledem k tomu, že na konci každého Sprintu je připravena nová inkrementální jednotka, jež se dostane ke kontrole zákazníkovi, vývojový tým by měl předem definovat, jaká „očekávaní“, kritéria a požadavky musí splnit výsledná část produktu. Z toho důvodů se používají kontrolní seznamy (checklists), jež dokážou vyhodnotit stav různorodých artefaktů nebo aktivit ve SCRUMu. Dle [1] má každá testovací úroveň, uživatelský příběh, funkcionalita, iterace, nebo dokonce nasazení svůj zformulovaný DoD.

V této kapitole projednáme stěžejní koncepty a vlastnosti testování, zdůrazníme vhodnost použití předem zmíněných testovacích technik z pohledu webových aplikací a zodpovíme si na otázky týkající se automatizace testovacích procesů v rámci vývoje a údržby webových produktů.

5.1 Aplikace testovacích technik v agilních metodikách vývoje

Z dříve popsaných testovacích technik se jich většina dá aplikovat na testování libovolné webové aplikace vyvinuté pomocí agilní metodiky. Ale i přesto každá z metodik má svoje vlastnosti a specifičnosti vývoje, díky čemu nabízí kombinaci nových technik, jejichž využití ve starších modelech neexistuje.

Všechny dynamické metodiky black box testování by se daly automatizovat pomocí psání UI nebo API testů. Psaní testů stejně jako vývoj probíhá na základě připravených uživatelských příběhů, popisujících **akceptační kritéria**[27] a jak funkční, tak i nefunkční chování webové aplikace, s výjimkou dynamických testů na základě zkušenosti, jež se vytváří během provedení testování a v žádné agilní metodice se neautomatizují, poněvadž by tak ztratily svoji podstatu. V testech mohou být použité tradiční testovací techniky, jako rozdělení na třídy ekvivalence, analýza hraničních hodnot a testování přechodů stavů.[1] Spolu se zrodem a vývojem agilních metodologií vznikají speciální testovací metodologie, které jsou dost často aplikovatelné v rámci aginího vývoje:

- Programování řízené testy (Test Driven Development nebo TDD)
- Programování řízené akceptačními testy (Acceptance Test Driven Development nebo ATDD)

[24]http://testlink.org/
[25]https://www.gurock.com/testrail
[26]https://www.getzephyr.com/
[27]Akceptační kritéria – výsledek kolektivního jednání testerů, programátorů a obchodních zástupců, díky němuž vývojový tým bude mít kompletní představu o chování systému a funkcionalit, které později zvaliduje zástupce zainteresovaných stran.

- Vývoj řízený požadavky na chování (Behavior Driven Development nebo BDD)

5.1.1 Programování řízené testy (TDD)

Daná pokročilá technika se nejčastěji používá v případě extrémního programování, dokonce patří k jeho základním principům. Rovněž programování řízené testy se neomezuje pouze na XP metodiku. Koncept takového vývoje aplikace spočívá ve změně vývojového přístupu, kdy celý tým dává větší prioritu promýšlení detailů a napsání testů kontrolujících novou funkcionalitu ještě před její implementací. Jedná se o psaní jednotkových testů, kde velkou roli hraje pokrytí kódu. Během plánovací schůzky se s celým týmem diskutuje jejich obsah, aby se vývojáři, testeři a manageři vyvarovali neúplnostem nebo nejasnostem vznikajícím při odlišném pojímání popisů požadavků. Na začátku se vytváří test, spuštění kterého vyvolá stoprocentní selhání. V další fázi proběhne implementace funkcionality nebo její změna, vůči níž byl jednotkový test napsán/opraven. Opakované spuštění sady takovýchto testů by mělo zaručit, že návrh a design nové funkcionality proběhl úspěšně. Poslední fáze vývoje jednotkových testů se věnuje jejich refaktoringu.

V agilních metodikách vývoje je psaní dokumentace poměrně omezenou činností, jejíž výstupem je množina textů a diagramů pouze nejpodstatnějších funkcionalit. Přednostní tvorba jednotkových testů může posloužit jako plnohodnotná dokumentace projektu vzhledem ke své udržitelnosti a aktuálnosti v porovnání s texty, na psaní kterých se ani dost často nealokuje čas během iterace. Druhá výhoda využití TDD může spočívat v tom, že po spuštění jednotkových testů, ještě před předáním nové funkcionality testerovi, má programátor představu, zda jeho změny ovlivnily dřívěji navržené funkcionality. To znamená, že daná technika šetří čas a odvedenou práci. Rovněž sada jednotkových testů funguje na principu regresních testů (což rozhodně neimplikuje, že je dokáže vyměnit). Další výhoda TDD spočívá v tom, že vývojový tým navrhuje API dle způsobu použití.

5.1.2 Programování řízené akceptačními testy (ATDD)

Pokus o vylepšení TDD zapříčinil vznik nové techniky, jejíž stěžejní principy zůstávají podobné jako v případě jeho předchůdce. Během workshopu předcházejícímu implementaci funkcionalit probíhá společné jednání agilního vývojového týmu s obchodními managery, jehož účel je stejný jako v případě TDD: vymezit a zdokonalit/změnit vize a popis existujících požadavků, během čehož se vytváří uživatelské příběhy. Forma nových testů může být jak manuální, tak i automatická. Testovací případy vycházejí z uživatelských příběhů. Postup jejich vytvoření není přesně stanovený, a proto mohou být vytvořeny testerem samotným anebo společně s vývojáři. Až po provedení kontroly výsledných testů se přechází k implementační fázi vývoje.

Dáno	Když	Výsledkem je
Uživatel nevyplnil žádné políčko cestovního formuláře	Klikne na tlačítko „Objednat“	Zvýrazní se povinná pole, objeví se upozornění v HTML <p> prvku

Tabulka 5.1: Použití „Given-When-Then“ šablony pro psaní testovacích případů.

Doporučuje se v počátečních etapách psaní akceptačních testů zaměřit se na tvorbu testů pozitivních, jež zkontrolují chování aplikace pro standardní vstupy. Pouze po jejich dokončení může vývojář/tester přejít ke psaní negativních případů. Ve firmách, které nepreferují zavedení automatizace testovacích případů, stále existuje možnost použití metodiky ATDD do vývoje.

5.1.3 Vývoj řízený požadavky na chování (BDD)

Vývoj řízený požadavky na chování představuje jinou odvětev „evoluce“ TDD, jejímž primárním záměrem je prohloubení komunikace mezi vývojovým týmem, obchodními managery a zainteresovanými osobami s využitím společného „netechnického“ jazyka. Seznam požadavků zpracovaný do tvaru uživatelských příběhů je použit pro tvorbu platných testovacích scénářů orientovaných na kontrolu chování aplikace.

BDD přichází s novými koncepty, jako je formule „Given-When-Then“ nebo „Role-Feature-Reason“ matrice.

Účelem formule „Given-When-Then“ je předložení šablony k psaní testovacích případů:

- **Given** (k dispozici, dáno) – konkretní scénář
- **When** (když) – se koná daná akce
- **Then** (zatím) – výsledek, dopad

Poskytneme příklad použití dané šamblony v rámci psaní testovacích scénářů webové aplikace, kde se jedná o pokus objednání zájezdu v check outu bez uvedení povinných údajů (viz tabulka 5.1).

Matrice „Role-Feature-Reason“ se používá jako šablona pro tvorbu uživatelských příběhů. Definice je následující:

- As a – jako
- I want – chci
- So that – aby, a proto

Jako zákazník cestovní agentury si **chci** objednat parkování před odletem na letišti, **abych** si mohl zaparkovat auto na přesný počet dnů, jako je délka zájezdu.

Existuje sada nástrojů, která může být použita pro úspěšnou a jednoduchou realizaci BDD jako framework Cucumber (použití angličtiny definované skrze jazyk Gherkin) nebo Speckflow, jenž z Cucumberu vychází.

5.1.4 End-to-end testování (E2E)

End-to-end testování je jedna z testovacích technik, jejímž cílem není pouze kontrola chování aplikace, ale ověřování integrace s externími rozhraními (např. REST API, SOAP API atd.) Takové testování by mělo být kompletní, rovněž může připomínat kontrolu na základě scénáře (od začátku až do konce). Vznik E2E byl odůvodněn postupným vývojem aplikací (nejenom webových) a zvýšením jejich komplikovanosti. Výsledné testy uživatelských příběhů a funkcionalit jsou tvořeny na systémové úrovni.

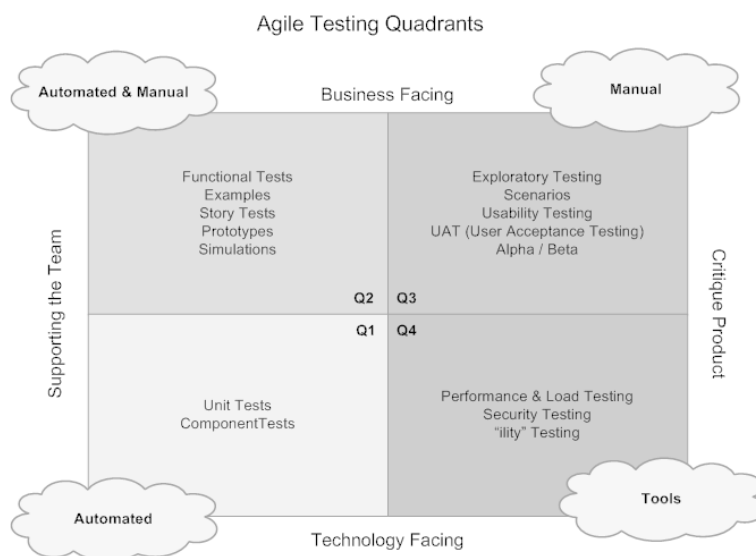
5.1.5 Exploratory testování

Exploratory testování v rámci agilního vývoje nabývá nový smysl, jehož odůvodnění spočívá v omezení času na detailní specifikaci uživatelských případů nebo v omezeném čase věnovaném analýze testů. [1] Exploratory testování se v mnoha případech kombinuje s jinými technikami testování (white nebo black box) nebo testovacími strategiemi. Hlavní podmínkou pro úspěšné vykonání exploratory testování je znalost produktu, vyvinutá intuice a kreativita. Je důležité, aby osoba provádějící testování dokumentovala vykonané kroky, aby objevená chyba byla opakovaně simulovatelná.

5.2 Automatizace testovacích procesů

Automatizace testovacího procesu je základním pilířem úspěšného agilního vývoje kvalitního softwaru. Moderní svět a jeho neustálé změny vyžadují od vývojového týmu nepřetržitou koncentraci a schopnost rychle reagovat na regulární transformace. V případě webových aplikací může zdržení pouze o několik vteřin způsobit zákazníkovi několikamilionové ztráty (např. vývoj burzové aplikace). Je z toho poměrně jednoduché odvodit, že manuální testování nevyhovuje současným požadavkům a je synonymem pojmu „ztráta času“, nicméně doteď takovýto druh testování hraje poměrně velkou roli v každé agilní metodice [16], neboť ještě nebyl vytvořen dokonalý nástroj schopný napodobit lidské rozhodování (i přes zapojení umělé inteligence do procesů [28]). Rovněž

[28] Automatické nástroje využívající umělou inteligenci (např. Applitools) jsou díky sofistikovanému algoritmu schopné porovnávat vizuální rozdíly mezi očekávaným výsledkem a reálným stavem aplikace (jedná se pouze o testování uživatelského rozhraní). Většina z nich není open source.



Obrázek 5.1: Kvadrant agilního testování [14]

uvedeme druhou nevýhodu manuálního testování, jež tkví v opakovatelnosti kroků, jako kupříkladu realizace regresních testů na konci každé iterace. Lidský faktor ovlivňuje veškeré testovací procesy, proto výsledky regresních či jiných testů ne vždy odráží aktuální stav aplikace. Díky zavedení automatizace se nejenom vyhneme podobnému druhu problému ale poskytneme i rychlou zpětnou vazbu o provedení testování a dáme testerovi čas na vykonání jiných činností, změníme zaměření jeho práce, jež bude vyžadovat větší představivost a hlubší pochopení chování koncových uživatelů webové aplikace. Jak už bylo řečeno předtím, testy, a nejenom jednotkové, jsou dobrým způsobem dokumentace projektu v případě, že se stále udržují.

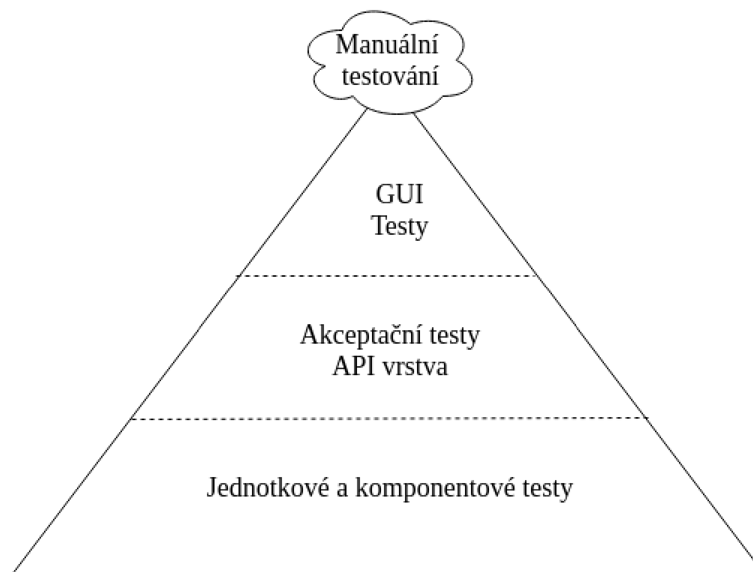
Při zavedení automatizace testů by tým měl počítat s tím, že nepromyšlená automatizace je schopná uškodit celému vývojovému procesu. Uvedeme několik příkladů běžných chyb při automatizaci testovacích procesů [32]:

- Vytvoření testovacích dat pomocí uživatelského rozhraní.
- Oddělení automatických testů (převážně UI) od vývojového procesu, neschopnost jejich provedení, dokud nebude celý systém nasazen.
- Kopírování velké části kódu a jeho redundantní opakování.
- Validace pouze viditelných prvků webových stránek, jež nezaručuje korektní kontrolu jejich chování.
- Snaha vše automatizovat.

- Testování pouze pomocí UI testů je pomalé, jak z hlediska doby psaní a údržby takovýchto testů, tak i z hlediska jejich vykonávání (spuštění prohlížeče, připojení na Internet). S nárůstem množství automatizovaných UI scénářů se čas strávený jejich provedením může rovnat času strávenému manuálním testováním.

Odpověď na otázku, jaké testovací úrovně by měly pokrývat automatické testy, poskytují L. Crispin a J. Gregory [14]. Vysvětlují ji pomocí kvadrantů agilního testování (viz obr. 5.1). Každý z nich nastiňuje, jaké druhy testů by měly projít procesem automatizace v projektu a jaké by se měly provádět manuálně nebo částečně manuálně. Z dalšího zkoumání vyplývá, že pouze třetí kvadrant (Exploratory testování, scénáře, testování použitelností atd.) by neměl podlehnout procesu automatizace až na výjimku použití automatických nástrojů pro inicializaci testovacích dat a scénářů. Použití kvadrantu agilního testování napovídá vývojovému týmu, jaké nástroje pro automatizace různorodých druhů testů na všech úrovních testování by se měly použít. Statické testovací techniky v rámci různých modelů jsou stejné. Nástroje pro jejich automatizaci byly zmíněny v kapitole 3.1. Testy využívající dynamické techniky s výjimkou testování založeného na zkušenostech se rovněž dají automatizovat v rámci agilního vývoje softwaru.

Velká množina projektů se potýká s pojmem technický dluh (algoritmicky nedokonalý způsob řešení různorodých problémů, zbytečný a redundantní kód apod.). Použití kvadrantů napomůže najít nový pohled na řešení s tím spojených záležitostí.



Obrázek 5.2: Pyramida automatizace testů

Po provedené analýze a rozhodnutí, jaké druhy testů budou v rámci aplikace automatizované, se tým zaměří na jejich rozsah. Pro daný účel se používá pyramida automatizace testů (viz obr. 5.2). Její rozdělení a velikost každé části odpovídá množství testů vytvořených v rámci každé z vrstev. Svislá osa vyjadřuje úroveň jejich abstrakce. Dolní stupeň pyramidy je základem držícím zbylé vrstvy. Nejčastější chybou při zavedení automatizace je špatný odhad rychlosti napsání různých druhů testů a nákladů spojených s nimi . Jednotkové a komponentové testy patří k nejlevnějším a rychlejším způsobům kontroly chování aplikace, jsou jádrem metodiky TDD. Při agilním vývoji produktu se tým snaží napsat co největší počet testů konkrétně dané vrstvy. Druhá úroveň pyramidy provádí kontrolu obchodní logiky, API a různorodých funkcionalit, jež nevyžadují projití GUI. Tyto testy jsou mnohem levnější, než testy kontrolující uživatelské rozhraní, ale i přesto jsou rozsáhlejší než jednotkové a potřebují náročnější údržbu a jsou o dost pomalejší (např. mohou vyžadovat propojení s DB), proto jejich množství je výrazně menší. Poslední vrstva pyramidy prezentuje ty nejdražší testy a časově náročnější testy. Jejich počet by se měl člen vývojového týmu snažit minimalizovat.

Z hlediska webových aplikací testování pouze kontrolerů nebo modelů neprokáže funkčnost produktu jako celku. Proto se využívají funkční nebo akceptační testy.

Kapitola 6

Kombinace testovacích technik na projektu vyvinutém ve SCRUMu

6.1 Seznámení s projektem

Vybraná aplikace je vyvíjená softwarovými odborníky po dobu čtyř let. Její tvorba doteď není ukončená, vzhledem k touze zainteresovaných osob o rozšíření vlivu na větším mezinárodním trhu, což znamená, že k existujícím jazykovým verzím požadované aplikace (české, slovenské, polské a německé) se v bližší době přidá i verze maďarská. Frontendová a backendová část jsou rozdělené mezi dvě firmy, společnost zajišťující backend je dodavatelem firmy poskytující zákazníkovi frontend.

Webová aplikace je tvořena pomocí agilního modelu, společně s vedením bylo dohodnuto použití metodiky SCRUM. Výsledky spolupráce všech členů týmů na daném projektu se zdají dost povedenými. Důvody jsou následující:

- Délka stanoveného sprintu je 2 týdny. V případě nespokojenosti klientů s výslednou inkrementací nasazení, proběhne v rámci dalšího sprintu, ale neovlivní (neodloží se) začátek nebo konec aktuální iterace.
- Dodržování 4 primárních druhů schůzek: plánování sprintu, retrospektivy a review, každodenního stand upu (jehož délka nepřesahuje 15 minut).
- Tým představuje samoorganizační jednotku.
- Základ produktového backlogu je seznam úloh určených během plánování sprintu.
- Testeři a programátoři spolupracují během každé iterace.

- Tým se skládá z vlastníka produktu, SCRUM mastera a vývojového týmu.[29]
- Zodpovědnost za zajištění kvality výsledného produktu se rozdělí mezi celý tým.
- Metrika vývoje – rychlost zpracování úloh, např. použití burndown grafů[30].
- Nepřetržitá komunikace se zákazníkem pro sběr nových požadavků a zajištění zpětné vazby.

Zároveň bychom měli upozornit na nedokonalosti realizace použité metodiky SCRUMu, ke kterým patří možná změna backlogu sprintu (což více přípomíná princip kanbanu) či nepřítomnost zákazníka na sprint review. Nedaří se dodržovat princip KISS (Keep It Stupid Simple) vzhledem k časté změně jednotlivců v týmu a dřívějšímu neoptimálnímu návrhu aplikace.

Za nástroj použitý pro evidenci problémů a chyb, jednodušší řízení změn a zároveň poskytující možnost vedení agilního managementu byl vybrán Jira tracking system. [31]

6.1.1 Funkční požadavky

Pro seznámení s jádrem webové aplikace poskytneme detailní popis množiny funkčních požadavků vytvořených zainteresovanými osobami, jenž je uveden v rámci uživatelských příběhů vlastníkem produktu (předložena pouze podmnožina všech funkcionalit, které jsou kontrolovatelné během provedení regresních testů).

- Vyhledávání zájezdů na základě různorodých odlišných filtrů:
  - termín
  - délka zájezdu
  - místo odletu
  - zvolená destinace
  - počet osob (dospělých a nezletilých)
  - druh stravování
  - transfer do hotelu
  - cestovní kancelář

[29] V rámci týmů existuje další rozdělení na vývojáře a testery. Firemní vize předpokládá, že každý člověk je unikátní a vyniká ve svém konkrétním oboru. Prvotřídní senior programátor nemusí mít předpoklady k testování a obráceně.

[30] Graf, který kombinuje aktuální stav s odhady a vizualizuje předpokládaný konec projektu. [34]

[31] https://www.atlassian.com/software/jira

  - speciální požadavky

- Vyhledávání pouze ubytování s použitím stejných filtrů jako v případě zájezdů.
- V listu nalezených hotelů existuje možnost třídění výsledků vyhledávání podle ceny, popularity nebo hodnocení.
- V detailu hotelů se objeví široký výběr nabídek zájezdů/ubytování odpovídající vyhledávaným kritériím.
- Přepnutí měn v české nebo slovenské jazykové verzi.
- Zadání osobních údajů v checkoutu webové aplikace. V případě objednávající osoby je povinností uvést oslovení, příjmení, jméno, adresu, telefon, email, občanství a datum narození, u spolucestujících plné jméno a datum narození odpovídající zadanému věku při vyhledávání.
- Objednávku může založit pouze dospělá osoba.
- Možnost volby druhu cestovního pojištění v checkoutu.
- Možnost přidání parkování na letišti v checkoutu.
- Možnost provedení offline objednávky (bez zaplacení).
- Možnost provedení online objednávky (platba kartou), která je omezená v případě, že zájezd je dražší než 45 000 Kč. [32]

6.1.2 Nefunkční požadavky

Poskytneme přehledný seznam nefunkčních požadavků pro detailnější popis aplikace cestovní agentury:

- Webová aplikace běžící v cloudu.
- Nasazení přes inteligentní server TeamCity pro kontinuální integraci.
- Backend napsaný ve frameworku PHP Nette.
- Aktuální verze PHP je 7.0 (během několika měsíců se očekává přechod na PHP 7.3).
- Čtyři různá prostředí pro vývoj: lokální, dvě testovací a produkční.
- Frontendová část: HTML, CSS, JQuery, Mustache.

[32] Reálný počet omezení může být mnohem větší v závislosti na typu zájezdů, počtu dětí, zdražení zájezdů, restrikcí ze strany hotelů nebo cestovní kanceláře. Logika aplikace se zkomplikovala během několika let vývoje.

- Relační databáze: MariaDB 10.
- Použití verzovacího systému Git.
- Propojení přes Web API s německými poskytovateli (Internet Booking Engine) cestovních zájezdů.
- Použití Codeception frameworku pro psaní automatických akceptačních testů.
- Asynchronní zpracování webových stránek pomocí JS knihovny Ajax.
- Aplikace poskytuje základní REST API.
- Použití Docker nástroje pro virtualizaci s účelem lokálního vývoje.
- Základem je MVC[33] architektura.
- Přehled využití a vytížení aplikace je poskytován přes Google Analytics.
- Aplikace včetně testů se vyvíjí v PhpStorm IDE.

6.2 Vytváření testovacího plánu regresních testů webové aplikace

V rámci vývoje aplikace cestovní agentury byly vytvořeny jak akceptační testy (UI), tak i jednotkové, komponentové testy, které jsou zodpovědností programátorů, a měly by se psát před vývojem nové funkcionality (TDD) nebo její opravy a spouštět se před merge requestem pro základní kontrolu, než se kód dostane na code review. Vývoj integračních API testů byl vývojářským týmem zamítnut.

Jednotkové testy (PHPUnit) pro použití techniky testování řídících struktur využívají automatické nástroje. Kontrola datového toku se provádí ve dvou formách: dynamická (díky PhpStormu) a manuální – procházení a kontrola kódu nezávislým vývojářem (code review) před mergováním do centrálního repozitáře a předání testerovi. Co se týče black box testů, používají je všechny uvedené metodiky. Exploratory testování se věnují 3 hodiny během každého sprintu. Regresní testování na projektu do sebe zahrnuje spuštění celé sady udržovaných jednotkových testů, testů komponent (programátorská tvorba) a akceptačních testů.

Účelem této práce byla tvorba veškerých regresních akceptačních testů a psaní pomocné PHP třídy pro propojení se Zephyrem pro Jiru, které se aktuálně spouští v lokálním prostředí v jednom z kontejnerů Dockeru. Tvorba regresních testů je založena na technikách testování případů užití, během jejichž vývoje byly použité kombinace analýzy hraničních hodnot a rozdělení

[33] Model View Controller – rozdělní aplikace do 3 vrstev.

Verze	Počet cestujících
CZ	1 dospělý + 1 dítě
SK	pouze 1 dospělý
PL	pouze 1 dospělý
DE	2 dospělí

Tabulka 6.1: Tabulka jazykových verzí základního průchodu aplikace a specifikace počtu cestujících.

na třídy ekvivalence. Daná realizace je proveditelná díky opakovatelnosti testovacích scénářů (stejná logika pro každou jazykovou verzi webové stránky). Regresní testy testovacího plánu mohou být vykonané jak manuálně, tak i automaticky. Regresní testy zahrnují do sebe pozitivní a negativní průchody aplikací.

První případ užití byl popsán ve třetí kapitole (viz tabulka **??**). Pro každou jazykovou verzi bude provedena korekce a specifikace bodu 12, hlavního úspěšného scénáře objednávky, kde se v případě různých jazykových verzí udává různý počet cesujících (viz tabulka 6.1).

Případ užití objednávky hotelu, vytvořený pomocí uživatelských příběhů napsaných vlastníkem produktu, je rozebrán v tabulkách 6.2 a 6.3.

Dané případy užití jsou založené pouze na pozitivním průchodu aplikací. Přidáme negativní, kde zkontrolujeme pomocí metodik technik hraničních hodnot a rozdělení na třídy ekvivalence např. datum narození cestujícího (viz tabulka 3.1) a ostatní pole v různorodých formulářích. Všechny testovací scénáře jsou vytvořeny v Test Management Systému Zephyr, odkud budou exportovány do souborů csv a přidány ke zdrojovým souborům.

6.3 Automatizace regresních testů

Automatizace uživatelského rozhraní libovolné webové aplikace je nepředstavitelná bez použití Selenium WebDriveru – specializované kolekce API, knihovny pro práci s prohlížečem. Selenium Web Driver podporuje široký výběr prohlížečů jako Firefox, Chrome, Safari a IE a je nezávislý na volbě konkrétní platformy. Tento nástroj nedokáže rozpoznat elementy webových stranek, proto pro jejich identifikaci používáme XPath[34], jméno, odkaz, *id* nebo CSS selektor. Selenium Web Driver nabízí podporu různorodých jazyků od Javy do PHP. Automatické testy pro aplikaci cestovní kanceláře byly vyvinuté v Codeception za pomoci Facebook Web Driveru, což je PHP „binding“ pro Selenium Web Driver.

[34] Syntaxe pro definici částí XML (eXtensible Markup Language) dokumentu. [35]

CZ, SK, PL, DE verze	0. Z: Přejít do zvolené jazykové verze aplikace. 1. Z: Změnit vyhledávací parametr „strava“. 2. S: Provést změnu parametru. 3. Z: Změnit vyhledávací parametr „transfer“. 4. S: Provést změnu parametru. 5. Z: Změnit vyhledávací parametr „cestonvní kancelář“. 6. S: Provést změnu parametru. 7. Z: Náhodně změnit vyhledávací parametry „speciální požadavky“. 8. S: Provést změnu parametrů. 9. Z: Kliknout na tlačítko „Hledat“. 10. S: Zobrazit seznam hotelů. 11. Z: Vybrat libovolný hotel. 12. S: Přejít do detailu hotelu. 13. Z: Zvolit vhodný termín ubytování. 14. S: Přejít do checkoutu. 15. Z: Vyplnit cestovní formulář. 16. Přidat cestovní pojištění, pokud existuje taková možnost. 17. Přidat parkování, pokud existuje taková možnost. 18. Z: Skrze tlačítko „Objednat“ provést objednávku. 19. S: Vytvořit novou objednávku.
Výjimky	10a. Chyba poskytovatele dat. Nejsou k dispozici žádné hotely. S: Zobrazí seznam hotelů neodpovídající parametrům vyhledávání. 12a. Chyba poskytovatele dat. Nejsou k dispozici žádné termíny zájezdů. S: Zobrazí seznam zájezdů neodpovídající předem uvedené délce pobytu nebo termínu. 19a. Proběhla změna ceny. S: Objednávka je uložena, ale není odeslána. 19b. Double booking. S: Objednávka je uložena, ale není odeslána. 19c. Chyba poskytovatele. S: Objednávka je uložena ale není odeslána.

Tabulka 6.2: Případ užití – scénář provedení objednávky hotelu

Název	Rezervace hotelu
Účel	Založení nové objednávky hotelu
Úroveň	Základní úloha
Hlavní aktér	Plnoletý zákazník
Stav před exekucí	Není definováno
Stav po exekuci (úspěch)	Plnoletý zákazník si úspěšně objednal hotel
Stav po exekuci (neúspěch)	Zákazníkovi se nepodařilo založit novou objednávku
Spouštějící akce	Vyhledávání zájezdu nebo pobytu
Priorita	Kritická
Frekvence použití	$\sim$ 1000 zákazníků týdně
Doba odezvy	10 sekund nebo méně
Jiní uživatelé	Žádné
Datum splatnosti	Není určen
Úroveň úplnosti	1.0 – všechna pole jsou vyplněna
Otevřené úlohy	Žádné

Tabulka 6.3: Případ užití – metadata objednávky hotelu

6.3.1 Využití Codeception

Testovací framework Codeception je ideální volba pro testování PHP aplikací, pro něž byl vyvinut, vzhledem k tomu, že podporuje všechny 3 druhy testů (jednotkové, funkční a akceptační), jež by měly být automatizované v rámci agilního vývoje. Jednou z důležitých vlastností Codeception frameworku je podpora BDD metodiky (BDD-style), což zároveň není primárním účelem Codeception, ale jeho doplňkovou možností. Testy jsou jednoduché a pochopitelné pro členy týmu s minimálními znalostmi programování, což poskytuje možnost se víc soustředit na psaní testovacích scénářů, než na jejich implementaci. Rozdělení mezi implementaci a prezentaci je klíčovou vlastností Codeception, která je umožněná pomocí high-level API, díky čemu se vytváří „čitelné“ testy s jednoduchým psaním a laděním.

Jejich spuštění automaticky otevře prohlížeč, kde by se měly vykonávat, v případě, že nebyl nastaven WebDriver do **headless** režimu[35]. Kód je čistý a jeho psaní připomíná tvorbu scénáře z uživatelského pohledu, díky čemuž ho zvládnou přečíst i lidé bez speciálního technického vzdělání.

Codeception se vyznačuje aktivním užitím page objektů, jež představují třídy, v nichž se implementují metody a alokátory logicky náležící dané konkrétní stránce. Použití objektového přístupu zčásti „znepřehlední“ kód, nicméně zaručí jeho znovupoužitelnost. Druhotným účelem page objektů je tvorba dat, interakce s nimi atd. Stejně jako psaní kódu aplikace má svůj styl, implemen-

[35]Simuluje spuštění bez grafického uživatelského rozhraní

tace Codeception testů má rovněž svoje pravidla, která jsou přesně definovaná v týmu či firmě.

Výstup spuštění testovacích scénářů nabízí detailní hlášení rozepsané krok po kroku, které jednotlivec pozoruje buď na konzoli, nebo ve formě XML nebo HTML dokumentů. Každý pád spuštění testů automaticky vygeneruje screenshot a hlášení.

Snapshoty, interaktivní konzole, použití modularity a vlastní Helpery je poměrně úzký výčet toho, jaké neomezené možnosti nabízí Codeception framework svým uživatelům.

Regresní testy se v popsaném projektu provádějí den před deployem, aby v případě nalezení kritických chyb blokujících nasazení projektu programátoři měli čas na jejich opravu. Část z nich patří k manuálnímu (testování vizuální stránky a obsahu emailů a PDF) nebo pouze částečně manuálnímu testování (kontrola reCAPTCHY, zbytek testu se provádí automaticky). Automatické regresní testy se spouštějí na lokálním vývojovém prostředí (s využitím XML feedů) v jednom z Docker kontejnerů. Používá se Google WebDriver v headless režimu, jenž je rychlejší než reálný prohlížeč, poněvadž eliminují režijní náklady na spuštění GUI. Takovýto headless režim je poměrně omezující pro testování v případě, že potřebujeme odladit prohlížeč, což náš případ ovšem není.

Page objekty, jež byly vytvořeny v rámci psaní testů, odpovídají struktuře procházení webovou aplikací pro vytvoření objednávky zájezdů: *HomepagePage, HotelListPage, HotelDetailPage, CheckoutPage, ThankyouPage.* Některé z popsaných objektů dědí od abstraktní třídy *FrontendAbstractPage.* Vzhledem ke složitosti logiky aplikace, např. v detailu hotelu po ověření ceny nemusí zbýt žádný zájezd, což vyžaduje návrat na předchozí stránku a volbu jiného hotelu. Při psaní byly využity základní znalosti programování v PHP.

Všechny testy napsané v rámci této práce se nachází ve třídě CustomerCest, kterou nalezneme v generované složce tests/acceptance. Všimneme si, že všechny třídy v dané složce mají povinnou příponu Cest. Každý napsaný test by měl být aplikovatelný pro všechny jazykové verze (CZ, SK, PL a DE), jež mají stejnou backendovou logiku, testování které by mělo do sebe zahrnout dříve zmíněné dynamické black box techniky. To znamená, že bychom mohli provést několik stejných testů s různými daty. Codeception přichází s example anotací pomocí JSON nebo Doctrine notací. V případě daných testů se rozhodlo o použití pouze JSON notace.

V rámci daného projektu se vytvářely i různorodé Helpery (rozšíření existujících modelů uživatelem). Jedním z nich je Acceptance, který nabízí kontrolu před a po spuštění každého z testů pomocí metod _before() a _after(). Hlavní objekt třídy AcceptanceTester $I může být rozšířen přidáním nových funkcionalit.

Obsah testů v CustomerCestu připomíná psaní scénářů. Nabídneme k prezentaci začátek funkce basicTouReservation, kde se přistupuje k public metodám page objektu HomepagePage. Je z kódu srozumitelné, že filtrace zájezdů je provedena na základě zvolené destinace, délky pobytu, počtu dětí a dospělých (v případě že se neliší od defaultního). Po odeslání vyhledavácího formuláře se dostaneme na seznam hotelů, zvolíme náhodnou stránku a hotel. Všechna vstupní data se načítají z objektu třídy \Codeception\Example.

```
public function basicTourTest(AcceptanceTester $I,
                ZapiHelper $client,
                \Codeception\Example $example)
  {
        $I->wantTo("Make a tour reservation for user");
        $I->amOnPage("/");

        $homePage = new HomepagePage();
        $homePage->changePage($I,
                        $client,
                        $queue,
                        $homePage::DOVOLENA,
                        $homePage::DEFAULT_TOUR_URL);

        $homePage->countFavoriteCustomerDestinations($I,
                        $client,
                        $queue,
                        $homePage::FAVORITE_DESTINATIONS,
                        8);
        $homePage->filterDestinationsByDestinationName($I,
                        $client,
                        $queue,
                        $example['country']);

        $duration = (int) $example['duration'];
        $adults = (int) $example['adults'];
        $children = (int) $example['children'];
        $homePage->filterDestinationsByDuration($I,
                $client,
                $queue,
                $duration);
        if ($adults !== $homePage::DEFAULT_AMOUNT_OF_ADULTS)
        $homePage->filterDestinationsByAmountOfAdults($I,
                        $adults);
        $homePage->submitHPForm($I,
              $client,
              $queue,
              $example);

        $hotelListPage = new HotelListPage();
        $hotelListPage->chooseRandomPage($I);
        )
  }
```

6.3.2 Propojení testů s testovacím nástrojem Zephyr pomocí ZAPI

Výsledky provedeného testování by se měly importovat do použitého testovacího nástroje/systému Zephyr (platforma pro správu testů), jenž byl vybrán z důvodů integrace s Jirou a možností přiblížit programátorský tým k testovacímu procesu. Každý test vytvořený v rámci projektu může být spuštěn samostatně nebo ve vybraném testovacím cyklu, což je pojem, jehož vznik byl iniciován firmou poskytující Zephyr a v použité testerské terminologii připomíná testovací sadu (test suite). Zephyr má značné nedostatky (jako nepřívětivé uživatelské rozhraní: nemožnost kopírování testů, exekuce jednotlivých kroků, nesmyslné přidání příloh) v porovnání s jinými testovacími nástroji jako je Test Rail nebo TestLink, jenž je navíc open source. Zephyr pro Jiru dokáže obousměrně odkazovat na úlohy popisující chybu nalezenou během exekuce a konkretní krok v rámci testovacího scénáře, což mu přidává velkou hodnotu na trhu.

Vzhledem k tomu, že chceme v reálném čase dokumentovat provedenou automatickou exekuci testů při minimálních ztrátách kapacit na dlouhodobém zadání výsledného stavu každého kroku, byla dodatečně vyvinutá krátká a jednoduchá PHP třída a několik enumů, jež provádí interakci se Zephyrem přes REST API – ZAPI. Za daným účelem byl vytvořen nový nezávislý účet pro realizaci automatického testování v Jiře.

Existuje několik stavů kroků, jejichž počet by se dal nastavit v rámci administrátorského nebo managerského postavení v Jiře. Na základě dohody se všemi testery ve firmě a stanovené dokumentaci se rozhodlo pro následující stavy:

- PASS – krok proběhl bezchybně, zápis případného komentáře
- FAIL – byla nalezena chyba, odchycena výjimka, jejíž obsah byl přidán do komentáře, byly zablokovány zbylé kroky
- BLOCKED – krok byl zablokován z důvodu nalezení chyby během dřívějších akcí
- WIP – krok je v procesu vývoje, což ovšem neznamená, že další kroky jsou zablokovány
- UNEXECUTED – krok v rámci teto exekuce zatím proveden nebyl

Na začátku každého testovacího scénáře se provede přidání vybraného testu pomocí unikátního id do předem vytvořeného testovacího cyklu s názvem „Automatické regresní testy“, díky čemuž se vytvoří nová exekuce, se kterou se později pracuje v testech. Komunikace probíhá na základě Basic Autentifikace. Requesty, jež se posílají pomocí API a response, jsou ve formatu JSON.

Závěr

Hlavním záměrem této práce bylo seznámení její čtenáře s procesem testování ve vybraných agilních metodikách, provedení výzkumu vhodnosti testovacích technik v podmínkách agilního vývoje webových aplikací a zdůraznění rozdílů fungování testera na tradičním a agilním projektu. Dalším primárním účelem, je na základě probrané teoretické části, vyvinout testovací plán pro aplikaci napsanou v Nette Frameworku a provést vhodnou automatizaci. Z mého hlediska byly plně splněny definované cíle, díky čemuž tato práce může sloužit jako příručka pro každého testera, ať začátečníka či odborníka, který se teprve seznamuje s agilním vývojem a odchází z vodopádového projektu. Bakalářská práce přináší velkou motivaci pro přechod k automatizaci testovacích procesů a poskytuje shrnutí několika pohledů na to, jaký by měly automatické testy mít rozsah a jakou část aplikace by měly pokrývat. Zároveň zpřehledňuje seznam testovacích nástrojů použitých v agilních projektech, což je zčásti nad rámec zadání bakalářské práce, ale v zásadě jinak neoddělitelným teoretickým základem pro fungování v agilním světě.

Účelem implementační části bakalářské práce byl vývoj testovacího plánu pro webovou aplikaci české cestovní agentury ve frameworku Codeception s použitím vybraných technik pro metodiku SCRUM. Druhotním cílem a zároveň požadavkem firmy vyvíjející vybranou aplikaci byla implementace knihovny pro propojení pomocí ZAPI s interním systémem Jira pro vylepšení automatizace testů. Stanovené cíle rovněž považuji za splněné.

Literatura

[1] Agile Tester Extension Material for Download. ISTQB® International Software Testing Qualifications Board [online]. [cit. 20-04-2019]. Dostupné z: https://www.istqb.org/certification-path-root/agile-tester-extension/agile-tester-extension-material-for-download.html

[2] KANER, Cem, BACH, James, PETTICHORD, Bret and ELDRIDGE, Margaret. Lessons learned in software testing A context-driven approach. New York : Wiley, 2002.

[3] Software Testing Tutorial: Free Course. Meet Guru99 - Free Training Tutorials & Video for IT Courses [online]. [cit. 2019-04-20]. Dostupné z: https://www.guru99.com/software-testing.html

[4] FOURNIER, GREG. ESSENTIAL SOFTWARE TESTING: a use-case approach. Place of publication not identified : CRC Press, 2017.

[5] WHITTAKER, James A. Exploratory software testing. Upper Saddle River, NJ : Addison-Wesley, 2010.

[6] PAGE, Alan, JOHNSTON, Ken and ROLLISON, Bj. Jak testuje software Microsoft. Brno : Computer Press, 2009.

[7] GREGORY, Janet and CRISPIN, Lisa. More agile testing: learning journeys for the whole team. Upper Saddle River, N.J : Addison-Wesley, 2015.

[8] RUBIN, Kenneth S. Essential Scrum: a practical guide to the most popular agile process. Upper Saddle River, NJ : Addison-Wesley, 2013.

[9] Software Testing CertificationsInternational Software Test Institute [online]. [cit. 20-04-2019]. Dostupné z: https://www.test-institute.org/Software_Testing_Roles_And_Responsibilities.php

[10] ČÁPKA, David. Lekce 1 - Úvod do testování softwaru v C# .NET. Ajťácká sociální síť a materiálová základna pro C#, Java, PHP, HTML, CSS, JavaScript a další. [online]. [cit. 20-04-2019]. Dostupné z: https://www.itnetwork.cz/csharp/testovani/uvod-do-testovani-softwaru-v-csharp-net

[11] ZÁVODSKÝ, Petr. Rozdíly mezi testováním, řízením/kontrolou kvality a zajištěním/prokazováním kvality. Rozdíly mezi testováním, řízením/kontrolou kvality a zajištěním/prokazováním kvality [online]. 07.07.2015. [cit. 19.04.2019]. Dostupné z: https://blog.nic.cz/2015/07/07/rozdily-mezi-testovanim-rizenimkontrolou-kvality-a-zajistenimprokazovanim-kvality/

[12] GELPERIN, David and HETZEL, Bill. The growth of software testing. Communications of the ACM. Červenec 1988. str. 687–695.

[13] KITNER, Radek. Přehled testovacích technik. Radek Kitner [online]. 28.06.2018. [cit. 18.04.2019]. Dostupné z: https://kitner.cz/testovani_softwaru/prehled-testovacich-technik/

[14] CRISPIN, Lisa and GREGORY, Janet. Agile testing a practical guide for testers and agile teams. Upper Saddle River : Addison-Wesley, 2014. Addison-Wesley, 2014.

[15] WATKINS, John. Agile testing: How to succeed in an extreme testing environment. Cambridge : Cambridge University Press, 2009.

[16] RINGS, Angela. A Beginner's Guide to Test Automation. StickyMinds [online]. 25.03.2019. [cit. 15.04.2019]. Dostupné z: https://www.stickyminds.com/article/beginners-guide-test-automation

[17] BLAŽEK, Jiří. Reverzní inženýrství. thesis. Praha, 2018.

[18] Best 25 Test Management Tools in 2019. Meet Guru99 - Free Training Tutorials & Video for IT Courses [online]. [cit. 14.04.2019]. Dostupné z: https://www.guru99.com/top-20-test-management-tools.html

[19] GARBAR, Dmitry. Custom Software Testing: How We Use the Pareto Principle and Escape the Murphy's Law. Belitsoft [online]. 07.04.2019. [cit. 22.04.2019]. Dostupné z: https://belitsoft.com/software-testing-services/how-we-use-pareto-principle-and-escape-murphys-law

[20] ANDREW. Home. Try QA [online]. [cit. 22.04.2019]. Dostupné z: http://tryqa.com/why-is-testing-necessary/

[21] TUTORIALSPOINT.COM. Test Suite. www.tutorialspoint.com [online]. [cit. 23.04.2019]. Dostupné z: https://www.tutorialspoint.com/software_testing_dictionary/test_suite.htm

[22] COCKBURN, Alistair. Writing effective use cases. Boston : Addison-Wesley, 2001.

[23] KANER, Cem. Testing computer software. Hoboken, NJ : Wiley, 2012.

[24] COPELAND, Lee. A practitioner's guide to software test design In-text: (Copeland 2004) Your Bibliography: COPELAND, LEE, 2004, A practitioner's guide to software test design. Boston, Mass. : Artech House.

[25] Foundation Level Syllabus. ISTQB® International Software Testing Qualifications Board [online]. [cit. 01 .04.2019]. Dostupné z: https://www.istqb.org/downloads/syllabi/foundation-level-syllabus.html

[26] TUTORIALSPOINT.COM. SDLC Quick Guide. www.tutorialspoint.com [online]. [cit. 30.04.2019]. Dostupné z: https://www.tutorialspoint.com/sdlc/sdlc_quick_guide.htm

[27] BECK, Kent, BEEDLE, Mike, COCKBURN, Alistair, CUNNINGHAM, Ward, GRENNING, James, HIGHSMITH, Jim, HUNT, Andrew and JEFFRIES, Ron. Manifest Agilního vývoje software [online]. [cit. 30.04.2019]. Dostupné z: https://agilemanifesto.org/iso/cs/manifesto.html

[28] POPPENDIECK, Mary and POPPENDIECK, Tom. The lean mindset: ask the right questions. Upper Saddle River, NJ : Addison-Wesley, 2014.

[29] HIBBS, Curt, JEWETT, Steve and SULLIVAN, Mike. The art of lean software development. Sebastopol, CA : OReilly Media, Inc., 2009.

[30] Feature Driven Development (FDD) and Agile Modeling [online]. [cit. 08.04.2019]. Dostupné z: http://agilemodeling.com/essays/fdd.htm

[31] The DSDM Agile Project Framework (2014 Onwards). Agile Business Consortium [online]. 18.04.2017. [cit. 09.04.2019]. Dostupné z: https://www.agilebusiness.org/resources/dsdm-handbooks/the-dsdm-agile-project-framework-2014-onwards

[32] HEUSSER, Matthew. 6 common software test automation mistakes and how to avoid them. TechBeacon [online]. 22.01.2019. [cit. 10.04.2019]. Dostupné z: https://techbeacon.com/app-dev-testing/6-common-test-automation-mistakes-how-avoid-them

[33] FAUDZIAH, Ahmad, BAHAROM, Fauziah and ALTARAWNEH, Moath. Agile Develop ment Methods for Developing Web Application in Small Software Firms. ResearchGate [online]. Červen 2012. [cit. 10.04.2019]. Dostupné z: https://www.researchgate.net/publication/281965654

[34] ŠOCHOVÁ Zuzana and RISING, Linda. The great scrummaster: #scrummasterway. Boston : Addison-Wesley, 2017.

[35] XML and XPath [online]. [cit. 15.04.2019]. Dostupné z: https://www.w3schools.com/xml/xml_xpath.asp

Příloha A

Slovník pojmů

Stakeholder Zájemce, zainteresovaná osoba, pro kterou se vyvíjí výsldný software

Jira Trekovací systém

Zephyr Nástroj pro správu testů

Příloha B

Seznam použitých zkratek

SDLC Software Development Lifecycle - Model životního cyklu vývoje software

JSON JavaScript Object Notation

AUT Application under test

QA Quality Assurance

GUI Gragraphical User Interface

API Application programming interface

UI User interface

Příloha C

Obsah přiloženého CD

- readme.txt .... stručný popis obsahu CD
- src
  - impl .... zdrojové kódy implementace
  - thesis .... zdrojová forma práce ve formátu LaTeX
- text .... text práce
  - thesis.pdf .... text práce ve formátu PDF