Универзитет у Београду
Математички факултет

Андрија Миљковић

Тестирање софтвера у агилном окружењу

Мастер теза

Ментор: **проф. др Владимир Филиповић**

Београд,
2018.

Ментор:

проф. др Владимир Филиповић
Математички факултет
Универзитет у Београду

Чланови комисије:

проф. др Филип Марић
Математички факултет
Универзитет у Београду

доц. др Александар Картељ
Математички факултет
Универзитет у Београду

Датум одбране:

Садржај

Глава 1

Увод

Данашње доба се може слободно назвати добом Интернета. Већина ствари које се користе сада могу бити повезане, могу комуницирати на више начина између себе. Иза сваког решења, производа који може бити повезано на Интернет, постоји неколико линија кода који заправо омогућава да то све буде могуће. Како бисмо били сигурни да све ради као што је иницијално замишљено, неопходно је тестирање.

Зарад бољег објашњења самог појма тестирања, нека фокус буде само на веб странама које (између осталог) омогућавају да повезаност између различитих система крајњи корисник види као платформу на којој може да самостално обави различите радње, преко куповине производа, резервације карата, летовања, итд. Развој веб стране се може обавити на више начина, међутим, у последњој деценији агилни приступ развоју софтвера постаје доминантан. Тестирање, било написано пре, током или након самог развоја софтвера, увек је пратило изабрани приступ, био то традиционални или неки од модернијих приступа. Тестирање софтвера у агилном окружењу самим тим има за циљ да испрати агилни приступ и омогући да развој софтвера буде стабилан.

У овом раду је описано тестирање у агилном окружењу са акцентом на мануелно и аутоматско тестирање веб стране. Софтвер развијен током израде овог мастер рада је, као слободан софтвер, доступан у GitHub репозиторијуму, на адреси: https://github.com/brackoandrija/automation.

Глава 2

Приступ развоју софтвера

Као што је у уводном поглављу написано, данас је доба Интернета, брзих рачунара и мобилних уређаја. Сваки од тих уређаја користи неко софтверско решење које је развијено помоћу различитих приступа.
Развој софтвера је сложен процес израде софтвера, који обухвата више различитих активности, међу којима су:

- анализа система
- препознавање и обликовање захтева
- анализа захтева
- детаљна спецификација захтева
- пројектовање софтвера
- имплементација софтвера
- тестирање
- одржавање

Сам развој софтвера и активности које су излистане изнад не могу да функционишу саме за себе, уколико не постоји **скуп вредности** и принципа који успостављају сам правац размишљања и **окружење** које дефинише активности и правила.

2.1 Агилност

Агилност (енг. Agile) је реч која се пречесто користи приликом модерног развоја софтвера, највише међу *startup* компанијама. Међутим, шта је то агилност? И још битније: шта то агилност није?

Уколико би одговор на ово питање давали људи из различитих бранши, одговори би били следећи (изражени кроз метафору):

- Продавци било ког производа би рекли да је њихов производ 100% агилан;
- Компанија за производњу папира би рекла да бити агилан значи да приче корисника (енг. user stories) морају бити написане на папиру;
- Консултант би рекао да је то методологија за развој софтвера коју свака организација може да научи уколико желите њихове услуге;
- Компанија за производњу ципела би рекла да је кључ за агилност састанци које проводите са људима који стоје.

Међутим, одговор на питање шта јесте, а шта није агилност можемо пронаћи у ***манифесту агилности (енг. Agile manifesto*** [1]***)***.

Агилност није:

- Методологија,
- Специфичан начин развоја софтвера,
- Развојни оквир или процес.

Агилност је **скуп вредности и принципа**.

Манифесто агилности у 68 речи открива боље начине за развој софтвера развијајући софтвер самостално и помажући другима при њиховом развоју. Кроз тај рад су научили да више вреднују:

- **Појединце и интеракције** од **процеса и алата**
- **Применљив софтвер** од **детаљне документације**
- **Сарадњу са клијентима** од **уговорних аранжмана**
- **Реакцију на промену** од **придржавања плана.**

Јако је битно да се нагласи, десна страна је битна, међутим лева страна је више цењена. Поред претходно наведених вредности, постоје принципи који подржавају вредности и тиме комплетирају слику која је потребна уколико желимо да неку праксу назовемо "Агилно". Сами принципи су општег типа и они не говоре шта је потребно да се уради, већ помажу приликом доношења одлука.

12 принципа Агилности су:

1. Задовољан клијент је врхунски приоритет, који се постиже благовременом и континуираном испоруком врхунског софтвера.
2. Спремно прихватање промена захтева, чак и у касној фази развоја. Агилни процеси омогућавају успешно прилагођавање измењеним захтевима што за резултат има предност у односу на конкуренцију.
3. Редовна испорука применљивог софтвера, у периоду од неколико недеља до неколико месеци, дајући предност краћим интервалима.
4. Пословни људи и програмери свакодневно да сарађују у току целокупног трајања пројекта. Пројекти се остварују уз помоћ мотивисаних појединаца којима је обезбеђен амбијент и подршка која им је потребна и којима је препуштен посао с поверењем.
5. За најпродуктивнији и најефикаснији метод преноса информације до и унутар развојног тима сматра се контакт лицем у лице.
6. Применљив софтвер је основно мерило напретка.
7. Агилни процеси промовишу одрживи развој.
8. Покровитељи, програмери и корисници морају бити у стању да континуирано раде усклађеним темпом, независно од периода трајања пројекта.

9. Стална посвећеност врхунском техничком квалитету и добар дизајн поспешују агилност.

10. Једноставност – вештина довођења до највишег степена количине рада који није потребно урадити – је од суштинске важности.

11. Најбоље архитектуре, захтеви и дизајн, резултат су рада само–организованих тимова.

12. Тимови у редовним интервалима разматрају начине како да постану ефикаснији, затим се усклађују и на основу тих закључака прилагођавају даље поступке.

Као што је већ речено, агилност је ***скуп вредности и принципа***, и као таква је јако корисна приликом доношења одлука са циљем развоја врхунског софтвера.

Приликом развоја производа, да би било могуће поистоветити се са вредностима и принципима, компаније су развиле раличите развојне оквире (енг. *frameworks*) у којима су примењивале агилност. Неретко, компаније се одлучују на више развојних оквира како би покриле све процесе у оквиру развоја производа. Оквири који су до сада дефинисани су описани у наставку.

2.1.1 Cinefin [1]

Моделирање различитих домена са циљем креирања смисла ствари које се развијају и приналажења правих активности. Све што се развија могуће је поделити на четири целине:

- Просте,
- Компликоване,
- Комплексне,
- Хаотичне.

[1] Cinefin је Велшка реч која представља станиште. Термин се односи на идеју да сви имају везе, као што су племенске, верске и географске, о којима можда нису ни свесни.

Ово је приступ који се користи приликом одабира који приступ користити – агилан, водопад или неки други. Овај развојни оквир је детаљније објашњен у оквиру секције 2.3.

2.1.2 Екстремно програмирање

Екстремно програмирање (енг. Extreme programing) има за циљ испоручивање максималне вредности за саму компанију пажљивим слушањем и професионалним кодирањем. Подстицање промене је једна од главних принципа екстремног програмирања. Као максимално итеративан приступ, екстремно програмирање у самом срцу процеса окупира програмере и бизнис људе, који у кратким итерацијама испоручују применљив софтвер што је брже могуће.

2.1.3 Scrum

Развојни оквир са фокусом на тим и инкрементални развој софтвера. Сам развојни оквир има дефинисане вредности које су кључ успеха:

- Фокус,
- Поштовање,
- Отвореност,
- Храброст и
- Привржености.

Један је од најпопуларнијих развојних оквира агилности, на први поглед јако сличан екстремном програмирању, међутим, уместо фокуса на само програмирање, фокус је више на теми управљања пројектом. Сам развојни оквир дефинише три улоге:

- **Развојни тим** (енг. Development team), сви људи који заправо развијају или тестирају сам производ;
- **Власник производа** (енг. Product owner), особа одговорна за повећање бизнис вредности производа који доставља тим. Особа која уређује листу спецификација потребних за развој производа;
- **Вођа тима** (енг. Scrum master), особа одговорна за спровођење и тренирање агилности код осталих чланова тима.

2.1.4 Kanban

Канбан је систем који ограничава број задатака на којима се може радити у истом тренутку. Настао је пре више од 50 година у Тојоти, у Јапану и повезује се са оптимизацијом масовне продукције аутомобила. Канбан је систем картица које тачно означавају где се један одређени материјал налази у процесу производње. Иако је развој софтвера креативна активност и нема много везе са масовном производњом, неки од механизама са производње су нашли примену у IT индустрији. У примени овог развојног оквира, највећа промена програмерима је да визуализују посао који је у току. Међутим, то је срж самог развојног оквира, који омогућава фокус на контролисање посла који је у току и транспарентно издвајање питања и проблема који би иначе били невидљиви.

2.1.5 Мапирање прича

Мапирање прича (енг. Story mapping) је развојни оквир који помаже свим интересним странама да визуелно виде комплетан производ представљен кроз структуру корисничких прича. Уместо стандардног списка спецификација које се завршавају једна за другом, овде имамо визуелну презентацију целог производа уређеног на основу вредности које доноси крајњем кориснику као и времену када ће бити завршено. Развојни оквир се базира на два типа спецификација представљених у виду радних налога:

- Активности, које представљају спецификације са највишег нивоа, које сам корисник може да разуме. Активности су затим поређане тако да креирају комплетну причу која је кориснику потребна. Уколико је то рецимо Интернет продавница, то би значило да имамо активности од тренутка када корисник долази на Интернет продавницу, преко саме куповине до тренутка када је корсник купио производ и затворио саму Интернет продавницу.
- Задаци, прецизирани радни налози који припадају активностима. Наводе се у оквиру активности и уређују тако да дају максималну вредност крајњем кориснику.

Сам развојни оквир поред тога што помаже приликом комуникације свим интересним странама производа који се развија, помаже приликом планирања и

брзе испоруке производа помоћу развоја задатака по приоритету. У пракси се овај развојни оквир користи као додатак на Scrum, односно као улазна информација. Да би власник производа могао да спреми јасне захтеве/спецификације, мапирање прича игра велику улогу у дефинисању свих активности поређаних по приоритету. Када су поређане активности по приоритету, вођа производа креира списак задатака који током планирања у Scrum развојном оквиру програмери ревидирају и испланирају колико ће им времена за сваку од активности бити потребно.

2.1.6 Ретроспективе

Ретроспективе (енг. Retrospectives) се користе се као саставни део *Scrum* развојног оквира, међутим издвојио се и као засебан развојни оквир, јер је јако моћан развојни оквир за константан напредак и учење на свим нивоима саме компаније. Овај развојни оквир није настао из агилности. Он, у психологији постоји одавно, и то је реч која се може користити за све активности 'гледања уназад', где је особа погрешила и шта може да уради боље следећи пут.

Највећи изазов са ретроспективама је што се најчешће користе прекасно. Зато су данас ретроспективе најчешће уметнуте унутар осталих развојних оквира.

2.1.7 Развој вођен тестовима

Развојни оквир **Развој вођен тестовима (енг. Test driven development)** означава писање тестова пре самог програмирања. Помаже како би се изградио модуларан и флексибилан систем. Брза повратна информација (енг. feedback), је кључ успеха у овом развојном оквиру.
Развој вођен тестирањем се највише повезује са тестирањем компоненти и аутоматским тестирањем, и јако је добра надоградња на развојни оквир екстремно програмирање.

Кораци које пропагира TDD су једноставни и ефективни:

(a) **Додај тест:** Пре него што се дода било каква функционалност у систем, креира се тест који ће поверавати жељено понашање апликације/функције која ће бити имплементирана након тога;

(b) **Пусти тест да "падне":** Пре него што се напише нови код, треба осигурати да написани тест "пада" приликом тестирања услова који ће бити када се напише нови кôд. На тај начин се осигурава да се тестира права функционалност;

(c) **Писање кôда:** Имплементирање функционалности на основу спецификација;

(d) **Покретање теста:** Након имплементације покренути тест опет и видети да ли је исправан. Уколико је исправан потребно је рефакторисати[2] кôд

[2]Рефакторисање кôда је побољшавање дизајна постојећег кôда.

и пробати опет. Уколико није добар, прво погледати кôд да ли функција ради то што треба да ради па се вратити на тест опет;

(е) **Рефакторисати:** Након што су написани и кôд и тест, јако је битно да коначни кôд буде читљив. Рефакторисање ту игра велику улогу.

2.1.8 Развој вођен понашањем

Развојни оквир **Развој вођен понашањем (енг. Behavior driven development)** се временом показао као практичнији. Развој вођен тестовима има велику комерцијалну вредност, међутим и даље је развојни оквир у коме особе које немају техничку позадину, не могу на најбољи начин да разумеју шта се дешава. Из тог разлога је настао развојни оквир Развој вођен понашањем.

Главна разлика у односу на TDD је:

- Тестови су написани у једноставој описној граматици енглеског језика;
- Тестови се објашњавају као понашање апликације и више су фокусирани на крајњег корисника;
- Примери се употребљавају за разјашњавање захтева.

За разлику од TDD, главне карактеристике BDD оквира су:

- Пребацивање са размишљања из угла "тестова" у угао "понашања";
- Сарадња између свих страна развоја самог производа: бизнис људи, аналитичара, програмера, тестера итд;
- Језик је општи, самим тим разумљив свима.

Сам развојни оквир је могуће применити на свим нивоима тестирања (биће више речи у поглављу 4), како на нивоу тестирања компоненти, тако и системско тестирање крајњег производа које види корисник.

2.1.9 Примена развојних оквира

Поред претходно описаних равојних оквира, постоји још агилних развојних оквира који неће бити покривени овим радом, а имају велику примену данас у развоју софтвера. Неки од њих су:

- I.N.V.E.S.T.
- Real Options
- Lean
- Lean Startup
- Impact Mapping
- Open Space
- Lean Coffee
- итд...

Развојни оквир дефинише понашања, правила и додатно прецизира вредности које су већ дефинисане коришћењем агилности. Као и за сва правила, потребно је пратити основна правила (са једне стране) али исто тако не треба се слепо придржавати свих правила и рећи да је развојни оквир у употреби због тога. Сваки тим има своју криву сазревања, и у складу са тим развојни оквир који примењује у датом тренутку ће сазревати заједно са тимом.

Применом једног развојног оквира тим може добити пуно дефинисаних активности, међутим не све. Данас, компаније користе више различитих развојних оквира приликом развоја једног производа. Нека пример буде компанија која развија Интернет продавницу. Да би се та продавница развила, потребно је дефинисати који приступ компанија жели да има, агилан, водопад или неки трећи. У томе ће развојни оквир **Cinefin** помоћи. Уколико се изабере агилан приступ, следеће на реду је дефинисање захтева и спецификација самог производа. За дефинисање захтева и визуелну презентацију се може користити **Мапирање прича**. У зависности од тога колико чланова развојног тима постоји, може се користити развојни оквир **Scrum** у комбинацији са **Екстремним програмирањем** и **TDD/BDD**. Финални и заједнички циљ је да корисник добије жељени производ што пре, тако да коришћењем ових развојних оквира компанија повећава шансу да то и испуни.

2.2 Водопад

До сада је било речи само о агилности, где је главни фокус на тиму и како сам тим доноси вредност свим интересним странама. Са друге стране имамо модел са фокусом на процесе. Водопад (енг. Waterfall) је модел који је настао међу првима и представљен је са животним циклусом који је секвенцијалан. То значи да свака фаза не може да почне док се не заврши претходна.

Слика 2.1: Водопад модел, илустративан пример процеса

Фазе у моделу водопада су:

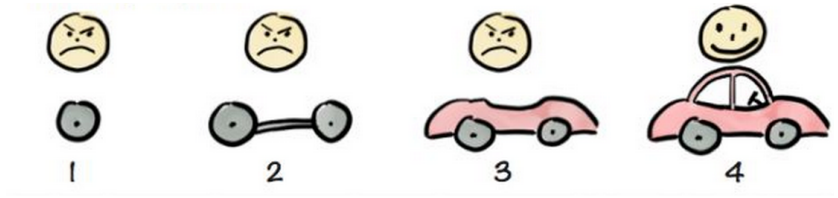
- **Прикупљање и анализа потреба** - Сви захтеви система који ће се развијати налазе се у овој фази и остаће документовани у спецификацији током ове фазе.
- **Дизајн система** - У овој фази се испитују спецификације захтева из прве фазе и припрема се дизајн система. Дизајн система помаже у одређивању хардверских и системских захтева и помаже у дефинисању укупне архитектуре система.
- **Имплементација** - Са информацијама из претходне фазе, систем се први пут развија у малим програмима званим јединице система (енг. units), које су интегрисане у следећу фазу. Свака јединица је развијена и тестирана по својој функционалности, што је уједно и тестирање јединице.
- **Интеграција и тестирање** - Све јединице развијене у фази имплементације интегрисане су у систем након тестирања сваке јединице. Након интеграције цео систем се тестира са циљем проналажења и уклањања свих грешака.

- **Развој система** - Када се уради комплетно тестирање свих компоненти система, производ се користи у окружењу купца и након тога пушта на тржиште.
- **Одржавање** - Након што се производ пусти на тржиште, корисници су последња линија тестирања, где се понекад добију и најбољи коментари како побољшати саму апликацију. Одржавање је неопходно како би се испратиле потребе на тржишту.

2.3 Упоредни однос агилног приступа и водопада, предности и мане

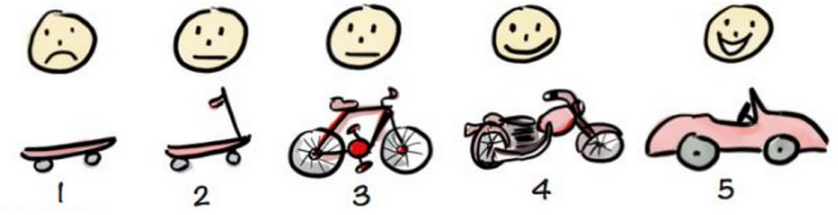
Постоји пуно разлика између ова два приступа, што ће бити илустровано сликама. У наставку можемо видети како изгледа развој софтвера са једним и другим приступом.

Водопад приступ:



Слика 2.2: Метафорички приказ развоја водопад приступом

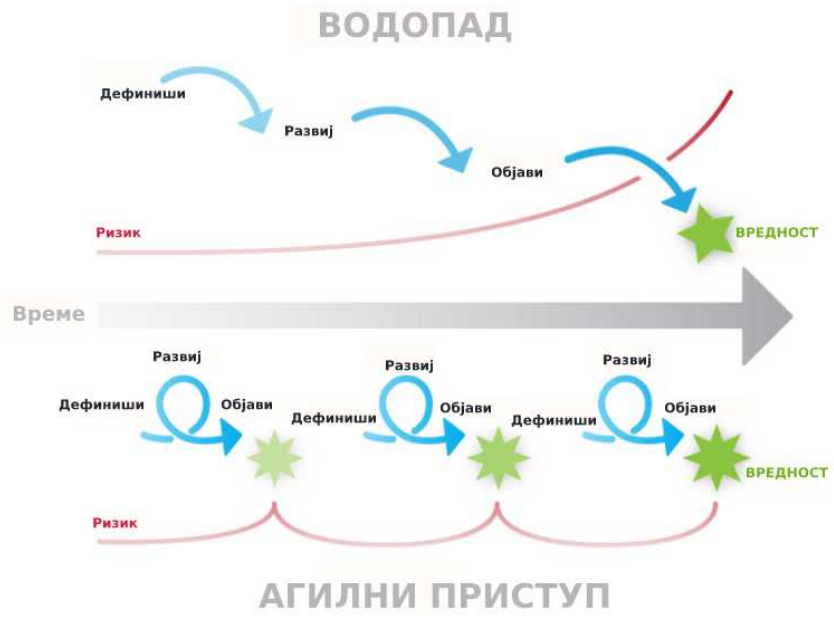
Агилни приступ:



Слика 2.3: Метафорички приказ развоја агилним приступом

Уколико би се направила паралела са фазама које се налазе у оба приступа, могло би се приметити да се исте фазе дешавају током развоја производа, међутим на другачији начин. Код приступа заснованог на водопаду вредност самог производа добија корисник на крају пројекта, док код агилног приступа добија

вредност константно али у мањим количинама.



Слика 2.4: Приказ извршавања истих активности код агилног и водопад приступа

Када се погледа слика 2.4, може се приметити да се тестирање у водопад приступу дешава у тачно одређеном тренутку, док се у агилном приступу дешава константно, паралелно са самим програмирањем и осталим процесима. То доводи до питања како изабрати прави приступ приликом развоја производа. Одговор на ово питање може дати граф који је направио Ралф Стејси (енг. *Ralph Stacey*) на слици 2.5.

Y оса представља ниво сагласности између свих интересних страна око развоја самог производа. X оса представља сигурност коју има развојни тим око техничких решења. На основу ове две димензије могуће је утврдити где се пројекат налази и на основу тога одлучити који приступ користити. Генерални приступ може бити такав да што је флексибилност самог пројекта мања и што је несигурније како ће технички бити изведен пројекат, треба одабрати Агилнији приступ док у обрнутом случају Водопад прави приступ.

Најчешћа четири стања у којима се могу наћи пројекти су:

- **Очигледан** - висок ниво сигурности и све је јасно око договора и потенцијалних проблема (водопад приступ препоручљив).

Слика 2.5: Графички приказ графа комплексности

- **Компликован** - стање у коме је најтеже одлучити, међутим постоје два најчешћа случаја, када је флексибилност око договора мала а техничка сигурност велика и када је флексибилност око договора велика али сигурност око техничких решења мала. У првом случају је добро користити водопад приступ јер је све познато са техничке стране и неће се пуно тога мењати са стране договора, тако да је могуће завршити развој производа по фазама. У другом случају имамо обрнуто, и овде је добро користити Агилан приступ, јер је флексибилност око договора велика и могуће је мењати ствари успут, док сигурност око техничких детаља није велика и самим тим ће вероватно и доћи до промене почетних договора.
- **Комплексан** - стање у коме је флексибилност око договора мала и сигурност око техничких решења мала, самим тим је доста незгодно предвидети шта ће се десити у току равоја производа. Препоручљив приступ је Агилан из разлога што је прилагодљивији у случају да се нешто успут промени.
- **Хаотичан** - стање које је јако непредвидиво и то су пројекти које је потребно довести на неки од претходна три стања како би се даље одлучило шта са њима радити. Како је све недефинисано, најбоље је кренути са било којим активностима и онда осетити у ком се смеру креће пројекат и кроз пар итерација (Активност – Осећај – Реакција) пројекат ће добити облик који ће више личити на неки од претходна три стања.

Као што се могло видети, најтеже је одлучити који приступ користити приликом комликованих пројеката. Како би се на крају одлучиле, компаније прелазе на поређење предности и мана између Агилног и Водопад приступа.

Предности Водопад приступа:

- Приликом развоја софтвера води се прецизно документација, која касније служи као водиља за унапређење процеса,
- Овим приступом корисник зна шта очекује. Тачно је договорен буџет и временски оквир
- Уколико је софтвер/производ већ прављен и потребно га је опет креирати само за другог корисника или истог само за потребе другог пројекта, овај приступ може помоћи тако што ће пројекат бити тачно испланиран, са јасним буџетом и биће достављен на време.
- У случају да дође до промене међу запосленима, документација омогућава да пројекат не заостаје превише

Мане Водопад приступа:

- Када се фаза заврши, на њу се враћа искључиво ако постоји блокада на следећој фази која изискује промене на претходној.
- Тестирање се ради тек у договореној фази, углавном пре завршетка пројекта. Како тестирање управо и служи да се пронађу све грешке, у случају да су захтеви били лоши од почетка, пројекат је опет осуђен на пропаст.

Предности Агилног приступа:

- Након почетног планирања, могуће је променити захтеве и њихове приоритете у било којој фази развоја софтвера.
- Производ се доставља у малим целинама и константно крајњем кориснику. Самим тим корисник је током целог развоја софтвера задовољан.
- Тестирање се дешава константно, паралелно са програмирањем.
- Са фокусом на развојни тим, овај приступ омогућава тиму да сазрева и проширује технолошка знања и постаје продуктивнији, самим тим сваки следећи пројекат може бити комплекснији док број чланова тима остаје исти.

Мане Агилног приступа:

- Како се овај приступ ослања на тимски менаџмент, уколико особа која дефинише производ није сигурна шта жели да види на крају, пројекат може бити неуспешан.
- Иницијалан план није дефинитиван, самим тим може се десити да је производ тотално другачији од идеје која је првенствено настала.

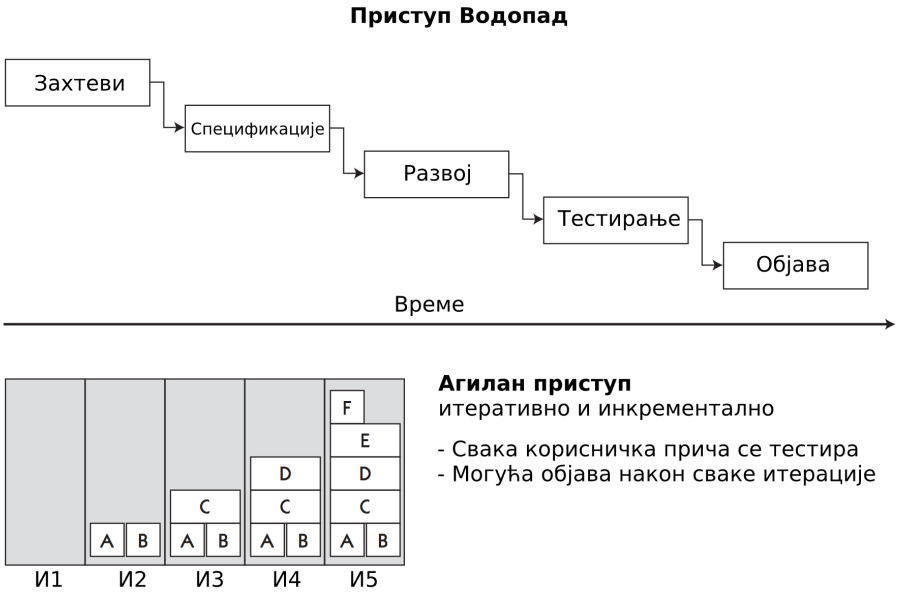
Глава 3

Основе тестирања

У претходном поглављу је направљен преглед приступа који су се показали као одлична пракса за развој софтвера. Међутим, када се прича о тестирању, традиционални водопад приступ и агилни приступ имају потпуно другачији поглед. У водопад приступу, тестирање је једна фаза пред сам крај пројекта, док је у агилном приступу саставни део свих активности. Самим тим главна разлика се одмах и уочава, где уколико нешто не ваља у фази тестирања водопад приступом, прелази се на фазу пре или у најгорем случају комплетно враћа на почетак пројекта. Агилни тимови тестирају константно. То је једини начин да се осигура квалитет софтвера који се развија током једне од итерација. У традиционалном водопад приступу, тестирање ради тим који је за то задужен, неретко изолован како би били максимално објективни приликом тестирања. Са друге стране, у агилном приступу сви тестирају. Постоје особе које су цело радно време задужене за тестирање, међутим главна предност тестирања у агилном приступу је што је свест свих страна подједнака и окренута ка тестирању и осигуравању квалитета.

На слици 3.1, може се јасно видети разлика у тестирању. У традиоционалном или водопад приступу, тестирање је изолована фаза пред сам крај пројекта. Као фаза је дефинисана да буде исте величине и вредности као остале фазе. Међутим, када је развој софтвера у питању, највећи број компанија има проблем са продужавањем самог програмирања из различитих разлога. Самим тим, време предвиђено за тестирање се смањује и на крају убрзава не би ли се достигао рок који је иницијално задат.

Са друге стране је агилни приступ који развој дели по итерацијама (које су величине од једне недеље до месец дана) и софтвер се развија инкрементално. У свакој итерацији софтвер је развијен, тестиран и пуштен у употребу. Снага овог приступа је што је тестирање неизоставни део развоја, јер захтев је готов тек оног тренутка када је кôд написан, документован и тестиран од стране тестера или других одговорних људи за тестирање.



Слика 3.1: Графички приказ водопад и агилног приступа

3.1 Грешке у развоју софтвера

Ако изузмемо било који приступ, комплексност развијања софтвера данас лежи у брзини којом се испоручују готови производи, како би корисници што пре добили жељени производ. Због те брзине, неретко се дешавају грешке и пропусти који касније узрокују обарање система, незадовољство корисника, итд. Све то наводи колико је тестирање неопходан и неизоставан фактор приликом развоја софтвера.

Грешке могу бити веома разноврсне:

- **погрешан редослед речи у реченици** – губитак смисла реченице, самим тим мали трошкови;
- **погрешна подешавања система** – престанак рада апарата у медицини, што може довести до губитка живота, самим тим прескупа грешка.

Постоји више речи који указују да је настао проблем у систему, те је најбоље разложити шта која реч значи:

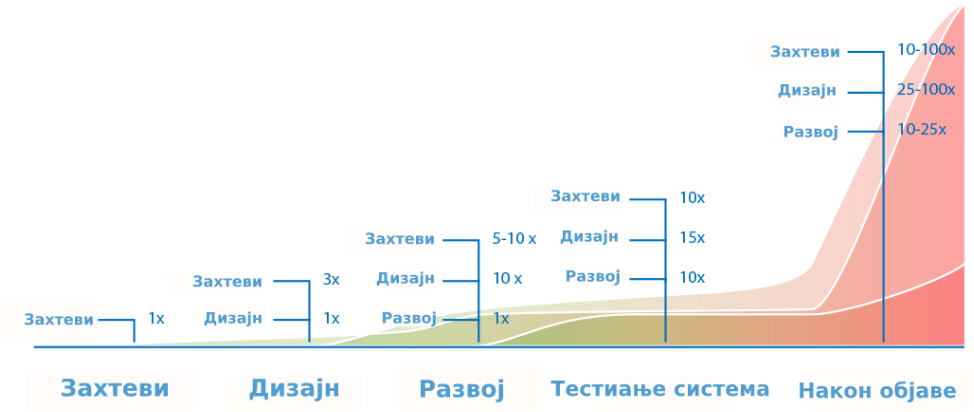
- **Пропуст (енг. Mistake)** – људска акција која доводи до погрешног резултата.
- **Недостатак (енг. Fault или Defect или Bug или Problem или Issue)** – погрешан корак, процес или дефиниција података у програму. Недостатак уколико се не пронађе и реши на време, проузрокује отказ.
- **Отказ (енг. Failure)** – немогућност система или његове компоненте да изврши захтевану функцију у складу са дефинисаним захтевима. Отказ може бити јако сакривен и да буде откривен тек приликом неке специфичне активности. Из тог разлога је јако битан за проналажење у што ранијој фази развоја софтвера.
- **Грешка (енг. Error)** – разлика између израчунате, уочене или измерене вредности или услова и стварне, дефинисане или теоретски исправне вредности или услова.

Тестирање као активност има за циљ да умањи могућност отказа у систему.

Сведоци смо свакодневних грешака које се јављају, од сигнала мобилних мрежа, банковних рачуна, проблема са рачунарима, серверима итд. Једини начин да се умањи број грешака који настаје је константним тестирањем, како током развоја производа тако и током коришћења самог производа.

Проналажење грешке у било ком тренутку је добро, тако да је сваки приступ који садржи тестирање као активност добар. Међутим сама грешка у зависности од тренутка када се пронађе има другачију тежину.

График који се може видети на слици 3.2 јасно показује да уколико се грешка нађе у фази спецификација може се брзо и лако уклонити, док што дуже та грешка остаје, постаје све већа и скупља.



Слика 3.2: Графички приказ пораста цене грешке приликом касног откривања

Пример 1 – Интернет продавница

Као и свака друга, сама интернет продавница би требало да дозвољава куповину више производа. Међутим, уколико се приликом програмирања изостави могућност куповине више производа у истом тренутку и функционалност постане таква да додавањем производа у корпу, брише се производ који је већ био у корпи, корисник ће моћи да купи највише један производ. Проналажење те грешке на нивоу спецификација, дефинисањем јасних критеријума за прихватање кôда (енг. Acceptance criteria) ова грешка се вероватно неће ни десити. Међутим уколико се та грешка провуче и прође фазу развоја и открије тек у фази тестирања, то значи да је потребно укључити опет пословне људе који морају да допуне спецификације, развојни тим мора да допуни кôд и на крају опет све мора да се тестира. То је прилично скуп процес за нешто што се могло предупредити још на почетку.

Агилни приступ игра велику улогу, јер дефинисање захтева креће од пословних људи, међутим ништа не улази у итерацију пре него што се испланира. Приликом планирања пролазе се сви захтеви још у фази спецификације и већ тада уклањају потенцијалне грешке као што је претходна. Чак и уколико се грешка појави у итерацији, како се тестирање дешава паралелно са програмирањем,

грешка ће бити врло брзо уочена и у следећој итерацији исправљена.

3.2 Колико тестирања је довољно

Главно питање које се приликом тестирања поставља је **'Када стати са тестирањем?'**.

Тестирање свега је готово немогуће. За било које тестирање, било какве апликације, интернет продавнице, производа, неопходно је време, људи, знање, итд.

Пример 2: Тестирање уноса вредности у поља.

Нека систем има 20 поља у која могу да се унесу 4 различите вредности. Тестирање свега би у овом случају значило тестирање 420 могућих комбинација. Уколико би за тестирање једног поља било потребно 10 секунди, комплетно тестирање би трајало око 350 хиљада година. То доводи до тога да исцрпно тестирање није могуће, већ да много већу улогу игра одабир шта ће бити тестирано.

У агилном окружењу, једна итерација траје од недељу дана до месец дана. То значи да у том периоду софтвер треба да буде написан, документован и тестиран. Самим тим време је врло ограничавајућ фактор. Да би тестирање било што релевантније, компаније су се временом окретале ка два правца тестирања током сваке итерације – тестирање софтвера који је развијен у току итерације и тестирање најбитнијих функционалности које су развијене у претходним итерацијама (енг. Regression testing). Биће речи у току рада о свим нивоима и врстама тестирања, међутим овде је кључно разумети да је неопходно прилагодити свест да је немогуће све тестирати, а са друге стране бити сигуран да то што је тестирано обезбеђује самоувереност како развојном тиму тако и крајњем кориснику.

3.3 Шта је тестирање?

Тестирање је комплексан процес који има пуно дефиниција и објашњења. То долази из самих активности који су разноврзне. Чак и сами циљеви тестирања нису увек исти. Неке од познатијих дефиниција тестирања су:

"*. . . процес извршавања програма са циљем потврде његовог квалитета..*"

Међутим нигде се не помиње да се приликом тестирања проналазе грешке. Тај део покрива следећа дефиниција.

"*. . . процес извршавања програма са циљем проналажења грешака..*"

Ова дефиниција не говори ништа о тестирању захтева са циљем њиховог побољшања и отклањања отказа још у раној фази развоја софтвера. Као што је описано у одељку 3.1, сваки проблем који се пронађе приликом разоја софтвера је скупљи што се касније открије, те је тестирање захтева јако битно. Да би се добила боља дефиниција тестирања потребно је спојити Милсову и Мајерсову дефиницију.

“. . . процес провере софтвера са циљем проналажења отказа и провере да исти задовољава задате функционлне и не-функционалне захтеве”.

Као закључак, једна од комплетнијих дефиниција тестирања је дефинисана од стране ISTQB[1]:

"Процес који садржи све животне циклусе пројекта са његовим активностима, статичким и динамичким, везано за планирање, припрему и извршавање софтвера и повезаних радних пакета са циљем задовољавања задатих захтева и демонстрирање да исти одговарају сврси и откривањем грешака које се у том процесу нађу.".

[1](енг. Internation Software Testing Qualification Board) интернационални одбор који се састоји од националних и регионалних тестинг одбора великих компанија, који су ISTQB представници у њиховим земљама

- **Тест** је скуп једног или више тест сценарија

- **Тест сценарио (енг. Test case)** је скуп улазних вредности, извршних предуслова, очекиваног резултата и услова извршења, развијен са циљем остваривања неког програма и верификације усклађености са одређеним захтевима.

- **Отклањање грешака (енг. Debugging)** је процес проналажења, анализирања и отклањања узрока који доводи до грешака у софтверу.

Границе између тестирања и отклањања грешака није увек јасна. Тестирање има за циљ да директно и систематично пронађе откази, који доводи до недостатака који су отказ проузроковали. Проналажење грешака има за циљ да лоцира грешку како би иста била обрисана.

3.4 Принципи тестирања

Као што је већ наглашено више пута, тестирање свега није адекватно. Такође, не постоји чаробни штапић којим се може обезбедити јединствено тестирање сваког производа јер на крају дана колико квалитетно ће бити тестирано у највећем броју случајева зависи од људи. Из тог разлога постоје принципи који приближавају тестерима да приступе тестирању на прави начин. Принципи тестирања описују следеће тезе:

(a) **Тестирање показује присуство недостатака, али не и њихово одсуство** *"Приликом пецања рибе, уколико се једна упеца, то говори да вода има рибу, међутим не говори да нема више рибе."* Из тог разлога тестирање треба користити на свим фазама развоја производа, што чешће и што више како би се умањила могућност проналажења грешака.

(b) **Тестирање свега није могуће**
Тестирање свега значи да је неопходно покрити све улазне вредности, све излазне вредности и комбинације између. Пример са улазним вредностима је већ објашњен у одељку 3.2.

(c) **Почети са тестирањем што раније**

Као што је описано у одељку 3.1, што се раније открију сви недостаци софтвера, то је сам пројекат финансијски стабилнији .

(d) **Грешке су често једна близу друге**

У пракси се показало да је највећи део грешака који је откривен био у близини осталих грешака јер приликом развоја софтвера једна грешка често проузрокује другу итд.

(e) **Парадокс пестицида**

Што је старији тест сценарио, велика је могућност да није више потребан нити квалитетан. Такође, тест сценарио који не проналази никакве грешке није добар тест сценарио.

(f) **Тестирање је зависно од контекста**

Немогуће је два система која су повезана тестирати на исти начин. Чак и два слична система не треба тестирати на исти начин. Сваки систем, свака компонента је прављена на основу јединствених захтева, и тако их треба и третирати.

(g) **"Непроналажење грешака", не значи да се систем може користити**

Тестирање се не завршава. Траје константно и понекад највећа ситница, ако довољно остане дуго у систему може проузроковати отказ у систему и престанак рада апликације, софтвера. Неопходно је коришћење прототипа, укључивање клијента у што ранијој фази итд.

Глава 4

Тестирање у агилном окружењу

До овог поглавља је било доста речи о томе које су разлике у приступима, како који приступ гледа на тестирање. Само тестирање као процес се не разликује пуно од приступа до приступа, међутим колико често и како се гледа на тестирање је нешто што прави разлику између приступа.

Тестирањем у агилном окружењу покушавају се постићи и верификација и валидација производа.

Верификација:

"Потврда путем испитивања и доказивање да су испуњени претходно дефинисани захтеви"

или једноставније речено: "***Да ли је систем развијен исправно***"

Валидација:

"Потврда путем испитивања и доказивање да су испуњени услови за одређену апликацију или намену"

или једноставније речено: "***Да ли је развијен исправан систем***"

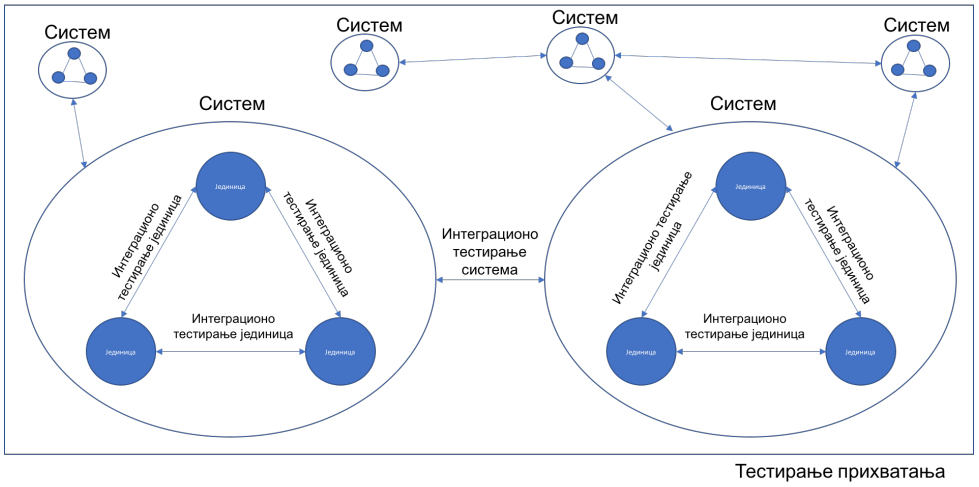
То се постиже константном укљученошћу тестирања и подизањем свести свих интересних страна о квалитету самог производа. Како се у Агилном окружењу ради по итерацијама, констатно се преиспитује да ли је производ добар крајњем кориснику, међутим додатни акценат се ставља и на то како је производ направљен и која је дугорочност истог.

4.1 Нивои тестирања

Када се погледа било који агилни оквир, може се приметити да је тестирање саставни део свих активности. Међутим, због лакшег разумевања и класификације тестирања, постоје четири нивоа тестирања која се испреплетано константно дешавају током развоја производа.

У овом раду ће бити обрађена сва четири нивоа тестирања која су неопходна за тестирање у агилном окружењу, међутим илустрација примером ће бити везана за аутоматско тестирање система. Поменута четири нивоа су:

- Тестирање јединице (енг. Unit/Component testing)
- Тестирање интеграције (енг. Integration testing)
- Тестирање система (енг. System testing)
- Одобравање (енг. Acceptance testing)



Слика 4.1: Визуелни приказ нивоа тестирања

4.1.1 Тестирање јединице

"**Јединица** је најмањи део софтвера који се може тестирати у изолацији."

Јединица може бити неколико линија кôда или може бити програм који садржи пуно линија кôда. Докле год постоји кôд који нешто смислено ради сам за себе то може бити јединица.

"**Тестирање јединице** је провера функционалности индивидуалне јединице софтвера"

Свака јединица је производ настао од захтева који су претходно дефинисани. Сврха тестирања јединица је да се потврди да се свака јединица понаша у складу са захтевима који су дефинисани.

Тестирање јединице је први и могло би се рећи и основнији ниво тестирања из разлога што се тестира одмах након што је свака јединица написана или пре него што је написана (нпр. **TDD**). Самим тим смањује се могућност каснијег откривања грешака и пробијања буџета самог пројекта. Како је неопходно програмерско знање, и контекст самог кôда, тестирање јединице се се у пракси показало као најбоље од стране аутора кôда који су писали програмски кôд. Као особе које јако добро познају то што су написали, уколико приступе деструктивно након или пре него што напишу јединицу кôда, тест који напишу квалитетно утиче на квалитет комплетног софтвера.

Како се тестови јединице пишу за сваки део кôда који чини једну целину, у току једног пројекта, програмери напишу велики број тестова. Како би могли да прате колико кôда је покривено тестовима, програмери користе неки од алата за мерење покривености кôда тестовима. Неки од примера су PyCharm, Coverity, SonarQube [2].

Када се погледа сам кôд, тестирање јединице може бити написана за функцију, методу, класу итд. Како су понекад потребни спољни улазни подаци како би сама функција имала смисла, то је случај када програмери пишу тест који као улазне параметре добија лажне (енг. mock) податке како би могао да изврши функцију до краја и провери да ли функција, метода или друга јединица ради.

4.1.2 Тестирање интеграције

Тестирање јединице је јако ефикасно и уколико је покривеност кôда тестовима велика, притом сами тестови квалитетни, квалитет софтвера је далеко већи. Међутим, тестирање јединице досеже најдаље до тестирања једне изоловане

јединице. Уколико је потребно проверити да ли комуникација између две јединице ради, ту тестирање јединице више нема смисла. Потребно је **тестирање интеграције** које покрива комуникацију између две или више јединице кода.

Ако се осврне на дефиницију [3] може се видети да:

"Тестирање које се извршава са циљем да покаже грешке у интерфејсима и интеракцијама између њихових интегрисаних компоненти или система зове се тестирање интеграције"

Дакле, тестирање интеграције помаже како би се разумело како подаци "путују" између система, како сам систем комуницира и да повезивањем више софтвера добија се један велики комплексан али и комплетан систем.

Тестирање интеграције се може поделити на две основне категорије:

- тестирање интеграције јединица
- тестирање интеграције система

4.1.2.1 Тестирање интеграције јединица

Развој софтвера се заснива на модуларном приступу, односно пише се програмски кôд којим се креирају модули који се повезују и заједно чине цео систем. Као што је претходно истакнуто, у тренутку када две јединице комуницирају, потребно је тестирати ту комуникацију. Сам софтвер чине стотине, хиљаде јединица кôда које комуницирају између себе. Тестирањем те комуникације добија се тестирање интеграције јединица.

4.1.3 Тестирање система

"Процес тестирања интегисаног система са циљем да се верификује да систем испуњава претходно дефинисане захтеве" [3]

Тренутак када се дође до тестирања система значи да се већ десило (барем делимично) тестирање јединице и тестирање интеграције. Као што је описано кроз рад, сва тестирања се дешавају паралелно у агилном окружењу, међутим

уколико се посматра једна функционалност, тестирање се мора десити секвенцијално кроз нивое тестирања од најнижег до највишег, уколико сама компанија не жели да дође у ситацију да грешку која се могла наћи у тестирању јединица нађе током тестирања система и самим тим плати много више него што је било потребно.

За тестирање система, потребно је неколико ствари као основа:

- Системски и софтверски захтеви
- Случајеви употребе
- Функционални захтеви

Тестирање система је први тренутак када су све претходно тестиране јединице и интеграције спојене заједно и тако чине један комплетан систем. Колико год се добро тестирају претходна два нивоа, постоји велика шанса да када се цео систем повеже, јављају се проблеми који се нису могли другачије уочити. Из тог разлога потребно је тестирати и фунцкионалне и нефункционалне захтеве система. Сами захтеви могу бити дефинисани на различите начине, и често су недовољно дефинисани.

4.1.3.1 Функционално тестирање система

Функционално тестирање система је тестирање функционалности које чине један систем.

Постоје два приступа приликом тестирања функционалности система:

- на основу функционалних спецификација (захтева)
- на основу пословних процеса

Корисничка прича

По традиционалном развоју софтвера, за све што се развија постоји пропратна спецификација која је написана пре самог почетка пројекта. У агилном развоју софтвера, корисник је на првом месту, самим тим спецификације се могу представити кроз корисничке приче (енг. User stories), и оне се могу написати на почетку развоја производа али се такође могу додавати касније у односу на потребе самог корисника. Као што је описано у поглављу 2.3, корисник жели

константан раст у самом производу, самим тим потребе се могу променити приликом развоја производа, што значи да спецификације морају да се прилагоде, могу бити скроз различите итд. Ово је све могуће из разлога што се планирање ради искључиво по спринтовима/итерацијама, самим тим на 2-4 недеље корисник може доћи са новим потребама које се планирају за кратак наредни период. Предност корисничких прича у развоју софтвера је што је корисник у фокусу самог развоја, али и зато што сама корисничка прича неће почети са развојем уколико цео тим није потврдио да разуме о чему се ради, прошао кроз све детаље и проценио колико ће требати да се исти заврши. Тестирање је овде кључан тренутак, из разлога што корисник/власник производа размишља из перспективе потреба и како нека функсионалност треба да изгледа, међутим често занемаре улогу тестирања саме функционалности. Приликом планирања, уколико не и раније, цео тим постаје свестан важности тестирања и колико је времена потребно да се тестира конкретна корисничка прича. Такође, може доћи до допуњавања из разлога што тек приликом размишљања о тестирању долази се до детаљнијих функционалности и потенцијалних проблема.

4.1.4 Тест прихватања

Већ у самом називу овог нивоа тестирања се може видети важност. Претходна три нивоа су везани пре свега на развој производа и како да буде што квалитетнији. Тест прихватања је ниво који се у тестирању у агилном окружењу понавља сваки пут пре него што се производ "пушта" на продукцију. Најчешће то ради сам корисник, власник производа или група људи који су најсличнији будућем кориснику, самим тим могу дати најрелевантнију информацију да ли је производ развијен исправно, односно верификовати сам производ.

Ако се осврне на дефиницију, тест прихватања би се могао представити као:

"Формално тестирање са акцентом на потребе корисника, спецификације, бизнис процесе тако да испуњавају задате критеријуме са циљем да корисник/власник производа финално прихвати или не прихвати производ" [3]

Сам циљ теста прихватања је да осигура поверење у систем, његове делове како функционалних тако и нефунционалних делова система. Налажење грешака није увек главни циљ теста прихватања, већ може бити верификација

идеје, да ли је оно што се прави заправо потребно.

За разлику од остала три нивоа, тест прихватања је неопходно добро планирати. Ако се узме SCRUM оквир као пример, тест прихватања се може десити на свака два, три спринта, што је између 4-8 и 6-12 недеља (у зависности колико недеља је један спринт).

Тест прихватања који укључује кориснике (енг. User Acceptance Testing или скраћено UAT) је најбитнији из горе поменутих разлога, и због тога је потребно добро планирати где ће се тестирати, када и колико људи ће тестирати. У односу на место где се обавља тестирање, разликују се две врсте UAT тестова:

Постоји пуно различитих начина како се ово ради, али међу познатијима су:

- Алфа тестирање (енг. Alpha testing)
- Бета тестирање (енг. Beta testing)

Алфа тестирање се обавља од стране потенцијалних корисника или изолованог тима у развојном окружењу (месту где се производ заправо развија), али без икаквог утицаја развојног тима.

Бета тестирање се обавља од стране потенцијалних корисника или стварних корисника на месту где ће се производ заправо користити.

Уколико је то интернет продавница која се развија из Србије за тржиште Велике Британије, Алфа тестирање би се обављало у Србији, док Бета тестирање мора бити у Великој Британији.

Глава 5

Технологије

У овом поглављу биће представљене све технологије које су коришћене за потребе израде различитих тестова.

5.1 Java

Јава (енг. Java) представља програмски језик опште намене који је конкурентан, класно заснован, објектно-оријентисан и од верзије Јава 8 са карактеристикама функционалних програмских језика, специјално дизајниран да има што је могуће мање имплементационе зависности. Компајлирани Јава код може да се покреће на свим платформама које подржавају Јаву без потребе за рекомпајлирањем, односно језик функционише по принципу "пиши једном, покрени било где". Јава апликације се преводе на бајткод, који се покреће на било којој Јава виртуелној машини (енг. JVM), без обзира на архитектуру рачунара. Јава је један од најпопуларнијих програмских језика у примени, користи се у разним типовима апликација попут Веб, мобилних, али и огромних серверских апликација [13].

Јаву је развио Џејмс Гослинг 1995. године у компанији Sun Microsystems, која је касније постала део корпорације Oracle. Синтакса великим делом потиче из језика C и C++, али за разлику од њих има објекте ниског нивоа. Selenium као алат за аутоматизацију је заснован на Јави.

Selenium има подршку за писање тестова у Groovy, Perl, PHP, C, Python, Rubi i Skala. Међутим како је сам написан у Јави, корисници се најчешће опредељују за писање тестова у Јави.

5.2 Selenium

Selenium је скуп алата различитих могућности за аутоматско тестирање апликација. Selenium има подршку за писање тестова у Groovy, Perl, PHP, C, Python, Rubi i Skala. Међутим како је сам написан у Јави, корисници се најчешће опредељују за писање тестова у Јави. Као софтвер отвореног кода, Selenium даје могућност свим корисницима да бесплатно користе сам алат.

Selenium укључује 3 компоненте:

- Selenium IDE
- Selenium WebDriver
- Selenium Grid

Selenium IDE (eng. Integrated Development Environment) је алат за креирање тест скрипти које се извршавају аутоматски. Другим речима, то је Mozilla Firefox додатак (енг. plugin) који омогућава снимање, модификацију и дебаговање скриптова.

Selenium WebDriver је друга компонента која се најчешће користи за аутоматско тестирање. Као компонента садржи све могућности Selenium IDE компоненте и има додатне могућности као што је тестирање свих претраживача, руковање различитим прозорима, тестирање динамичких компоненти на Web страници, итд.

За тестирање веб продавнице у раду ће се користити компонента WebDriver.

5.2.1 Иницијализација WebDriver-a

Да би се покренула било која интернет страница, неопходно је користити неки од предефинисаних WebDriver-a. У примеру овог рада користиће се ChromeDriver, у коме ће се отворити интернет страница и наставити тестирање. ChromeDriver је фајл који се мора сачувати негде на локалној машини и покретати сваки пут пре почетка теста. Детаљније отварање стране и почетак тестирања ће бити објашњен у наредном поглављу.

```
public static ChromeDriver selectDriver(){
    System.setProperty("webdriver.chrome.driver","/Users/
andrijamiljkovic/Downloads/chromedriver");
    WebDriver driver = new ChromeDriver();
    return (ChromeDriver) driver;
}
```

Скрипта 5.1: Дефинисање ChromeDriver-a

5.2.2 Проналажење и дефинисање елемената по странама

Selenium WebDriver подржава више могућности за проналажење елемената неке веб стране, и то су помоћу

- ID-a;
- Атрибута;
- Имена класе;
- Xpath-a;
- други.

Најчешће у употреби су претраживање помоћу ID-a и Xpath-a. Претраживање помоћу ID-a је најпрецизније и најстабилније, из разлога што чак и уколико дође до промене у развоју кˆда, WebDriver ће и даље проналазити тачно тај елемент који се тражи. ID се везује директно за само један елемент, и самим тим приликом претраге елемената резултат ће увек бити тачно један. Приликом претраге помоћу Xpath-a, уколико дође до промене кˆда, премештања изнад

или испод по веб страни, Xpath може пронаћи нешто што уопште није било тражено. Како је у агилном развоју софтвера промена саставни део развоја, у овом раду ће се користити претрага путем ID-а.

```
 driver.findElement(By.xpath("/select[@id=category]/option[@id=
cat2]")).click();
 driver.findElement(By.xpath("/select[@id=category]/option[text
()='Auto']")).click();

 ...
```

Скрипта 5.2: Проналажење и дефинисање елемената помоћу Xpath-а

```
@FindBy (id = "cart-item-quantity-field-")
WebElement QuantityFieldCardItem;

@FindBy (id = "submit")
WebElement EnterAddressButtonCheckout;

@FindBy (id = "continue-shopping")
WebElement ContinueShoppingCheckout;

...
```

Скрипта 5.3: Проналажење и дефинисање елемената помоћу ID-а

5.2.3 Имплицитно и експлицитно чекање

WebDriver може чекати или не да се у потпуности учита садржај неке веб странице. Како то зависи од различитих фактора, може се десити да страница није учитана скроз а извршавање теста се настави и дође до грешке. Зато је синхронизација између страница које се тестирају и WebDriver-а неопходна. То се постиже увођењем појма "чекања" на свакој страни у ситуацијама када је то потребно.

Постоји неколико различитих врста чекања:

- Имплицитно - систему се одређује одређен број секунди које мора да сачека пре него што настави са тестом;

- Експлицитно - систему се даје јасан услов који мора бити испуњен како би се тестирање наставило;
- Природно - комбинација имплицитног и експлицитног чекања, тако што постоји одређен услов који мора да се сачека, игноришући одређене грешке и проверавајући на одређени период да ли се елемент учитао.

На примеру тестирања Дремел интернет продавнице, користиће се природно чекање.

```
public WebElement fluentWait(final WebElement locator){
    Wait<WebDriver> wait = new FluentWait<WebDriver>(driver)
       .withTimeout(90, TimeUnit.SECONDS)
       .pollingEvery(2, TimeUnit.SECONDS)
       .ignoring(NoSuchElementException.class);

    WebElement foo = wait.until(
        new Function<WebDriver, WebElement>() {
            public WebElement apply(WebDriver driver) {
                return locator;
        }
    }
);
return foo;
```

Скрипта 5.4: Пример природног чекања

5.3 Junit

JUnit је слободног кода и представља оквир за тестирање [4]. Написан је за Јава програмски језик, међутим постао је толико популаран да је преведен и за остале језике (C, Python, Fortran, C++). Како би сами тестови имали смисла потребно је имати окружење у ком би се они писали користећи програмски језик у одређеној форми. JUnit је интегрисан у многа развојна окружења (Eclipse, NetBeans, JBuilder ...) самим тим олакшава развој програмерима из разлога што су многе операције могуће преко самог графичког интерфејса а компајлирање и извршавање тестова аутоматизовани.

Предности тестирања коришћењем JUnit-а:

- Једноставан за коришћење
- Сваки програмер може да научи да пише тестове
- Сви тестови се чувају на једном месту, самим тим одржавање и писање нових тестова је знатно олакшано
- Тестови написани у JUnit-у врше аутоматску проверу и пријаву грешака.

Најчешћа структура Junit-а може се видети у наредном примеру.

@Before део:

```
@Before
    public void setUp() throws Exception {
        driver = Constants.selectDriver();
        driver.manage().timeouts().implicitlyWait(60, TimeUnit.
   SECONDS);
        driver.manage().window().maximize();
    }
```

@Test део:

```
@Test
    public void CheckoutContentValidation_en_GB() throws
   InterruptedException, IOException{
        try {
        DOMConfigurator.configure("log4j.xml");
        Log.info("-- -- -- -- -- -- -- -- -- -- -- -- -- ");
        Log.info("GB Checkout | Content validation - Started");
        LandingPage landingPage = new LandingPage(driver);
        landingPage.openGb();
        landingPage.confirmCookieManager();
        landingPage.clickGlueGun930BuyNow();
        GlueGun glueGun = new GlueGun(driver);
        glueGun.addToCartGlueGun930Gb();
        glueGun.order();
        if(!glueGun.page.getText().contains(ContentGb.
   ShoppingCartTitle)){
            Log.warn(ContentGb.ShoppingCartTitle + " not shown");
            landingPage.takeScreenshot(new Object(){}.getClass().
   getEnclosingMethod().getName());
        }
        ...
```

@After део:

```
@After
    public void tearDown() {
        driver.quit();
    }
```

5.4 Мејвен

Мејвен (енг. Maven) је софтверски алат за управљање и изградњу пројекта [5]. Помоћу овог алата можемо припремити код за дистрибуцију. Такође, уз помоћ Мејвена можемо на лак начин укључити Java библиотеке (jar-ове) који су нам потребни у пројекту. Сва правила дефинишемо у датотеци pom.xml, која мора садржати групу, предмет за употребу, верзију и назив пројекта. У истој датотеци дефинишемо и начин паковања, да ли ће бити, на пример, jar (Java Application Arhive) или war (Web Application Arhive). Ово је веб-пројекат тако да ће начин паковања бити war. У dependencies делу pom.xml-а се дефинишу библиотеке које су потребне пројекту. Без Maven-а се све библиотеке (jar-ови) морају поставити ручно.

Пример укључивања потребних библиотека у пројекат:

```
<!-- https://mvnrepository.com/artifact/org.seleniumhq.selenium/
   selenium-java -->
<dependency>
 <groupId>org.seleniumhq.selenium</groupId>
 <artifactId>selenium-java</artifactId>
 <version>3.6.0</version>
</dependency>

<!-- https://mvnrepository.com/artifact/junit/junit -->
<dependency>
 <groupId>junit</groupId>
 <artifactId>junit</artifactId>
 <version>4.12</version>
 <scope>test</scope>
</dependency>

<!-- https://mvnrepository.com/artifact/log4j/log4j -->
<dependency>
 <groupId>log4j</groupId>
 <artifactId>log4j</artifactId>
 <version>1.2.17</version>
</dependency>
```

Постоји неколико фаза животног циклуса изградње пројекта:

- **validate** (врши се валидација пројекта)
- **compile** (врши се компилација извршног кода, компајлиран код се смешта у директоријум target у пројекту)
- **package** (узима се компајлиран код из target директоријума и пакује се у формат за дистрибуцију, на пример, war)
- **install** (направљен пакет се инсталира на локалном репозитпријуму да би пројекат могао да се користи локално у другим пројектима као dependency)
- **deploy** (копира се пакет на удаљени репозиторијум и на тај начин пакет постаје доступан осталим развојним инжењерима и пројектима)

5.5 Лог4j (енг. Log4j)

Између осталог, аутоматски тестови имају за сврху да смање време које се користи за мануелно тестирање, пре свега функционалности које се константно на исти начин тестирају. Такође, постоје ствари које нема ни смисла тестирати мануелно, самим тим, аутоматски тестови ступају на снагу. Као такве, могуће их је бити јако пуно, и могу се извршавати цео дан. Особу задужену за тестирање интересује финално стање самих тестова као и у томе може помоћи Log4j.

Због тога је битно да се направи добро логирање извшавања апликације, како би се лакше пронашао проблем и приликом развоја, а посебно при одржавању апликације у току извршавања. У Јава свету најпознатији алат за логирање је Лог4j (енг. Log4j). Дистрибуира се под лиценцом Apache Software [16]. Лог4j се може конфигурисати преко екстерних конфигурационих датотека у току извршавања, обично као .xml или .properties формат. Надгледа процес логирања у смислу дефинисања нивоа приоритета и нуди механизме за усмеравање информација о евиденцији до великог броја дестинација, као што су база података, датотека, конзола, системски логови, итд. Користи више нивоа, у које по приоритету спадају: ALL, TRACE, DEBUG, INFO, WARN, ERROR, FATAL.

Пример како се подешава упис логова у датотеку:

```
# Root logger option
log4j.rootLogger=DEBUG, stdout, file

//  # Redirect log messages to console
// log4j.appender.stdout=org.apache.log4j.ConsoleAppender
// log4j.appender.stdout.Target=System.out
// log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
// log4j.appender.stdout.layout.ConversionPattern=%d{yyyy-MM-dd HH:
   mm:ss} %-5p %c{1}:%L - %m%n
```

Пример исписа log4j лога:

```
2017-10-20 20:15:19 INFO    -- -- -- -- -- -- -- -- -- -- -- -- --
2017-10-20 20:15:24 INFO  GB Links verification - Started
2017-10-20 20:15:24 INFO  Press link ok
2017-10-20 20:15:24 INFO  T&C link ok
2017-10-20 20:15:24 INFO  Privacy statement link ok
2017-10-20 20:15:24 INFO  Warranty link ok
2017-10-20 20:15:25 INFO  Dremel Europe link ok
2017-10-20 20:15:25 INFO  Sitemap link ok
2017-10-20 20:15:25 INFO  GB Links verification - Ended
   SUCCESSFULLY

2017-10-20 20:17:30 INFO  -- -- -- -- -- -- -- -- -- -- -- -- --
2017-10-20 20:17:30 INFO  GB Checkout | Content validation -
   Started
2017-10-20 20:18:37 INFO  GB Checkout | Content validation - Ended
   SUCCESSFULLY
2017-10-20 20:24:42 INFO  -- -- -- -- -- -- -- -- -- -- -- -- --
2017-10-20 20:24:42 INFO  GB Checkout | Guest | Express shipping -
   Started
2017-10-20 20:25:49 INFO  GB Checkout | Guest | Express shipping -
   Ended SUCCESSFULLY
```

Глава 6

Аутоматизација у агилном окружењу

6.1 Аутоматизација

Једно од најчешћих питања која могу доћи како од стране руководиоца тако и од стране програмера: "Како одлучити шта аутоматизовати?".

На почетку је доста битно прихватити чињеницу да није могуће све аутоматизовати, чак није ни пожељно јер се у Агилном окружењу ствари брзо мењају, самим тим сваки вид аутоматизације је потребно одржавати константно. Број аутоматских тестова који је потребно одржавати се пропорцијално повећава повећању нових функционалности и било којим променама к^да у оквиру производа.

Један од начина који се користи приликом приоритизације аутоматских тестова и смернице које ствари треба прво аутоматизовати и у којој мери је ***Тест пирамида (енг. Test pyramid)***

Тест пирамида јасно показује који ниво колико ресурса захтева и колико је скуп сам ниво. Приликом креирања производа, само програмирање креће од најмање јединице к^да, када се најчешће и пишу јединични тестови. Након што се више јединица к^да може повезази у смислену целину, пишу се интеграциони тестови који показују да је све у реду у комуникацији између различитих компоненти. Када се заврши програмирање система, пре свега се раде мануелни тестови који се морају испратити са аутоматским тестовима.

Слика 6.1: Визуелни приказ тест пирамиде аутоматизације

Проблеми/заблуде до којих наилази већина компанија су:

- Аутоматизација се пише једном и након тога ради сама од себе
- Аутоматизација једног система се после лако употреби за аутоматизацију другог система
- Све се може аутоматизовати
- итд.

Аутоматизација је процес који се као и сам код апликације мора одржавати, преправљати, брисати, рефакторисати итд. Сваки систем је за себе јако специфичан, и већина ствари се изнова мора креирати за други систем. Не може се све аутоматизовати, и то је чињеница на коју менаџмент већине фирми тешка срца пристаје. Као што се може видети на пирамиди изнад, јединично и интеграционо тестирање треба максимално аутоматизовати, јер је то покретач саме апликације. Кориснички интерфејс (енг. user interface или UI) је најтеже аутоматизовати из разлога што је то крајњи приказ саме апликације и узрок сваке грешке може бити било где у систему. Зато се приликом аутоматизације корисничког интерфејса пре свега треба договорити са менаџментом приоритете апликације и тим редоследом аутоматизовати. Такође, највише се измена ради на самом корисничком интерфејсу, што даље доводи до тога да се сваки тест мора преправити у складу са изменама.

6.2 Веб страна која ће се користити за потребе аутоматизације

Дремел веб продавница је интернет продавница преко које крајњи корисник може купити било који производ бренда Дремел[1]. Крајњи корисник се преко ове интернет продавнице може информисати о самом производу, претраживати више производа и финално купити жељени производ.

Дремел интернет продавница функционише по сличном принципу као и већина других интернет продавница, саме процесе можемо поделити на два дела:

- Куповина производа
- Враћање производа

Уколико би се поделила продавница по нивоима тестирања, могло би се представити на следећи начин:

- ***Јединично тестирање*** - то би могло бити тестирање функционалности *"додавање у корпу"*, односно тестирање самог кôда који се односи на додавање производа у корпу;
- ***Интеграционо тестирање*** - то би могло бити тестирање комуникације између две функције (тестирање интеграције у малом) или између два система, нпр. интернет продавнице и сервиса за плаћање (тестирање интеграције у великом);
- ***Тестирање система*** - тестирање система који ће корисници користити. У овом случају то је Дремел интернет продавница;
- ***Тестирање одобравања*** - одобравање од особа/фирме која заправо шаље захтеве и на основу чега се и креирао производ. Најчешће је то финални производ који ће корисници користити. У овом случају то је такође тестирање крајњег система који ће корисници користити, међутим без фокуса на делове самог производа, већ праћење случајева који ће имати крајњи корисник: Претрага производа, одабир производа, куповина производа, испорука производа, враћање производа.

[1] Дремел је амерички производ електричних алата познатијих као ротирајући алати. Референца на бренд: https://en.wikipedia.org/wiki/Dremel, и на продавницу у наставку https://shop.dremeleurope.com/de/de/produkte

У случају Дремел веб странице, сами приоритети корисничког интерфејса се могу поставити у односу на циљеве крајњег корисника:

(a) Куповина производа;

(b) Логовање корисника;

(c) Претраживање производа.

У овом раду ће се аутоматизовати процес куповине једног Дремел производа (до тренутка плаћања).

6.3 Аутоматизација веб стране

6.3.1 Модел странице као објекта (енг. Page object model)

За једну веб продавницу, зарад провере критичних функционалности, могуће је написати и преко 200 тестова. Ти тестови већим делом користе скуп истих или сличних функција. Да би се избегло дуплирање к^да и неконзистентност између тестова, користи се модел странице као објекта.

Модел странице као објекта је објектно орјентисана класа која је главна класа за све тестове. То значи да се странице које су написане да се понашају као интерфејси морају бити додати као библиотека у сваки тест, и позивањем истих могу се користити приликом креирања нових функција. Предност овог приступа је што све промене се дешавају на једном месту уколико је неопходно променити нешто.

Модел странице као објекта садржи наредне предности:

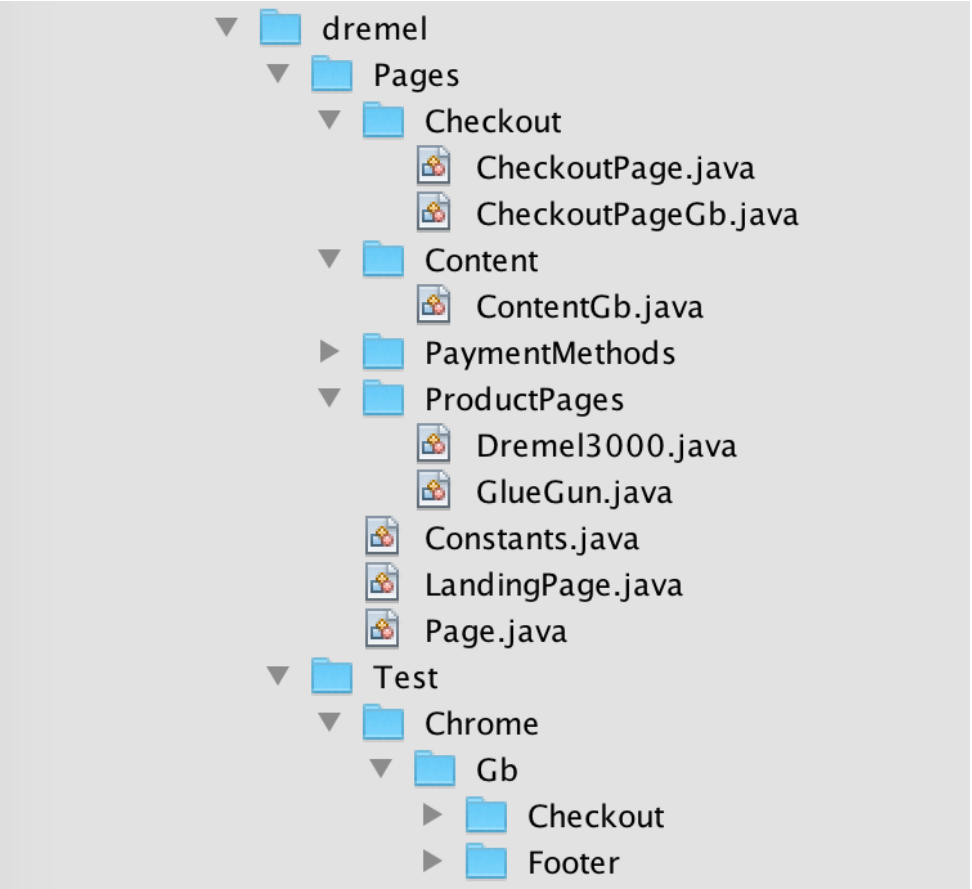
- Јасно су раздвојени тестови од специфичног к^да као што су идентификатори;
- Постоји само једно место који садржи већину сервиса и операција који се користе уместо да за сваки тест буде појединачно декларисан.

6.3.2 Архитектура тестова Дремел интернет продавнице

Пројекат је подељен у два дирекнторијума: Pages и Test.

Pages фолдер садржи класе које описују елементе и методе Дремел продавнице.

Test фолдер садржи класе са тестовима који се извршавају.



Слика 6.2: Визуелни приказ Pages и Test фолдера

6.3.2.1 Pages/Page.java

Ово је главна класа која описује све елементе који се налазе на свим странама интернет продавнице, самим тим понављаће се у сваком тесту који се напише. Све остале стране наслеђују ову класу.

Класа Page.java наслеђује PageFactory у конструктору. PageFactory је предефинисана библиотека која се највише користи као подршка Page Object модела.

```
public WebDriver driver;

 // MAIN CLASS CONSTRUCTOR
 public Page (WebDriver driver){
   this.driver=driver;
   PageFactory.initElements(driver, this);
 }
```

Скрипта 6.1: Дефинисање главне класе

Извршавање к^да се дешава истог тренутка, међутим како је интернет продавница састављена од пуно различитих сервиса може доћи до кашњења са стране корисничког интерфејса. Да би сам аутоматски тест изгледао што природније и да би уопште био могућ, потребно је увести појам "чекања" на свакој страни у ситуацијама када је то потребно.

```
public WebElement fluentWait(final WebElement locator){
    Wait<WebDriver> wait = new FluentWait<WebDriver>(driver)
       .withTimeout(90, TimeUnit.SECONDS)
       .pollingEvery(2, TimeUnit.SECONDS)
       .ignoring(NoSuchElementException.class);

    WebElement foo = wait.until(
        new Function<WebDriver, WebElement>() {
            public WebElement apply(WebDriver driver) {
                return locator;
        }
    }
);
return foo;
```

Скрипта 6.2: Класа за дефинисање чекања током аутоматског тестирања

6.3.2.2 Pages/LandingPage.java

Корисник приликом приступа на интернет продавницу види такозвану почетну страну (енг. Landing Page). У класи почетне стране су дефинисани сви елементи које корисник види када приступи интернет продавници.

```
// Elements

 // DREMEL 3000
 @FindBy (id = "7736-buy-now")
 WebElement Dremel3000BuyNow;

 // DREMEL Glue Gun 930
 @FindBy (id = "7894-buy-now")
 WebElement GlueGun930BuyNow;


 // DREMEL Glue Gun 940
 @FindBy (id = "7895-buy-now")
 WebElement GlueGun940BuyNow;
```

Скрипта 6.3: Проналажење ID-јева елемената и додељивање веб елементу

Методе које су дефинисане у оквиру ове странице су:

- Отварање прве стране - стандардни метод који ће се понављати у већини тестова из разлога што је непходно отворити интернет претраживач.
- Прихватање менаџера колачића (енг. Cookie manager) - Приликом сваког првог појављивања на интернет продавницу, потребно је прихватити колачиће како би кориснички основни кораци кроз саму продавницу били запамћени, поготово уколико дође до прекида мреже или гашења рачунара итд. Како већина интернет продавница захтева прихватање колачића, неопходно је имати методу која ће регулисати тај део, у којој год страници да се нађе тест приликом отварања прве стране.
- Клик на један од два изабрана производа - обе методе су исте, потребно је сачекати да се страна учита и онда направити клик на претходно идентификовани елемент (помоћу идентификатора претходно описаних).

```
// Methods

    // OPEN FIRST PAGE (LANDING PAGE)

    public void openGb(){
        driver.get(Constants.LandingPageGb);
    }

    public void confirmCookieManager(){
        fluentWait(OkCookieManagerHeader).click();
    }

    // BUY NOW ON LANDING PAGE

    public void clickDremel3000BuyNow() throws InterruptedException
   {
        Thread.sleep(Constants.SmallWaitingTime);
        scroll_element_into_view(Dremel3000BuyNow);
        new Actions(driver).moveToElement(fluentWait(
   Dremel3000BuyNow)).click().build().perform();
    }

    public void clickGlueGun930BuyNow() throws InterruptedException
   {
        Thread.sleep(Constants.SmallWaitingTime);
        scroll_element_into_view(GlueGun930BuyNow);
        new Actions(driver).moveToElement(fluentWait(
   GlueGun930BuyNow)).click().build().perform();
    }
```

Скрипта 6.4: Методе за отварање прве стране, прихватање колачића и клик на један од два производа

6.3.2.3 Pages/Constants.java

Класа **Constants.java** дефинише већину статичких елемената који ће се користити у тестовима.

Као што је описано у другом поглављу, само тестирање се дешава паралелно са развојем производа. То значи да се сав код који је написан прво компајлира на рачунару за тестирање па тек онда на продукционој. То се одражава на тест

тако што је неопходно написати линкове за више инстанци и онда се приликом коришћења само коментаришу инстанце које се не користе, док је једна активна. У овом тесту активно окружење је продукционо [2], док су тестна окружења стављена само као пример:

```
  // Instance

   // P INSTANCE public static final String INSTANCE_NAME = "https
  ://shop.dremeleurope.com";
 // Q INSTANCE public static final String INSTANCE_NAME = "https
  ://testshop.dremeleurope.com";
// S INSTANCE
 public static final String INSTANCE_NAME = "https://testshop.
  dremeleurope.com";
```

Скрипта 6.5: Одабир са које инстанце ће се тест покретати, тестно или продукционо окружење

Када је инстанца позната, остаје да се надгради са коначном дестинацијом почетне стране. Постоји неколико могућности одакле тест може да почне, те је најбоље написати све комбинације и касније користити оне које су неопходне.

Неколико различитих које постоје су:

- LandingPageGb → основна почетна страна;
- Step1PageGb → страна након почетне стране, такозвани први корак;
- Step2PageGb → друга страна, страна за уписивање адресе и података корисника;
- Step3PageOrderConfirmationGb → трећа страна, на којој корисник потврђује поруџбину финално пре плаћања;
- Step4PageGB → страна за плаћање.

[2] У развоју софтвера, окружење је рачунарски софтвер у коме се програм или софтверска компонента развијају и извршавају. Постоји више различитих окружења као што су: продукционо, тестно, домаће програмерско, итд.

```
// PAGES FOR GB

   public static String LandingPageGb = INSTANCE_NAME + "/gb/en/
  products";
   public static String Step1PageGb = INSTANCE_NAME + "/gb/en/
  checkout/step1/-/cart/cartSummary";
   public static String Step2PageGb = INSTANCE_NAME + "/gb/en/
  checkout/step2";
   public static String PriceChangePageGb = INSTANCE_NAME + "/gb/
  en/checkout/step1/-/cart/cartSummary";
   public static String Step3PageOrderConfirmationGb =
  INSTANCE_NAME + "/gb/en/checkout/order-confirmation?
  orderConfirmation";
   public static String Step4PageGb = "https://test.adyen.com/hpp/
  pay.shtml";
```

Скрипта 6.6: Наставак на странице, како би тест могао директно да приступи некој од страница интернет продавнице

Како би сваки тест био проверен уникат, као што је сваки корисник (уколико је нерегистровани корисник), потребно је да се генерише сваки пут име, презиме, адреса итд. То се може урадити на начин описан у наставку.

Како интернет продавнице потврде куповине шаљу на мејл адресу наведену приликом куповине, за тестирање ће се користити једна мејл адреса.

```
// GUEST USER DATA

    public static String NameGuest = RandomStringUtils.
   randomAlphabetic(7).toLowerCase();
    public static String LastnameGuest = RandomStringUtils.
   randomAlphabetic(9).toLowerCase();
    public static String AddressGuest = RandomStringUtils.
   randomAlphabetic(9).toLowerCase() + RandomStringUtils.
   randomNumeric(3);
    public static String SecondLineAddress = RandomStringUtils.
   randomNumeric(3);
    public static String TownGuest = RandomStringUtils.
   randomAlphabetic(6).toLowerCase();
    public static String TelephoneGuest = RandomStringUtils.
   randomNumeric(9);
    public static String PostalCodeGuestGb = "aa12 3" +
   RandomStringUtils.randomAlphabetic(2).toLowerCase();
    public static String AccountDataGuest = "dremeltester@gmail.com
   ";
```

Скрипта 6.7: Генерисање података за попуњавање странице за адресу корисника

Како је аутоматско тестирање код који се извршава оног тренутка када се покрене, а за извршавање је потребно неколико секунди, неопходно је да се предвиде сва могућа кашњења са стране интернет продавнице како се рад аутоматског теста не би угрозио. То значи да је потребно дефинисати места где иако тест стигне кроз секунд, стане и сачека да се учита комплетна страна како би несметано наставио са радом. Испод се могу видети сва времена која су дефинисана у односу на део интернет продавнице када се до њих долази. Тако нпр постоји време чекања да се учита сервис за плаћање, док уколико је дошло до промене цене, потребно је сачекати да се поново генерише цела страна са новом ценом, итд.

```
// Sleep timeouts

    public static int AdyenLoadingTime = 16000;
    public static int FillInStep123LoadingTime = 8000;
    public static int SmallWaitingTime = 2800;
    public static int PriceChangeUrlWaitingTime = 20;
    public static int ValidationUrlWaitingTime = 30;
```

Скрипта 6.8: Чекања током аутоматског тестирања

Са стране корисника, можда и највећа фрустрација долази у тренутку када линк који је корисник желео да погледа не ради. Линкови које већина корисника жели да погледа налазе се на свим интернет продавницама истакнути у заглављу или подножју сваке странице. У подножју Дремел интернет продавнице, линкови који су кориснику потребни се налазе у подножју и како се већина њих односи на теме као што су права корисника, сервисирање корисника итд, неопходно је да раде у сваком тренутку. Зато ће постојати тестови који ће искључиво проверавати рад линкова у подножју странице.

```
// FOOTER HYPERLINKS

    public static String PressOfficeHyperlinkGb = "https://dremel-
   relaunch.kittelberger.net/gb/en/support/customer-service/press-
   office/";
    public static String TermsAndConditionsHyperlinkGb = "https://
   shop-s.dremeleurope.com/gb/en/customerservice/terms-condition";
    public static String PrivacyStatementHyperlinkGb = "https://
   shop-s.dremeleurope.com/gb/en/customerservice/dataprivacy";
    public static String WarrantyHyperlinkGb = "https://dremel-
   relaunch.kittelberger.net/gb/en/support/customer-service/
   warranty/";
    public static String DremelEuropeHyperlinkGb = "https://dremel-
   relaunch.kittelberger.net/gb/en/";
    public static String SitemapHyperlinkGb = "https://dremel-
   relaunch.kittelberger.net/gb/en/sitemap/";
```

Скрипта 6.9: Пример линкова из подножја стране, такозвани футер линкови

Последња функција у овој класи је можда и најважнија са стране покретања тестова. Да би се било који тест покренуо неопходно је дефинисати који претраживач ће се користити. У примеру Дремел интернет продавнице ће се користити Chrome интернет претраживач, међутим могуће је користити било који други једноставним додавањем и дефинисањем тих претраживача у овој класи.

```
public static ChromeDriver selectDriver(){
    System.setProperty("webdriver.chrome.driver","/Users/
andrijamiljkovic/Downloads/chromedriver");
    WebDriver driver = new ChromeDriver();
    return (ChromeDriver) driver;
}
```

Скрипта 6.10: Дефинисање ChromeDriver-а

6.3.2.4 Pages/Checkout

CheckoutPage.java је главна класа која дефинише све елементе завршне стране.

Од тренутка када се "убаци производ у корпу", до тренутка куповине постоји више страна које садрже пуно елемената по страни. Како су те странице у сваком случају исте, без обзира колико и који производи се купују, најбоље их је све дефинисати на једном месту као константе. Класа CheckoutPage.java ради управо то.

Елементи су дефинисани истим редом којим путује корисник кроз интернет продавницу од тренутка када пређе и стране где је бирао производ у страну која му омогућује куповину.

```
// ELEMENTS - STEP 1

    // Card item quantity field (Step 1)
    @FindBy (id = "cart-item-quantity-field-")
    WebElement QuantityFieldCardItem;

    // Enter address (Step 1)
    @FindBy (id = "submit")
    WebElement EnterAddressButtonCheckout;

    // Continue shopping button (Step 1)
    @FindBy (id = "continue-shopping")
    WebElement ContinueShoppingCheckout;

    // ELEMENTS - STEP 2

    // Checkout (Step 2)
    @FindBy (id = "without-account-guest-radio-option")
    WebElement WithoutAccountCheckoutRadioButtonCheckout;
    ...

    // STEP 3

    // Edit Billing address Step 2 from Step 3 (Step 3)
    @FindBy (id = "edit-checkout-step-2")
    WebElement EditBillingAddressCheckout;

    // Edit Delivery address Step 2 from Step 3 (Step 3)
    @FindBy (id = "edit-delivery-address-checkout-step-2")
    WebElement EditDeliveryAddressCheckout;
    ...
```

Скрипта 6.11: Проналажење и дефинисање елемената по странама

```
    public void checkOnExpressField() throws InterruptedException{
        Thread.sleep(Constants.SmallWaitingTime);
        fluentWait(ExpressCheckboxUnregisteredCheckout).click();
        Thread.sleep(Constants.SmallWaitingTime);
    }

    // Fill in all the fields on Step 1
    public void fillInStep1() throws InterruptedException{
        Thread.sleep(Constants.SmallWaitingTime);
        new Actions(driver).moveToElement(fluentWait(
   EnterAddressButtonCheckout)).click().build().perform();
//        fluentWait(EnterAddressButtonCheckout).click();
    }

    // Confirmation of Step 2 and moving to Step 3
    public void confirmStep2() throws InterruptedException{
        Thread.sleep(Constants.SmallWaitingTime);
//        fluentWait(ContinueButtonUnregisteredCheckout).click();
        scroll_element_into_view(ContinueButtonUnregisteredCheckout
   );
        new Actions(driver).moveToElement(fluentWait(
   ContinueButtonUnregisteredCheckout)).click().build().perform();
    }

    // Confirmation of Step 3 and moving to step 4
    public void confirmStep3(){
        fluentWait(GoToPaymentPageButton).click();
    }
```

Скрипта 6.12: Методе за попуњавање више Checkout страница

CheckoutPageGb.java је класа која наслеђује CheckoutPage.java класу и додатно дефинише специфичне методе које су везане само за конкретну земљу, односно језик користећи њене методе. Уколико би се променио језик на било коју другу земљу, потребно би било направити додатне класе које су специфичне за те земље. У примеру Дремел интернет продавнице је покривена само земља Велике Британије, јер је најлакше за разумевање како је енглески језик у питању.

Након добро дефинисане CheckoutPage.java класе, позивањем разних метода може се доћи до комплетног теста више страна. Пример такве методе је у наставку:

```
  // Integrated Step 2 and Step 3
 public void fillInSteps123GuestWithExpress() throws
InterruptedException{
     fillInStep1();
     fillInStep2withExpress();
     confirmStep2();
     try {
        if (new WebDriverWait(driver, Constants.
PriceChangeUrlWaitingTime).until(ExpectedConditions.urlToBe(
Constants.PriceChangePageGb))){
         fillInStep1();
         confirmStep2();
     }
     } catch (Exception e) {
     }
         fillInStep3();
 }
```

Скрипта 6.13: Пример попуњавања више страна позивањем метода из две класе

6.3.2.5 Pages/ContentGb.java

Постоји пуно садржаја на страницама који се не мења током било ког теста. Тај садржај је најбоље држати на једном месту и када год је потребна провера или коришћење истог, лако се долази до тог садржаја.

```
// SHIPPINGS

    public static String ExpressShipping = " 3 .00";
    public static String NoShipping = " 0 .00";
    public static String StandardAndExpressShipping = " 8 .95";
    public static String StandardShipping = " 5 .95";

    // Step 1
    public static String ShoppingCartTitle = "SHOPPING CART";
    public static String PleaseCheckYourShoppingCart = "Please
   check your shopping cart, enter the missing information and send
    your order!";

    // Step 2
    public static String TitleIsRequired = "Title is required";
    public static String FirstNameIsRequired = "First name is
   required";
    public static String LastNameIsRequired = "Last name is
   required";
    public static String StreetIsRequired = "Street is required";
    public static String CityIsRequired = "City is required";
    public static String EmailIsRequired = "Email is required";
    public static String InvoiceAddress = "INVOICE ADDRESS";
    public static String DeliveryAddress = "DELIVERY ADDRESS";
    public static String AccountData = "ACCOUNT DATA";
    public static String WantToReceiveYourProduct = "WANT TO
   RECEIVE YOUR PRODUCTS EVEN FASTER?";
    public static String PostalIsRequired = "Postal code is
   required";

    // Step 3
    public static String BillingAddress = "BILLING ADDRESS";
    public static String ShippingOption = "SHIPPING OPTION";
```

Скрипта 6.14: Садржај који се појављује на свим страницама

6.3.2.6 Pages/Dremel3000.java

Свака интернет продавница садржи јако пуно производа. На нивоу функционалног аутоматског тестирања, фокус је на фукционалностима, самим тим потребно је изабрати пар производа који су констатно у продаји и њих аутоматизовати како би аутоматски тестови били што дуже валидни и без одржавања исправни. У случају Дремел интернет продавнице, производ који је изабран је Dremel 300. Самим тим класа Dremel3000.java дефинише све елементе који се налазе на страни до које се долази кликом на производ Dremel 3000.

```
// DREMEL 3000 4 Star Kit (3000-3/55)

@FindBy (id = "F0133000MJ-add-to-cart")
WebElement Dremel30004StarKitAddToCartGB;

// DREMEL 3000 (3000-15)
@FindBy (id = "F0133000JB-add-to-cart")
WebElement Dremel300015AddToCartGB;

// DREMEL 3000 (3000-1/25 EZ)

@FindBy (id = "F0133000JR-add-to-cart")
WebElement Dremel300025EzAddToCartGB;

// DREMEL 3000 5 Star Kit (3000-5/75)

@FindBy (id = "F0133000MP-add-to-cart")
WebElement Dremel30005StarKitAddToCartGB;

// METHODS

public void addToCartDremel3000(){
    fluentWait(Dremel300015AddToCartGB).click();
}
```

Скрипта 6.15: Елементи на интернет страници за дефинисан производ

6.3.2.7 Pages/Adyen.java

Када је плаћање у питању, неопходно је имати исправне картице за тестирање како би цео процес могао да се тестира. У примеру Дремел интернет продавнице, тестира се директно на продукцији, самим тим последњи корак тестирања је управо провера уписа података на страници за плаћање без клика на дугме за плаћање. Како се плаћање у највећем броју случајева интегрише у интернет продавницу, на продукцији је довољно проверити да ли се страница за плаћање појављује и да ли је могуће уписати податке за куповину.

```
// Payment
 public void creditCardPaySingleCard() throws
InterruptedException{
     new WebDriverWait(driver, Constants.
PriceChangeUrlWaitingTime).until(ExpectedConditions.
elementToBeClickable(fluentWait(CardNumberField))).click();
     fluentWait(CardNumberField).clear();
     fluentWait(CardNumberField).sendKeys(Constants.
CreditCardNumber);
     fluentWait(CardHolderNameField).click();
     fluentWait(CardHolderNameField).clear();
     fluentWait(CardHolderNameField).sendKeys("Thanks for
watching!");
     new Select(fluentWait(CardExpiryMonthField)).
selectByVisibleText(Constants.CardExpiryMonth);
     new Select(fluentWait(CardExpiryYearField)).
selectByVisibleText(Constants.CardExpiryYear);
     fluentWait(CvcCardField).click();
     fluentWait(CvcCardField).clear();
     fluentWait(CvcCardField).sendKeys(Constants.CvcCard);

     Thread.sleep(Constants.AdyenLoadingTime);
 }
```

Скрипта 6.16: Пример странице за плаћање

6.3.2.8 Test/FooterLinkVerification.java

Као што је описано већ у оквиру Constants.java класе, футер линкови су контантни и неопходно их је проверавати јер лако може доћи до грешке приликом

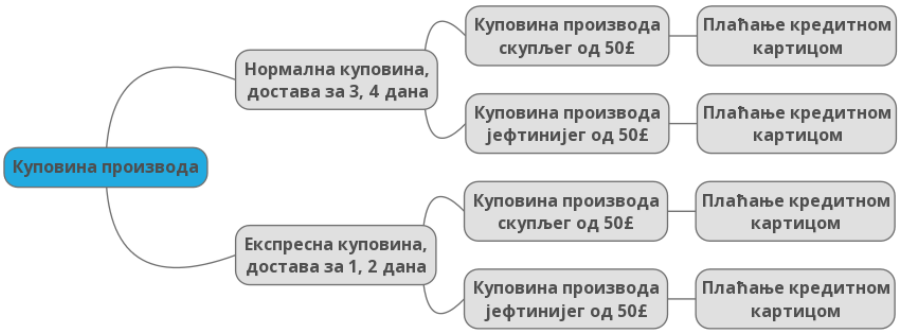
копирања новог кˆда на продукцију или тестни систем. Како се ту налазе линкови који су кориснику неопходни, тест који је везан само за футер је издвојен од осталих тестова. Може се приметити да је тест крајње једноставан. Пролази се кроз сваки линк и упоређује да ли је исправан. Уколико било који од линкова није исправан, тест пада.

```
@Test
    public void FooterLinkVerificationGb() throws
   InterruptedException{
        try {
        DOMConfigurator.configure("log4j.xml");
        Log.info("-- -- -- -- -- -- -- -- -- -- -- ");
        Log.info("GB Links verification - Started");
        LandingPage landingPage = new LandingPage(driver);
        landingPage.openGb();
        if(!landingPage.getPressOfficeHyperlink().equals(Constants.
   PressOfficeHyperlinkGb)){
            Log.warn(Constants.PressOfficeHyperlinkGb + " not shown
   ");
        } Log.info("Press link ok");
        if(!landingPage.getTermsAndConditionsHyperlink().equals(
   Constants.TermsAndConditionsHyperlinkGb)){
            Log.warn(Constants.TermsAndConditionsHyperlinkGb + "
   not shown");
        } Log.info("T&C link ok");
        if(!landingPage.getPrivacyStatemenHyperlink().equals(
   Constants.PrivacyStatementHyperlinkGb)){
            Log.warn(Constants.PrivacyStatementHyperlinkGb + " not
   shown");
        } Log.info("Privacy statement link ok");

        ...

        Log.info("GB Links verification - Ended SUCCESSFULLY");
        } catch (WebDriverException e) {
            Log.error("Failed with message: " + e.getMessage().
   substring(0, e.getMessage().indexOf("\n")));
        }
    }
```

Скрипта 6.17: Тестирање подножја интернет странице, футера

6.3.2.9 Test/CheckoutGuestUserExpressShipping.java

Већина интернет продавница нуди могућност различитих врста куповине истог производа, као што су нпр. нормална или експресна куповина, са или без доплате за доставу, различите могућности плаћања итд. На примеру Дремел интернет продавнице у наставку се може видети које су све могућности куповине производа. За сваку од тих могућности потребно је написати различит тест како би тест прошао кроз цео процес.



Слика 6.3: Визуелни приказ различитих могућности куповине преко Дремел интернет продавнице

Финално, комплетан процес се може одразити кроз ову класу, самим тим је најбоље проћи кроз кораке које заправо корисник у највећем броју случајева мора да уради како би купио производ:

(a) Отварање интернет претраживача и Дремел интернет продавнице

(b) Прихватање колачића

(c) Клик на жељени производ

(d) Убацивање производа у корпу

(e) Попуњавање странице са корисничким подацима

(f) Прихватање услова куповине, провера података и прелазак на сервис за куповину

(g) Уписивање података за куповину и куповина самог производа

У наставку се може видети сваки од корака изражен кроз код:

```
@Test
    public void CheckoutGuestUserExpressShipping_en_GB() throws
   InterruptedException, IOException{
        try {
        DOMConfigurator.configure("log4j.xml");
        Log.info("-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
   -- -- -- -- -- -- -- -- -- -- ");
        Log.info("GB Checkout | Guest | Express shipping - Started"
   );
        LandingPage landingPage = new LandingPage(driver);
        landingPage.openGb();
        landingPage.confirmCookieManager();
        landingPage.clickDremel3000BuyNow();
        Dremel3000 dremel3000 = new Dremel3000(driver);
        dremel3000.addToCartDremel30004StarKitGb();
        dremel3000.order();
        CheckoutPageGb checkoutPage = new CheckoutPageGb(driver);
        checkoutPage.fillInSteps123GuestWithExpress();
        if (!checkoutPage.currentUrl().equals(Constants.
   Step3PageOrderConfirmationGb)){
            Log.warn("Order Confirmation step of checkout not shown
   ");
            landingPage.takeScreenshot(new Object(){}.getClass().
   getEnclosingMethod().getName());
            assertTrue(false);
        }
        if (!checkoutPage.page.getText().contains(ContentGb.
   ExpressShipping)) {
            Log.warn(ContentGb.ExpressShipping + " for Express
   SHIPPING not shown");
            landingPage.takeScreenshot(new Object(){}.getClass().
   getEnclosingMethod().getName());
            assertTrue(false);
        }
        checkoutPage.confirmStep3();
        Adyen adyen = new Adyen(driver);
        adyen.creditCardPaySingleCard();
```

```
        Log.info("GB Checkout | Guest | Express shipping - Ended
    SUCCESSFULLY");
        } catch (WebDriverException e) {
            new Page(driver).takeScreenshot(new Object(){}.getClass
    ().getEnclosingMethod().getName());
            Log.error("Failed with message: " + e.getMessage().
    substring(0, e.getMessage().indexOf("\n")));
            assertTrue(false);
        }

     }
```

Скрипта 6.18: Тестирање комплетног процеса куповине производа

Глава 7

Закључак

Уколико би поставили питање какав производ желе да виде различите интересне стране вероватно би одговори били следећи:

- Фирма која продаје проивод: Производ који се брзо развија, лако продаје,
- Корисник: Производ који се не квари и доживотно се може користити
- Програмер/Тестер: Производ без грешака

Чињеница је да је у пракси ово готово немогуће. Тестирање игра велику улогу у свакој фази развоја производа. Свест о квалитету и тестирању производа сваке особе укључене у развој на било који начин је неопходна. Тестирање у агилном окружењу управо дозвољава напредак кроз итерације, из разлога што у самом старту можда ни самом кориснику није скроз јасно какав производ жели. Такође, јако је тешко гарантовати да ће на дужи временски период исти људи радити на пројекту, да ће квалитет бити константан. Тестирање у агилном окружењу омогућава напредак у свакој итерацији, и дозвољава промене тако што ће у складу са смањеним људством, искуством, могућностима бити смањена и очекивања, спецификације итд.

На примеру конкретне интернет продавнице је показано колико је константно тестирање неопходно, како мануелно тако и аутоматско. Тестирање свега је немогуће, међутим у највећем броју случајева крајњем кориснику то није ни потребно, већ да у сваком тренутку зна да може да користи интернет продавницу и да ће бити у могућности да користи све најважније функционалности саме интернет продавнице. Софтвер за тестирање који је развијен у оквиру разматрања датог примера је доступан као софтвер отвореног кода.

Библиографија

[1] Манифест Агилног Развоја Софтвера, Јун 19, 2015. URL http://agilemanifesto.org/iso/sr/.

[2] Best static code analysis software. URL https://www.g2crowd.com/categories/static-code-analysis.

[3] International Software Testing Qualifications Board. Foundation level. URL https://www.istqb.org/downloads/category/2-foundation-level-documents.html.

[4] JUnit. All about junit. URL http://junit.org/junit5/.

[5] Apache Maven. Apache maven project. URL https://maven.apache.org/.

[6] Lisa Crispin Janet Gregory. *Agile Testing*. Addison-Wesley, Boston, 2008.

[7] Lisa Crispin Janet Gregory. *More Agile Testing*. Addison-Wesley, Boston, 2014.

[8] Jeff Sutherland Ken Schwaber. The definitive guide to scrum: The rules of the game, November 2017. URL http://scrumguides.org/scrum-guide.html.

[9] Ognjen Pejovic. Scrum framework, September 2016. URL https://webbootcamp.eu/prvo-predavanje-ognjen-pejovic-scrum-framework/.

[10] NOOP.NL. Simple vs. complicated vs. complex vs. chaotic, August 2008. URL http://noop.nl/2008/08/simple-vs-complicated-vs-complex-vs-chaotic.html.

[11] Axosoft. Implementing scrum guide, 2016. URL http://www.scrumhub.com/implementing-scrum-guide/.

[12] Elisabeth Hendrickson. Agile testing, nine principles and six concrete practices for testing on agile teams, August 2008. URL http://testobsessed.com/wp-content/uploads/2011/04/AgileTestingOverview.pdf.

[13] Selenium HQ Browser Automation. Documentaton for selenium. URL http://www.seleniumhq.org/.

[14] Java wiki, Март 04, 2015. URL https://en.wikipedia.org/wiki/Java_(programming_language).

[15] The Apache Software Foundation. Apache log4j 2. URL https://logging.apache.org/log4j/2.x/.