ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ ЕКОНОМІЧНИЙ УНІВЕРСИТЕТ ІМЕНІ СЕМЕНА КУЗНЕЦЯ

ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНФОРМАЦІЙНИХ СИСТЕМ

Пояснювальна записка

до дипломної роботи

МАГІСТРА

на тему: "ВПРОВАДЖЕННЯ ЗАСОБІВ BDD В ПРОЦЕС ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ"

Виконав**:**
студент 2 року навчання
групи 8.04.126.010.20.1
спеціальності 126 "Інформаційні системи та технології"
Гаркавий Станіслав Олегович

Керівник:
к.е.н. доц., Беседовський Олексій Миколаєвич

Харків – 2021 рік

РЕФЕРАТ

Пояснювальна записка до дипломного проекту: 69 с., 13 рис., 6 табл., 54 джерела.

Об'єктом дослідження являється процес контролю та забезпечення якості програмного продукту (також відомого як процес тестування програмного продукту), який включає в себе автоматизоване та ручне тестування.

Мета дослідження полягає в аналізі моделей організації процесу контролю та забезпечення якості програмного продукту, дослідженні альтернативних методологій та виявленні впливу використання засобів Behaviour Driven Development на даний процес на основі зібраної інформації.

В процесі дослідження використовувались як емпіричні (спостереження, опис) так і логічні методи пізнання (аналіз, порівняння, аналогія, моделювання).

В результаті даного дослідження можна зробити висновок, що використання підходів та засобів методології Behaviour Driven Development дозволяє зменшити затрати часу на процес ручного та автоматизованого тестування, що в свою чергу призводить до більш ефективного використання людського ресурсу і як наслідок до зменшень фінансових витрат.

Також завдяки результатам даного дослідження подальший розвиток дістала методологія Behaviour Driven Development, імплементація якої може бути поширена не лише на процес автоматизації тестування, а й на контроль та забезпечення якості загалом.

Результати досліджень можуть бути використані для організації процесу тестування на IT-підприємствах, а саме в командах, які займаються тестуванням та підтримкою великих програмних продуктів протягом більш ніж 2 років.

КОНТРОЛЬ ЯКОСТІ, ЗАБЕЗПЕЧЕННЯ ЯКОСТІ, РУЧНЕ ТЕСТУВАННЯ, АВТОМАТИЗАЦІЯ ТЕСТУВАННЯ, TEST LAST DEVELOPMENT, TEST FIRST DEVELOPMENT, TEST DRIVEN DEVELOPMENT, BEHAVIOUR DRIVEN DEVELOPMENT

ABSTRACT

Thesis report: 69 pages, 13 figures, 6 tables, 54 sources.

The objects of research are processes of quality assurance and quality control (also known as software testing process), which includes manual testing and test automation.

The purpose of research is analysis of existing models of software testing process, learning about alternative methodologies and revealing influence of Behaviour Driven Development tools on this process based on available information.

There were used empirical (such as observation and description) as well as logical methods of cognition (such as analysis, comparation, analogy and modeling).

As a result of research we can conclude that usage of Behaviour Driven Development tools and practices allow to decrease time spent for both manual and automated testing, which allows to use available human resources more efficiently and decrease financial expenses.

Also results of the research allow to get further development for Behaviour Driven Development methodology, because it was proved, that it can be implemented not only in test automation, but in whole process of quality control and assurance.

Результати досліджень можуть бути використані для організації процесу тестування на IT-підприємствах, а саме в командах, які займаються тестуванням та підтримкою великих програмних продуктів протягом більш ніж 2 років.

The obtained results can be applied for organizing process of software testing at IT-enterprises. Especially it can be used by teams, that take part in testing and supporting of large products for at least 2 years.

QUALITY ASSURANCE, QUALITY CONTROL, MANUAL TESTING, AUTOMATED TESTING, TEST LAST DEVELOPMENT, TEST FIRST DEVELOPMENT, TEST DRIVEN DEVELOPMENT, BEHAVIOUR DRIVEN DEVELOPMENT

ЗМІСТ

ВСТУП

Безсумнівним являється той факт, що XXI століття – епоха комп'ютерних технологій. Вони вже давно стали невід'ємною складовою будь-якої сфери людського життя. Той чи інший гаджет супроводжує людей і на роботі, і протягом годин дозвілля. І варто визнати, що без досягнень в області діджиталізації світова пандемія спричинила б значно більше проблем як в економічному плані, так і з точки зору людських жертв.

Протягом того часу, коли соціальні контакти були неймовірно небезпечні, важко уявити скільки людей втратили б робочі місця, якби не було можливості працювати дистанційно. В повсякденне життя багатьох людей також увійшли сервіси доставки їжі та покупок, електронна комерція, інтернет-банкінг тощо. Таким чином майже рік людство буквально жило в павутині мобільних додатків та веб-сайтів, що в свою чергу сприяє розвитку IT-індустрії загалом.

Так наприклад за даними журналу Форбс з десяти найбагатших людей світу станом на 2021 рік шестеро так чи інакше пов'язані з інформаційними технологіями. Отже можна легко зробити висновок, що галузь IT – справжня золота жила XXI століття. Але тільки шукач знайде це золото.

Розробка будь-якого програмного продукту – процес довгий і складний. Він включає в себе дуже багато різноманітних аспектів, про які рідко задумуються звичайні користувачі: від розгортання складної мережі оточень та планомірної імплементації функціоналу до моделювання поведінки користувача. І часто залежно від тих чи інших характеристик цільових аудиторій певними моментами можна знехтувати. Проте незалежно від статі, віку, соціального статусу кожен користувач хоче отримати в першу чергу якісний продукт.

Контроль якості програмного забезпечення – планомірний і систематичний план дій, що покликаний гарантувати відповідність системи бажаним характеристикам. Для забезпечення якості програмного продукту весь процес розробки пронизує тестування. Починаючи від перших вимог до продукту і закінчуючи готовим сайтом чи додатком на етапі супроводу і підтримки – кожна деталь проекту піддається ретельному тестуванню.

Тестування програмного забезпечення є невід'ємною складовою створення будь-якого продукту. Саме так гарантується відповідність реального продукту до тих вимог, які були до нього поставлені. Проте ситуація, коли програмний продукт існує в ринкових умовах більше п'яти-десяти років, зустрічається повсюди і з плином часом зустрічатиметься все частіше. І нема такого продукту,

який би протягом такого періоду обходився без приросту функціоналу, що в свою чергу спричиняє більші затрати на процес тестування.

Дану проблему покликана вирішити автоматизація тестування. Дана практика дозволяє залучити спеціалістів з метою створення спеціалізованого фреймворку автоматизованого тестування, який покликаний замінити ручного тестувальника під час виконання рутинних та однотипних завдань, дозволяючи використовувати людський ресурс більш ефективно. Такий варіант дозволяє суттєво знизити часові та людські ресурси, задіяні в процесі контролю якості.

Проте проблемою даного рішення є значні затрати ресурсів на створення та впровадження самого фреймворку. Фактично команда постає перед необхідністю розробки двох проектів замість одного, який окупить себе тільки на пізніх стадіях роботи. Проте часто буває так, що обсяги роботи, яку з кожною ітерацією необхідно виконувати команді тестувальників, є недостатньо великими для створення автоматизації, але достатньо відчутними, щоб ефективність роботи ручних тестувальників знизилась.

Але навіть у випадку великих проектів проблема раціонального розподілу обов’язків між тестувальниками автоматизації та ручними тестувальниками все одно лишається актуальною. Оскільки кількість кваліфікованих автоматизаторів на ринку нижча, то їх послуги зазвичай обходяться дорожче, що в купі з неочевидністю переваг автоматизації з точки зору бізнесу призводить до бажання обійтись мінімально можливою їх кількістю. Що в свою чергу призводить до більш повільного впровадження автоматизованого фреймворку і як наслідок меншої ефективності роботи всієї команди.

Об’єктом даного дослідження є процесс тестування програмного забезпечення, що включає в себе як і ручне тестування, так і створення автоматизованого фреймворку.

Предметом даного дослідження являється управління процесом тестування, а також методології, які дозволили б поєднати переваги ручного та автоматизованого тестування.

Завданням даного дослідження є проведення аналізу доцільності використання засобів поширених методологій для оптимізації процесу тестування програмного забезпечення, а також дослідження можливості впровадження альтернативних методологій, зокрема BDD.

Мета даного дослідження полягає в тому, щоб адаптувати методологію Behaviour-Driven Development для організації процесу тестування, що послужило б основою для уніфікації процесів автоматизованого та ручного тестування в повсякденній діяльності IT-підприємств.

1. ОПИС ОРГАНІЗАЦІЇ ПРОЦЕСУ ТЕСТУВАННЯ НА ІТ-ПІДПРИЄМСТВІ

1.1. Загальний опис учасників процесу тестування

В першу чергу необхідно зазначити, що попри той факт, що проходження практики відбувалось на базі ФОП «Столяр К. М.» більша частина інформації даного розділу не буде мати стосунку до неї. Дана фізична особа-підприємець являється інженером програмного забезпечення в сфері автоматизації тестування і надає свої послуги з проектування та розробки спеціалізованого програмного продукту для тестування.

Надання вищезазначених послуг відбувається за принципами аутсорсингу і на момент проходження переддипломної практики дана особа виконувала завдання, делеговані їй на підставі договору однією з провідних ІТ-компаній України. На жаль, відповідно до пункту в договорі про нерозголошення більш конкретна інформація про дану компанія являється конфіденційною і як наслідок не може бути детально описана в рамках даної роботи. Тому вся подальша інформація про всі сторони, які приймають участь в даному процесі окрім безпосередньо вищезазначеної фізичної особи, буде подана в деперсоніфікованій формі.

Так в процесі тестування програмного процесу, який являється об’єктом даного дослідження, можна було б виділити три основних організаційних рівні. На нижчому з них знаходяться безпосередньо виконавці, фізичні особи, які надають послуги в сферах контролю та забезпечення якості програмного продукту, а також займаються розробкою програмних засобів для тестування. Так чи інакше вони пов’язані договором з вищезазначеним ІТ-підприємством.

Саме це підприємство і являє собою другий рівень організаційної структури. Співробітники або такі ж фізичні особи виконують контролюючі функції та посідають керівні місця на рівні команд, а також займаються менеджерською діяльністю. Таким чином, компанія створює прошарок власного керівництва, яке здатне вирішувати повсякденні проблеми без залучення відповідних спеціалістів з боку замовника.

Проте попри наявність другого рівня організації, керівництво з боку замовника також присутнє в рамках даного процесу. Так на глобальному рівні замовник приймає безпосередню участь не лише в процесі створення та затвердження вимог до програмного продукту, а також і грає значну роль в контексті планування та загальній організації діяльності. На повсякденному рівні

до кожної проектної команди, яка працює над створення відповідної частини програмного продукту, (далі такі команди будуть називатись стрімами) представлений щонайменше один співробітник з боку замовника, який займається валідацією всіх змін в продукті.

Залежно від стріма, обсягів функціоналу, який має бути зробленим, а також кількості спеціалістів в командах розробки, тестування тощо залучення ресурсів з боку замовника може різнитись. В деяких випадках окрім контролюючого менеджера може паралельно функціонувати окрема команда, створена повністю з працівників замовника.

1.2 Загальна характеристика підприємства

Компанія, яка використовує послуги вищеназваної фізичної особи, являється одним із найбільших виробників програмного забезпечення в Східній Європі. Вона займається розробкою проектного (замовного) програмного забезпечення, а також лишається одним із провідних гравців в сфері консалтингу в Центральній та Східній Європі.

Виконуючи проекти для найбільших корпорацій та співпрацюючи з провідними світовими розробниками програмного забезпечення дане підприємство набуло досвід в таких галузях як:

- Розробка за замовленням провідних виробників програмного забезпечення для систем корпоративного користування, керування життєвим циклом, корпоративних інформаційних порталів, систем керуванням відношеннями з клієнтами, серверів інтеграції додатків, систем керування контентом та систем керування знаннями
- Під час ручного виконання, людина може зробити помилку
- Розробка додатків, що відповідають вимогам новітніх сервіс-орієнтованих архітектур
- Створення та розгортання систем керування закупками та збутом
- Створення порталів великих підприємств і холдингів з розвинутими засобами аналізу даних та керування знаннями
- Інтеграція додатків в розподілених системах, проектування, консолідація та налаштування корпоративних довідників та каталогів
- Інтеграція ERP, PLM, CRM, SCM рішень і систем аналітики, стратегічного планування та бюджетування в ряді галузей.

- Аналіз інфраструктури та інформаційних ресурсів, проектування та реінжиніринг бізнес-процесів, управління проектами модернізації та розвитку інформаційних систем.

Дана компанія має обширний список престижних клієнтів з усього світу, серед яких є значна кількість членів списку Fortune 500, а також провідні компанії-розробники програмного забезпечення.

Підприємство успішно впровадило свої рішення в найрізноманітніших галузях, серед яких можна відмітити:

- страхування та фінанси;
- програмне забезпечення;
- телекомунікації та високі технології;
- роздрібна торгівля та споживчі товари;
- туризм та індустрія розваг;
- ЗМІ, наукові дослідження та дистанційна освіта;
- виробництво, транспорт та енергетика.

Також це підприємство неодноразово успішно проходило міжнародну сертифікацію відповідності четвертому рівню CMMI (SEI CMMI v.1.1 Maturity Level 4).

Відповідність даному стандарту означає, що підприємство на міжнародному рівні визнається надійним та ефективним розробником програмного забезпечення, а також постачальником послуг в сфері IT. В процесі проходження іспиту на відповідність вимогам CMMI організація покращує свої бізнес-процеси і розвиває якість розробки програмного продукту, що в свою чергу дозволяє забезпечити стабільно високу якість послуг, що надаються, і являється однією із основних засад підвищення конкурентоспроможності і подальшого розвитку компанії.

На даний момент в компанії працює більш ніж 40 000 співробітників в понад тридцяти країнах світу. Офіси компанії розташовані в таких країнах як Білорусь, Росія, Україна, Казахстан, Сполучені Штати Америки, Угорщина, Великобританія, Німеччина, Швеція та Швейцарія.

Також варто зазначити, що кількість працівників з кожним роком зростає, що спричинено розвитком та розширенням компанії. Регулярно мають місце курси по цілеспрямованому підбору персоналу як для співробітників компанії, так і для зовнішніх кандидатів.

Велика увага керівництво компанії приділяє питанням підвищення кваліфікації співробітників, для чого регулярно проводяться тематичні та проблемні курси та семінари як на базі компанії так і в спеціалізованих навчальних центрах.

Компанія прагне забезпечити оптимальні умови праці для своїх працівників. З цією метою створена прогресивна система мотивації та стимулів, гнучкий розклад робочого часу та відпусток, робочі місця, оснащені сучасним обладнанням.

Також заохочується спортивний розвиток, наявна добре розвинена корпоративна культура. Серед її особливостей можна віднести демократичність та відкритість комунікацій, вільний обмін знаннями, стимулювання ініціативи та відповідальності працівників.

1.3 Загальна характеристика замовника

Замовником в рамках даного проекту виступає одна із найбільших компанії в сфері роботи з авторськими правами для корпоративних та академічних користувачів матеріалів, що були захищені авторським правом.

Дана компанія являється брокером в контексті використання авторського права. Вона укладає договори з володарями авторських прав на ті чи інші публікації і після цього виступає в ролі агента, пропонуючи можливість придбання матеріалів іншим користувачам.

Задля забезпечення даного процесу замовник володіє рядом програмних продуктів, розміщених в мережі інтернет, які створюють цілісний інтерфейс взаємодії як для авторів, так і для шукачів матеріалу. Ці продукти надають можливості з пошуку, покупки, а також маніпуляцій з рядом об’єктів авторських прав з більш ніж сотні країн світу.

Головне управління замовника знаходиться в Сполучених Штатах Америки. Там же провадиться і переважна більшість послуг під егідою замовника. Зокрема замовник ліцензує такі об’єкти авторського права, як книги (в тому числі і такі, що вже не видаються), журнали, газети, різноманітні статті та наукові праці, кінофільми, телевізійні шоу, зображення, електронні книги, музичні твори тощо. Після цього автори того чи іншого контенту будуть отримувати компенсацію за кожне використання свого твору.

Проте попри свою направленість на Сполучені Штати, замовник через ряд своїх дочірніх підприємств має можливість надавати всі вищезазначені послуги для користувачів інших країн та континентів.

На даний момент серед основних сфер діяльності замовника окрім вищезазначених функцій посередника в питаннях ліцензування контенту, можна відмітити здобутки в наступних сферах:

- автоматизація надання дозволів на використання матеріалів;

- розробка централізованого спеціалізованого пошукового домену;
- дослідження в сфері текст-майнінгу.

1.4. Опис організаційної структури проекту

В першу чергу варто зазначити, що серед сучасних IT-підприємств є три найпоширеніших підходи до організаційної структури компанії: лінійна, проектна та матрична [38].

В першому випадку особливістю організаційної структури є наявність невеликих крос-функціональних команд. Основною перевагою лінійної структури є стійкість в кризових ситуаціях та можливості швидкого реагування на виникнення проблемних ситуацій, завдяки залученню додаткових людських ресурсів від інших команд.

Проте зворотнім ефектом є той факт, що часом вирішення проблем підрозділу знаходиться за межами повноважень безпосереднього керівника даного підрозділу. Так наприклад проблема нестачі ресурсів в окремій команді протягом певного періоду часу вирішується на рівні проекту, а не на рівні команди.

Проектна структура дуже подібна до лінійної. Вона також складається з невеликих крос-функціональних команд, в яких повноту влади має проектний менеджер або lead. Основна різниця полягає в тому, що при даному варіанті структури підрозділи та співробітники набираються виключно під конкретний проект і після його завершення співпраця припиняється, а підрозділ розформовується.

Такий підхід дає переваги з точки зору економіки та керування людськими ресурсами, адже гарантовано підприємство не має працівників, які не мають корисного робочого навантаження. З іншого ж боку такий підхід не сприяє покращенню корпоративного іміджу, а також формуванню досвідчених спеціалістів

Матрична структура вносить поділ в функції влади. Проектні менеджери виділяються в окремий відділ, натомість обов’язки організації функціонування проекту лягають на плечі проектного керівництва, а також керівників команд-підрозділів.

Дана структура є найважчою до впровадження, адже вона найбільш піддається впливу проблем комунікації між відділами. Проте у випадку вдалої імплементації матрична структура сприяє накопичуванню знань в рамках кожного окремого функціонального відділу [4].

Компанія притримується лінійної структури організації, схематичне зображення якої наведено на рис. 1. 1. Схематичне зображення команди тестувальників відповідно до цього підходу наведено на рис. 1.2.

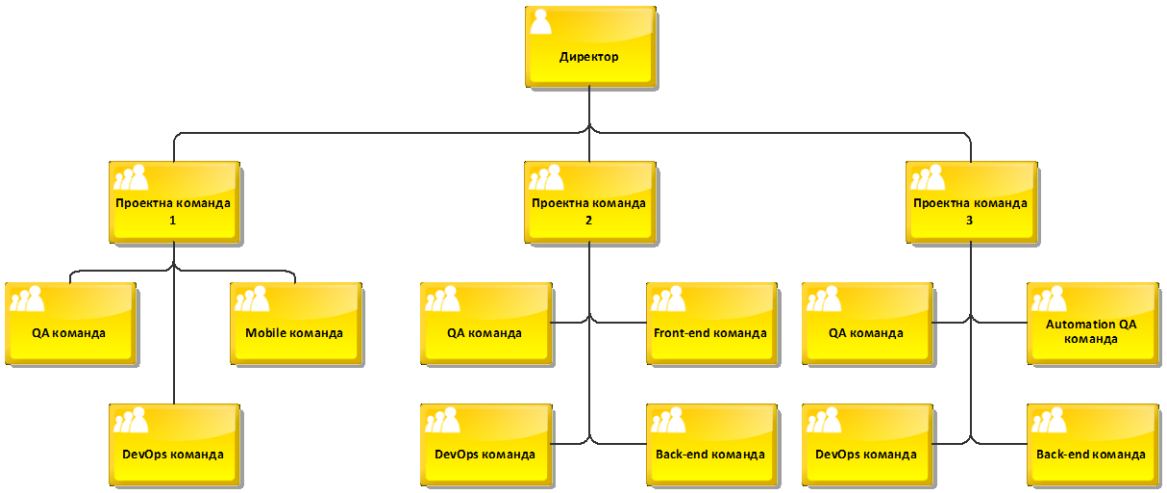

Рис. 1.1. Схема лінійної структури

Рис 1.2. Схема стандартного складу Automation QA команди

Особливістю роботи конкретного цього проекту є багатошаровість та незалежність команд. Загальні риси організації даного проекту були вже наведені в пункті 1.1. Тепер варто розглянути організації в рамках кожного із трьох рівнів керування.

Найнижчою функціональною одиницею являється окрема функціональна команда. Як правило вона має в своєму складі від п’яти до десяти осіб, які об’єднані за своєю роллю в проектній команді. Так на чолі кожної команди обов’язково є лід, який є керівником і відповідальним за роботу команди. Саме він презентує результати діяльності, а також бере участь в плануванні та комунікаціях з представниками інших організаційних рівнів.

Сама ж команда залежно від обсягів роботи і побажань замовника може бути дуже неоднорідною, тому якісь спільні риси різних функціональних команд доволі непросто. Проте неписаним правилом являється наявність так званого «заступника» ліда. Не будучи повноважним на керування процесом роботи групи, «заступник» зазвичай являється одним із найбільш давніх і досвідчених учасників команди, який часто виконує функції наставника для новачків, а також займається вирішенням повсякденних технічних питань.

Якщо говорити з точки зору процесу тестування, то команда тестувальників також може відрізнятись. Залежно від імплементації фреймворку автоматизованого тестування, а також затрат на його утримання автоматизаторів може не бути взагалі, вони можуть бути членами єдиної команди тестувальників, або ж навіть виокремлюватись в другу команду. Проте в загальному випадку схема зображена на рис. 1.2 лишається справедливою.

На один крок вище від функціональної команди стоїть проектна команда, або стрім. По суті стрім являє собою сукупність функціональних команд, які так чи інакше взаємодіють з однією і тою ж частиною функціоналу. Діяльність кожного стріма значною мірою автономна та має небагато точок перетину з іншими стрімами.

Подібний підхід справедливий і для різних команд в рамках одного стріма. Значною мірою команди розробників, тестувальників та автоматизаторів працюють незалежно одне від одного. Окрім екстрених випадків комунікація відбувається через лідів команд.

Керівництво стрімом провадиться відповідним представником менеджменту з боку компанії, а також представниками замовника, кількість яких залежить від самого стріма. В даному випадку корпоративний менеджер більше відповідає за контроль та організацію процесу, в той час як представник замовника займається верифікацією та валідацією програмного продукту протягом його створення.

Кількість учасників зі сторони замовника може помітно варіюватись. Так інколи одна особа може контролювати всю діяльність усіх команд в рамках стріму, а часом для кожної команди є окрема відповідальна людина. Даний підхід розповсюджується і вгору по організаційній ієрархії аж до головуючих посад.

Схематично організаційна структура стріму зображена на рис. 1.3

Сукупність стрімів представляє собою весь проект загалом. Якась спільна діяльність в рамках проекту зазвичай є явищем рідким і пов’язаним здебільшого або зі змінами технічного плану (корегування конфігурації хмарних сервісів тощо), або з форс-мажорними подіями. Саме тому з точки зору рядового розробника чи тестувальника на цьому рівні не відбувається нічого цікавого.

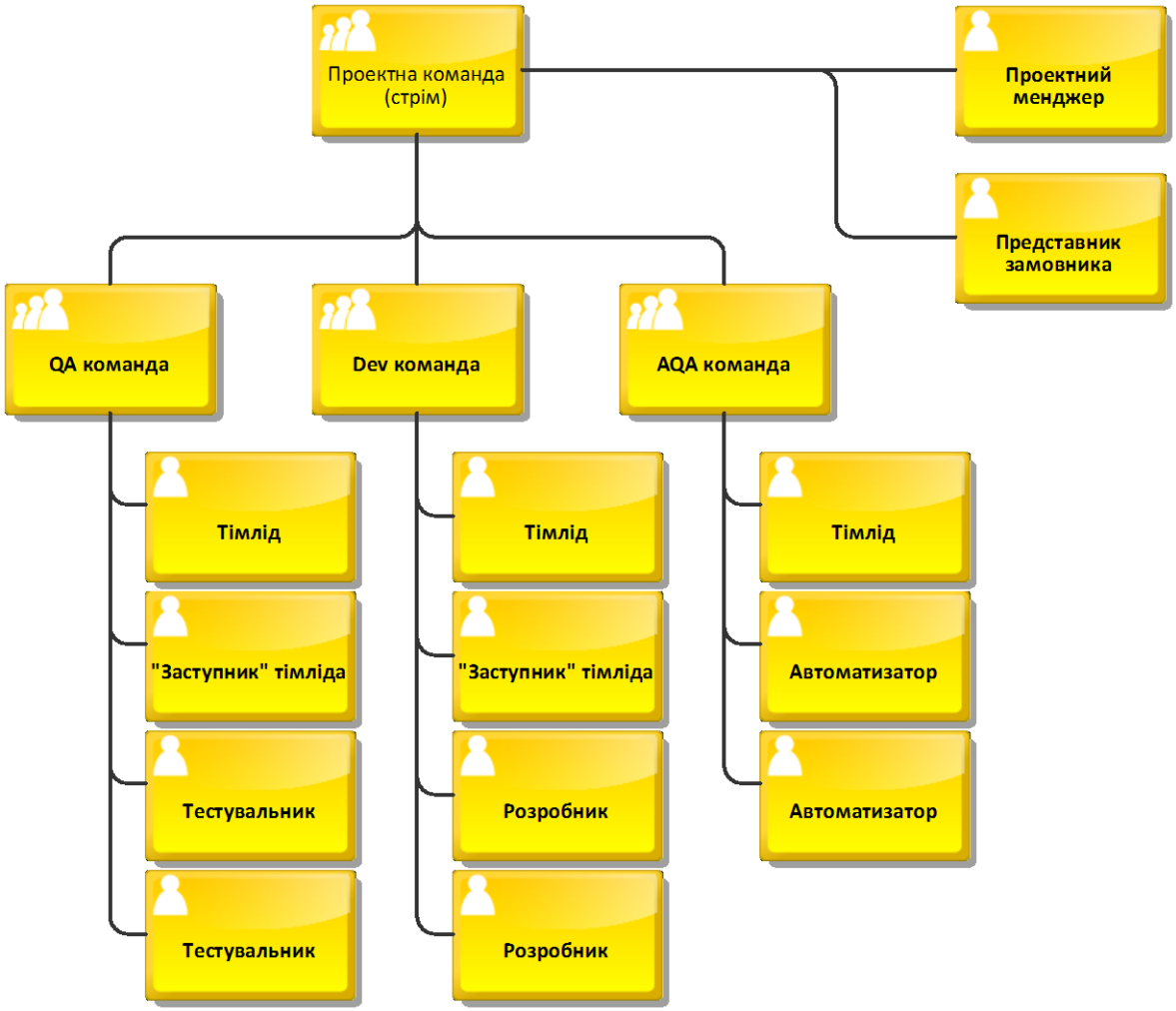

Рис 1.3. Схема організації стріму

Проте саме на етапі проекту має місце діяльність менеджерського складу, який знаходиться на другому та третьому рівнях організаційної драбини. Так часто вже керівництво стрімів приймає участь в розробці планів та вирішенні тих

чи інших проблем бізнесу. Проте зазвичай останнє слово лишається за делівері-менеджерами (далі - ДМ), а також керівництвом з боку замовника.

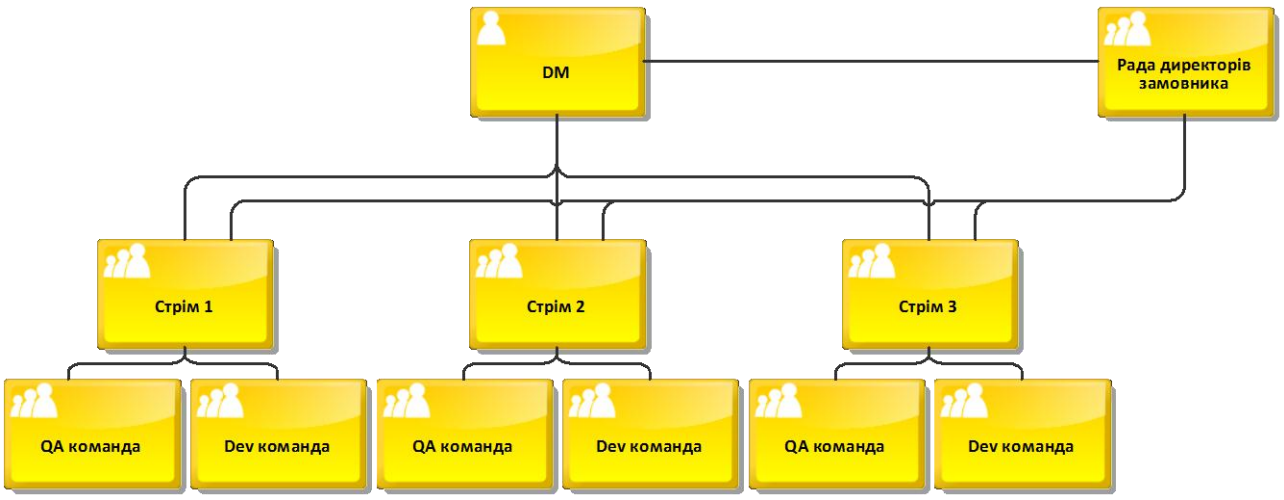

Рис. 1.4. Схема організації керування проектом

Попри те, що на перший погляд ДМ і керівництво замовника в рамках проекту вирішують дуже схожі питання, проте працюють вони з ними з різних точок зору.

Так ДМ являється нижчим організаційним рівнем, адже саме вони займаються безпосереднім контролем та координацією задля найбільш ефективного виконання поставленої задачі. Також ДМ більшою мірою займається вирішенням кадрових питань, а також форс-мажорних випадків. Таким чином можна сказати, що ДМ – це своєрідний тактик, який веде програмний продукт до вдалого релізу.

В свою чергу керівництво замовника являється по суті стратегом в рамках проекту. Будучи першоджерелом вимог зі сторони бізнесу, представники замовника формують загальне бачення того, як програмний продукт має виглядати на момент закінчення періоду приросту функціоналу.

Завдяки інформації отриманої від ДМ, а також наглядачів серед стрімів, представники замовника мають можливість швидко орієнтуватись в роботі процесів на проекті і реагувати на ризики, що виникають, вносячи глобальні корективи навіть на рівні бізнес-процесів, якщо це знадобиться.

Саме тому представники замовника аж до керівників відділів знаходяться саме на третьому рівні організаційної драбини даного проекту. Схематично організація управління проектом зображена на рис. 1.4

2. ТЕОРЕТИКО-МЕТОДИЧНЕ ЗАБЕЗПЕЧЕННЯ ВИКОРИСТАННЯ ЗАСОБІВ BDD ДЛЯ ОПТИМІЗАЦІЇ ПРОЦЕСУ ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Опис процесу тестування програмного забезпечення

Відповідно до стандарту IEEE SWEBOOK v3.0 «тестування програмного забезпечення» (Softwaretesting) – це перевірка відповідності між реальною та очікуваною поведінкою програмного продукту, здійснюваного на кінцевому наборі тестів, обраним певним чином [3].

Проте попри те, що процес тестування може видатись простим виконанням певної кількості перевірок, насправді він являється значно більш комплексним і провадження тестів є лише одною із складових частин даного процесу. Як було зазначено в матеріалі з сайту QAtestlab.com [13], в ході роботи тестувальник повинен максимально підвищити вірогідність того, що продукт, який піддається тестуванню, буде працювати коректно за будь-яких умов і відповідатиме наявним вимогам. А оскільки згідно одної з аксіом тестування тестування ніколи не може бути вичерпним, то для підвищення цієї вірогідності необхідні попередні приготування.

Протягом останніх двадцяти років було створено безліч варіантів опису процесу тестування. І серед них нема єдиного правильного. В залежності від автора та часового проміжку, коли автор вів активну діяльність, їх сутність може докорінно різнитись. Проте більшість сучасних варіантів сходяться в думці, що процес тестування програмного забезпечення нараховує від п’яти (дані QAtestlab.com) до восьми етапів (дані посібника «Тестирование програмного обеспечения» [1]). Проте найбільш загальновживаним являється варіант представлений ресурсом SoftwareTestingFundamentals [6] . Відповідно до цього ресурсу весь процес тестування програмного забезпечення поділяється на шість основних етапів (рис. 2.1).

Першим етапом є аналіз вимог. На цьому етапі менеджер проекту чи бізнес-аналітик в процесі спілкування з клієнтом визначає умови та критерії роботи системи. Саме в цей проміжок часу потреби бізнесу конвертуються в чіткі вимоги до розроблюваного продукту. По ходу їх формулювання тестувальник перевіряє цілісність, однозначність та логічність створених вимог. У разі неоднозначності має місце їх уточнення. В іншому випадку на основі отриманих даних формується основна керівна документація.

Після остаточного затвердження документації, починається етап планування тестування. На цьому етапі, маючи загальне уявлення про поточний стан продукту, а також про особливості розробки та прирощення функціоналу, створюється загальний план тестування, оцінюються метрики та ризики, визначаються необхідні ресурси і відповідальні особи, описуються формальні вимоги до процесу тестування програмного продукту, а також складається розклад тестування. Саме на основі цього плану відбуватиметься весь подальший процес.

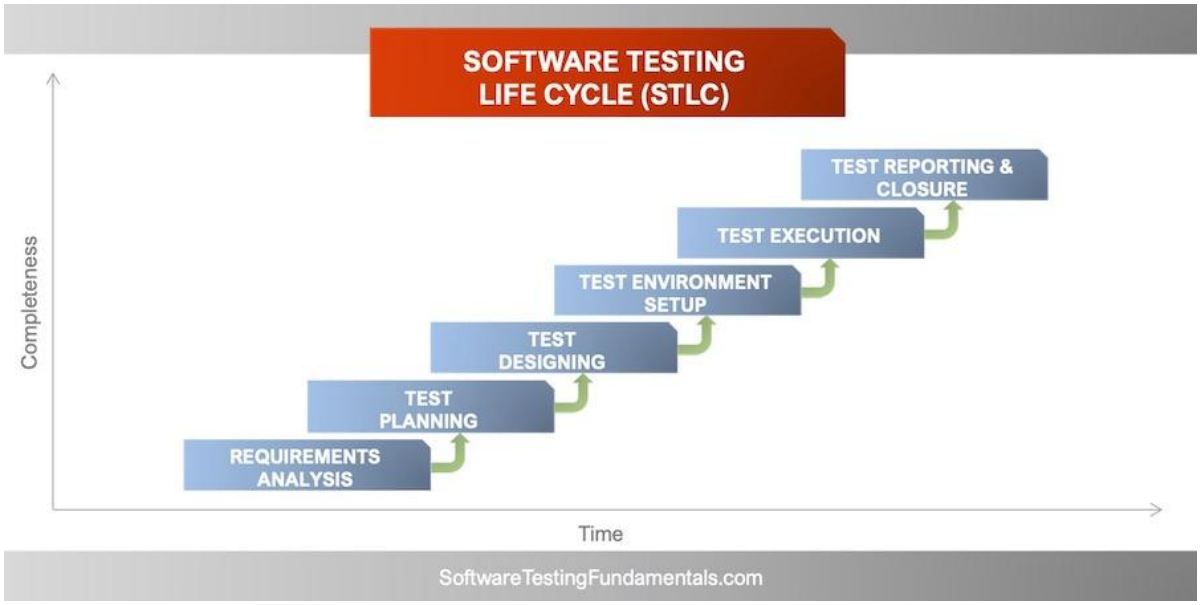

Рис. 2.1 – Життєвий цикл тестування програмного забезпечення

Далі відповідно до створеного плану має місце етап дизайну тестів, протягом якого тестувальник розроблює, переглядає, уточнює, допрацьовує та виконує будь-яку іншу діяльність, націлену на створення тестових сценаріїв та формування на їх основі тестових наборів, які будуть використовуватись протягом безпосередньо процесу виконання тестів. Це може бути як і покриття тестами нового функціоналу, так і підтримка та актуалізація вже наявних сценаріїв.

Проте перед виконанням тестів також необхідним є етап установки тестового оточення (сервер, мережа, клієнт, операційна система, браузер тощо). Зазвичай це декілька різних наборів налаштувань, які дозволяють ефективно змоделювати взаємодію типового користувача з системою. Зазвичай тестові оточення покривають основні комбінації налаштувань, які використовуються принаймні п’ятьма відсотками користувачів.

Коли вимоги до тестового оточення виконано, відбувається етап виконання тестів. На даному етапі відбувається неодноразовий виконання в різних умовах тестових сценаріїв та наборів з метою виявлення та фіксації максимальної кількості дефектів. Дані стосовно дефектів валідуються представниками бізнесу і в разі успішного проходження і підтвердження необхідності виправлення передаються розробникам для подальшого виправлення.

Заключним же етапом процесу тестування є аналіз отриманих даних та формування звітної документації. На цьому етапі відбувається порівняння тестувальником очікуваних на етапі планування результатів з отриманими даними. Після цього підсумки роботи оформлюються у вигляді результатів тестування та тестових метрик і передаються менеджеру проекту чи бізнес-аналітику для того, щоб стати основною для першого етапу наступної ітерації тестування. Таким чином цикл замикається.

Таким чином, стає помітно, що тестування охоплює практично всі аспекти діяльності програмного продукту. І тому кількість тестів і тестових наборів росте з великою швидкістю по мірі додавання нової функціональності до програмного продукту. Тому сучасні проекти, які функціонують протягом більш ніж двох років, можуть мати більш ніж вісімсот тестових сценаріїв. Відповідно ростуть і затрати людських ресурсів, які витрачаються на розробку, провадження та обробку результатів тестів. Так для підтвердження працездатності чергового релізу програмного продукту може знадобитись понад сто годин роботи середньостатистичної команди тестувальників з чотирьох осіб. З урахуванням того, що згідно Agile-методологій, які зараз є загальновживаними в сфері IT, випуск нових версій програмного продукту може відбуватись навіть щотижня.

Таким чином стає очевидно, що проблема оптимізації процесу тестування в умовах проектів, що стрімко розростаються, є однією з найбільш актуальних для сучасних IT-підприємств.

Процес тестування є процесом довгим та об’ємним. Навіть на рівні теоретичного бачення даного процесу нема однозначного бачення алгоритму роботи. Тому в реальних умовах кожна команда, керуючись загальними принципами, намагається впровадити своє унікальне бачення цього процесу.

Зазвичай команди рухаються в бік двох основних варіантів: оптимізації процесу ручного тестування з метою впровадження найбільш ефективної схеми діяльності команди тестувальників або ж залучення спеціалістів-автоматизаторів з метою подальшого впровадження відповідного фреймворку автоматизованого тестування, який взяв би на себе виконання значної частини рутинних тестових сценаріїв.

В першому випадку більшість команд тестувальників схиляються до впровадження Agile-практик в процес тестування, що робить процес, описаний в підрозділі 1.1, ще більш ітеративним.

Дані практики ставлять на меті впровадження ітеративності не лише до процесу в цілому, а й до кожного окремого етапу. Таким чином замість одного довгого етапу з’являється набір коротких спринтів, протягом яких відбувається попередньо запланований процес нарощення функціоналу. Часом в одному спринті навіть можуть поєднуватись риси декількох етапів. Діаграма IDEF0 для типового варіанту імплементації процесу тестування наведена на рис. 2.2. Декомпозиція діаграми IDEF0 наведена на рис. 2.3

Як можна спостерігати на рис. 2.2, на вхід даного процесу подається актуальний дистрибутив програмного продукту (зазвичай це відбувається за допомогою системи CI/CD, яка надає доступ до актуальної версії в будь-який момент часу), документація для продукту, а також вимоги стосовно того функціоналу, який має бути протестований.

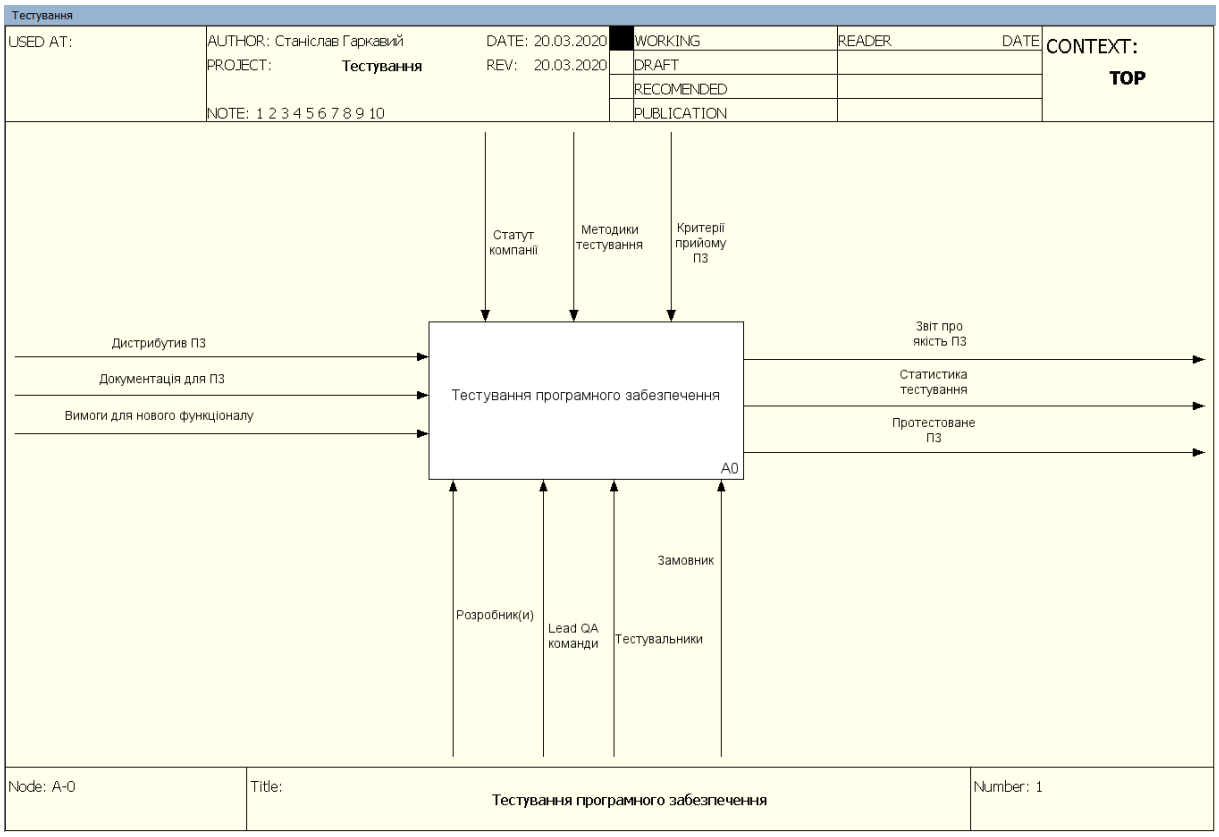

Рис. 2.2. Діаграма IDEF0 процесу «Тестування ПЗ»

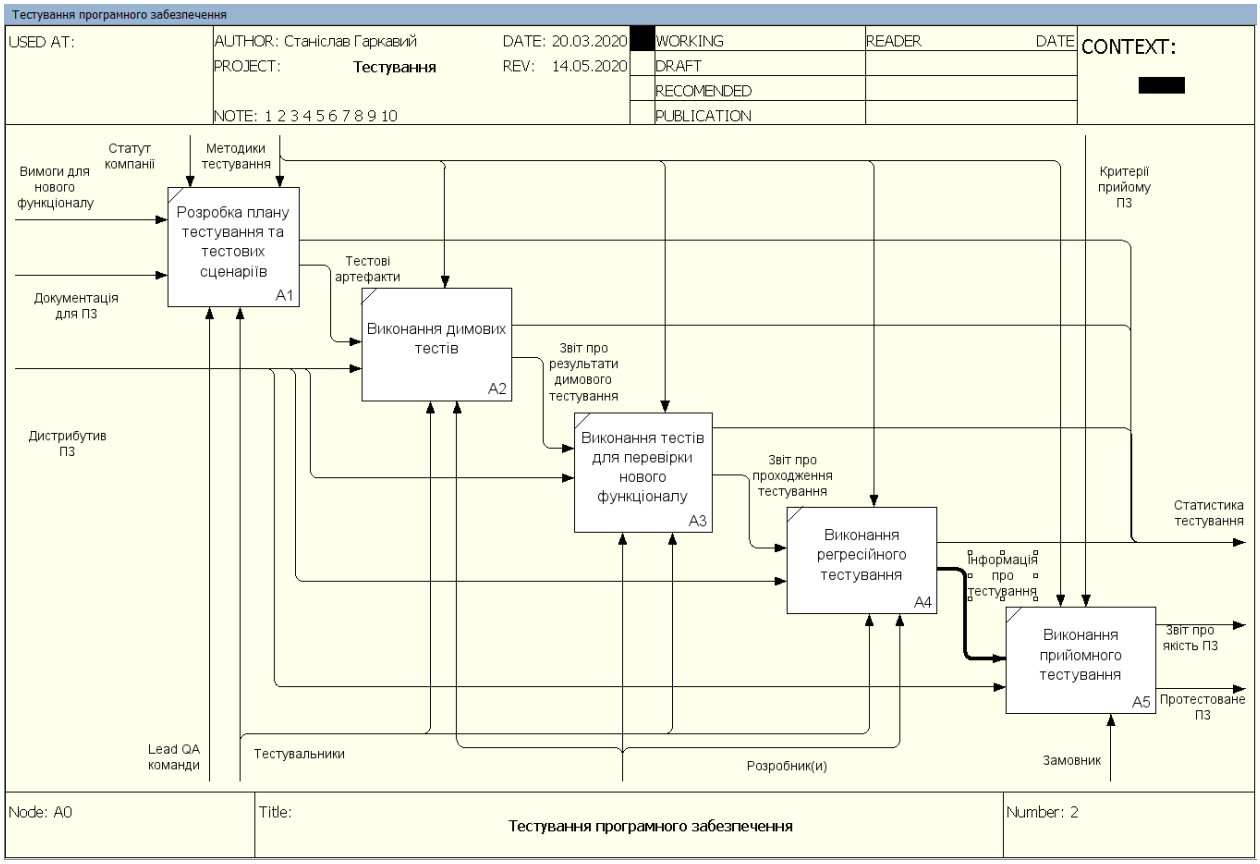

Рис. 2.3. Декомпозиція діаграми IDEF0

Серед функцій керування для цього процесу варто відмітити Статут IT-компанії, методики тестування програмного забезпечення та критерії прийому програмного забезпечення. Механізмами ж даного процесу можна вважати тестувальників, lead команди тестувальників, замовника та розробників.

На виході даного процесу є інформація про якість програмного продукту, статистика тестування, а також протестована версія актуального програмного продукту.

На декомпозиції (рис. 2.3) можна побачити, що даний процес має в собі 5 етапів: створення плану тестування та розробка тестів, димове тестування, тестування нового функціоналу, регресійне тестування та прийомне тестування.

На етапі планування команда тестувальників отримує документацію та вимоги до програмного продукту. На основі цього відбувається планування та актуалізація тестових сценаріїв, які далі будуть проведені та потенційно можуть бути впроваджені в фреймворку автоматизованого тестування.

На етапі димового тестування відбувається перевірка тестів, що відповідають за основну функціональність програмного продукту. Для цього повторно виконуються вже існуючі димові тести.

На етапі тестування нового функціоналу буде проведено тестування відповідно до тестових сценаріїв, актуалізованих (або створених з нуля) протягом першого етапу. Таким чином будуть отримані дані про ефективність роботи конкретного функціонального блоку.

На етапі регресійного тестування командою тестувальників буде повторено димові тести, виконані на другому етапі. Метою цього є перевірка відсутності дефектів основного функціоналу програми, пов'язаних з впровадженням нового функціоналу. Після виконання цього етапу формується статистика тестування, в якій зберігаються необхідні метрики тестування.

На етапі прийомного тестування відбувається повне тестування програмного продукту. Зазвичай цей етап проходить за участі замовника. Окрім цього засвідчується відповідність розробленого програмного продукту відносно критеріїв прийому програмного продукту.

Попередній етап є в основному формальним засвідченням виконаної роботи перед стороною замовника. Однак навіть формалізованість даного етапу не зменшує його важливості для самого процесу. Саме завдяки їй після завершення всіх інших видів діяльності в обов'язковому порядку формується звіт про тестування, в якому зазначаються всі можливі метрики, отримані в ході тестування. Серед них основними можна назвати такі метрики, як кількість тестових сценаріїв, відсоток проходження тестів, кількість знайдених дефектів програмного продукту, тощо.

2.2 Формулювання проблем процесу тестування

Як помітно з опису, процес тестування програмного забезпечення є затратним процесом, адже являється основний інструментом забезпечення якості програмного продукту. Окрім того необхідно відмітити, що тестування відбувається не лише в сфері безпосередньо функцій, які має виконувати програмний продукт. Існує велика кількість підвидів та рівнів тестування, які мають на меті забезпечити якнайбільшу вичерпність тестування і високий показник покриття тестами. Відповідно до довідника стандартизації та сертифікації програмного забезпечення всі види тестування, відповідно до мети проведення, можна умовно розділити на функціональне, нефункціональне та тестування пов'язане зі змінами [12].

Функціональні тести базуються на функціях та особливостях системи, а також взаємодії з іншими системи. Серед них виділяють такі підвиди як власне

функціональне тестування, тестування інтерфейсу користувача, тестування безпеки, і тестування взаємодії.

Нефункціональне тестування описує тести, необхідні для визначення характеристик програмного забезпечення, які можуть бути виміряні певними величинами. Найкращим прикладом нефункціонального тестування будуть усі види тестування продуктивності, а також тестування інсталяції, зручності користування, тестування відмови та відновлення, конфігураційне тестування тощо.

Таким чином можна зробити висновок, що функціональні тести можуть бути представлені на будь-якому рівні тестування: модульному, інтеграційному, системному, операційному та прийомному. Натомість нефункціональне тестування відбувається лише на останніх трьох рівнях.

Модульне тестування перевіряє функціональність та шукає дефекти в окремих частинах програмного продукту, які доступні та можуть бути протестовані окрема від інших частин програми.

Інтеграційне тестування проводиться після успішного затвердження якості роботи окремих модулів. На цьому рівні перевіряється взаємодія між компонентами системи.

Системне тестування має на меті перевірку як функціональних так і нефункціональних вимог в системі загалом. При цьому виявляються дефекти такі як неоптимальне використання ресурсу системи, непередбачені комбінації даних, які можуть бути подані на вхід на користувацькому рівні, відсутня сумісність з оточенням, непередбачені сценарії використання, відсутня або невірна функціональність, невдалий дизайн, який спричиняє незручності використання програмного продукту тощо [18].

Операційне тестування дозволяє переконатись, що програмний продукт відповідає вимогам, потребам користувача і виконує відведену йому роль в сфері своєї експлуатації згідно до того, як це описано в бізнес-моделі. Оскільки не виключені проблеми в побудові і самої бізнес моделі, то перевірка продукту в реальних умовах стає необхідною. Окрім того на даному етапі можна виявити такі нефункціональні проблеми як конфлікти з суміжними системами в даній області бізнеса чи програмному оточенні; недостатня продуктивність в експлуатаційному оточенні тощо.

Прийомне тестування являється більше формальним етапом, під час якого остаточно перевіряється відповідність критеріям прийому та виноситься рішення замовником або повноважним лицем про приймання програмного продукту.

Димове тестування являє собою короткий набір основних тестів, які засвідчують працездатність засобів установки, запуску, а також роботи обмеженого основного функціоналу після випуску оновленої збірки коду (нового чи виправленого).

Регресійне тестування має на меті підтвердження того, що після внесення змін в код програмного продукту або оточення функціональність, яка була вже була імплементована на момент внесення змін, працює як і раніше.

Повторне тестування проводиться для виконання тестів чи тестових наборів, в яких було виявлено помилки під час попереднього запуску і підтвердження того, що дані помилки були успішно усунуті.

Згідно даних Аміра Гхахраї можна зробити висновок, що переважна більшість тестувальників проводить регулярно димове, регресійне та/чи повторне тестування паралельно з роботою над більш спеціалізованими функціональними чи нефункціональними наборами [5]. На основі чого можна зробити висновок, що так чи інакше присутнє повторне виконання робіт через великі обсяги матеріалу, який необхідно опрацювати. Для полегшення цього процесу зазвичай використовується автоматизація тестування.

Автоматизація тестування («Software Automation Testing») – це процес верифікації програмного забезпечення, під час якого основні функції і кроки тестування виконуються автоматично за допомогою інструментів автоматизованого тестування.

За словами Андрія Реброва [14], автоматизація тестування є логічним наслідком процесу оптимізації процесів всередині команди, які мають місце на етапі так званого «бережного» виробництва. Особливістю цього етапу є успішна взаємодія в команді на базовому рівні, яка приносить успіхи в роботі. І з тим як базовий прогрес закріплюється, стартує пошук шляхів для модифікації процесів та зменшення зайвих затрат людських, часових та матеріальних ресурсів. Для команди тестувальників даним шляхом являється автоматизація.

Так чому саме команди беруться за автоматизацію тестів:

- Ручне виконання тестів займає надто багато часу
- Під час ручного виконання, людина може зробити помилку
- Автоматизація звільняє час команди для виконання важчих задач
- Автоматизовані регресійні тести першими знаходять дефекти програми
- Автотести дозволяють швидко отримати інформацію про продукт
- Автотести дозволяють підвищити рівень якості написаного коду

Як би чудово не звучало все вищеперераховане, проте проблеми з впровадженням автоматизованого тестування були відомі вже понад два десятки років тому. Брет Петікорд, експерт в області тестування та розробник продукту для автоматизації тестування Watir, в своїй праці «Сім кроків до успішної автоматизації» [4] ще в далекому 1999 році виділяв наступні проблеми автоматизованого тестування:

- Неможливість автоматизації тестів у вільний час
- Відсутність чітких цілей тестування
- Малий досвід роботи в цій сфері
- Якщо людина починає займатись автотестами, то втрачає досвід основної професії (на той час окремої професії автоматизатора тестування не існувало)
- Необхідність в високій мотивації, адже автотести важко закінчить і доводиться підтримувати і переписувати
- Багато людей займаються автоматизацією заради автоматизації, а не заради покращення якості продукту
- Занурення в технічні проблеми призводить до втрати мети тестування

Загалом даний список проблем має місце і в сучасному світі, проте більш ніж два десятки років змінили індустрію та підхід до проектних процесів, тому не можна не відмітити той факт, що він все ж застарів. Тому було б доречно звернутись до більш сучасних джерел.

Непогано список проблем автоматизованого тестування було сформульовано Лізою Кріспін в книзі «Agile Testing: A Practical Guide for Testers and Agile Teams» [2]. Виглядає він наступним чином:

- Розробники не розуміють навіщо потрібні автотести і тим паче не бажають працювати над ними
- Довгий період впровадження, який не всі можуть подолати
- Потреба в значному аналізі засобів стратегії та багатьох інших ресурсів в умовах вже існуючого робочого процесу, коли є чіткий графік
- Необхідність в підтримці та корегуванні тестів
- Легасі код, який стає недоторканим і ніхто не береться за покриття його тестами
- Страх того, що автоматизація не окупить себе
- Старі звички, які витікають з попереднього пункту, створюючи аргументи в стилі «Навіщо писати автотести, якщо зроду і без них обходилось?»

Як боротись з цими проблемами і чи дійсно вигода від автоматизації переважує всі її недоліки? Питання важливе, проте досі в доступних джерелах

нема однозначної відповіді на нього. В рамках agile-розробки ці проблеми падають на плечі всієї команди, а отже команда в цілому і повинна їх вирішувати. За два десятиліття, які компанії працюють над ними, вибудувалось декілька основних підходів до інтеграції автоматизованих тестів, які і набули основного поширення.

Таким чином на основі фактів, наведених в підрозділі 2.1 можна зробити висновок, що процес тестування програмного забезпечення в умовах сучасного конкурентного ринку являється об’ємним та таким, що вимагає залучення значних ресурсів.

Основним завданням даного дослідження являється пошук найбільш оптимального підходу до організації даного процесу, який дозволив би знизити затрати людських, матеріальних та нематеріальних ресурсів та був би універсальним для різних команд незалежно від специфіки сфери розробки, розміру команди чи обсягу проекту.

Таким чином необхідно, щоб шуканий підхід дозволив максимально ефективно інкапсулювати в процесі тестування як розробку тестової документації, так і провадження ручних перевірок, а також за необхідності дозволяв би імплементувати етап автоматизації створених тестів. Така необхідність з’являється внаслідок того, що розробка проектної документації (до якої входить і тестова) дуже слабко пов’язана з розробкою та провадженням безпосередньо тестових сценаріїв.

Більше того у випадках, коли проектні команди приймають рішення приступити до автоматизації тестових сценаріїв, то в більшості своїй цим займається ще один окремий підрозділ, який теж працює у відносному робочому вакуумі. І таким чином комунікація між командами, які по суті виконують одну і ту ж роботу, лишає бажати кращого.

Саме тому особливістю бажаного підходу до оптимізації процесу тестування програмного забезпечення є той факт, що він дозволяє максимально інтегрувати ці три види діяльності між собою.

Для ефективного дослідження необхідно виконати декілька основних завдань.

По-перше, необхідно проаналізувати поточний стан справ. Для цього варто зібрати інформацію щодо тих підходів та методологій, які використовуються в реальних проектах, оцінити їх сильні та слабкі сторони, а також визначити те, наскільки універсальними вони являються. На основі отриманих даних можна сформулювати конкретний список проблем, які мають бути вирішені.

По-друге, варто провести дослідження, яке дозволить оцінити наскільки засоби методології BDD здатні оптимізувати слабкі місяці, які були знайдені в існуючих робочих моделях. За необхідності скорегувати засоби даної методології з метою підвищення ефективності її впровадження.

По-третє, варто провести моделювання оптимізованого процесу і порівняти його ефективність з ефективністю підходів та методологій, які являються поширеними на ринку. На основі цього можна зробити висновок про те наскільки використання засобів BDD дозволяє оптимізувати процес тестування програмного забезпечення, а також оцінити сильні і слабкі сторони даної методології.

2.3 Короткий опис моделей розробки програмного забезпечення

2.3.1 Водоспадна модель розробки програмного забезпечення

Попри те, що процес тестування програмного продукту є довгим і об’ємним, тим не менш він лишається лише складовою частиною процесу розробки ПЗ. Тому для того, щоб краще зрозуміти ті правила, якими керуються команди тестувальників під час організації процесу, необхідно зрозуміти місцезнаходження тестування в ланцюгу розробки продукту. Для цього треба розглянути існуючі моделі розробки програмного забезпечення.

Модель розробки ПЗ - структура, що систематизує різні види проектної діяльності, їх взаємодію і послідовність в процесі розробки ПЗ. Вибір тієї чи іншої моделі залежить від масштабу і складності проекту, предметної області, доступних ресурсів і безлічі інших факторів. Вибір моделі розробки серйозно впливає на процес тестування, визначаючи вибір стратегії, розподіл ресурсів, розклад тощо [47].

На даний момент існує значна кількість різноманітних методологій, проте для наглядності буде зручно виділити три основні «архетипи»: водоспадна модель, ітераційна модель та гнучка модель.

Дана модель є найстарішою з усіх та інтуїтивно зрозумілою. Її основна суть полягає в тому, що кожен наступний етап розробки слідує виключно після виконання попереднього. Кожен етап виконується лише один раз і більше ніколи до нього не можна повертатись. Якщо говорити спрощено, то в рамках даної моделі команда в будь-який момент часу бачить тільки результат попереднього етапу та мету, яку необхідно досягнути для початку наступного етапу.

В реальному ж світі під час розробки програмного забезпечення необхідно мати чітке бачення проекту загалом і повертатись до попередніх етапів у випадку необхідності коректив чи уточнень. З точки зору тестування дана модель

вважається неефективною через те, що безпосередньо процес тестування починається ближче до кінця розвитку проекту, що в сучасних реаліях призводить до значного зростання вартості розробки програмного продукту. Саме через цю громіздкість та неефективне використання ресурсів дана модель вважається нині застарілою [5].

Проте з іншого боку не можна ігнорувати і очевидні переваги водоспадної моделі. Зокрема її простоту та інтуїтивну зрозумілість для будь-якої люди навіть без спеціалізованих знань. Управління проектами, що використовують цю модель, стає значно простішим завдяки чіткій детермінації точок переходу між етапами розробки, рівно як і тому факту, що етапи між собою не перетинаються, створюючи відчуття впорядкованості в роботі.

Попри те, що дана модель і вважається застарілою, саме простота і зрозумілість дозволяє їй продовжувати існування на рівні повсякденних задач. Так на рис. 1.3 можна помітити, що в розрізі окремого спринту процес тестування має багато ознак водоспадного методу. Саме на основі цього методу базується методика Test-Last Development, яка і нині являється одним із найбільш поширених засобів організації процесу тестування програмного забезпечення.

2.3.2 Ітераційна модель розробки програмного забезпечення

Дана модель являється фундаментальною основою для сучасного підходу до розробки програмного забезпечення. Як можна зрозуміти з назви, ця модель ґрунтується на ітераціях, тобто на багаторазовому повторенні одних і тих самих стадій [7]. В деяких джерелах дана модель називається ітераційно-інкрементальною, акцентуючи увагу також на прирості корисних функцій продукту з кожною ітерацією. Проте офіційний глосарій ISTQB розділяє ці два поняття, а в розрізі процесу тестування програмного забезпечення більшу цінність являє собою саме ітераційна складова моделі.

2.3.3 Гнучка модель розробки програмного забезпечення

Дана модель є по суті сукупністю різноманітних підходів до розробки програмного забезпечення, які були зібрані в так званому «Agile-маніфесті». По суті своїй даний документ закликає відійти від формальності в усіх аспектах діяльності проекту і зосередитись на тих моментах, які забезпечують ефективність роботи («Люди та взаємодія важливіші інструментів та процесів», «Продукт важливіший від документації», «Готовність до змін важливіша від початкового плану» тощо) [50].

Саме через те, що дана модель є сукупністю багатьох підходів, можна сказати, що вона увібрала в себе все те, що протягом десятиліть було

протестовано та імплементовано іншими моделями розробки програмного забезпечення.

Проте якщо говорити дуже спрощено суто технічно можна сказати, що гнучка модель являє собою полегшену в контексті документацію версію ітераційної моделі, яка керується «agile-маніфестом».

З точки зору тестування на відміну від ітераційної версії, яка імплементувала процес тестування лише на певних етапах ітерації, гнучка модель дозволяє задіяти даний процес буквально в будь-який момент, коли це буде необхідно. Саме ця гнучкість стала причиною того, чому на противагу Test-Last Development з'явились так звані Test-First Development. Наявний підхід до тестування не дозволяв ефективно провадити даний процес в умовах постійних змін. Саме тому ідея TFD була імплементована в 2001 році, коли було створено Test Driven Development.

2.4. Опис та порівняння основних методологій до управління процесом тестування програмного забезпечення

2.4.1 Основні засади підходів до управління процесом тестування програмного забезпечення

Проаналізувавши основні моделі розробки програмного забезпечення, стало очевидно, що їх розвиток та поширення стало причиною для появи двох діаметрально протилежних в своїй ідеології підходів до проведення процесу тестування: Test-Last Development (створений на основі водоспадної моделі) та Test-First Development в формі моделі Test Driven Development (заснований на основі гнучкої моделі). Тепер необхідно розглянути окремо кожен з цих підходів.

Як можна зрозуміти з назви, Test-Last Development описує такий метод, за якого тестування програмного продукту відбувається безпосередньо після того, як було написано код необхідного функціоналу. В часи, коли водоспадна модель була основною на ринку, подібний метод був єдиним, який вписувався в дану модель розробки. Проте навіть зараз TLD являється популярним методом і часто є першим методом, який команди намагаються впровадить. Як було зазначено вище, в цьому випадку тести пишуться вже після того, як написано код. Тестові сценарії мають на меті перевірку функціональності написаного коду, після чого вони використовуються розробниками для відпрацювання коду, доки всі тести не будуть успішно пройдені.

В противагу TLD ідеологія TFD полягає в тому, щоб писати тести до того, як буде написано код, і тільки після цього робити код, який створений для того,

щоб завідомо пройти тести. Варто відмітити, що така особливість даного підходу призводить до того, що процес тестування стає тісно пов’язаним з процесом розробки коду і тому часто імплементація TFD набуває рис моделей, що застосовуються під час створення коду.

Найбільш відомим прикладом TFD є модель Test Driven Development (TDD). В своєму оригінальному вигляді ця модель по суті є ітеративним втіленням TFD. В першу чергу розробник створює тест, який продукт не проходить. Після того як тест провалено, він створює мінімальну необхідну кількість коду для того, щоб цей тест став успішним, після чого впроваджений функціонал інтегрується до системи. Так продовжується доки код не буде написаний повністю.

Проте в реальному світі дуже нечасто послуговуються інструкцією, наведеною вище. Зараз TDD більше нагадує широкий набір практик, які певною мірою намагаються слідувати ідеям, закладених в оригінальному вигляді моделі.

Кожен з цих підходів має свої переваги та недоліки. Оскільки вони створені на основі протилежних моделей розробки програмного забезпечення, то в більшості випадків переваги одного підходу в той самий час являються недоліками іншого.

З точки зору часу, затраченого на розробку, TDD потребує в середньому на 16% більше часу ніж TLD [43]. Причиною цьому слугує ітеративність процесу, який часто рухається між тестуванням, написанням коду, рефакторингом тощо. Розробник не може сфокусуватись виключно на написанні коду, оскільки необхідно постійно оглядатись на створені тестові сценарії. Що приводить до наступної категорії.

З точки зору важкості входження TDD має великі проблеми. Навіть для спеціалістів в своїй галузі необхідність думати про тести до того, як написати код, є неочевидною. Для ефективного впровадження такого підходу від розробників та тестувальників необхідно багато практики та дисципліни. В цьому плані TLD виграє за рахунок своєї простоти та інтуїтивності.

З точки зору продуктивності та вартості підтримки TDD стає більш вигідним. Даний метод робить підтримку більш дешевою та надійною. Зв’язок між тестуванням та написанням коду дозволяє провадити тестування вже на етапі розробки коду. Окрім цього покриття тестами збільшується в середньому на 52%.

З точки зору розміру коду, код, створений відповідно до TLD, має відносно невеликий розмір. З урахуванням того, що кількість тестів при використанні TDD більша, то відповідно більшими стають і обсяги коду. Проте з іншого боку даний

метод сприяє створенню мінімально необхідного коду, тому ефективність коду може бути вищою.

З точки зору внесення змін в код, TDD також є ефективнішим. Значна кількість повторних перевірок може фактично гарантувати те, що внесення змін та імплементація нового функціоналу не призведе до порушень в роботі програмного продукту. Натомість TLD не провадить часті повторні перевірки і гарантувати справність вже впровадженого функціоналу може зі значно меншою вірогідністю.

З точки зору простоти коду, перевагу отримує TLD. Прямолінійність підходу дозволяє робити код більш прямолінійним та легким для сприйняття. Натомість циклічність TDD потребує використання більш спеціалізованих принципів проектування таких як інтерфейси, паттерни проектування тощо.

Таким чином, підбиваючи підсумки можна сказати, що вищеописані підходи в багатьох аспектах доповнюють одне одного і так чи інакше мають право на існування. Проте в умовах сучасного ринку кожен підхід потребує власного контексту. TLD часто використовується для невеликих проектів, які створюються невеликою командою та орієнтовані на невелику аудиторію. Натомість TDD поширений серед великих проектів, які мають велику професійну команду та націлені на довготривалий життєвий цикл продукту.

Після того, як було проведено аналіз наявних літературних джерел, які дали уявлення про сутність ручного та автоматизованого тестування, їх переваги та недоліки, а також проблеми їх співіснування в рамках діючих проектів сучасної сфери IT, можна приступити до безпосередніх досліджень варіантів рішення даної проблеми, які виходять за рамки BDD підходу.

За останні двадцять років, які минули з першої публікації про проблеми автоматизованого тестування, спеціалісти розробили незліченну кількість практик розробки, котрі мають на меті подолання тих чи інших проблем. Деякі з них запозичувались із підходів до розробки, які практикувались на підприємствах, деякі були синтезовані штучно. Проте важливо розуміти, що часто привабливість опису не відображає того, наскільки дієвою є та чи інша практика.

Підходи до розробки поділяються за складністю, областю застосування та метою. Таким чином в першу чергу необхідно з’ясувати навіщо вони потрібні, чому їх так багато і чим вони можуть бути корисні в контексті озвучених проблем.

2.4.1 Методологія TDD – Test Driven Development

Test Driven Development – це методологія розробки програмного забезпечення, яка ґрунтується на повторенні коротких циклів: в першу чергу

пишеться тест, який покриває бажаний функціонал, після цього пишеться програмний код, який реалізує цей функціонал і дозволяє успішно пройти створений тест. Далі відбувається рефакторинг коду з подальшою перевіркою проходження тестів.

Даний вид розробки є одним із самих популярних і вважається одною із форм правильного метода побудови додатку, адже тести в такому випадку стають по суті специфікацією того, як працює продукт. Якщо тестовий набір є обов'язкова частина процесу збірки, то якщо тести не проходять – програма не зможе зібратись, адже працює невірно.

В даному підході обмеження полягає в тому, що правильність роботи програми в значній мірі полягає в повноті тестів, які її описують. Проте попри це TDD широко поширена, адже дозволяє значно скоротити кількість помилок під час виробництва.

Ця методологія дозволяє досягнути створення зручного з точки зору покриття автоматизованими тестами коду, який до того ж буде заздалегідь покритий модульними та частково інтеграційними тестами. Також TDD гарантує працездатність основного функціоналу і як наслідок стабільність роботи продукту загалом, яка постійно перевіряється. В свою чергу це покращує ефективність на етапі підтримки програмного продукту, адже всі дефекти, які з'явились в результаті внесення нових змін в код чи змін оточення, локалізуються дуже швидко завдяки високому відсотку покриття коду тестами.

Попри те, що дана методологія надзвичайно ефективна на етапах модульного та інтеграційного тестування, вона не приносить багато користі, коли необхідно перевірити роботу системи та її взаємодію з кінцевим користувачем. Навіть димове тестування не може бути покритим цією методологією, адже вона гарантує працездатність основного функціоналу, а не правильну його взаємодію з користувачем.

Тому часто Test Driven Development являється методологією, яка поширена серед розробників для покращення працездатності коду та формування тестового покриття вже на етапі розробки.

2.4.2 Методологія Type Driven Development

Type Driven Development – методологія розробки, яка ґрунтується на використанні типів даних і сигнатур в якості специфікації програми. Хід розробки аналогічний до TDD (Type Driven Development має аналогічне скорочення з Test Driven Development, тому зазвичай використовується без скорочення). Типи також слугують документацією, яка гарантовано оновлюється.

Типи являють собою невеликі контрольні точки, завдяки яким на виході отримується велика кількість невеликих тестових випадків, які розкидані по всьому програмному продукту. Окрім того затрати на створення типів мінімальні, оскільки нема потреби в їх створенні та актуалізації, оскільки вони являються частиною кодової бази.

Type Driven Development за аналогією з TDD вважається ще одним правильним методом побудови програмного продукту. Як і в минулому випадку, розробка на основі типів дозволяє значно знизити кількість помилок, які були допущені в процесі написання коду, створити відчутне покриття коду тестами на етапі модульного та інтеграційного тестування.

Окрім того Type Driven Development дозволяє зекономити час при внесенні змін в велику кодову базу, які моментально будуть застосовані до всіх типів в програмному продукті.

Будучи в питанні недоліків схожою до TDD, Type Driven Development має свою специфічну проблему. Через те, що в основі методології лежить використання типів, то імплементація даного підходу важча для мов програмування з динамічною типізацією. Так, наприклад, Type Driven Development відчутно важче впровадить на мові JavaScript ніж на мові TypeScript.

2.4.3 Методологія FDD – Features Driven Development

Features Driven Development – це методологія, яка являє собою спробу об'єднати найбільш визнані в індустрії розробки програмного забезпечення методики, що беруть за основу важливу для замовника функціональність розроблюваного програмного забезпечення. Основною метою даної методології є розробка реального, працюючого програмного забезпечення систематично, в поставлені терміни.

Як і інші адаптивні методології, вона робить основний наголос на коротких ітераціях, кожна з яких служить для опрацювання певної частини функціональності системи. Згідно FDD, одна ітерація триває два тижні. FDD налічує п'ять процесів (рис. 3.1 [36]). Перші три з них відносяться до початку проекту:

- Розробка загальної моделі
- Створення списку необхідних властивостей системи
- Планування роботи над кожною з властивостей
- Проектування кожної властивості
- Конструювання кожної властивості

Останні два кроки виконуються під час кожної ітерації, при цьому кожен процес розбивається на задачі зі своїми власними термінами та критеріями верифікації. Розглянемо детальніше кожен з етапів.

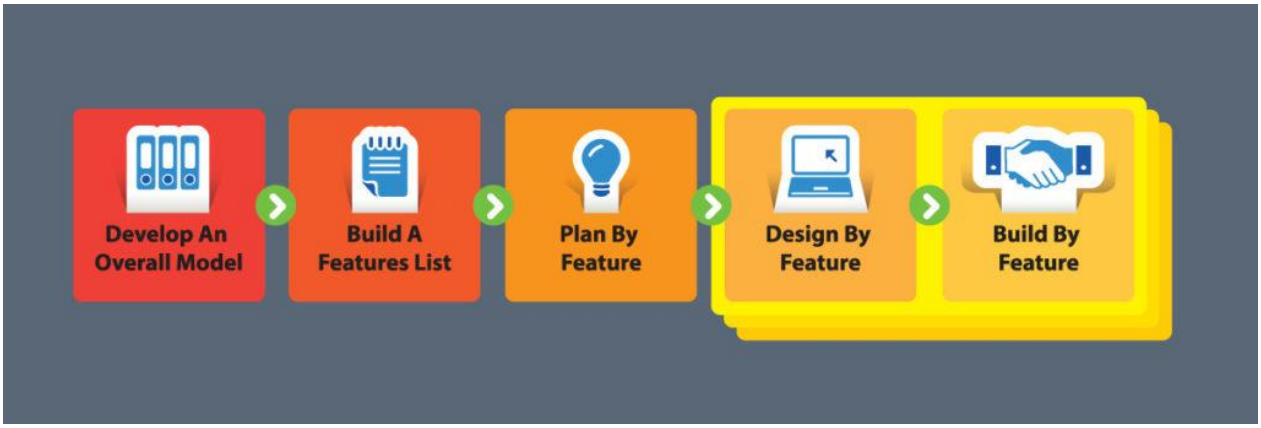

Рис. 2.4. Схема циклу згідно FDD

Розробка починається с аналізу широти наявного кола завдань і контексту системи. Далі для кожної моделюється області робиться більш детальний розбір. Попередні опису складаються невеликими групами і виносяться на подальше обговорення і експертну оцінку. Після одна із запропонованих моделей або їх сукупність стає моделлю для конкретної області. Моделі кожної області завдань об'єднуються в загальну підсумкову модель, яка може змінюватися протягом роботи.

Інформація, зібрана при побудові загальної моделі, використовується для складання списку функцій. Функції об’єднуються в «області», а ті в свою чергу поділяються на «підобласті» за ознакою функціональності. Кожна підобласть відповідає певному бізнес-процесу, а його кроки стають списком функцій (властивостей). Функції представлені у вигляді «дія - результат - об'єкт», наприклад, «перевірка пароля користувача». Розробка кожної функції повинна займати не більше двох тижнів, інакше завдання необхідно розбити на більш дрібні ітерації. Список властивостей в FDD - те ж саме, що і product backlog в SCRUM.

Далі відбувається розподіл наявних функцій між програмістами чи командами. Після цього створюється пакет проектування. Виділяється група властивостей для розробки протягом двох тижнів. Після залишаються детальні діаграми послідовності для кожної властивості, уточнюючи загальну модель. Далі пишуться «заглушки» класів і методів.

Останнім етапом пишеться код, прибираються «заглушки» і відбувається безпосереднє тестування.

На виході отримуємо документацію властивостей системи, детальне проектування, покращену оцінку невеликих задач, тести, орієнтовані на бізнес-задачі, детальний процес створення програмного продукту, значний приріст продуктивності за рахунок ітеративності розробки.

Проте найбільшим мінусом даного підходу є той факт, що невеликі команди на відміну від великих проектів не відчують всі переваги даної методології.

2.4.4 Методологія MDD – Model Driven Development

Model Driven Development – стиль розробки програмного забезпечення, коли моделі стають основними артефактами розробки, на основі яких генерується код та інші артефакти. Інакше кажучи, вся суть розробки зводиться до побудови необхідних діаграм з яких в подальшому буде згенеровано безпосередній працюючий код проекту. Схематично суть Model Driven Development зображено на рис. 2.5 [48].

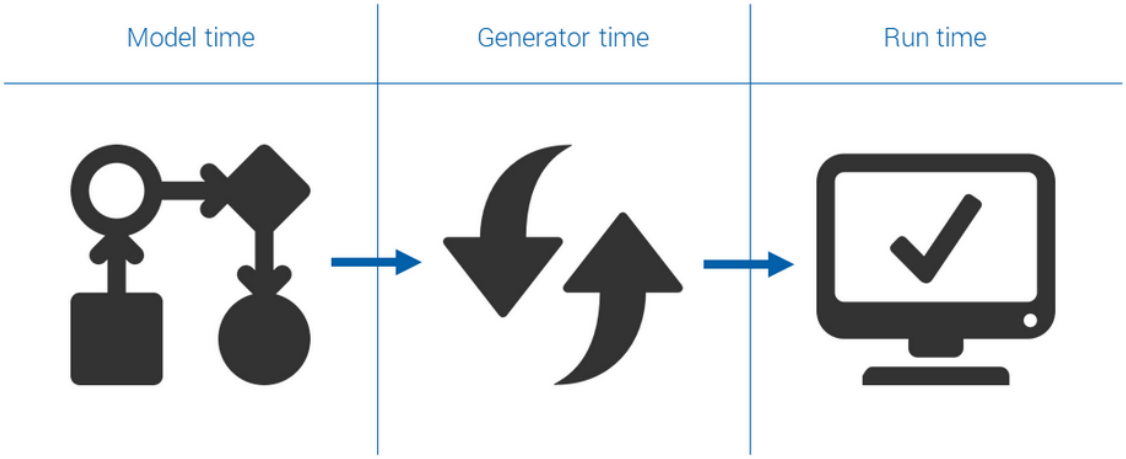

Рис. 2.5. Схема роботи за MDD

Основна мета MDD - мінімізація витрат, пов'язаних з прив'язкою до конкретних системним платформ і програмних інфраструктур. Адже основна бізнес-логіка міститься в діаграмах і не сковує розробників рамками вибору мови програмування і інструментів розробки.

У MDD діаграми - це ще один рівень абстракції, який не дозволяє загрузнути в деталях розробки, а подивитися на картину в цілому. Діаграми виступають в якості своєрідних «креслень», з яких різні автоматизовані і напівавтоматизованого процеси витягують програми і відповідні моделі. Причому

автоматична генерація коду варіюється від вилучення простого скелета додатку до отримання кінцевої кодової бази (що можна порівняти з традиційною компіляцією).

Ідея MDD не є новою і значний час використовувалась з різною мірою успіху. Причиною росту інтересу до неї є той факт, що автоматизації з плином часу піддається все більше процесів. Цей розвиток відображається в появі стандартів MDD, яскравим прикладом яких є UML 2.0

Класичний приклад застосування MDD, який використовується вже давно, - моделювання баз даних. На основі однієї концептуальної моделі даних інженер баз даних може підтримувати декілька пов'язаних з нею фізичних моделей для різних СКБД.

Серед основних переваг MDD варто відмітити швидкий вихід мінімального життєздатного продукту на ринок, скорочений час генерації каркасу програмного продукту (моделі класів, бази даних тощо), документація, яка постійно оновлюється, і наглядність діаграм порівняно з кодом.

Натомість в мінус можна занести необхідність використання спеціалізованих програмних рішень, яких на ринку не дуже багато (зокрема Rational Software Architect, Simulink або Sirius), також від програмістів необхідне вміння працювати з діаграмами. І на додаток до цього, Model Driven Development зазвичай являється чи не найдорожчим з усіх підходів з точки зору затрат на інтеграцію.

2.4.5 Методологія DDD – Domain Driven Design

Domain Driven Design – це набір принципів та схем, які направлені на створення оптимальних систем об’єктів. Процес розробки зводиться до створення програмних абстракцій, які називаються моделями предметних областей. У ці моделі входить бізнес-логіка, що встановлює зв'язок між реальними умовами області застосування продукту і кодом.

Основна мета Domain Driven Design - це боротьба зі складністю бізнес-процесів, їх автоматизації та реалізації в коді. «Domain» перекладається як «предметна область», і саме від предметної області відштовхується розробка і проектування в рамках даного підходу.

Ключовим поняттям в DDD є «єдина мова» (ubiquitous language). Ubiquitous language сприяє прозорому спілкуванню між учасниками проекту. Єдиний вона в контексті того, що всі учасники спілкуються на ній, все обговорення відбувається в термінах єдиної мови, і всі артефакти максимально повинні викладатися в термінах єдиної мови, тобто, починаючи від ТЗ, і, закінчуючи кодом.

Наступним поняттям є "доменна модель". Дана модель являє собою словник термінів з ubiquitous language. І доменна модель, і ubiquitous language обмежені контекстом, який в Domain-Driven Design називається bounded context. Він обмежує доменну модель таким чином, щоб всі поняття всередині нього були однозначними, і всі розуміли, про що йде мова.

Приклад: візьмемо сутність "чоловік" і помістимо його в контекст "публічні виступи". У цьому контексті, за DDD, він стає спікером або оратором. А в контексті "сім'я" - чоловіком або братом.

З точки зору Domain-Driven Design абсолютно все одно, яку архітектуру ви виберете. Але DDD майже неможливий без чистої архітектури проекту, так як при додаванні нової функціональності або зміні старої потрібно намагатися зберігати гнучкість і прозорість кодової бази. Приклад підходящої для DDD архітектури наведено на рис. 3.3 [30].

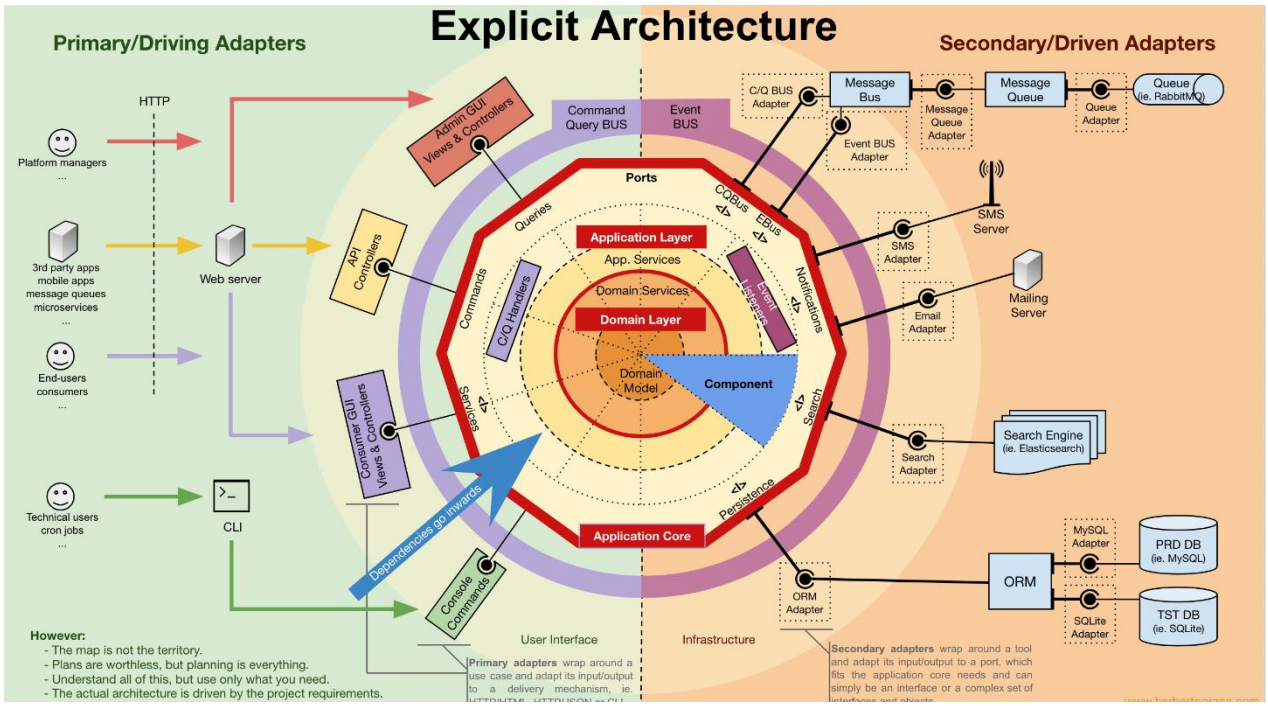

Рис. 2.6. Архітектура для впровадження DDD

Серед основних переваг Domain Driven Design можна виділити наступні:

- Код проекту написаний на зрозумілій для всіх учасників проекту мові
- Постановка задач стає більш явною
- Дефекти бізнес-логіки стають очевиднішими
- Пошук дефектів тестувальниками значно покращує завдяки легкості читання коду

Мінусів такого підходу відносно небагато, проте своєю наявністю вони здатні легко переважити всю легкість розуміння коду та вигоду з точки зору пошуку дефектів. Зокрема важливим недоліком є необхідність в спеціалістах найвищого класу для вдалого запуску подібного типу проекту, що являється наслідком неортодоксальності самого підходу, а також використання специфічної архітектури.

Для замовника ж основною перепоною на шляху використання Domain Driven Design стає той факт, що попри грошові затрати на пошук спеціаліста, який був би готовий провадить дану методологію, DDD являється таким підходом, який потребує участі всієї команди. Тобто таким чином після його впровадження вся команда буде змушена навчатись розробці програмного продукту відповідно до цих правил, що безсумнівно вплине на витрату як грошових, так і людських ресурсів.

2.5 Порівняння основних методологій тестування програмного забезпечення

Варто відмітити, що в пунктах 2.4.1-2.4.5 були описані різні методології розробки програмних продуктів. Проте це не означає, що цим списком все закінчується. Існує значна кількість ще більш рідких видів проектування. Деякі з них являють собою логічне продовження чи модифікацію вищеописаних підходів. Деякі з них мають значно вужчу спеціалізацію. Так окрім вищеназваних підходів було також розглянуто Documentation Driven Development, Defect Driven Development, Responsibility Driven Development та User Guide Driven Development. Проте жоден з цих підходів не був достатньо унікальним чи корисним для вирішення проблем оптимізації процесу тестування.

З шести методологій описаних в попередніх пунктах є три основні, які розповсюджені серед найбільшої кількості компаній. Цими підходами є Test Driven Development, Feature Driven Development та Behaviour Driven Development. Але чи являються вони найбільш корисними для оптимізації всього процесу тестування? І чи не являються DDD, MDD і Type Driven Development незаслужено забутими?

Говорячи про Domain Driven Design в першу чергу варто відмітити його громіздкість і високі затрати матеріальних та людських ресурсів на етапі впровадження, які далеко не відразу почнуть окупатись. Таким чином DDD являє собою приклад концептуально нового підходу до розробки, який в реальному житті виявляється не найпрактичнішим.

Model Driven Development має значно більший потенціал. Як і BDD цей підхід дозволяє ширшому колу спеціалістів бути залученим до процесу розробки завдяки тому, що діаграми являються значно більш зрозумілими ніж код. Автоматична генерація коду та швидша поява працездатного продукту фактично нівелюється обов'язковістю використання спеціалізованих продуктів. А оскільки ринок MDD-інструментів не є надто великим, то існуючі рішення можуть мати як завищену ціну, так і незадовільну функціональність. Тому не кожен проект буде готовий вкладати реальні гроші за використання сумнівних інструментів.

Type Driven Development концептуально являється похідною методологією від Test Driven Development і тому логічно було б його не розглядати в рамках даного дослідження поряд з його ж попередником. Проте Type Driven Development запропонував корисну ідею кодової бази, яка зберігає в собі інформацію про наявні типи, таким чином виносячи частину функціоналу за межі самого тесту. Тобто таким чином нема необхідності дублювати внутрішню частину тесту, натомість для цього можна використати дані, інкапсульовані всередині типу, який можна отримати з кодової бази.

Test Driven Development являється найпоширенішим підходом з усіх вищеназваних. Він благополучно доводить свою ефективність і покращує процес розробки в сотнях компаній по всьому світу. В контексті ж тестування ця методологія не дуже практична, адже створення тестів в автоматизації є самоціллю і досягнення її за допомогою створення нових тестів є абсурдною. Тому дана практика ніяким чином не допоможе в питанні оптимізації тестування.

В свою чергу Behaviour Driven Development являється неймовірно корисним, коли діло до ходить до оптимізації тестування. В теорії BDD надає просто неймовірну кількість переваг порівняно з іншими методологіями. Універсальність методології, її простота для розуміння, можливість залучати до процесу тестування аналітиків, розробників та навіть представників клієнтів, проста і легка документація – все це мало б з легкістю вивести дану методологію на перший рядок в списку найпопулярніших.

Більшість IT компаній не використовують BDD через те, що перебудування проекту під неї займає час, подовжує цикл розробки і йде поперек класичних уявлень про процес ведення розробки. Інакше кажучи, проблема Behaviour Driven Development з точки зору організації полягає не в недоліках самої методології, а в реалізації та існуючих інструментах.

Саме тому ті підприємства, які все ж вирішують імплементувати BDD, зазвичай користуються готовими фреймворками на кшталт Cucumber, Squish, JBehave, Yulup тощо. Які самостійно реалізують всі необхідні функції.

Проте в рамках конкретно даного проекту було проведено опитування між представниками майже двох десятків проектних команд на тему використання тих чи інших методологій в процесі тестування програмного забезпечення.

Відповідно до даних опитування майже третина стрімів практикує в своїй діяльності Test Driven Development. Стільки ж використовують підхід Behaviour Driven Development, який по суті являється продовженням TDD і містить в собі його елементи. Решта ж розподілилась між іншими методологіями.

Повні дані результати дослідження наведені в табл. 2.1

Таблиця 3.1

Розподіл використання методологій серед проектних команд

Назва	Опис	Кількість стрімів, що використовують методологію
Test Driven Development	Імплементація функціоналу відбувається на основі створених тестів	6
Type Driven Development	Розвиток TDD, який передбачає групування тестів в типи відповідно до функціоналу, що перевіряється	1
Feature Driven Development	Робота відбувається виключно в рамках одної групи функціональності	3
Model Driven Development	В основі роботи лежить модель бізнес-процесу	2
Domain Driven Design	Всі процеси проекту працюють в рамках однієї доменної області	1
Test Last Development з використанням мови Gherkin	Написання тестів відбувається нативною мовою	6

Проте як показало дослідження, використання на базі Behaviour Driven Development корисних аспектів з інших методологій таких, як Type Driven

Development та Model Driven Development дозволить оптимізувати і уніфікувати не лише процес автоматизації тестування, а й процеси ручного тестування і створення тестової документації.

2.6 Основні засади методології Behaviour Driven Development

Як було описано раніше для вирішення проблеми недостатньо ефективної взаємодії процесів ручного та автоматизованого тестування існує два основних підходи, які покладаються на оптимізацію використання людських ресурсів і на покращення інструментарію для тестування завдяки використанню автоматизованих фреймворків.

Кожен з цих варіантів ліг в основу двох діаметрально протилежних підходів. Так Test First Development (TFD) в спробах підвищити продуктивність тестування за рахунок впровадження розробки через тестування (Test-Driven Development).

Методологія TDD розраховувала, що початковий рівень якості програмного продукту буде вищий, адже ряд першочергових тестів було проведено ще під час безпосередньо розробки.

Натомість Test Last Development (TLD) завдяки інтуїтивній зрозумілості проведення тестування готової частини програмного продукту, а не спроби перевірити те, що ще не було створено, і досі залишається популярною серед багатьох команд.

Цьому сприяє поява все більш ефективних інструментів для автоматизації тестування, які дозволяють отримувати хороші показники продуктивності не вдаючись до внесення змін в класичний Scrum.

Проте з плином часу змінюються вимоги ринку і тому на основі кожного з цих підходів з’являється все більше методологій, які вважають той чи інший аспект процесу тестування першочерговим для впровадження.

Це можуть бути як прості варіанти розвитку класичних підходів (зокрема Type Driven Development і Feature Driven Development, які пролонгують ідеї TDD, проте фокусуються на типах даних та окремих частинах функціоналу відповідно), так і самостійні рішення (зокрема Model Driven Development, яка полягає в створенні моделей, які потім будуть конвертуватись у відповідний код, і таким чином тестування проводитиметься не над самим кодом, а над моделлю). Проте більшість з цих підходів являються вельми специфічними і тому не набули широкого використання. Із усього дещо виокремлено знаходиться методологія Behaviour Driven Development або BDD.

Behaviour Driven Development – це методологія, яка заснована на описі поведінки. Створена в 2003 році Доном Нортоном у якості відповіді на TDD, ця методологія включала в себе не лише прийомне тестування з боку клієнта, а й активне залучення його (або інших неспеціалістів, задіяних на проекті) до безпосереднього створення тестових сценаріїв

В своїй суті BDD являється духовним продовженням TDD, тобто в ній закладено, що в першу чергу створюються тести, а потім реалізуються в програмному коді. Проте в ній є одна важлива відмінність. Якщо TDD використовується для програмування та тестування на рівні технічної реалізації продукту, то BDD більше спрямоване на опис тестів з точки зору сценаріїв користувача на природній мові.

На відміну від інших методологій BDD сприймає тест не як набір кроків для перевірки певної функціональності, а як повноцінний сценарій, якому може слідувати користувач системи. Таким чином нівелюється проблема підвищеної кількості тестів, яка властива розробці за класичним TDD.

Окрім того для написання користувацьких сценаріїв використовуються не мови програмування, а нативна мова Gherkin. За її допомогою певна людина або група людей, яким навіть не обов’язково бути спеціалістами в тій чи іншій мові програмування, здатна створити тест за допомогою формули «Given-When-Then» і простого опису подій відповідно до цієї формули.

Далі за допомогою спеціальних інструментів, або силами тестувальників сценарій переводиться в код. Після цього відбувається стандартна розробка з тестами.

BDD дозволяє відмовитись від legacy-документації, яка має неактуальну інформацію, і отримувати оновлення дуже швидко, відразу переводячи її на мову Gherkin і зберігаючи її разом з проектом. Таким чином дана методологія сприяє тісній співпраці і близькості тестувальників та бізнес-аналітиків до процесу розробки коду, що підвищує їх замученість і втілює одну із ключових засад сформульованих в класичному описі TDD.

Behaviour Driven Development являється по суті процесом, який має на меті зменшення вартості впровадження нового функціоналу. Документація за такого підходу стає зрозумілою всім учасникам процесу розробки, що знижує поріг входження для нових учасників проекту.

Навіть при короткому огляді можна помітити, що BDD, будучи нащадком TDD доволі ефективно нівелює принаймні деякі недоліки свого попередника. Використання нативної мови Gherkin дозволяє легше увійти в проект підготовленому спеціалісту.

Той факт, що тести являють собою реальні користувацькі сценарії дозволяє знизити кількість тестів, а розподіл їх на специфікацію та імплементацію (інакше кажучи на опис тесту та робочий код) дозволяє зробити тести простішими і більш зрозумілими для кожного члена команди незалежно від його ролі та професійних навичок.

Окрім цього важливою технічною перевагою даної методології є той факт, що тести самі по собі не залежать від цільової мови програмування і тому можуть бути з легкістю портовані під інші умови розробки.

Проте навіть з такими очевидними перевагами, Behaviour Driven Development має також і очевидні недоліки. І якщо більший час на розробку можна списати на таку ж проблему, яку мав оригінальний TDD (хоча впровадження додаткових специфікацій і залучення тестувальників на більш ранніх етапах розробки збільшує період розробки саме по собі), то значні затрати ресурсів часу і грошей на імплементацію даної методології є проблемою виключною і самостійною.

3. МОДЕЛЮВАННЯ ПРОЦЕСУ ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ЗА ДОПОМОГОЮ ЗАСОБІВ BDD

3.1 Опис процесу тестування відповідно до методології BDD

На початку роботи команди, які зазвичай працювали раніше класичними TDD або TLD методиками, стикаються як з проблемами інструментарію, так і з труднощами реалізації. З одного боку впровадження будь-якого фреймворку вимагає спеціалізованих навичок та додаткових компетентностей від тестувальників. З іншого ж внесення значних змін безпосередньо в організацію робочого процесу призводить на перших етапах до того, що проектна команда так чи інакше буде використовувати старі звички, які лишились з часів, коли використовувались інші підходи.

Зокрема, неочевидною особливістю є робота зі специфікацією, яка по суті являється користувацьким сценарієм і тестом одночасно. Зазвичай розробкою самого сценарію, його втіленням у вигляді тесту та написанням коду автоматизованого тесту займаються відповідно аналітик, мануальний тестувальник та автоматизатор. В класичних методологіях результат роботи одного спеціаліста стає відправною точкою для наступного. Таким чином все відбувається поступово і не призводить до хаосу. Натомість під час використання BDD все інакше.

Причиною цьому стає той факт, що специфікація по суті і є тестом, програмування якого не потрібне. Натомість тестувальнику необхідно автоматизувати не тест в цілому, а розробити механізми роботи кожного з кроків, які в подальшому можуть бути використані при написанні інших тестів. Наприклад, якщо в специфікації описується вхід на сайт, то автоматизатор замість написання тесту створює код, який буде реалізовувати відкриття сторінки входу, введення логіну-паролю, натискання кнопки входу тощо.

Таким чином, поки тестувальник займається програмною реалізацією, змінюється сенс функціональності. Тому виходить, що комунікації між командою мінімум, адже тестувальник робить реалізацію, аналітик вносить зміни в документацію і десь між цим всім змінюються самі користувацькі сценарії в специфікаціях. І через цю неузгодженість часто процес тестування переходить в те ж саме русло, яке було і раніше, не додаючи ніяких переваг.

Таким чином важливим аспектом оптимізації тестування згідно даної методології є написання тестів безпосередньо аналітиками. Тобто одна людина повинна розробляти документацію і провадити створення тестів. Проте навіть

просте створення тестових сценаріїв потребує відповідних навичок. Тому виконання цієї умови часто призводить до наступної проблеми: відсутність спеціалізованих технічних знань у аналітика.

В ідеальному випадку аналітику було б бажано мати знання як в області тестування, так і в області програмування та автоматизації для того, щоб ефективно комунікувати з різними членами команди. Проте лише дуже незначна частка аналітиків може мати рівень знань програмування, тестування та автоматизації на рівні з самими тестувальниками. Тому є необхідність зробити використання фреймворку для тестування максимально легким для розуміння та дружнім до неспеціаліста.

Основними напрямками вирішення цієї проблеми є використання спеціалізованих плагінів для роботи з тестовими кроками, або залучення автоматизаторів для розробки конкретних кроків тесту.

В першому випадку на проекті впроваджуються спеціалізовані інструменти на кшталт Masquerade, які максимально полегшують процес написання механізму кожного кроку за допомогою величезних бібліотек, які мають вбудовані функції на будь-який випадок життя. Завдяки цьому будь-хто буде здатен побудувати реалізацію тестових кроків, використовуючи заздалегідь розроблені функції в якості будівельних блоків і не роздумуючи про те, як вони реалізовані на більш глибинному рівні. Таким чином розрахунок йде на те, що неспеціалісти зможуть за максимально короткий проміжок часу освоїти мінімально необхідний обсяг інформації з програмування та автоматизації за підтримки конкретного продукту.

В другому же випадку, для імплементації кроків тестового сценарію використовують послуги окремої команди автоматизаторів. Такий варіант дещо більш небезпечний, адже є вірогідність розділення єдиного процесу тестування на два підпроцеси (створення безпосередньо специфікації користувацького сценарію та створення механізмів її роботи). В такому випадку зростає вірогідність того, що ці два процеси будуть працювати паралельно і слабко взаємодіяти одне з одним, тим самим нівелюючи ряд переваг від використання методології. Проте за умови належної комунікації, використання кваліфікованих спеціалістів з автоматизації дозволяє створювати більш надійні та більш ефективні реалізації тестових кроків, адже наявність спеціалізованих навичок та вмінь в області автоматизації дозволить вирішувати поставлені задачі з використанням значно ширшого спектру програмних та проектних інструментів.

Незалежно від того, який варіант було обрано, після його імплементації можна відчути переваги BDD. Зокрема відчутно зменшується час на розробку вимог, специфікації та тестової документації. Окрім того ця ж тестова

документація по суті являється тестом, що вилучає етап написання безпосередньо тестового сценарію. Натомість для автоматизації цього тесту не треба створювати з нуля всю його імплементацію, натомість потрібно лише імплементувати механізм (або використати існуючий) для конкретних чітко виражених кроків.

Різницю між процесами згідно TLD і BDD можна помітити на рис. 3.1 [11] і 3.2 відповідно.

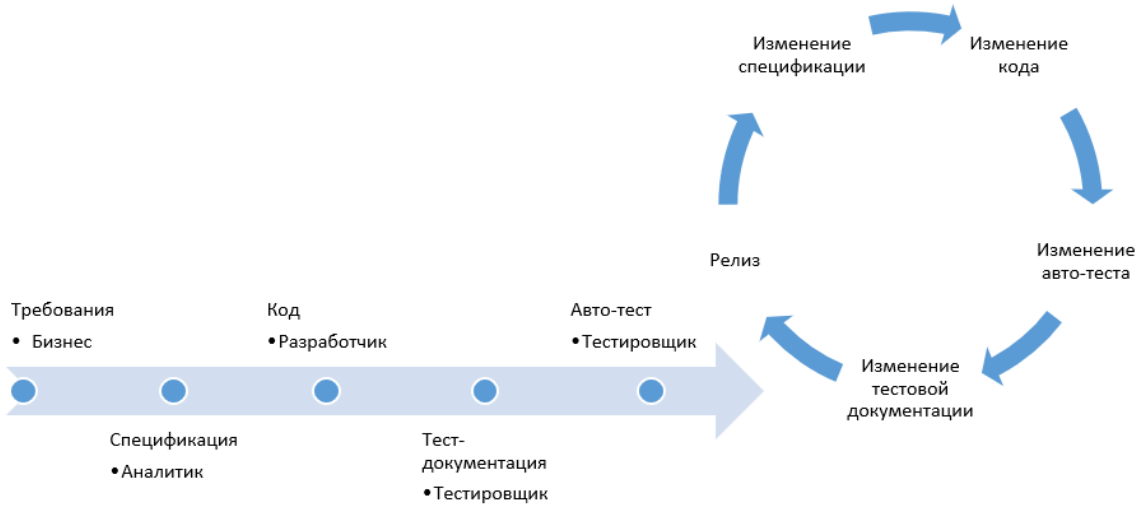

Рис. 3.1. Процес тестування згідно підходу TLD

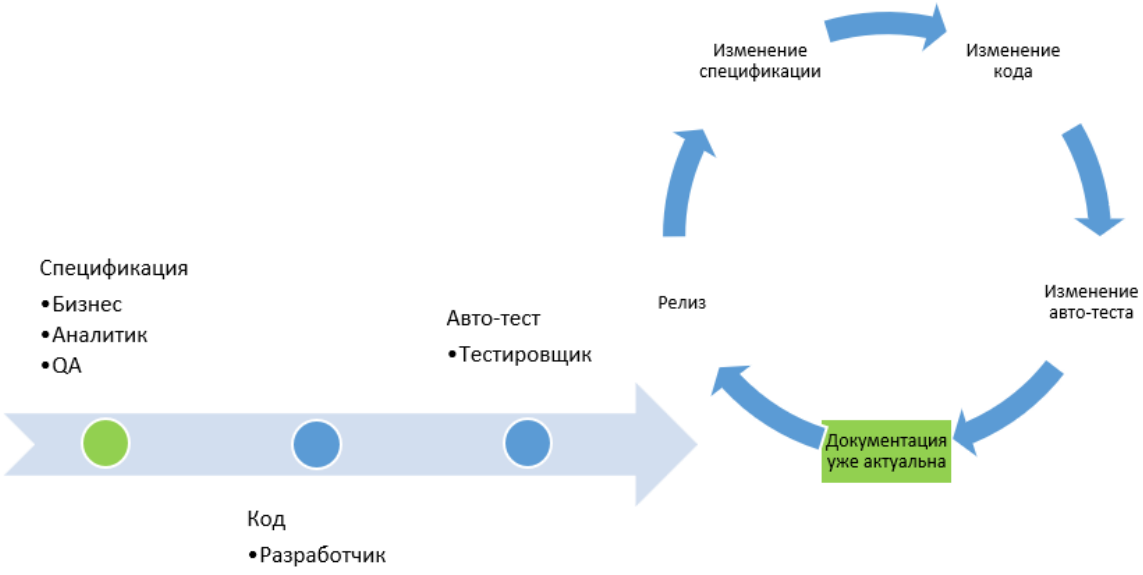

Рис. 3.2. Процес тестування згідно методології BDD

Таким чином варто відмітити, що методологія Behaviour Driven Development дозволяє суттєво знизити затрати часу на підтримку документації,

оскільки зміна специфікація (тобто логіки системи) тягне за собою відразу зміну в усій тестовій конфігурації, економлячи час та ресурси на внесенні змін в кожну її складову. Також за рахунок правильного використання програмних та людських ресурсів час на підтримку тестів може бути зниженим майже вдвічі.

Загальновідомим є факт, що найбільш затратним є усунення дефектів вимог та аналізу. І чим пізніше ця помилка виявляється тим більше ресурсів необхідно витратить на її усунення. Тому механізм, який дає можливість легше знайти цей дефект і усунути його за менший час, суттєво може понизити вартість розробки в цілому. І саме доступ до такого механізму є унікальною та беззаперечною перевагою BDD.

3.2 Аналіз впливу засобів BDD на процес тестування програмного забезпечення

Попри те, що у пункті 3.2 методологія BDD розглядалась в більшій мірі через призму процесу тестування, не варто забувати, що насправді вона охоплює процес проектування та розробки програмного продукту загалом. Про цей аспект часто забувають, вважаючи BDD ще одним способом автоматизації тестування, для якого достатньо встановити відповідний програмний продукт.

Саме тому перш ніж проводити подальші дослідження було важливо визначити згідно якого підходу (чи методології) насправді провадиться процес тестування програмного продукту. Оскільки заявленою методологією стріма і проекту загалом є BDD, то фактично необхідно з’ясувати чи має процес тестування на даному стрімі характерні ознаки даної методології.

З цією метою було проаналізовано процес тестування від моменту отримання вимог і до імплементації або автоматизованого тестового сценарію у складі фреймворку, або до надання тесту статусу «Will not be automated», що свідчить про необхідність провадження даної перевірки виключно силами ручних тестувальників.

На основі опису процесу тестування згідно TLD і BDD, описаному в пункті 3.2 було виділено ряд параметрів за якими можна було б визначити яким чином функціонує процес тестування програмного продукту на даному стрімі. Серед цих параметрів можна виділити наступні:

1) Робота з вимогами

Отримання вимог до програмного продукту або певної частини його функціоналу та подальша їх обробка і уточнення – це те, з чого починається як процес розробки так і процес тестування.

У випадку з Test Last Development обробкою вимог перед створенням специфікації займається бізнес-аналітик або ж повністю віддається на відкуп стороні замовника. Натомість для Behavior Driven Development такий етап окремо не виділяється.

2) Формулювання специфікацій

На основі отриманих вимог створюється специфікація – основний документ, який описує поведінку системи або її частини. Саме на основі специфікацій відбувається як розробка, так і подальше написання тестових сценаріїв для майбутніх перевірок.

Розробкою специфікацій згідно до TLD займається лише бізнес-аналітик, оскільки є або автором оновлених виправлених вимог, або ж тією особою, що виконує обов'язки комунікаційного моста між функціональними командами та стороною замовника.

В свою чергу BDD пропонує поєднати обробку вимог від бізнесу і формування специфікацій. Таким чином потреби замовника відразу перетворюються на інструкції згідно яким можна провадити подальшу імплементацію і тестування. І саме через те, що на основі цих документів буде відбуватись робота команд розробників та тестувальників на цьому етапі необхідне залучення представників від обох підрозділів для формування чіткого і несуперечного плану роботи.

3) Формулювання прикладів поведінки системи

Серед загальноприйнятої документації нема окремого документа, який би описував конкретно приклади роботи системи. Частково цю функцію виконують специфікації, проте через відсутність конкретного готового програмного продукту вони часто мають значний рівень абстракції, який може нівелюватись використанням різноманітних моделей (мокапи, лендінги тощо). Іншим абсолютом цього документу є поле «Очікувані результати» в тестовому сценарії, який описує конкретний результат кожної дії вчиненої над готовим програмним продуктом.

Саме до другого пункту тяжіє TLD. Після того як команда розробки імплементувала функціонал, описаний в специфікації, лише тоді команда тестувальників приступає до створення тесту. І саме в процесі написання сценарію тестувальник орієнтується на очікувану поведінку згідно специфікацій і реальну реакцію системи на певні дії. Таким чином приклад поведінки системи з'являється вже після того як система отримала приріст функціоналу.

Натомість BDD на основі специфікацій, які буквально являються і тестовими сценаріями, створює необхідні конкретні приклади роботи системи. Це

відбувається через те, що сценарії, написані відповідно до стандартів мови Gherkin, потребують чіткого визначення кожного наступного кроку і не дають простору для абстракції. Тому створення прикладів функціонування системи відбувається до безпосереднього впровадження необхідного функціоналу.

4) Залученість команди до створення тестових сценаріїв

Традиційно склалось, що створення тестів між собою розділяють команди розробників та тестувальників. На плечі перших зазвичай звалюються обов'язки по модульному тестуванню, яке по суті є інструментом базового забезпечення якості коду. Другі ж відповідають за інші функціональні та не функціональні тести, а також потенційно за їх подальшу автоматизацію.

TLD чітко притримується подібного розподілу ролей і не заохочує до регулярної взаємодії між командами. Відповідно до цього підходу кожна команда повинна займатись виконанням своїх обов'язків і не втручатись в справи інших. Те саме стосується аналітиків та інших представників бізнесу з боку компанії чи замовника.

BDD ж все сприймає діаметрально протилежно. Оскільки специфікації одночасно являються і тестовими сценаріями, то по факту після проведення певного часу за навчанням норм мови Gherkin, до створення тестових сценаріїв може долучитись практично будь-який член команди незалежно від його ролі та знань в області тестування. На практиці це частіше за все стосується представників бізнесу

5) Автоматизація тестування

Автоматизація тестових сценаріїв зазвичай являється доволі спеціалізованою сферою тестування, яка являється прерогативою спеціалістів-автоматизаторів і яку важко опанувати ручному тестувальнику. Ключовим моментом тут являється необхідність знань мов програмування вкупі з навичками роботи з відповідними інструментами автоматизації.

З точки зору TLD автоматизоване тестування повинно провадитись насамкінець, після імплементації функціоналу та проведення ручного тестування. Таким чином автоматизатори отримають підтвердження від ручних тестувальників, що даний функціонал працює і готовий до автоматизації, що по суті своїй є пережитком водоспадної моделі розробки і тягне за собою відповідні проблеми характерні для цієї моделі.

Натомість BDD проголошує те, що автоматизація тестування має відбуватись паралельно з ручним тестуванням. Для цього на етапі формування специфікації наявні представники автоматизаторів та ручних тестувальників мають заздалегідь виділить тести, які можуть і які не можуть бути автоматизовані.

Таким чином вони страхують одне одного в плані проведення тестових запусків і вберігають від повторного виконання задач.

6) Зв’язок між імплементацією функціоналу та покриття його тестами

Згідно стандартів водоспадної моделі існує чітка послідовність, яка показує, що покриття певного функціоналу тестами не може бути зроблене до того, як цей функціонал буде заімплементовано. Це спричинено тим, що за такої моделі вихідні дані попереднього етапу являються строго вхідними для етапу наступного.

Саме тому TLD теж свідчить, що повноцінне покриття функціоналу тестами можливе тільки після його безпосереднього впровадження. Попри те, що такого строго взаємозв’язку як у класичній моделі даний підхід не має, але все одно вважається, що до створення відповідної частини програмного продукту тестування не може вийти за рамки тестування вимог чи створення тестової документації на конкретний спринт.

В свою чергу BDD завдяки універсальності специфікацій дозволяє створити значний відсоток покриття тестами ще до впровадження самого функціоналу. Окрім того варто згадати, що дана методологія була створена на основі методології Test Driven Development, а отже потенційно передбачає розробку через тестування (попри те, що в умовах сучасних проектів повноцінна розробка через тестування має місце нечасто). Таким чином є можливість паралельного створення автоматизованих тестів і розробки програмного продукту.

Це може виражатись у поділі тестових сценаріїв на менші тестові кроки, кожен з яких по суті являється модульним тестом, який можуть використовувати розробники в процесі створення коду. Таким чином у них зникне необхідність витрачати час на створення модульних тестів, які будуть інтегровані в загальний фреймворк автоматизованих тестів.

3.3 Дослідження ефективності розробленого підходу на основі засад BDD в умовах комерційного проекту

Тепер, сформулювавши конкретні характеристики кожного з етапів тестування відповідно до методології BDD, можна провести повноцінний аналіз того наскільки реальний процес тестування, який називають BDD, дійсно являється таким. Для цього необхідно дати опис процесу відповідно до вищеназваних характеристик.

В першу чергу необхідно зазначити, що вся діяльність описаного стріма відбувається згідно Scrum зі спринтами довжиною в три тижні. Також на проекті

практикується інкрементальний підхід до розробки програмного продукту. Це означає, що окрім спринтів існують періоди приросту функціоналу, протягом якого створюється набір нововведень та виправлень, які будуть опубліковані з новим релізом. Зазвичай період приросту функціоналу складає п’ять спринтів або ж п’ятнадцять тижнів між релізами.

Під час аналізу процесу тестування за основу буде взято саме діяльність протягом спринта, адже різниця порівняно з періодом приросту функціоналу виражається лише в більших проміжках часу та більших обсягах завдань, що виливається в циклічність описаного нижче процесу.

Таким чином першим етапом тестування є планування. Як правило, це один або декілька мітингів, протягом яких в повному складі збираються функціональні підрозділи, а також представники замовника та бізнес-аналітики. Під час подібних зборів провадиться повний аналіз функціоналу, який замовник прагне імплементувати протягом спринта. На основі цієї інформації формуються специфікації на майбутній спринт, а також відбувається поділ на менші підзавдання, на основі яких команди проводять оцінку ресурсних затрат на їх провадження і доводять її до відому інших підрозділів. Таким чином залежно від озвучених оцінок в плани на спринт вносяться корективи.

Після розробки завдань для всіх команд команда розробників береться за імплементацію нового функціоналу. В цей час команда тестувальників, як правило, займається переглядом та виправленням проблем серед автоматизованих та ручних тестів, які вже були створені раніше, але з точки зору приросту функціоналу ніяких робіт зазвичай на цьому етапі не провадиться.

Приблизно через тиждень після початку спринта на оточенні команди розробників з’являється чорнова версія нового функціоналу. На основі створених специфікацій, а також готового програмного продукту починається створення та провадження ручних тестів, які покривали б цей функціонал. Саме тут знаходиться та виправляється більша частина початкових дефектів, щоб в подальшому реліз на тестовому оточенні мав менше проблем. На деяких стрімах також практикують використання третього передрелізного оточення, куди випускаються зміни, схвалені замовником для подальшого випуску в продакшен з новим релізом.

Чим же весь цей час займається команда автоматизації? Зазвичай імплементація автотестів трохи відстає від основного робочого процесу. Це відбувається через те, що список тестових сценаріїв, які можна автоматизувати потрапляє до рук команди автоматизації тільки після проведення ручного тестування. Тому значну частину часу автоматизатори займаються рефакторингом

існуючих автоматизованих тестів, а також відповідають за проведення щотижневих «прогонів чистоти». Ці запуски проводяться наприкінці кожного тижня і являються по суті регресійним тестуванням, перевіряючи роботу системи загалом після внесених в неї змін. Після виконання цих прогонів провадиться аналіз автоматизованих тестів, які не пройшли. У випадку дефекту самого тесту випускається хотфікс, якщо ж проблема полягає в самій системі, то інформація про це або передається ручним тестувальникам, або ж створюється відповідний баг-репорт.

Наприкінці кожного стріму формуються звіти про діяльність всіх функціональних підрозділів, а також провадяться мітинги, де обговорюються результати діяльності і вносяться корективи в плани у зв'язку з тими труднощами, з якими стикнулись ті чи інші команди. Для команди тестувальників в звітах зазвичай відзначається кількість знайдених дефектів, а також кількість створених тестових сценаріїв (ручних чи автоматизованих відповідно). І саме кількість імплементованих тестових сценаріїв протягом одного спринта будемо в подальшому вважати показником ефективності роботи команди тестування, адже кількість знайдених дефектів залежить не лише від роботи тестувальників, а й від якості роботи розробників, а також складності новоствореного функціоналу. Проте зараз необхідно визначити чи відповідає описаний вище процес тестування програмного забезпечення методології BDD.

Порівняльна характеристика підходу TLD, методології BDD, а також реального стану справ на проекті наведена в табл. 3.1

Таблиця 3.1

Порівняльна таблиця характеристик проекту з TLD і BDD

Назва характеристики	Опис характеристики для підходу TLD	Опис характеристики для методології BDD	Опис характеристики на проекті
1	2	3	4
Робота з вимогами	Виконується представниками замовника або бізнес-аналітиком	Не виділяється	Виконується представниками замовника
Розробка специфікації	Виконується бізнес-аналітиком	Виконується спільно представниками всіх функціональних підрозділів	Виконується представниками функціональних підрозділів в рамках одного стріма

1	2	3	4
Приклади поведінки системи	Формується після імплементації необхідного функціоналу	Описується під час розробки специфікацій	Формується після імплементації функціоналу
Створення тестових сценаріїв	Виконується тестувальниками	Залучаються також представники бізнесу і розробки	Виконується тестувальниками
Автоматизація тестування	Виконується після імплементації функціоналу і проведення ручного тестування	Виконується паралельно ручному тестуванню	Виконується після імплементації функціоналу і проведення ручного тестування
Зв’язок тестів і функціоналу	Тестове покриття формується після імплементації функціоналу	Тестове покриття формується одночасно з імплементацією функціоналу і може слугувати в якості модульного тестування під час розробки	Тестове покриття формується після імплементації функціоналу

Таким чином розглянувши отримані результати дослідження можна прийти до висновку, що процес тестування на даному проекті не підпадає під заявлене використання методології Behaviour Driven Development.

Провівши дослідження відповідності реального процесу тестування до вимог BDD і виявивши значні розбіжності між очікуваним та реальним результатом з’явилась необхідність додатково визначити потенційні проблемні моменти та «вузькі місця» методології BDD, які не дозволяють проектним командам правильним чином впровадити її та скористатись усіма перевагами даної методології.

Очевидно, що дослідження діяльності в рамках одного стріма і одної функціональної команди є недостатньо для отримання валідних даних. Тому серед учасників інших стрімів було проведено опитання стосовно того, як в їх командах було впроваджено процеси відповідно до Behaviour Driven Development. В опитуванні взяло участь сорок дві особи з шести різних стрімів, на яких заявлено використання методології BDD. На основі отриманих відповідей було отримано висновок, що жоден із стрімів не дотримується вимог даної методології.

Повний список відповідності процесів тестування різних стрімів до вимог методології BDD представлено в табл. 3.2

Таблиця 3.2

Таблиця відповідності процесів тестування стрімів до вимог BDD

Назва	Стрім 1	Стрім 2	Стрім 3	Стрім 4	Стрім 5	Стрім 6
Робота з вимогами	+	+	-	+	+	+
Розробка специфікації	+	+	-	+	+	+
Приклади поведінки системи	-	-	-	-	-	-
Створення тестових сценаріїв	-	-	+	-	-	+
Автоматизація тестування	-	-	-	-	-	+
Зв’язок тестів і функціоналу	-	-	-	-	-	-

Таким чином можна зробити висновок, що в більшості випадків процес тестування відповідає BDD лише на етапі створення документації, залучаючи до цього процесу не лише представників замовника та аналітиків, а й безпосередньо учасників (або окремих представників) функціональних підрозділів.

Проте варто відмітити, що виконання даних вимог є характерним не лише для BDD, а й для усіх методологій, які будуються на основі Agile-маніфесту, адже одним із основних постулатів останнього є активне спілкування і взаємодія між підрозділами всередині проектної команди. Тому для констатування факту використання BDD виконання лише перших двох вимог не є достатнім.

Також на основі вищезгаданого опитування вдалось систематизувати інформацію про процеси на стрімах і виділити ряд основних проблем та недоліків, які не дозволяють імплементувати методологію в умовах реального процесу.

Серед основних проблем варто виділити:

- Надмірний зворотній зв’язок
- Нестача спеціалізованих знань у команди тестувальників
- Мінімальна бізнес-логіка
- Надмірна кількість зустрічей
- Розподілена команда
- Завищені очікування
- Сприйняття BDD виключно як методологію тестування

Тепер розглянемо окремо кожну з цих проблем і варіанти змін, які необхідно внести в процес тестування за методологією BDD, щоб мінімізувати ризики і уникнути цих складнощей.

На перший погляд може здатись, що велика кількість зворотного зв'язку – безумовно піде на користь проекту, проте на ділі все виявляється важче. Так, наприклад, якщо затвердження від бізнесу необхідно для початку нової користувацької історії чи внесення змін в існуючі тестові сценарії, то зворотній зв'язок займатиме значну кількість часу, що в свою чергу буде гальмувати процес тестування і як наслідок знижувати ефективність роботи. Тому важливо всі аспекти зворотного зв'язку заздалегідь обговорити на етапах планування і за можливості надати функціональним підрозділам якнайбільше автономності.

Нестача спеціалізованих знань у команди тестувальників також може стати значною проблемою до покращення ефективності роботи. Зазвичай подібні складнощі виникають в тих випадках, коли BDD сприймається у якості дешевої альтернативи повноцінній автоматизацій через те, що є можливість автоматизувати тестування за допомогою готовий програмних рішень. В такому випадку керівництво проекту залучає ручних тестувальників до розробки фреймворку і створення автоматизованих тестів, що виходить за межі їх компетенцій. Таким чином у випадку найму додаткових ручних тестувальників кошти замовника будуть використовуватись нераціонально, а якщо створення автоматизованих тестів буде покладено на плечі вже існуючих тестувальників, то це призведе лиш до збільшення навантаження і відчутного зниження ефективності за рахунок необхідності самонавчання новим сферам діяльності і робочого використання незнайомих інструментів.

Оскільки BDD в суті своїй бере за основу поведінку за якою ховається певна бізнес-логіка, то нема сенсу використовувати цю методологію для продуктів, де подібної логіки нема, адже в такому випадку будуть значні проблеми зі створенням специфікацій та описом подальших тестових сценаріїв, якщо основна складність проекту полягає в технічних аспектах, а не в розумінні та спілкуванні.

В світі, де особливо популярні Scrum і Kanban, надмірна кількість різноманітних зустрічей стає поширеною проблемою. В рамках BDD це питання може стати ще більш гострим, адже розробка документації, а також її подальше оновлення та затвердження може займати багато часу. В таких випадках є загроза, що замість реальної роботи тестувальники будуть просто сидіти більшу частину робочого дня на ділових зустрічах, де від них користі буде небагато. Тому доволі логічним рішенням є делегувати одного представника, який би провадив

комунікації між підрозділами та з представниками замовника. Найбільш логічним було б на цю роль обирати ліда команди.

Інша проблема сучасної IT-сфери – це розподілені команди. Очевидно, що неможливість швидкого контакту між учасниками є значною перепоною на шляху до якісної комунікації, а отже і знижує ефективність роботи. І ситуація не стає критичною, якщо різниця у часі становить одну-дві години, проте коли співпраця йде між людьми з різних півкуль, то виникають проблеми, які для методології побудованій на комунікації є майже фатальними. Саме тому найкращим рішенням було б формувати функціональні підрозділи в рамках одного-двох часових поясів, натомість у разі відсутності альтернатив виділити окремий час для спільної роботи, коли б комунікація була максимально доступною.

Часто побутує думка серед неспеціалістів, що використання BDD може стати панацеєю від усіх проблем проекту, а потім реальність стає гірким розчаруванням. І хоч дійсно дана методологія є сильним інструментом оптимізації процесу тестування, проте вона потребує змін на всіх стадіях даного процесу і життєвого циклу програмного продукту. Тому дуже важливо донести до менеджерського складу і представників бізнесу, що перед тим як пожинати плоди використання цієї методології необхідно в першу чергу закласти міцний теоретичний та практичний базис в повсякденній діяльності функціональних підрозділів.

І останнім, але не за значимістю, є той факт, що Behaviour Driven Development навіть попри те, що в даному дослідженні розглядається з точки зору процесу тестування не являється виключно методологією тестування. Оскільки вона впливає на всіх учасників проекту як з боку замовника, так і з боку функціональних команд, то в свою чергу і відповідальність за забезпечення якості кінцевого програмного продукту лягає не лише на тестувальників, але й на всю команду загалом. BDD здатна приносити користь лише у випадку залучення всієї проектної команди. Написання і використання сценаріїв мовою бізнес-логіки буде ефективним лише тоді, коли і тестувальники, і розробники, і представники бізнесу користуються цими сценаріями. Натомість, якщо весь процес тестування провадиться виключно ручними тестувальниками і автоматизаторами, то використання даної методології просто перетвориться на додаткову роботу з переписування тестових сценаріїв без особливої потреби.

Проаналізувавши класичний підхід до організації процесу тестування відповідно до методології BDD та виявивши проблемні моменті, які перешкоджають ефективному впровадженню засобів цієї методології на практиці необхідно перед проведення безпосереднього вивчення впливу методології

сформулювати нове бачення процесу тестування з урахуванням усіх вищезазначених даних.

Для того, щоб весь процес тестування став більш оптимізованим і філософія універсального тестувальника стала робочою необхідно працювати над взаємодією двох команд тестувальників, а також подумати про залучення до процесу менеджменту на щоденній основі.

Так в першу чергу варто чітко визначити першочерговий етап підготовки проекту до переходу на новий процес. Протягом цього етапу поступово усі члени функціональних підрозділів мають хоча б на базовому рівні ознайомитись з робочим процесом, який в подальшому буде імплементовано. Також на цьому етапі варто чітко продумати всі джерела комунікації між бізнесом та робочими командами, які б дозволяли нівелювати всі особисті, часові та технічні проблеми в процесі спілкування. Протягом цього періоду кожен новий спринт має все більше застосовувати засади BDD. Приклад оптимального процесу тестування протягом одного спринта описано нижче.

Так в першу чергу необхідно поставити фреймворк автоматизованого тестування в якості окремого і основного програмного забезпечення в процесі тестування, з яким мають взаємодіяти всі члени QA команди. Таким чином він слугував би своєрідним інформаційним хабом, де знаходяться вхідні дані і куди заносяться вихідні дані діяльності кожного функціонального підрозділу в рамках стріма.

Після отримання вхідних даних на спринт у вигляді вимог до функціоналу, який має бути імплементовано, на початку нового спринта або наприкінці попереднього має відбутись зустріч представників функціональних команд (бажано лідів або найбільш досвідчених учасників) і бізнесу. Під час цієї зустрічі має бути складено план роботи, вимоги мають бути описані за допомогою мови Gherkin і описані в специфікаціях, які далі будуть поміщені в спільний інформаційний хаб. Також на основі специфікацій командою тестувальників має бути створено список тестових сценаріїв, які слугуватимуть для розробників джерелом імплементації та, за можливості, модульного тестування.

Оновлені тестові сценарії написані мовою Gherkin тепер не вимагають від автоматизаторів витрачати час на переписування відповідно до вимог мови, а отже вони мають змогу раніше приступити до імплементації безпосередньої логіки кожного кроку тестового сценарію. Цей процес може початись як і паралельно процесу розробки нового функціоналу (якщо наявного функціоналу достатньо для створення заготовки кроків тестового сценарію), так і після появи на оточенні розробників нової версії програмного продукту.

Натомість поки провадиться процес автоматизації ручні тестувальники мають змогу по цим же тестовим сценаріям проводити перевірки функціоналу вручну, або навіть за необхідності запускати автоматизовані перевірки імплементованих раніше тестових сценаріїв чи наборів і відразу після цього засобами фреймворку отримати згенерований звіт про тестовий запуск, який можна передати менеджменту.

Таким чином на момент завершення спринту ручні тестувальники зможуть з більшою вірогідністю провадити перевірку всіх попередньо створених тестових сценаріїв, автоматизатори тепер не мають переписувати сценарії заново для фреймворку автоматизованих тестів, а можуть просто додати програмний код до вже існуючих кроків.

Все вищесказане вкупі з налагодженою комунікацією з представниками замовника, а також загальним доступом всієї проектної команди до інформаційного хаба, в теорії повинно значно покращити як взаємодію між командами так і загальну продуктивність роботи стріма чи проекта. Проте чи на практиці подібний підхід приносить свої дивіденди?

В рамках переддипломної практики було проведено дослідження згідно якого команда тестувальників, яка складалась з п’яти осіб (три ручних тестувальника та два автоматизатора), протягом одного спринта провадила процес тестування максимально відповідним до методології BDD чином. Після трьох тиждень проведення експерименту було порівняно кількість тестів створених кожним учасником команди протягом контрольного спринта і спринта за методологією BDD.

Варто зазначити, що зміни в процесі тестування не стосувались створення повноцінного хаба на основі фреймворку автоматизованого тестування, який був би доступний представникам бізнесу і команди розробників. Окрім того документація розробників не підпадала під стандарти мови Gherkin і була описана більш класичним чином. І також до створення тестових сценаріїв не вдалось залучити представників бізнесу. Тому значною мірою експеримент стосувався взаємодіє підрозділів всередині команди тестування. Дані ефективності кожного члена кожного підрозділу протягом обох спринтів наведено в таблицях 3.3 і 3.4. При зборі результатів експерименту, як і було зазначено раніше використовувалась кількість створених тестів протягом спринту, розділених по групам A, B і C відповідно до рівню складності та оцінки затрат людських ресурсів на створення та проходження відповідного тестового сценарію в ручному чи автоматизованому вигляді (залежно від підрозділу).

Також варто зазначити, що в таблицях 3.3 і 3.4 тестом класу A вважається тест, написання якого оцінюється в вісім годин, тестом класу B вважається тест, написання якого оцінюється в чотири години і тестом класу C вважається тест, написання якого оцінюється в дві години. Також пріоритетом задач можна знехтувати.

Таблиця 3.3

Кількість тестів створених кожним членом команди протягом контрольного спринта

Член команди	Кількість завдань класу A	Кількість завдань класу B	Кількість завдань класу C
Ручний тестувальник 1	4	8	11
Ручний тестувальник 2	1	7	4
Ручний тестувальник 3	2	10	9
Автоматизатор 1	2	6	18
Автоматизатор 2	3	7	17

Таблиця 3.4

Кількість тестів створених кожним членом команди протягом експериментального спринта

Член команди	Кількість завдань класу A	Кількість завдань класу B	Кількість завдань класу C
Ручний тестувальник 1	6	9	15
Ручний тестувальник 2	3	14	6
Ручний тестувальник 3	3	10	22
Автоматизатор 1	4	13	7
Автоматизатор 2	2	15	16

Так оскільки кожен тест має вартість в вісім, чотири чи дві години, які розраховувалось на виконання цього тесту, то можна порівняти яка була середня вартість тестів, написаних одним членом команди тестування. Дані про вартість тестів протягом двох спринтів приведені на рис 3.3.

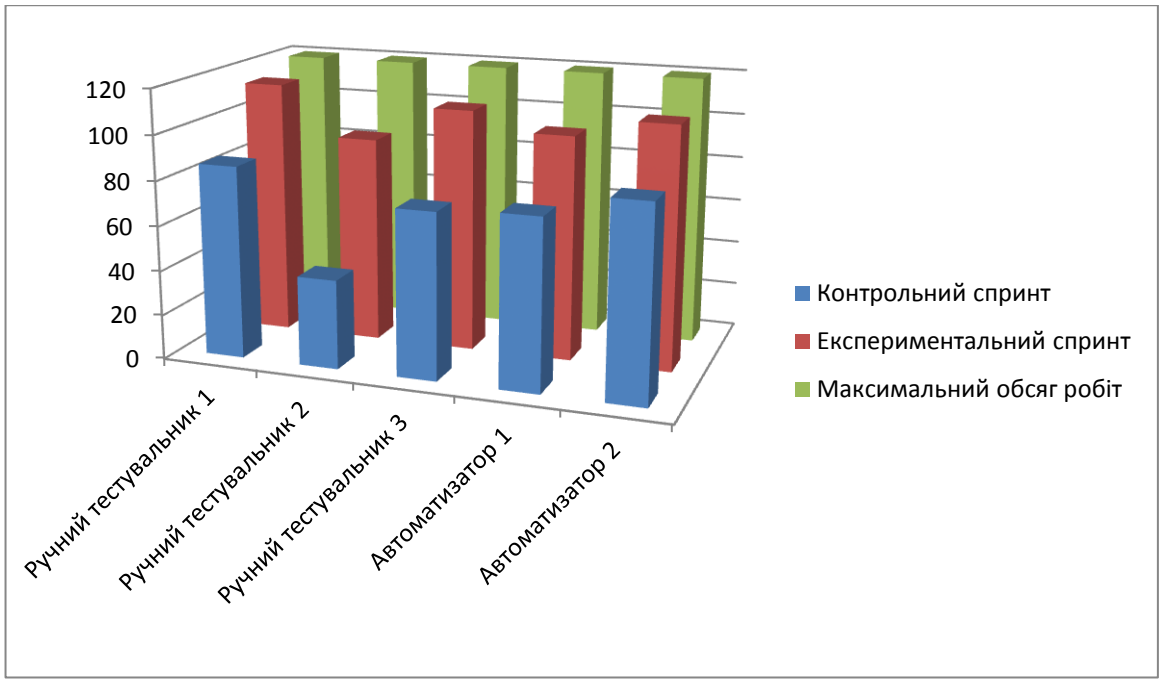

Рис. 3.3. Показники ефективності роботи протягом двох спринтів

Якщо взяти до уваги, що протягом обох спринтів обсяг взятої роботи на кожну людину складав 120 оціночних годин, то можна легко вирахувати ефективність роботи кожного із учасників команди. Для цього необхідно виразити у відсотках співвідношення виконаних робіт і всіх запланованих. Дані наведені в табл. 3.5

Таким чином порівнявши дані двох спринтів можна зробити висновок, що після впровадження в процес тестування засад методології BDD тільки на рівні команди тестування відбувся приріст ефективності кожного її члена, адже обсяг роботи виконаний протягом трьох тижнів в годинному еквіваленті зріс щонайменше на 15%, а щонайбільше – на 54%.

Тому справедливим буде припустити, що в середньому даний підхід може дати приріст ефективності від 25% до 50%.

Таблиця 3.5

Порівняння вартості виконаних тестів протягом двох спринтів, відсотків

Член команди	Відсоток завдань, виконаних протягом контрольного спринту	Відсоток завдань виконаних протягом експериментального спринту	Різниця між ефективністю протягом двох спринтів
1	2	3	4
Ручний тестувальник 1	72%	96%	+24

1	2	3	4
Ручний тестувальник 2	33%	83%	+54
Ручний тестувальник 3	62%	94%	+32
Автоматизатор 1	63%	91%	+28
Автоматизатор 2	72%	87%	+15

Таким чином неозброєним оком помітно, що під час експериментально спринту із застосуванням методології BDD кількість імплементованих тестів зросла. Проте для того, щоб говорити про ріст ефективності необхідно взяти більш однозначний параметр.

ВИСНОВКИ

Протягом переддипломної практики було проведено дослідження процесу ручного тестування, а також створення автоматизованого тестового фреймворку.

Даний проект існує вже більше десяти років і через це складається з великої кількості менших проектів, над кожним з яких працює окрема проектна команда, що називається стрімом. В рамках кожного стріму існують свої функціональні підрозділи, які відповідають за різні аспекти процесу створення програмного продукту (QA команда, Dev команда тощо). І команди всередині стріму, і стріми між собою підтримують обмежену комунікацію, яка часто провадиться за посередництва менеджменту. Також для кожного стріму є як мінімум одна відповідальна особа з боку замовника, яка контролює діяльність стріму загалом, або конкретних функціональних підрозділів.

На більшості стрімів вже впроваджено певний функціонал фреймворку автоматизованого тестування і під час нарощення функціоналу самого програмного продукту відбувається приріст автоматизованих тестів. Проте попри це більшу частину стандартної QA команди складають ручні тестувальники, а кількість автоматизаторів рідко перевищує двох-трьох осіб. Це відбувається через фокус уваги на короткострокові перспективи тестування протягом періоду приросту функціоналу, а також загальної філософії керівництва про тестувальника-універсала, який здатен провадити як ручне так і автоматизоване тестування.

Дана філософія є наслідком популярних нині Agile-підходів, які закликають до універсальності кожного члена команди. Проте з точки зору тестування це впровадити важче, адже автоматизатор являється по суті розробником специфічного програмного продукту всередині програмного продукту, а отже потребує знань в ряді додаткових областей з якими ручні тестувальники зазвичай стикаються рідко. Таким чином залучення ручних тестувальників в екстрених випадках є доволі проблематичним через доволі високий поріг входження.

Процес тестування на даному проекті провадиться методом чорного ящика і в більшості своїй підпадає під опис класичного підходу Test Last Development, за якого тестування відбувається вже після розробки. Проте команда автоматизаторів для створення фреймворку послуговується методологією Behaviour Driven Development, інструменти якої розглядаються в даному дослідженні.

Так вже після короткого проміжку часу стають очевидними переваги використання BDD в процесі створення автоматизованих тестів. Зокрема оскільки

при написанні тестових сценаріїв використовуються кроки, написані мовою Gherkin, а безпосередньо логіка кроків винесена в окремий модуль, то по ходу росту і розвитку фреймворку написання автоматизованих тестів все більше зводиться до використання ряду вже імплементованих кроків, які необхідно вибудувати у відповідному порядку.

Проте назвати даний результат задовільним важко, адже окрім нативності мови Gherkin, яка робить тестовий сценарій легшим для розуміння, даний підхід слабко відрізняється від ідеї використання класів-хелперів, де так само імплементується логіка кожного кроку, а вже потім кроки компілюються в цілий сценарій.

Таким чином ігнорується основна перевага, яку надає Behaviour Driven Development, а саме інтеграція процесів ручного та автоматизованого тестування. Так в оригінальному вигляді ручні тестувальники роблять тестові сценарії у відриві від потреб автоматизаторів, чим ускладнюють останнім роботу. Це виражається, наприклад, в необхідності по суті писати тестовий сценарій для автоматизації заново, адже сценарії, що використовуються ручними тестувальниками окрім того, що ніяк не інтегровані в фреймворк, так і просто непридатні для автоматизації в існуючому вигляді.

Для того, щоб весь процес тестування став більш оптимізованим і філософія універсального тестувальника стала робочою необхідно працювати над взаємодією двох команд тестувальників, а також подумати про залучення до процесу менеджменту на щоденній основі.

Так в першу чергу необхідно поставити фреймворк автоматизованого тестування в якості окремого і основного програмного забезпечення в процесі тестування, з яким мають взаємодіяти всі члени QA команди. Таким чином він слугував би своєрідним інформаційним хабом, де знаходяться вхідні дані і куди заносяться вихідні дані діяльності кожного функціонального підрозділу в рамках стріма.

Так, наприклад, менеджмент і представники замовника могли б подавати вимоги та специфікації не за допомогою інструментів на кшталт Google Spreadsheets, а завантажувати конкретні файли в систему фреймворка. Звідти бізнес-аналітики або ручні тестувальники (залежно від складну проектної команди) брали б звідти сирі дані і конвертували їх безпосередньо в тестові сценарії і тестові набори, які там же і зберігались проте вже в новому форматі і відповідно до вимог мови Gherkin.

Оновлені тестові сценарії тепер не вимагають від автоматизаторів витрачати час на переписування відповідно до вимог мови, а отже вони мають

змогу раніше приступити до імплементації безпосередньої логіки кожного кроку тестового сценарію.

Натомість поки провадиться процес автоматизації ручні тестувальники мають змогу по цим же тестовим сценаріям проводити перевірки функціоналу вручну, або навіть за необхідності запускати автоматизовані перевірки імплементованих тестових сценаріїв чи наборів і відразу після цього засобами фреймворку отримати згенерований звіт про тестовий запуск, який можна передати менеджменту.

Спроба обмеженого впровадження даного підходу на невеликому колі тестувальників показала значний приріст ефективності кожного. Так протягом двох спринтів, на які припали терміни практики, ефективність роботи автоматизаторів зросла приблизно в 2,5 рази.

Таким чином можна зробити висновок, що запропонований підхід до організації та управління процесом тестування програмного забезпечення показав свою ефективність та може стати основою для подальших досліджень в цьому напрямку.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Бек, К. Шаблоны реализации корпоративных приложений / К. Бек. – М. : Вильямс, 2017

2. Блек Р. Ключові процеси тестування / Р. Блек. - Лорі, 2006. - 544 с

3. Р. Калбертсон, К. Браун, Г. Кобб. Быстрое тестирование. - Вильямс, 2004. – 379 с

4. Канер, С. Тестирование программного обеспечения. Фундаментальные концепции менеджмента бизнес-приложений: Пер. с англ. / С. Канер, Д. Фолк, Е. К. Нгуен. – К. : ДиаСофт, 2001. – 544 с

5. Куликов С.. Тестирование програмного обеспечения. Базовый курс. - Минск: Четыре четверти, 2015.

6. Тампе, Л. Введение в тестирование программного обеспечения / Л. Тампе. – М. : Вильямс, 2003. – 368 с

7. Beck, K. 2000. Extreme Programming Explained. Addison Wesley.

8. Beck, K. 2003. Test-Driven Development: By Example. Addison-Wesley Professional.

9. Copeland, Lee (December 2001). "Extreme Programming". Computerworld. Archived from the original on August 27, 2011. Retrieved January 11, 2011.

10. Copland. L. Practitioner’s Guide to Software Test Design. - London: STQE Publishing, 2004.

11. Duvall Paul, Matyas Stephen M., Glover Andrew. Continuous Integration: Improving Software Quality and Reducing Risk (The Addison-Wesley Signature Series). — Addison-Wesley Professional, 2007. — ISBN: 0321336380.

12. Erdogmus H., M. Morisio, M. Torchiano, On the effectiveness of the test-first approach to programming, IEEE Transactions on Software Engineering 31 (3) (2005) 226–237.

13. Faught, Danny R. (November 2004). "Keyword-Driven Testing". Sticky Minds. Software Quality Engineering. Retrieved September 12, 2012.

14. Feathers, M. Working Effectively with Legacy Code, Prentice Hall, 2004

15. Ferguson, John Smart (2014). BDD in Action: Behavior-Driven Development for the Whole Software Lifecycle. Manning Publications. ISBN 9781617291654.

16. Freeman Elisabeth, Freeman Eric, Bates B., Sierra K. Head First Design Patterns — O’ Reilly & Associates, Inc., 2004. — ISBN: 0596007124

17. Garousi, Vahid; Mäntylä, Mika V. (2016-08-01). "When and what to automate in software testing? A multi-vocal literature review". Information and Software Technology. 76: 92–117. doi:10.1016/j.infsof.2016.04.015

18. George B., Williams L., A structured experiment on test-driven development, Information and Software Technology 46 (2004) 337–342.

19. Glenford Myers J., The Art of Software Testing. Revised and Updated by Tom Badgett, Todd M.Thomas, Corey Sandler. - 2nd ed. - Hoboken, New Jersey.: John Wiley & Sons, Inc., 2004

20. Guide to the Software Engineering Body of Knowledge [Текст] : SWEBOK, Version 3.0, IEEE; введ. 20.12.13

21. Haring, Ronald (February 2011). de Ruiter, Robert (ed.). "Behavior Driven development: Beter dan Test Driven Development". Java Magazine (in Dutch). Veen Magazines (1): 14–17. ISSN 1571-6236.

22. Hayes, Linda G.. Automated Testing Handbook. – Software Testing Inst; 2nd edition, 2004. – 182 p

23. Petersen, Kai, Claes, Wohlin: Context in industrial software engineering research. ESEM 2009: 401-404.

24. Keogh, Liz (2009-09-07). "Introduction to Behavior-Driven Development". SkillsMatter.

25. Keogh, Liz (June 27, 2011). "ATDD vs. BDD, and a potted history of some related stuff".

26. Kolawa, Adam; Huizinga, Dorota (2007). Automated Defect Prevention: Best Practices in Software Management. Wiley-IEEE Computer Society Press. p. 74. ISBN 978-0-470-04212-0.

27. Koskela, L. "Test Driven: TDD and Acceptance TDD for Java Developers", Manning Publications, 2007

28. Lübke, Daniel; van Lessen, Tammo (2016). "Modeling Test Cases in BPMN for Behavior-Driven Development". IEEE Software. 33 (5): 15–21. doi:10.1109/MS.2016.117

29. Mandurrino, José L. (July 2014). "Gestione e approccio alla validazione in sistemi RT (Real-Time)"

30. Marick, Brian. "When Should a Test Be Automated?". StickyMinds.com.

31. M.M. Mueller, O. Hagner, Experiment about test-first programming, IEE Proceedings – Software 149 (5) (2002) 131– 136.

32. Newkirk, JW and Vorontsov, AA. Test-Driven Development in Microsoft .NET, Microsoft Press, 2004.

33. Osherove Roy. The Art of Unit Testing: With Examples in .Net. — 1st edition. — Greenwich, CT, USA : Manning Publications Co., 2009. — ISBN: 1933988274, 9781933988276

34. O'Connor, Rory V.; Akkaya, Mariye Umay; Kemaneci, Kerem; Yilmaz, Murat; Poth, Alexander; Messnarz, Richard (2015-10-15). Systems, Software and Services Process Improvement: 22nd European Conference, EuroSPI 2015, Ankara, Turkey, September 30 -- October 2, 2015. Proceedings. Springer. ISBN 978-3-319-24647-5.

35. Solis, Carlos; Wang, Xiaofeng (2011). "A Study of the Characteristics of Behaviour Driven Development". Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on: 383–387. doi:10.1109/SEAA.2011.76. hdl:10344/1256. ISBN 978-1-4577-1027-8.

36. Tharayil, Ranjith (15 February 2016). "Behavior-Driven Development: Simplifying the Complex Problem Space". SolutionsIQ

37. B. Turhan, A. Bener, P. Kuvaya, M. Oivo, A quantitative comparison of test-first and test-last code in an industrial project, in: A. Silliti et al. (Eds.), XP 2010, LNBIP 48, Springer-Verlag, Berlin, 2010, pp. 232–237.

38. Pettichord Bret, "Seven Steps to Test Automation Success" URL: http://www.testpoint.com.au/attachments/093_Seven%20Steps%20to%20Test%20Automation%20Success.pdf

39. Problems with Test Automation and Modern QA URL: https://devqa.io/problems-test-automation-modern-qa/

40. Software Testing Life Cycle URL: https://softwaretestingfundamentals.com/software-testing-life-cycle

41. Автоматизация тестирования: выбор инструмента. // OpenQuality.ru. Качество програмного обеспечения. - http://blog.openquality.ru/tool-choice

42. Введение в BDD. http://agilerussia.ru/practices/introducing-bdd/

43. Журнал «Форбс»: https://www.forbes.ru/milliardery-photogallery/425297-20-bogateyshih-lyudey-mira-2021-reyting-forbes

44. Метрики в тестировании — URL: http://blog.shumoos.com/archives/271.

45. Пирамида автоматизации и другие геометрические фигуры // AT.Info http://automated-testing.info/t/piramida-avtomatizacz..,

46. Портал Про Тестинг – Тестирование Программного Обеспечения http://www.protesting.ru.

47. Тестирование. Фундаментальная теория URL: https://dou.ua/forums/topic/13389/

48. Этапы тестирования ПО: https://training.qatestlab.com/blog/technical-articles/software-testing-stages

49. Graham Dorothy. — ROI of test automation:benefit and cost, 2010. — URL: http://www.dorothygraham.co.uk/downloads/generalPdfs/ProfTesterROI.pdf

50. QA Automation, часть 2: Автотестирование – Начало.URL: http://agilerussia.ru/articles/qa-automation-part-2-autotesting1/

51. ltd Cucumber. — Cucumber. Simple, human collaboration. — URL: https://cucumber.io

52. Modern Test Case Management Software for QA and Development Teams. — URL: http://www.gurock.com/testrail/

53. Test Design Considerations // Selenium HQ. Browser automation. https://www.selenium.dev/documentation/en

54. TIOBE Index [Электронный ресурс]. https://tiobe.com/tiobe-index/