Вінницький національний технічний університет
Факультет інформаційних технологій та комп’ютерної інженерії
Кафедра комп’ютерних наук

Пояснювальна записка

до магістерської кваліфікаційної роботи

на тему «Інформаційна технологія автоматизованого тестування»

Виконав: студент 2 курсу, групи 2КН-20 м
спеціальності 122 «Комп'ютерні науки»
Василевський В. О.

Керівник: завідувач кафедри КН, професор,
д.т.н. Яровий А.А

Рецензент: канд. техн. наук, доцент каф.
ПЗ Бабюк Н. П.

Вінниця
2021

ЗАТВЕРДЖУЮ
Завідувач кафедри КН
д. т. н., проф. Яровий А. А.
(наук. ст., вч. зв., ініц. та прізв.) (підпис)
"_____"_____2021 р.

ЗАВДАННЯ

на магістерську кваліфікаційну роботу на здобуття кваліфікації магістра наук зі спеціальності: 122 «Комп'ютерні науки»
(шифр – назва спеціальності)

08-22.МКР.022.20.000.ПЗ

Магістранта групи 2КН-20м Василевського Володимира Олеговича
(назва групи) (прізвище, ім'я і по батькові)

Тема магістерської кваліфікаційної роботи: «Інформаційна технологія автоматизованого тестування»

Вхідні дані: мінімальна кількість динамічних аналізаторів тестових методів – 50, кількість інтерфейсів запуску процесу тестування – не менше 2, час на опрацювання алгоритму знаходження тестових методів в кількості 100 одиниць не більше 4 секунди, мінімальна кількість вхідних параметрів – 25, розмір тестових проєктів, що підтримує технологія – не менше 10 проєктів, що містять по 500 тестових методів.

Короткий зміст частин магістерської кваліфікаційної роботи

1. Графічна: демонстрація аналогів, структурна схема компонентів інформаційної технології, схема алгоритму пошуку тестових методів, приклади вихідного коду розробленої технології, результати тестування.
2. Текстова (пояснювальна записка): вступ, аналіз сучасного рівня розвитку інформаційної технології автоматизованого тестування, моделювання інформаційної технології автоматизованого тестування, програмна реалізація інформаційної технології автоматизованого тестування, економічна частина, висновки, перелік використаних джерел, додатки.

КАЛЕНДАРНИЙ ПЛАН ВИКОНАННЯ МКР

№ етапу	Назва етапу	Термін виконання		Очікувані результати
		початок	кінець	
1	Аналіз сучасного рівня розвитку інформаційних технологій автоматизованого тестування. Постановка задач дослідження			Розділ 1
2	Моделювання інформаційної технології автоматизованого тестування			Розділ 2
3	Програмна реалізація інформаційної технології автоматизованого тестування			Розділ 3
4	Підготовка економічної частини			Розділ 4
5	Апробація результатів дослідження			Тези доповідей
6	Оформлення пояснювальної записки, графічного матеріалу та презентації			Пояснювальна записка, графічний матеріал, презентація

Консультанти з окремих розділів магістерської кваліфікаційної роботи

1. Науковий керівник _____ д. т. н., проф., зав. кафедри КН
(підпис) наук. ступінь, вчене звання (посада)

"___" _____ 2021р А.А. Яровий
ініціали та прізвище

2. Економічна частина _____ к. ек. н., проф., зав. кафедри ЕПВМ
(підпис) наук. ступінь, вчене звання (посада)

"___" _____ 2021р М. І. Небава
ініціали та прізвище

Дата попереднього захисту роботи "___" _____ 2021р

3. Рецензент _____ к. т. н. доцент кафедри ПЗ
(підпис) наук. ступінь, вчене звання (посада)

"___" _____ 2021р Н. П. Бабюк
ініціали та прізвище

4. Завдання видав науковий керівник _____ д. т. н., проф., зав. кафедри КН
(підпис) наук. ступінь, вчене звання (посада)

"___" _____ 2021р А.А. Яровий
ініціали та прізвище

5. Завдання отримав магістрант _____ В.О. Василевський
(підпис) ініціали та прізвище

"___" _____ 2021р

АНОТАЦІЯ

Магістерська кваліфікаційна робота присвячена розробці інформаційної технології автоматизованого тестування. Технологія забезпечує програмні засоби, що реалізовані з метою підвищення ефективності проведення процесу автоматизованого тестування.

У роботі досліджено предметну область автоматизованого тестування. Проаналізовано основні методи та засоби реалізації технології. Визначено основні складові модулі та побудовано оптимальну структурну схему технології для максимального забезпечення інформаційних потреб користувачів. Запропоновано інформаційну технологію автоматизованого тестування, що відрізняється від існуючих застосуванням удосконаленої інформаційної моделі динамічного аналізу тестових модулів, що забезпечило підвищення ефективності автоматизованого тестування.

Розроблено мультикомпонентне програмне забезпечення для реалізації та проведення автоматизованого тестування. Програмні засоби реалізовано з використанням інтегрованого середовища розробки Microsoft Visual Studio Professional 2019, мови програмування C# та програмної платформи .NET.

ABSTRACT

The master's thesis is devoted to the development of information technology for automated testing. The technology provides software tools implemented to improve the efficiency of the automated testing process.

The subject area of automated testing is investigated in the work. The main methods and features of technology implementation are analyzed. The main components have been identified and an optimal structural diagram of the technology was developed to satisfy the informational needs of users. The information technology of automated testing is proposed, which differs from the existing ones by the application of the improved information model of dynamic analysis of test modules, which provided an increase in the efficiency of automated testing.

Multicomponent software for implementation and conducting of automated testing has been developed. The software is implemented using the integrated development environment Microsoft Visual Studio Professional 2019, C# programming language and .NET software platform.

ЗМІСТ

ВСТУП

Актуальність теми дослідження. Незважаючи на покращення засобів реалізації програмного забезпечення, складність та комплексність кінцевих продуктів має неуклінну тенденцію збільшуватись щорічно. Відповідно ускладнюється процес підтримки, можливості внесення змін та тестування систем. Визначено, що лише економіка США загальні перевитрати пов’язані із недосконалостями програмного забезпечення склали близько 60 мільярдів доларів [1].

Інститут системних наук в IBM встановив, що вартість виправлення критичних проблем значно збільшується із кожним переходом на нову стадію розробки [2]. Вартість виправлення помилки, виявленої після випуску продукту, була в чотири-п'ять разів більшою, ніж та, що виявлена під час етапу розробки, і до 100 разів більше, ніж одна, виявлена на етапі проєктування програмного забезпечення.

Розробники, маючи за мету мінімізувати такі витрати, намагаються застосовувати різні види тестування. Залежно від системи використовуються комбінації різного виду ручного та автоматизованого тестування. Ручне виконується безпосередньо людьми. Автоматизоване охоплює в себе модульне (юніт-тестування), функціональне, навантажувальне та інші.

Сьогодні одним з найпоширеніших підходів у автоматизованому тестуванні є модульний, що базується на ідеї написання невеликих тестових методів, котрі реалізують перевірку невеликих ізольованих елементів системи (окремих функцій) на правильність виконання. Цей метод дає можливість визначати значну кількість помилок на ранніх стадіях розробки, і надалі значно прискорити процес регресійного тестування при внесенні змін в існуючий код, оскільки великі обсяги програмного коду тестуються автоматично.

Юніт-тестування або модульне тестування – метод тестування програмного забезпечення, що дозволяє перевірити на правильність окремі

модулі вихідного коду програми. Ідея полягає в написанні тестів для кожної функції або методу. Перевагою такого методу є ізоляція окремих частин додатку і підтвердження, що кожна окрема функція, метод та модуль коректний і здійснює очікувану поведінку [3].

Для реалізації модулів автоматизованого тестування розробникам необхідно використати програмну систему (фреймворк, бібліотеку) модульного тестування. Вони різняться за наявністю різноманітного функціоналу, методом організації та проведення тестування, тому важливо правильно вибрати ту бібліотеку, котра найповніше покриє усі вимоги до процесу автоматичного тестування.

Сучасні бібліотеки модульного тестування містять ряд допоміжних функцій, що дозволяють їх користувачам краще застосовувати засоби автоматизованого тестування. Проте, як правило системи аналізу та виконання бібліотек є незадовільними і не розкривають всю повноту статичного та динамічного опрацювання тестових модулів, а тому застосовуються для більш загальних випадків. Саме тому розробка власної інформаційної технології автоматизованого тестування з використанням засобів динамічного та статичного аналізу тестових модулів є актуальною.

Зв'язок роботи з науковими програмами, планами, темами. Магістерська кваліфікаційна робота здійснювалась відповідно плану виконання наукових досліджень на кафедрі комп’ютерних наук.

Мета і задачі дослідження. Основною метою є підвищення ефективності процесу модульного тестування програмного забезпечення.

Основними задачами є:

- Проаналізувати сучасний рівень розвитку інформаційних технологій автоматизованого тестування.
- Виконати моделювання інформаційної технології автоматизованого тестування.

- Реалізувати компоненти інформаційної технології автоматизованого тестування.
- Оцінити та спрогнозувати економічний потенціал розробки.

Об'єктом дослідження є процес автоматизованого тестування програмних засобів.

Предметом дослідження є програмні засоби автоматизованого модульного тестування.

Методи дослідження. У процесі досліджень використовувались: теорія алгоритмів для розробки алгоритмів виконання автоматичного тестування; комп'ютерне моделювання для аналізу та перевірки отриманих теоретичних положень, комбінаторний аналіз, лінійна алгебра.

Наукова новизна одержаних результатів.

– удосконалено інформаційну модель динамічного аналізу тестових модулів, що відрізняється від існуючих введенням додаткових аналізаторів тестових методів, що забезпечило підвищення ефективності процесу модульного тестування;

– запропоновано інформаційну технологію автоматизованого тестування, що відрізняється від існуючих застосуванням удосконаленої інформаційної моделі динамічного аналізу тестових модулів, що забезпечило підвищення ефективності автоматизованого тестування.

Практичне значення одержаних результатів Практична цінність одержаних результатів полягає в тому, що на основі отриманих в магістерській кваліфікаційній роботі теоретичних положень запропоновано інформаційну технологію та розроблено програмні засоби для підвищення ефективності процесу автоматизованого тестування програмного забезпечення, що досягається удосконаленням інформаційної моделі динамічного аналізу тестових методів та програмної реалізації мультикомпонентної технології модульного тестування.

Достовірність теоретичних положень магістерської кваліфікаційної роботи підтверджується строгістю постановки задач, коректним застосуванням методів під час доведення наукових положень, порівнянням результатів із відомими, та збіжністю результатів моделювання з результатами, що отримані під час впровадження розроблених програмних засобів.

Особистий внесок автора. Усі наукові результати, наведені у магістерській кваліфікаційній роботі, отримані автором самостійно. У друкованих працях, опублікованих у співавторстві, автору належать такі результати: аналіз структури компонентів інформаційної технології автоматизованого тестування, створення інтелектуальної системи автоматизованого тестування на основі фреймворку юніт-тестування, методи вирішення задачі надання доступу до програмних засобів інформаційної системи автоматизованого тестування.

Апробація результатів роботи. Результати досліджень опубліковано у L Науково-технічній конференції факультету інформаційних технологій та комп'ютерної інженерії (2021), секція комп’ютерних наук у м. Вінниці в березні 2021 р., V Міжнародній науково-практичній конференції «Modern directions of scientific research development» (2021), секція технічних наук у м. Чикаго в жовтні 2021 р., Молодь в науці: дослідження, проблеми, перспективи (МН-2022), секція Інформаційні технології та комп’ютерна інженерія у м. Вінниця в грудні 2021 р.

Публікації. За результатами магістерської кваліфікаційної роботи опубліковано три тези доповідей на науково-технічних конференціях [4-6]. За результатами досліджень прийнята до друку стаття у міжнародний науково-технічний журнал «Інформаційні технології та комп’ютерна інженерія».

1 АНАЛІЗ СУЧАСНОГО РІВНЯ РОЗВИТКУ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ

1.1 Аналіз предметної області автоматизованого тестування

Незалежно від розміру, цілі та складності, кожний програмний проєкт підлягає тестуванню, яке може бути мануальним, тобто продукт безпосередньо перевіряється людиною функціонально та візуально на правильність виконання або автоматичне. Перше ще має назву інтеграційне, але сучасні проєкти доволі великі та містять велику кількість залежностей. Інтеграційне тестування дозволяє лише перевірити коректність роботи декількох модулів в одній системі. Така процедура не може гарантувати, що система в цілому на всіх рівнях працює вірно, адже окрім можливих механічних помилок такі тести припускають, що все середовище вже коректно налаштоване та розвернуте на апаратному забезпеченні. Для багатьох програм, що включають в себе багато компонентів, на практиці не завжди зручно налаштовувати компоненти щоразу для проведення тестування.

Для подолання цих проблем та атомарно безпомилкового тестування необхідно переконатись, що компоненти системи працюють правильно кожен окремо. Модульне тестування, або юніт-тестування – процес програмування, який дозволяє ізольовано перевірити кожен окремий модуль вихідного коду. Тести пишуться для кожного базового компонента – функції або методу. Такий підхід дає можливість завчасно виявляти помилки, до яких призвів новий код, або зміни у вже існуючому, й в результаті усувати їх [7].

Модульне тестування має ряд переваг перед іншими методами тестування, через що в сучасності більшість розробників прагнуть покрити юніт-тестами якомога більшу частину своїх програмних проектів. Проте не на кожному проєкті можливе використання юніт-тестування. Серед основних причин відмовитись від модульного тестування в процесі розробки є [8]:

1. Необхідність тестування складного коду. Багато задач в

програмуванні мають комбінаторний характер, тобто вихідний результат може набувати різних значень. У випадку з булевою змінною це лише два варіанти, але за необхідності опрацювати дерево, або іншу складну структуру даних, варіантів кінцевого результату може бути незліченно. Кожен такий варіант необхідно опрацювати та перевірити за допомогою тесту, кожен з яких займає мінімум 3-5 стрічок коду. Виходячи з того, що на практиці неможливо передбачити усі можливі шляхи виконання програми, модульне тестування не дозволяє відловити усі помилки, окрім найпростіших випадків.

2. Очікуваний результат є лише приблизним. В модульному тестуванні кожен окремий тест повинен повертати твердження про відповідність очікуваного результату до отриманого. Це стає неможливим коли не можна завчасно знати який результат є очікуваним за певних вхідних даних, як, наприклад у математичному моделюванні, або за використання машинного навчання та добування даних.

3. Використання коду, що працює зі системою. Код, що потребує взаємодії з апаратними портами, таймерами або іншими нестабільними частинами системи дуже складно перевірити в ізольованому стані. В цьому випадку з’являється необхідність переходити до більш абстрактних засобів та імітувати використання апаратних та елементів системи, хоча і це не дає можливість повністю повторити часто невизначений стан системи [9].

4. Помилки в інтеграції та швидкості виконання. Модульне тестування не підходить для перевірки системи в цілому, оскільки відбувається тест кожного модулю окремо. Це означає, що помилки інтеграції, системного рівня, функцій, які виконуються в різних модулях, не будуть визначені. Окрім цього, юніт-тести не придатні для тестування швидкості виконання та загальної оптимізації системи.

5. При загальнонизькій культурі програмування. Для отримання максимальної користі від модульного тестування необхідно чітко слідкувати за технологією тестування протягом всієї розробки. Необхідно зберігати записи про всі проведенні тести та про всі зміни вихідного коду. В результаті,

якщо більш пізня версія програми не проходить конкретний тест, можна звернутись до записів і перевірити на якій версії цей тест проходився і які зміни були внесені, тобто виявити зміни, що привели до помилки та легко усунути її. Якщо ігнорувати ці вимоги, це призведе до накопичення інформації, яку вже неможливо перевірити, а тому й виправити. Кожен учасник, залучений в розробці програмного забезпечення повинен нести відповідальність за загальну технологію тестування та її підтримку

Таким чином визначено, що автоматичне юніт-тестування може бути важливою частиною процесу розробки програмного забезпечення, проте потрібно завчасно зважати на недоліки та ситуації при яких модульне тестування не може бути використане як єдина практика перевірки правильності роботи системи.

1.2 Аналіз методів та засобів вирішення задачі автоматизованого тестування

Задача використання модульного тестування вирішується шляхом застосування програмних бібліотек (фреймворків), реалізація та метод розповсюдження яких залежить від обраної платформи та інтегрованого середовища програмування. Такі бібліотеки можуть бути представлені у різному вигляді, наприклад, окремі застосунки із графічним інтерфейсом, яким необхідно передати посилання на реалізовані тестові модулі, додатки із консольним інтерфейсом або програмні розширення, інтегровані у середовище розробки [10]. Рішення про доцільність реалізації того чи іншого формату надання функціоналу фреймворку перш за все залежить від планованого подальшого методу розповсюдження бібліотеки.

Для використання методів автоматизованого тестування необхідно отримати доступ до модулів, що його надають. Він залежить від обраного формату представлення функціоналу. Якщо додаток містить інтерфейс, графічний або консольний, основним методом його розповсюдження є

надання користувачам окремого файлу або комплексної збірки, які можливо необхідно запустити на робочій машині, на якій заходяться тестові модулі. Недоліком такого методу є відсутність інтеграції з методами написання тестових методів та складність поширення, оскільки для того, щоб користувача отримав можливість використати програмне забезпечення, йому необхідно отримати його із середовища Internet або з фізичного носія, попередньо дізнавшися про існування такого продукту.

Найпоширенішим методом розповсюдження програмних бібліотек автоматизованого тестування є надання їх функціоналу у вигляді програмних розширень для інтегрованих середовищ розробки. Таким чином доступ до фреймворку можливо отримати напряму методом, що реалізує середовище розробки. Найпопулярнішим таким методом є поширення програмного забезпечення з допомогою пакетних менеджерів, наприклад Nuget для середовища .NET, npm для Node.js, Gradle для Java та багато інших [11]. В більшості випадків вони встановлюються разом з інтегрованим середовищем програмування та є легкими для освоєння, що дозволяє користувачеві швидко отримати доступ до необхідного програмного продукту. Іншою перевагою саме бібліотек автоматизованого тестування, що розповсюджуються як програмні розширення є можливість їх інтеграції із засобами тестування самого середовища розробки. Це дозволяє користуватись функціоналом фреймворку стандартними методами інтегрованого середовища, звичними користувачеві. Таким чином, людина, що хоче налаштувати та використовувати методи автоматизованого тестування у своєму продукті, не потребує окремих засобів, або компонентів та має можливість запускати та аналізувати результати роботи програмної бібліотеки напряму в середовищі програмування з використанням стандартизованого для середовища інтерфейсу. Прикладом такої інтеграції є можливість реалізувати розширення, що стане частиною стандартного засобу автоматизованого тестування Test Explorer в середовищі Visual Studio [12].

Окрім формату представлення доступу до функціонала та виду розповсюдження, розробнику програмної бібліотеки юніт-тестування необхідно чітко визначити повний набір методів та засобів, що надає фреймворк користувачеві. Адже саме якісні та кількісні показники засобів, що надає технологія, є визначним у рішенні користувачів чи задовільняє їхні потреби доступний функціонал. До якісних показників можна віднести швидкодію технології, наявність декількох інтерфейсів запуску тестування, можливість розширити або модифікувати існуючу логіку проведення тестування та аналізу тощо.

Кількісним показником, що вказує на загальну якість технології автоматизованого тестування та можливість її використання у різноманітних проєктах є об’єм множини різноманітних функцій та засобів тестування, що надає бібліотека. Для забезпечення стабільності системи при зміні її компонентів та алгоритмів застосовуються різні засоби проведення та аналізу тестування в рамках однієї програмної бібліотеки [13].

Серед основних методів, що реалізуються фреймворками тестування варто виділити:

- Безпосередньо проведення модульного тестування, тобто запуск, опрацювання та отримання результатів тестових функцій, написаних з використанням бібліотеки. Його ціллю є перевірка, що код працює правильно, тобто для заданих вхідних даних після виконання функція виконує очікувані дії, наприклад повертає відповідний результат, закінчується винятковим випадком, або коректно змінює визначені змінні всередині методу;
- Аналіз покриття коду тестами. Оскільки метою впровадження модульного тестування в програмний проєкт є максимальне покриття всієї кодової бази ізольованими тестовими методами, то важливим є розумінням кількості програмних функцій та модулів проєкту, що не перевіряються тестами та не задіяні в модульному тестуванні, а тому є більш схильними до несподіваних поломок, оскільки не є частиною процесу регресії. Такий аналіз

може бути статичним та динамічним. Результатом статичного аналізу, що отримується після процесу модульного тестування є повний список функцій та модулів, що не покриті тестовими методами, з вказанням тих випадків та вхідних даних, що варто було б перевірити. Динамічним аналізом є відображення в режимі реального часу функцій, для яких не знайдено відповідних тестових методів. Тобто при написанні нової функції, або зміни уже існуючої, програмна бібліотека враховує це, і над функцією, або в окремому вікні одразу відображає необхідність покрити новий метод тестами. Ціллю засобу є перевірити, що весь код відпрацьовує в процесі модульного тестування, та відсутні ділянки коду, що не беруть участі в тестуванні;

- Динамічний аналіз тестових методів. Цей засіб тестування перш за все допомагає розробникам не допускати семантичних помилок та використовувати найкращі практики написання тестових методів, що визначені технологією. Зазвичай реалізуються у вигляді візуальних підказок, із вказанням коректного варіанту виправлення, безпосередньо у середовищі розробки, або у текстовому аналізі (логові) після виконання процесу тестування. Є одним з найважливіших компонентів інформаційної технології автоматизованого тестування, оскільки значно покращує процес модульного тестування.

- Аналіз стилістики коду. Сьогодні нечасто можливо зустріти великий програмний продукт, що написаний однією людиною, сучасні проєкти зазвичай реалізуються групами розробників. Ці групи легко можуть сягати понад 500 людей, що займаються однією кодовою базою. Саме тому часто на рівні компанії або проєкту встановлюються уніфіковані вимоги до стилістики коду. Зазвичай вони описуються і є обов’язковими до виконання. Оскільки ціллю процесу модульного тестування є опрацювання якомога більшого об’єму коду, то для розробників може бути зручною перевірка виконання вимог стилістики коду тих методів, що були ними написані, одразу під час проведення автоматизованого тестування відповідною програмною

бібліотекою. Таким чином розробник може перевірити не лише правильність реалізації нових функцій, а й дотримання усіх стилістичних вимог за один запуск бібліотеки.

- Аналіз швидкодії та ефективності є складним методом, що може застосовуватись в процесі модульного тестування. Такий аналіз зазвичай є комплексним та включає в себе великий набір засобів, ціллю яких є перевірка швидкості виконання ключових операцій, стійкості до навантажень, споживання пам’яті під час роботи додатку тощо. Метод може включати в себе профілювання та заміри ключових показників, що впливають на роботу та успішність реалізації програмного продукту. Гарною практикою є застосування сторонніх профільних аналізаторів для таких цілей, хоча сучасні фреймворки модульного тестування також можуть містити елементи таких засобів.

Вище перераховані основні засоби, що зазвичай реалізуються сучасними бібліотеками автоматизованого тестування. Цей список не вичерпний та не обов’язковий, оскільки з урахуванням цілей, задач та складності фреймворку юніт-тестування, засоби, що він надає можуть доповнюватись або функціонально різнитись [14].

Отже, перед початком реалізації інформаційної технології необхідно визначити доступні формати представлення функціоналу, засіб розповсюдження фреймворку та функціональні можливості, що повинна надавати система автоматизованого тестування, адже від цього значно залежить її реалізація. Таким чином важливим етапом проєктування є визначення задач, що вирішуються бібліотекою.

1.3 Аналіз існуючих програмних засобів автоматизованого тестування

На сьогодні існує багато систем, призначених для автоматизації тестування, як платних, так і безкоштовних. Серед найпопулярніших

інформаційних технологій автоматизованого тестування для платформи .NET на сьогодні є NUnit, MSTest та xUnit [15]:

NUnit (рисунок 1.1) – безкоштовна модульна система тестування з відкритим кодом для .NET Framework та Mono [16]. NUnit.Forms доповнює NUnit засобами тестування елементів користувацького інтерфейсу Windows Forms. NUnit.ASP виконує цю саму задачу для елементів інтерфейсу в ASP.NET. Система має різні способи запуску, але не є інтегрованою за замовчуванням в основний засіб програмування мовою C# – Visual Studio. Сьогодні бібліотека широко застосовується для модульного, інтеграційного та автоматизованого тестування.

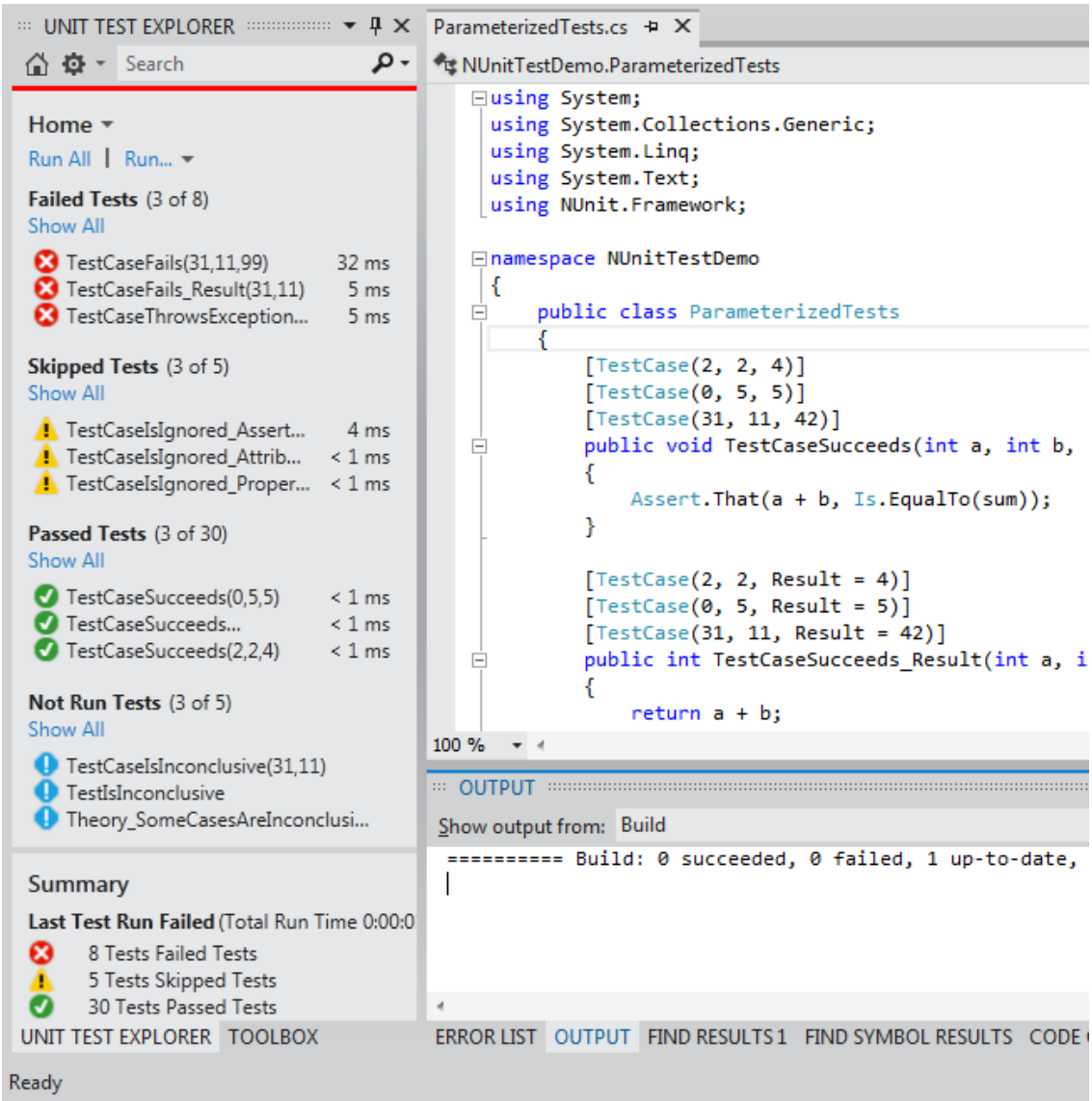


Рисунок 1.1 – Синтаксис та використання NUnit

xUnit (рисунок 1.2) – це загальна назва декількох модулів тестування, які виводять свою структуру та функціональність із SUnit Smalltalk. Система написана у високоструктурованому об’єктно-орієнтованому стилі, який дає можливість функціонування сучасним мовам, таким як Java та C # [17]. Технологія відрізняється основою суті юніт-тестів, поділяючи їх на факти та теорії, що розділяють логіку на “завжди вірно” та “вірно для правильних вхідних даних”. Таке рішення реалізовано відповідно до ідеї про поширення можливостей виконання модульного тестування від просто наявності модульних тестів до опису семантичної логіки.

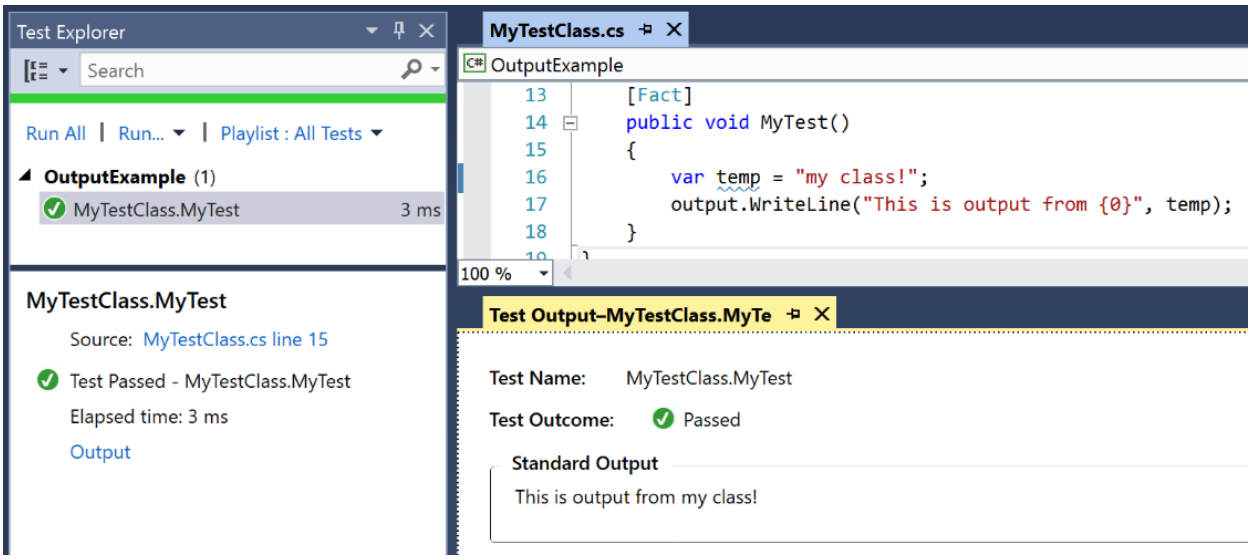


Рисунок 1.2 – Синтаксис та використання xUnit

MSTest (рисунок 1.3) – стандартний фреймворк тестування, що по замовчуванням є додатком для Visual Studio, який встановлюється разом із редактором [18]. Відсутність додаткових дій для отримання функціоналу бібліотеки юніт-тестування однозначно є головною перевагою фреймворку від Microsoft.

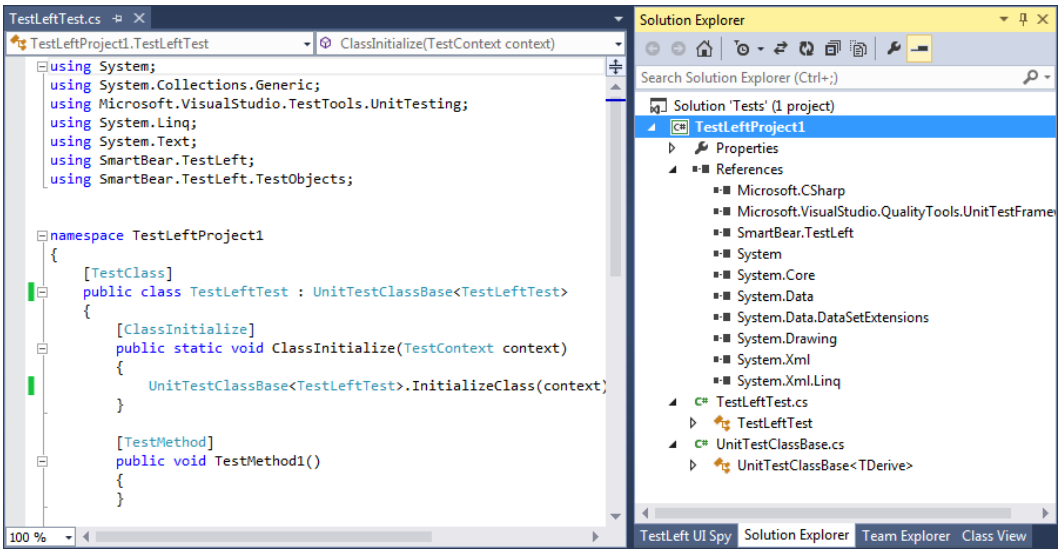


Рисунок 1.3 – Синтаксис та використання MSTest

Основними параметрами вибору бібліотеки юніт-тестування є швидкість виконання, легкість інтеграції, набір необхідних функцій та повнота інформації, що супроводжує систему [19]. Необхідно зважати на ці критерії порівнюючи бібліотеки. Проведемо критеріальний аналіз наступних систем: MSTest, NUnit та xUnit.

Першою спільною характеристикою можна назвати можливість обробки величезних обсягів тестів та вхідних даних. Однак відносно інших, основною проблемою MSTest є невисока продуктивність бібліотеки. Для невеликих задач питання інтеграції сторонніх бібліотек ускладнює рішення. Розглядаючи Visual Studio як основну платформу для .NET розробки, MSTest є стандартною бібліотекою вбудованою в середовище програмування, а отже надає усі основні функції одразу після встановлення Visual Studio. В порівнянні, сторонні бібліотеки, NUnit та xUnit необхідно додатково встановлювати через пакетні менеджери. Являючись фреймворками з відкритим кодом, xUnit й NUnit надають ширший функціонал та можливості, такі як запуск тестів не лише з середовища програмування, а й окремо від нього, покращений паралелізм, гнучкість системи, специфічні покращення та семантичні зміни.

Кожен з прикладів має широку базу користувачів та людей, які доповнюють документацію та пропонують покращення та розширення

функцій бібліотек з відкритим кодом. MSTest підтримує корпорація Microsoft, в результаті чого майже усю потрібну інформацію щодо використання фреймворку та написання тестів легко знайти на їх офіційному сайті. Документація усіх бібліотек є достатньою для роботи на будь-якому рівні. Водночас xUnit є більш розширеним та логічно зміненим фреймворком і тому він є складнішим для початку роботи з ним.

Проаналізувавши усі аналоги, визначено їхні можливості та недоліки, які враховувались при створенні власного програмного забезпечення з назвою «ISUnit» (табл. 1.1).

Таблиця 1.1 – Порівняльні характеристики програмних продуктів

Критерій	MSTest	NUnit	xUnit	ISUnit
Швидке виконання	0	1	1	1
Легкість інтеграції	1	0	0	1
Можливість запуску з декількох середовищ	0	1	0	0
Семантичний програмний інтерфейс	1	1	0	1
Паралельне виконання тестів	0	0	1	1
Наявність консольного інтерфейсу	0	0	0	1
Можливість розширення функціоналу бібліотеки	0	1	1	1
Всього	2	4	3	6

Таблиця порівняльних характеристик показала, що розробка програмного продукту є доцільною. В результаті розробки продукт буде покривати недоліки наявних рішень і надавати широкий набір функцій, які необхідні у використанні інформаційної технології автоматизованого тестування покращеним паралельним виконанням тестів та великою швидкістю роботи.

1.4 Постановка задачі реалізації інформаційної технології автоматизованого тестування

Однією з цілей інформаційної технології, що розробляється є автоматизація процесу автоматизованого тестування та надання засобів статичного та динамічного аналізу тестових методів. Такі задачі потребують попередньої підготовки перед реалізацією, оскільки необхідно коректно визначити методи, що використовуються для реалізації функціоналу, що буде надаватись інформаційною технологією. Саме тому, важливо після проведеного аналізу стану питання та порівняння аналогів правильно визначити завдання для виконання розробки програмного рішення:

1. Провести аналіз завдання і вибрати метод вирішення поставленої задачі;

1.1 Провести аналіз існуючих програмних реалізацій;

1.2 Виконати огляд відомих методів вирішення задачі реалізації програмного продукту;

2. Провести моделювання технології автоматизованого тестування;

2.1 Визначити набір складових компонентів технології;

2.2 Розробити стуктурну схему компонентів;

2.2 Проаналізувати особливості реалізації компонентів;

2.4 Розробити алгоритми роботи програмного забезпечення;

3. Реалізувати програмне забезпечення;

3.1 Провести аналіз і обґрунтування вибору програмних засобів для розробки;

3.2 Реалізувати вихідний код компонентів технології;

3.3 Провести тестування платформи;

4. Оцінити комерційний потенціал розробки;

4.1 Спрогнозувати витрати на виконання науково-дослідної, дослідно-конструкторської та конструкторсько-технологічної роботи;

4.2 Розрахувати ефективність вкладених інвестицій та період їх окупності.

Отже, описавши задачі розробки, можливо розпочинати визначення та аналіз методів й засобів автоматичного тестування, моделювання й реалізацію інформаційної технології автоматизованого тестування та опис відповідних технічних та економічних показників, що відносяться до розробки.

1.5 Висновок

У першому розділі розглянуто актуальність створення пропонованої інформаційної системи автоматизованого тестування. Проаналізовано сучасні методи розв'язання проблеми та проведено їх порівняння з розроблюваною інформаційною технологією. У результаті порівняння доведено доцільність розробки власного програмного продукту. Проведено дослідження відомих методів вирішення задачі реалізації інформаційної технології автоматизованого тестування.

Зазначено завдання для початку реалізації розробки технології.

2 МОДЕЛЮВАННЯ ІНФОРМАЦІЙНОЇ ТЕХНОЛОГІЇ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ

2.1 Аналіз та обгрунтування вибору складових компонентів інформаційної технології

Інформаційна технологія – сукупність організаційних і технічних засобів для збереження та обробки інформації з метою забезпечення інформаційних потреб користувачів [20].

Інформаційна технологія автоматизованого тестування (ІТАТ) – це сукупність програмних компонентів, що мають на меті забезпечити потребу користувачів у організації та проведенні процесу модульного тестування, шляхом надання відповідних технічних засобів.

Залежно від складності, цілей та особливостей реалізації технологій автоматизованого тестування, набір компонентів, що вони містять, може значно різнитись.

Основою ІТАТ є програмна бібліотека модульного тестування, котра містить повний набір функцій, атрибутів та інших елементів, представлених у вигляді вихідного коду, що доступні користувачеві та надають вичерпні можливості користування технологією [21]. Цей компонент представлений у вигляді чітко визначеного та зрозумілого програмного інтерфейсу, знаючи який, користувач може реалізувати усі тестові модулі свого проєкту, написати необхідні тестові методи, а також програмно запустити виконання процесу автоматизованого тестування. Іншим функціоналом, що може надавати програмна бібліотека може бути можливість розширення або зміни особливостей реалізації процесу тестування, написання підготовчих або завершаючих задач, що виконуються відповідно до та після завершення проведення тестування, спеціалізоване логування та інші різноманітні елементи, що можуть бути необхідні користувачеві для задоволення своїх задач.

Формат програмного інтерфейсу визначається розробниками, на основі аналізу та особистих уподобань реалізації в рамках можливостей, що надають обрані мова та платформа програмування [22].

Наступним необхідним елементом ІТАТ є програмний двигун системи. Двигуном є компонент технології, що в її архітектурі займає найнижчий рівень, не доступний користувачеві, та на основі технічних специфікацій та реалізації якого, будуються решта програмних засобів технології. Для інформаційної технології автоматизованого тестування двигун повинен містити реалізацію засобів пошуку тестових методів, опрацювання вхідних даних, виконання, налаштування системи, роботи з програмними потоками, їх одночасним виконанням (мультипоточністю), роботи з файловою системою, низькорівневе спеціалізоване логування, отримання результатів виконання тестів та проведення їх динамічного та статичного аналізу. Повний перелік задач, що реалізуються двигуном залежить від потреб технології, специфіки та обмежень обраної платформи програмування.

Саме двигун визначає загальний рівень швидкодії технології, оскільки він містить операції, що мають найвищий час виконання [23]. Тому, від якості реалізації двигуна залежить можливість якісної імплементації вищерівневих компонентів, та загальний рівень технології.

Наступним компонентом ІТАТ є користувацькі інтерфейси (КІ). В більшості випадків вони поділяються на консольні та графічні [24]. КІ є необов’язковим елементом, їх наявність визначається розробниками на етапі проєктування, та залежить від обраного формату розповсюдження та надання доступу до функціоналу технології, варіанти яких описані у розділі 1.2.

Одним із важливих компонентів технології автоматизованого тестування є програмне розширення для обраного інтегрованого середовища програмування, з допомогою якого користувачі технології можуть запускати процесу модульного тестування, отримувати результати роботи та аналізу в зручному форматі у вигляді стандартних засобів підтримки виконання тестових модулів самого середовища [25]. Цей компонент є одним з

найзручніших з точки зору користувача, тому для інформаційної технології, що має за мету максимально задовольнити потреби тих, хто її використовує, наявність такого розширення є надзвичайно важливим.

Серед інших компонентів, котрі можуть містити сучасні ІТАТ, можна виділити:

- Компонент, що містить набір аналізаторів, котрі виконують функцію статичного аналізу тестових методів. Вони можуть бути як представлені окремим елементом, так і інтегровані в двигун системи автоматизованого тестування, або в програмне розширення.
- База даних, що зберігає необхідну інформацію про тестові методи та результати їх виконання.
- Компонент, що виконує інтеграцію із різноманітними сторонніми сервісами, наприклад з сервісами безперервної інтеграції та доставлення TeamCity, або Jenkins, веб-сервісами, що виконують запуск процесу тестування з допомогою хмарних технологій та багато інших бібліотек та застосунків, що застосовуються в проєкті.
- Інші додаткові компоненти [26].

За результатами аналізу, перед початком моделювання інформаційної технології автоматизованого тестування, необхідно визначити усі компоненти, що вона буде містити.

Основним елементом розроблюваної ІТАТ буде двигун системи. В якісній технології до нього висуваються вимоги високої швидкодії та можливості розширення за потреб користувача. Він повинен виконувати усі головні функції, визначені в процесі аналізу.

Розроблювана технологія повинна мати зрозумілий та зручний програмний інтерфейс бібліотеки з високим різноманіттям функцій перевірки та атрибутів тестування.

Наявність хоча б одного користувацького інтерфейсу є великою перевагою ІТАТ, оскільки не усі розробники мають доступ до програмних

засобів проведення тестування, тому, визначено, що одним з компонентів технології буде консольний інтерфейс. Він повинен містити зручний метод запуску процесу тестування, та зрозумілий користувачеві інтерфейс, де легко та швидко можна визначити результати проведення тестування.

Інформаційна технологія автоматизованого тестування, що розробляється має на меті якнайповніше задоволення потреб користувачів у проведенні якісного модульного тестування, тому в сучасній технології наявність компоненту програмного розширення для найбільш популярного інтегрованого середовища програмування є обов'язковим. Таке розширення повинне бути максимально інтегрованим у засоби середовища для його зручного використання.

Останнім компонентом, що буде містити розроблювана ІТАТ є набір аналізаторів, що виконуватимуть функцію статичного аналізу тестових функцій, та допомагатимуть користувачем коректно та повноцінно використовувати засоби програмної бібліотеки [27]. Збільшення якісних та кількісних показників цього елементу технології є значним покращенням процесу модульного тестування, що є метою цієї роботи.

Отже, проаналізовано елементи, що має містити сучасна мультикомпонентна інформаційна технологія автоматизованого тестування та визначено ті компоненти, що має містити розроблювана ІТАТ для максимального покращення процесу модульного тестування та задоволення потреб користувачів.

2.2 Побудова структурної схеми інформаційної технології автоматизованого тестування

На етапі моделювання технології, необхідно чітко визначити структурні та архітектурні залежності між її компонентами. Існує пряма залежність між кількістю елементів, що необхідно імплементувати та кінцевою складністю проєкту, адже в більшості випадків додавання лише одного компоненту значно

впливає на усі ті, що вже були створені, якщо між ними є якісь взаємозв'язки. На рисунку 2.1 представлена структурна схема компонентів технології, що включає в себе усі елементи інформаційної технології автоматизованого тестування, що були визначені вище.

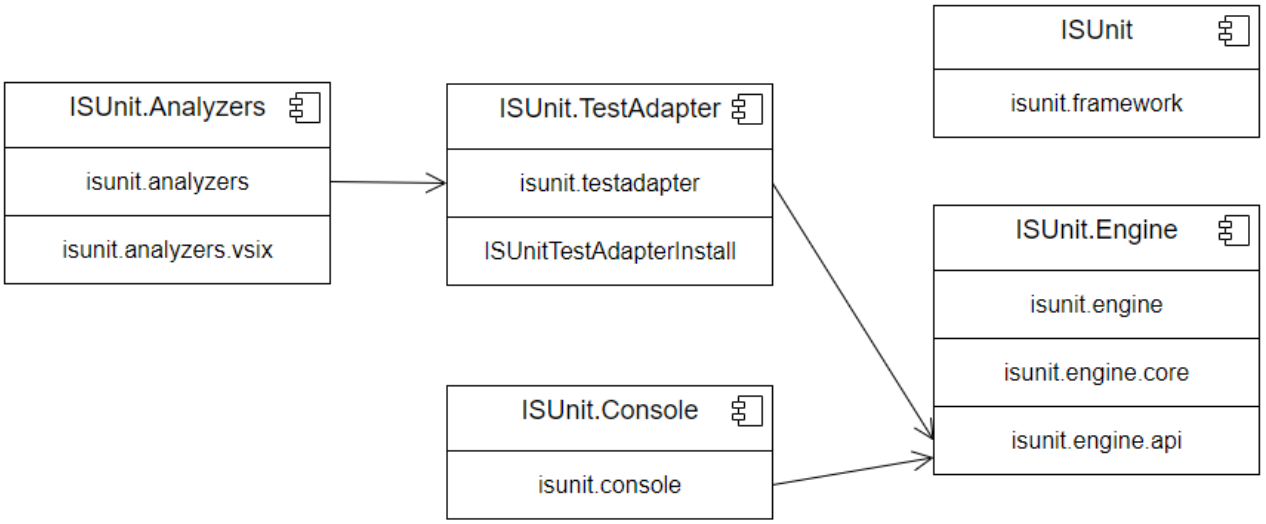


Рисунок 2.1 – Структурна схема компонентів розроблюваної інформаційної технології автоматизованого тестування «ISUnit»

У побудованій схемі кожен компонент є окремим структурним блоком, його назва вказана у верхній частині. Нижче описані усі модулі, що міститиме компонент. Стрілками визначено залежності між модулями. Опишемо, які задачі виконує кожен окремий компонент технології та їх модулі:

1. ISUnit – компонент програмного інтерфейсу ITAT.

1.1 isunit.framework – модуль програмного інтерфейсу, що містить реалізацію програмних засобів необхідних для написання тестових методів.

2. ISUnit.Engine – компонент двигуна ITAT.

2.1 isunit.engine – модуль, що відповідає за базове виконання великої частини функціоналу компоненту двигуна.

2.2 isunit.engine.api – модуль, котрий виконує роль публічного інтерфейсу до функціоналу компоненту двигуна.

2.3 isunit.engine.core – модуль, що містить програмну логіку найбільш

низького рівня та допоміжний вихідний код, яким користується система.

3. ISUnit.Console – компонент консольного інтерфейсу ITAT.

3.1 isunit.console – модуль, що містить вихідний код реалізації консольного інтерфейсу технології

4. ISUnit.TestAdapter – компонент програмного розширення ITAT.

4.1 isunit.testadapter – модуль, що містить вихідний код, котрий реалізує зв'язок між технологією, що розробляється та засобами інтеграції середовища програмування.

4.2 ISUnitTestAdapterInstall – модуль, що відповідає за можливість розповсюдження компонента методом вбудованих розширень інтегрованого середовища.

5. ISUnit.Analyzers – компонент динамічних аналізаторів ITAT.

5.1 isunit.analyzers – модуль, що містить відповідну документацію та вихідний код реалізації динамічних аналізаторів для технології, що розробляється.

5.2 isunit.analyzers.vsix – модуль, що реалізує можливість розповсюдження компонента методом вбудованих розширень інтегрованого середовища та методом пакетних менеджерів.

2.3 Обгрунтування вибору формату написання тестових методів для програмного інтерфейсу інформаційної технології автоматизованого тестування

В процесі моделювання компоненту програмного інтерфейсу бібліотеки модульного тестування важливо визначити стандартизований формат написання тестових методів. Формат написання тестових методів – це узагальнений шаблон структури тестової функції, можливість реалізації якого повинен бути наданий програмними засобами бібліотеки.

Використання одного формату дозволяє розробникам легше розуміти структуру тестового методу, а відповідно полегшує імплементацію та

внесення в нього змін. Саме тому, з метою покращення процесу проведення автоматизованого тестування розроблювана програмна бібліотека повинна надавати можливість використання оптимального та відомого підходу до написання тестових методів.

Натепер найвідомішим шаблоном написання модульних тестів є модель «AAA» («Arrange – Act – Assert») [28]. В його основі закладена ідея розділення тестового методу на три секції, кожна з яких виконує визначену функцію. Його зручність заключається у візуальному розділені етапів тестового методу, що полегшує розробникам процес їх написання. Кожна секція повинна містити таку виокремлену логіку:

- В секції «arrange» виконується підготовка об'єктів, що мають тестуватись. На цьому етапі необхідно привести інфраструктуру системи до бажаного стану та налаштувати залежності. Це виконується двома шляхами: прямим створенням екземпляру необхідного класу, або створення його двійника, завчасно реалізованого для тестування.
- В секції «act» відбуваються дії направлені на виклик відповідного методу системи, що тестується. Зазвичай, в цій частині шаблону викликається одна функція, котрій передаються налаштовані раніше залежності, та отримується результат її виконання для подальшої перевірки.
- Секція «assert» містить перевірочні функції, що дають змогу визначати правильність виконання тестового методу, на предмет відповідності очікуваному результату. На етапі може відбуватись перевірка результату виконання секції «act», кінцевого стану системи, що підлягає тестуванню або інших елементів, що брали участь в процесі тестування.

На рисунку 2.2 наведено приклад застосування шаблону «Arrange – Act – Assert» з використанням високорівневої мови програмування C#.

```
[Test]
0 references
public static void ContainsInt()
{
    // arrange
    var list = new List<int>(5);
    var expected = 3;

    // act
    list.Add(expected);

    // assert
    Assert.Contains(expected, list);
}
```

Рисунок 2.2 – Приклад використання шаблону написання тестових методів «Arrange – Act – Assert»

Серед інших особливостей застосування шаблону «AAA» варто виділити:

- Уникнення використання декількох однакових секцій в рамках одного тестового методу, що свідчить про те, що функція виконує перевірку декількох різних поведінкових випадків [29]. В цьому випадку варто розділити цей метод на кілька таких, кожен з яких перевіряє окремий випадок. Приклад структури тестового методу, що відповідає цій неправильній особливості реалізації наведено на рисунку 2.3.
- Уникання використання умовних тверджень в тестових методах. Для правильної та однозначної перевірки очікуваного результату, тестовий метод повинен бути простою послідовністю чітко визначених кроків перевірки. Наявність розгалужень свідчить, що метод виконує занадто багато перевірок. Використання умовних тверджень не дає жодних переваг, а лише додаткові витрати на підтримку тестових модулів, оскільки такі функції важче розуміти та модифікувати.
- Секція «act» зазвичай складається з одного рядка коду. Якщо етап складається з двох або більше рядків, це може свідчити про проблему з

публічним інтерфейсом тестування системи, що тестується.

- Для кращого розуміння тестового методу, кожну секцію варто відділяти між собою пустою стрічкою, та на її початку використовувати коментарі з назвою секції, що послідує.

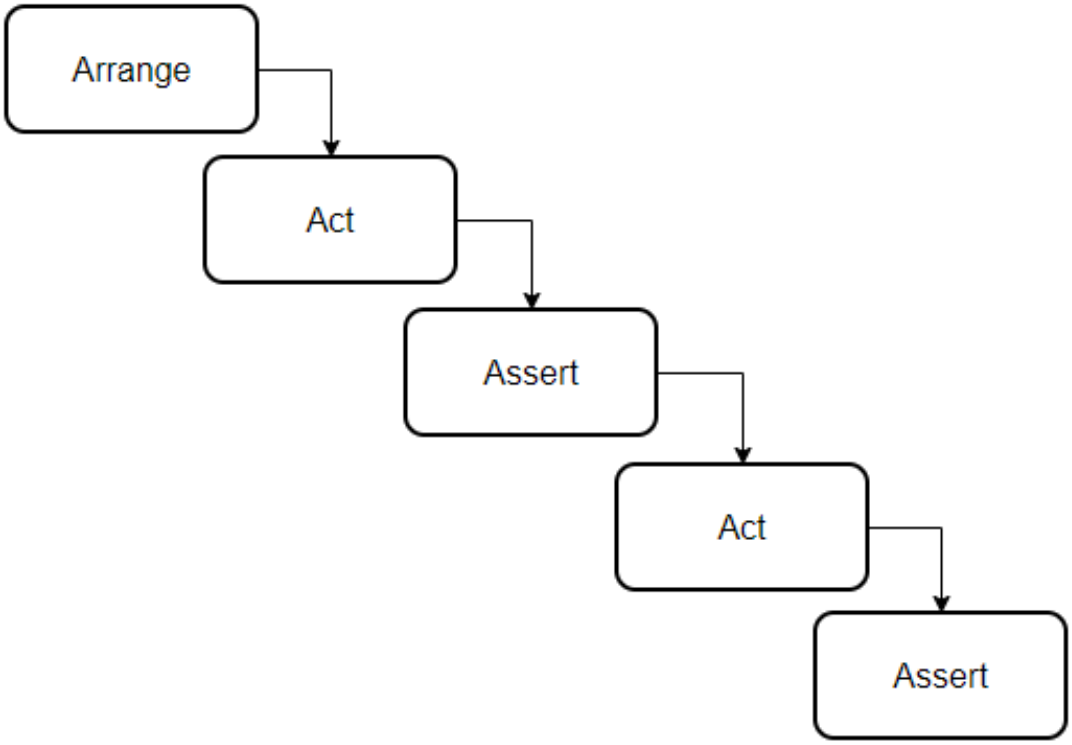


Рисунок 2.3 – Варіант використання декількох однакових секцій в одному тестовому методі

Таким чином визначено, що компонент програмного інтерфейсу інформаційної технології автоматизованого тестування для покращення процесу модульного тестування повинен надавати можливість та заохочувати використовувати шаблон «Arrange – Act – Assert» при написанні тестових методів користувачем.

2.4 Аналіз програмної моделі та вигляду подання результатів динамічного аналізу тестових методів з використанням компоненту аналізаторів інформаційної технології

Як визначено, компонент інформаційної технології, що містить аналізатори є одним з найголовніших для покращення процесу автоматизованого тестування, шляхом значного збільшення коректності використання програмної бібліотеки модульного тестування.

Динамічним аналізатором є програмний елемент, що в режимі реального часу аналізує новий, або щойно змінений програмний код на предмет невідповідності правилам описаним у ньому [30]. Такий аналізатор зазвичай містить назву, повний опис знайденої проблеми та рекомендацію, щодо її усунення. В рідких випадках такий аналізатор містить варіанти виправлення, при виборі яких, код автоматично змінюється, відповідно запропонованій корекції.

Реалізація цього компоненту значно різниться залежно від обраних інтегрованого середовища, платформи та мови програмування. Ця залежність є вагомою, оскільки не усі інтегровані середовища програмування підтримують можливість відображення результатів роботи аналізаторів в реальному часі. Тому на це варто зважати на етапі реалізації розроблюваної технології.

Різні платформи та мови програмування надають різні можливості реалізації аналізаторів. Наприклад компанія Microsoft, розробник платформи .NET та мов програмування C#, F# та інших, з допомогою спеціальних програмних бібліотек надає широкий доступ до потужностей своєї платформи, таким чином дозволяючи отримувати та аналізувати широкий спектр перевірок на глибокому рівні аналізу програмного коду [31]. Значною перевагою також є реалізація прямої інтеграції інтерфейсу отримання результатів аналізаторів в найпопулярніше інтегроване середовище програмування платформи .NET – Visual Studio. Приклад такого відображення

результатів показано на рисунку 2.4. Компанія Microsoft не єдина, що пропонує власні програмні інтерфейси реалізації аналізаторів.Варто виділити популярні середовища програмування IntelijIDEA, WebStorm та PyCharm, від компанії JetBrains, що мають можливість впровадження аналізаторів для мов програмування Java, JavaScript та Python та ряд інших [32]. Проте рівень можливостей при реалізації цих аналізаторів та прямої інтеграції зі середовищем не є оптимальним для їх широкого використання.

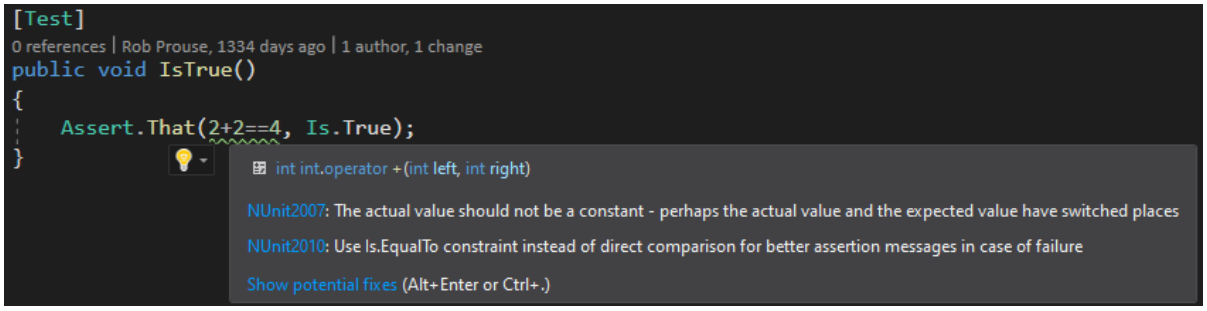


Рисунок 2.4 – Приклад візуального відображення результатів динамічного аналізу в середовищі програмування Visual Studio з використанням мови програмування C#

На етапі моделювання компоненту варто визначити типи аналізаторів, що будуть реалізовуватись. Це допоможе ширше покрити випадки некоректного використання бібліотеки, а отже збільшити якість перевірок.

Першим типом є структурні аналізатори. Їх правила відповідають за структурну цілісність тестових методів. Цей тип може містити перевірки використання асинхронних блоків коду, передачі коректних типів та кількості аргументів у тестовий метод, взаємодії з інтерфейсом підготовки та завершення виконання тестової функції, та інший аналіз.

Другим типом, що зазвичай являється найкількіснішим, є ствердджувальні. Вони містять правила, які покращують використання тверджень в тестовому коді. До них відносяться правила перевірок неправильно використаних функцій ствердження, що надаються програмним інтерфейсом бібліотеки модульного тестування.

Третім, можливим типом аналізаторів є приховуючі. Вони містять ряд правил, котрі приховують помилки компілятора мови програмування на основі контексту, та відображають свої припущення та рекомендації, щодо виправлення цих помилок.

Отже, визначено, що для значного покращення процесу автоматизованого тестування, інформаційна технологія повинна містити компонент аналізаторів. Описані правила, що опрацьовуються аналізаторами, вони мають містити максимально повну інформацію про проблему та, за можливості надавати варіанти їх вирішення. Також важливим фактором реалізації цього компоненту є його можливість прямої інтеграції із відомими інтегрованими середовищами програмування для полегшення отримання візуальних результатів аналізаторів.

2.5 Розробка алгоритму пошуку тестових методів

Важливим алгоритмом, що впливає на загальну якість інформаційної технології автоматизованого тестування є алгоритм пошуку тестових методів. Враховуючи складність та великий час виконання, якість моделювання та реалізації цього алгоритму є визначальною для загальної швидкодії процесу модульного тестування, що виконує технологія.

Схема розробленого алгоритму представлена на рисунку 2.5.

Алгоритм складається із 12 кроків. Вхідними даними для алгоритму є попередньо згенерований об’єкт збірки, створений на основі переданого у вхідних параметрах посилання на файл та структура, що містить інформацію про фільтрацію результатів знайдених тестових методів. На основі отриманих даних викликається метод «Build» класу «TestAssemblyBuilder», котрий відповідає за процес пошуку тестових методів у збірці. Далі відбувається створення об’єкту класу «TestSuite», котрий міститиме повну інформацію про знайдені тестові методи, що будуть включені в подальший процес тестування.

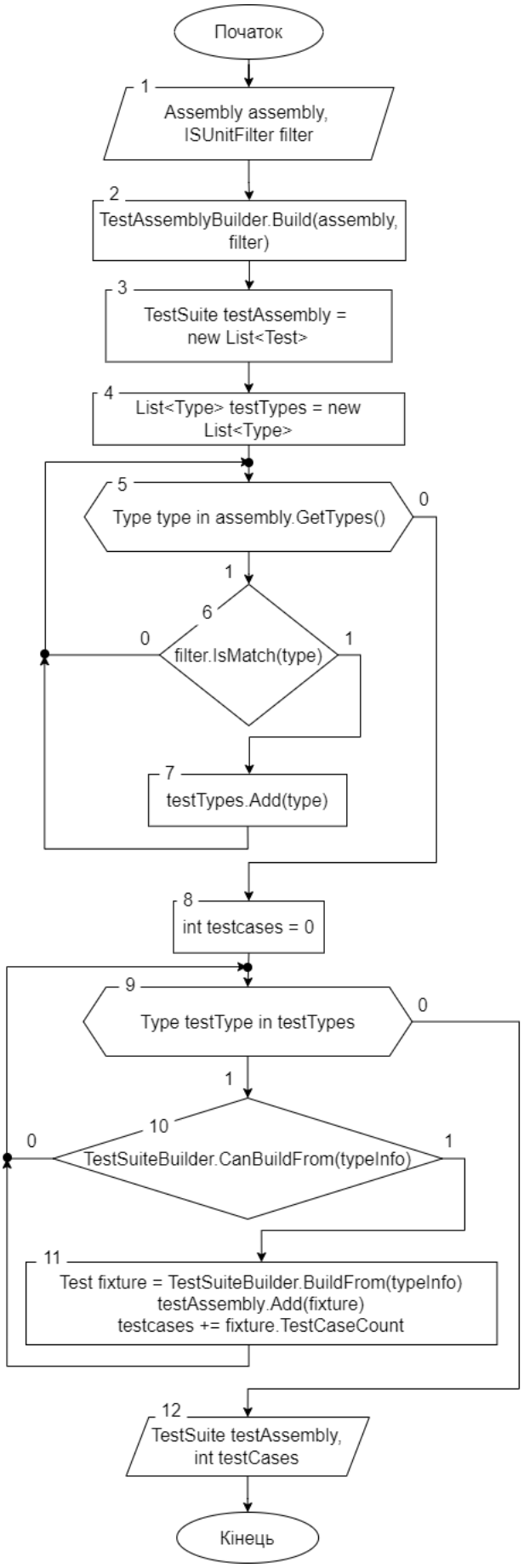


Рисунок 2.5 – Схема алгоритму пошуку тестових методів

Наступним кроком алгоритму є цикл, що, використовуючи засоби рефлексії мови програмування, отримує інформацію про усі класи, що містяться у збірці та перевіряє кожен з них на відповідність вказаному фільтру, й у випадку відповідності додає клас у виществорений список типів, що варто досліджувати далі. Далі вводиться числова змінна «testcases», що міститиме загальну кількість тестових методів знайдених у збірці. Завершаючим етапом алгоритму є програмний цикл, що опрацьовує список типів, отриманих на попередніх кроках. Для кожного знайденого типу відбувається перевірка чи являється він тестовим класом, написаним з використанням технології, тобто чи відзначений він відповідним атрибутом. Якщо визначено, що клас є тестовим, отримується інформація про його усі методи, з них визначаються ті, що являються тестовими та записуються у відповідний список, а їх кількість додається у змінну «testcases». Таким чином вихідними даними алгоритму є отриманий список знайдених тестових методів та їх загальна кількість. На основі цих даних технологія має можливість продовжувати процес тестування.

Отже, розроблено алгоритм пошуку тестових методів, котрий має високу ефективність та гнучкість завдяки можливості його застосування різними компонентами технології та для різних програмних середовищ.

2.6 Висновок

В процесі моделювання інформаційної технології були визначені та проаналізовані основні компоненти технології, описані їх ключові особливості та умови реалізації. Обґрунтовано використання шаблону «Arrange – Act – Assert», як такого, що надає програмна бібліотека для стандартизації написання тестових методів. Також був визначений формат відображення та типи динамічних аналізаторів, що будуть імплементуватись. Був розроблений оптимізований алгоритм пошуку тестових методів.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ІНФОРМАЦІЙНОЇ ТЕХНОЛОГІЇ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ

3.1 Варіантний аналіз і обґрунтування вибору мови та середовища програмування

Для того щоб апаратне забезпечення комп’ютера виконало бажаний набір дій, необхідно передати процесору чітко сформульовані програмні інструкції. Реалізація цих інструкцій відбувається за допомогою мов програмування, які зрештою перетворюються в машинний код зрозумілий процесору. Кожна програма являє собою набір команд, корті визначають роботу, яку потрібно виконати апаратному забезпеченню.

Машинний код складається з двійкових чисел (бітів) і є складним для розуміння і написання [33]. Через це використовуються високорівневі мови програмування, які дозволяють реалізовувати й виконувати програмне забезпечення не зважаючи на специфіку архітектури процесора, та не знаючи як буде виглядати кінцевий машинний код. Кожна інструкція у високорівневій мові програмування включає в себе багато інструкцій низькорівневих мов, які зрештою будуть виконуватись. Вихідний код таких мов є зрозумілим для написання та редагування розробником.

До реалізації програмного забезпечення інформаційної технології автоматизованого тестування необхідно проаналізувати відомі мови програмування та обрати ту, яка найбільше буде відповідати потребам розробки програмного продукту. Технології автоматизованого тестування існують для багатьох платформ та високорівневих мов програмування. Зазвичай можливість використання технології доступна лише мовою програмування для якої та якою вона була написана. Тому важливо обрати мову програмування, використовуючи котру, розробка мультикомпонентної технології буде найбільш оптимальною.

Для порівняння оберемо чотири відомі мови програмування: C#, C++, Java та Python. Вибір зумовлений їх великим поширенням та використанням об’єктно-орієнтованої парадигми програмування, що зарекомендувала себе однією з найкращих для реалізації технологій автоматизованого тестування.

Мова програмування C# [34] – об’єктно-орієнтована мова програмування, створена на початку 2000 року в компанії Microsoft. Розроблялась як мова розробки для платформи програмування .NET Framework і сьогодні є головною та найбільш розповсюдженою мовою програмування для цієї платформи. C# створювалась як мова прикладного рівня для одного з компонентів .NET – віртуальне середовище виконання CLR (Common Language Runtime). Саме CLR багато в чому визначає можливості мов програмування, які виконуються на платформі.

C# є мовою з C-подібним синтаксисом та наступником мов програмування C++, Java, Delphi. Опираючись на практику використання попередників, Microsoft виключила із реалізації C# деякі моделі програмування, які зарекомендували себе як проблематичні при розробці програмного забезпечення. Одним з прикладів є відсутність множинного наслідування класів, на відміну від C++, де ця можливість вважається однією з найбільших недосконалостей мови.

Однією з переваг об’єктно-орієнтованої мови C# є статична типізація, що дозволяє уникнути багатьох помилок кодування ще на етапі компіляції, а не виконання. Мова постійно оновлюється, покращує швидкодію, розширює список можливостей та мовних особливостей і виправляє наявні нелогічності.

C# однією з базових можливостей підтримує мультиплатформеність, що дозволяє запускати додатки на різних платформах, таких як Android, IOS, Windows, веб-додатки тощо. Це досягається тим, що платформа під час компіляції перетворює код написаний на C# у проміжну мову IL (Intermediate Language), яку й виконує CLR [35]. Серед інших особливостей мови можна виділити високу інтеграцію з наявними середовищами програмування, можливість метапрограмування, підтримку мультипарадигменості,

реалізоване керування пам’яттю. Великою перевагою C# є велика вбудована в .NET бібліотека базових класів для взаємодії з більшістю необхідних апаратних та програмних інтерфейсів та різноманіття технологій для яких можна використовувати C#, таких як ASP, .NET Core, Blazer, Razor та інші.

C++ [36] – високорівнева мова програмування яка має вбудовану підтримку декількох парадигм програмування. Розроблена як розширення до мови С Б'ярном Страуструпом у 1979 році. Розроблялась C++ для ефективного системного програмування, для використання додатків з обмеженими ресурсами та великих систем. Головними особливостями дизайну були швидкість виконання, ефективність, гнучкість мови та підтримка різних стилів програмування: імперативного, узагальненого та функціонального. C++ має ряд нововведень порівняно з С, серед яких підтримка об’єктно-орієнтованого програмування, підтримка узагальненого програмування за допомогою шаблонів, доповнення до стандартної бібліотеки класів С, можливість обробки виняткових ситуацій, вбудовані функції та оператори управління вільнорозподіленою пам’яттю.

Сьогодні C++ є однією з найпотужніших мов програмування завдяки своїм програмним можливостям взаємодіяти із системою на нищому рівні й безпосередньо керувати апаратною пам’яттю. Завдяки цьому вона використовується для багатьох видів розробок.

Мова знаходить своє використання у додатках, яким необхідна висока обчислювальна продуктивність та контроль виконання, вбудованих додатках, програмних серверах та відеоіграх з високим використанням графічних ресурсів. Серед основних прикладів використання мови у системному програмуванні є факт, що операційна система Windows написана в більшості на C++. Періодично виходять нові офіційні стандарти мови, які оптимізують виконання та розширюють можливості C++.

Java [37] – статично типізована, об’єктно-орієнтована, високорівнева мова програмування, яка створена в 1996 році в компанії Sun Microsystems, а сьогодні розвивається в компанії Oracle. Особливістю реалізації Java є

використання JVM, або ж віртуальної Java машини. Це дозволяє транслювати додатки в байт-код, який можна запустити на будь-якій комп'ютерній архітектурі. Програми написані на Java мають репутацію повільніших та більш вимогливих до оперативної пам'яті, аніж відповідні, написані мовою C.

Першочергово Java розроблялась як мова програмування незалежна від платформи на якій вона запускається, тому вона слабо використовується для системного та прикладного програмування, в якому часто необхідно застосовувати низькорівневі можливості роботи з апаратним забезпеченням. Проте Java сьогодні є однією з найпопулярніших мов програмування і завдяки своїм основним перевагам має одну з найбільших користувацьких баз та кількість проектів написаних з її використанням. Завдяки цьому мова зазвичай використовується для написання мобільних, прикладних, веб та серверних додатків, а також багатьох допоміжних компонентів, таких як системи керування базами даних, хмарні сервіси та інші [38].

Python – скриптова мова програмування високого рівня, результат виконання якої отримується шляхом інтерпретації вихідного коду. Мова була розроблена в 1991 році. Надає широкі можливості за рахунок гнучкості та підтримки багатьох парадигм програмування [39]. Код в Python організований у зручні для користування модулі, які сприяють легкому перевикористанню вже написаних компонентів. Головною особливістю мови є реалізація динамічної типізації та динамічного зв'язування, які в поєднані з великою швидкодією роблять мову потужною для використання. Головна сфера використання мови програмування це розробка веб-сервісів, серверних модулів, невеликих компонентів, що потребують швидкісного виконання всередині системи. Сьогодні Python знаходить широке використання в клієнтських та серверних сервісах завдяки великому набору розроблених для нього технологій.

Таблиця 3.1 містить результат порівняння розглянутих мов програмування за критеріями, що є головними для розробки бакалаврської роботи

Таблиця 3.1 – Порівняння мов програмування

Назва критерію	C#	C++	Java	Python
Підтримка об’єктно-орієнтованої парадигми програмування	1	1	1	1
Висока швидкодія	0	1	0	1
Легке поширення програмних засобів	1	0	0	0
Можливість розробки динамічних аналізаторів	1	1	0	0
Автоматична інтеграція програмних розширень із середовищами розробки	1	0	1	1
Наявність широкої бібліотеки класів для взаємодії зі системою	1	0	1	0
Результат	5	3	3	3

З таблиці порівняння отримано результат, що свідчить про те, що мова програмування C# має значні переваги для реалізації змодельованої мультикомпонентної технології. C# має можливості та задовільняє усі вимоги до мови програмування для розробки інформаційної технології автоматизованого тестування.

Окрім вибору мови програмування, іншою значною складовою засобів реалізації програмного забезпечення є середовище розробки. Інтегроване середовище розробки (IDE) – спеціалізований програмний додаток, який надає комплексні засоби для розробки програмного забезпечення. Базовими частинами IDE є текстовий редактор коду, автоматизовані засоби збірки й виконання та відлагоджувач. Багато середовищ розробки пропонують більш розширені можливості, такі як функціональні редактори коду з підсвічуванням та підказками, вбудовані системи контролю версій, тощо.

Найкращими інтегрованими середовищами розробки для програмування мовою C# є Microsoft Visual Studio, Jetbrains Rider та Visual Studio Code [40].

Microsoft Visual Studio (рис. 3.1) [41] – програмний продукт, який містить інтегроване середовище розробки та ряд інших інструментальних засобів розробки.

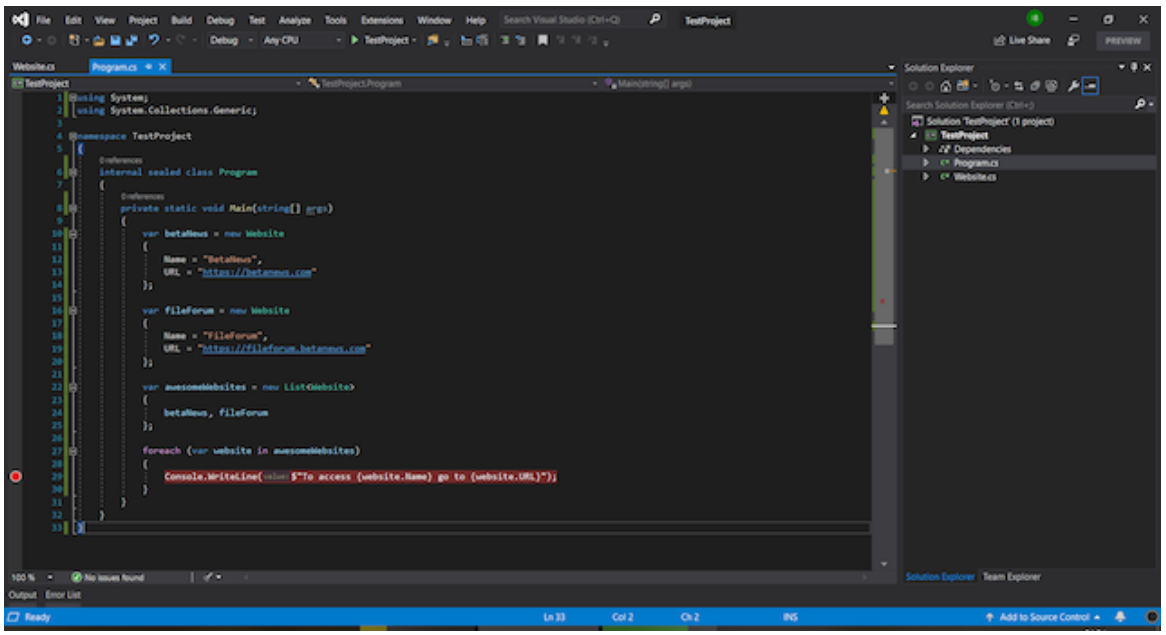


Рисунок 3.1 – Інтерфейс IDE Visual Studio 2019

Цей продукт дозволяє проводити розробку як базових консольних додатків, так і програм з широким графічним інтерфейсом, веб-служб, веб-сервісів, веб-сайтів та програмних серверів.

Visual Studio має велику підтримку платформи .NET та реалізації ПЗ з використанням її технологій, оскільки є розробкою компанії Microsoft, які також є розробниками платформи та мови програмування C#.

Visual Studio включає в себе редактор коду с підтримкою потужної технології IntelliSense, яка допомагає коректно писати вихідний код за допомогою рефакторингу, розумного підсвічування, автодоповнення, автоматичного вибору ім’я елементів та інших функцій. Вбудований відлагоджувач дозволяє працювати як з вихідним кодом C#, так і кодом на нижчих рівнях та є найкращим по повноті інформації про додаток під час виконання. Інші вбудовані інструменти включають в себе редактор візуальних форм для спрощення створення графічного інтерфейсу, веб-редактор,

дизайнери класів та схеми бази даних. Середовище програмування надає повний функціонал для розробки за допомогою технологій .NET Framework, .NET Core, ASP.NET, Silverlight, ADO.NET, тестових фреймворків та інших. Окрім цього Visual Studio дозволяє створювати на встановлювати сторонні програмні розширення функціоналу.

JetBrains Rider (рисунок 3.2) [42] – мультиплатформенне середовище розробки для платформи .NET, що розробляється компанією JetBrains. Середовище підтримує платформи .NET Framework, .NET Core та Mono, чого достатньо для проведення розробки різнотипних додатків.

В основі Rider лежить інший продукт компанії – ReSharper [43], який починався як розширення для підвищення продуктивності роботи в Microsoft Visual Studio. ReSharper проводить статичний аналіз коду й надає додаткові засоби пошуку, підсвічування синтаксису, форматування, оптимізації, рефакторингу, автозаповнення та генерації коду. Зрештою ReSharper був доопрацьований, його функціонал був розширений і перенесений в інтегроване середовище розробки – Rider. IDE містить в собі широкий функціонал редагування й аналізу коду, інструменти запуску юніт-тестування, роботи з базою даних, відлагоджувач, підтримку фронтенд-технологій та можливість встановлювати та реалізовувати сторонні програмні розширення.

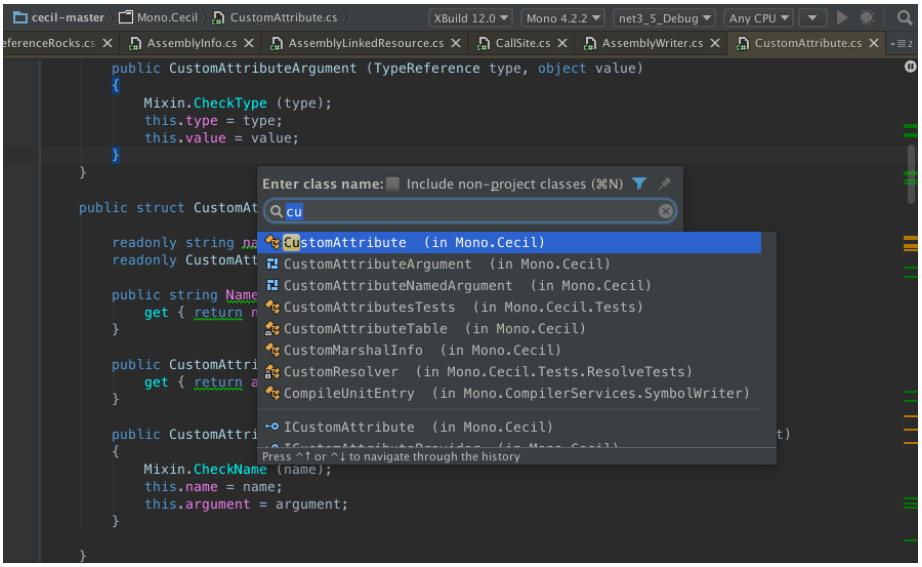


Рисунок 3.2 – Інтерфейс IDE JetBrains Rider

Visual Studio Code (рис. 3.3) – “легкий” редактор коду створений для розробки хмарних та веб-технологій [44]. Редактор не створювався для повного циклу розробки великих систем, а задумувався як гнучка платформа з широкими можливостями налаштування середовища. Сьогодні Code містить безліч готових розширень, які не лише змінюють інтерфейс, або розширюють аналіз та редагування коду, а додають підтримку цілих мов та платформ програмування.

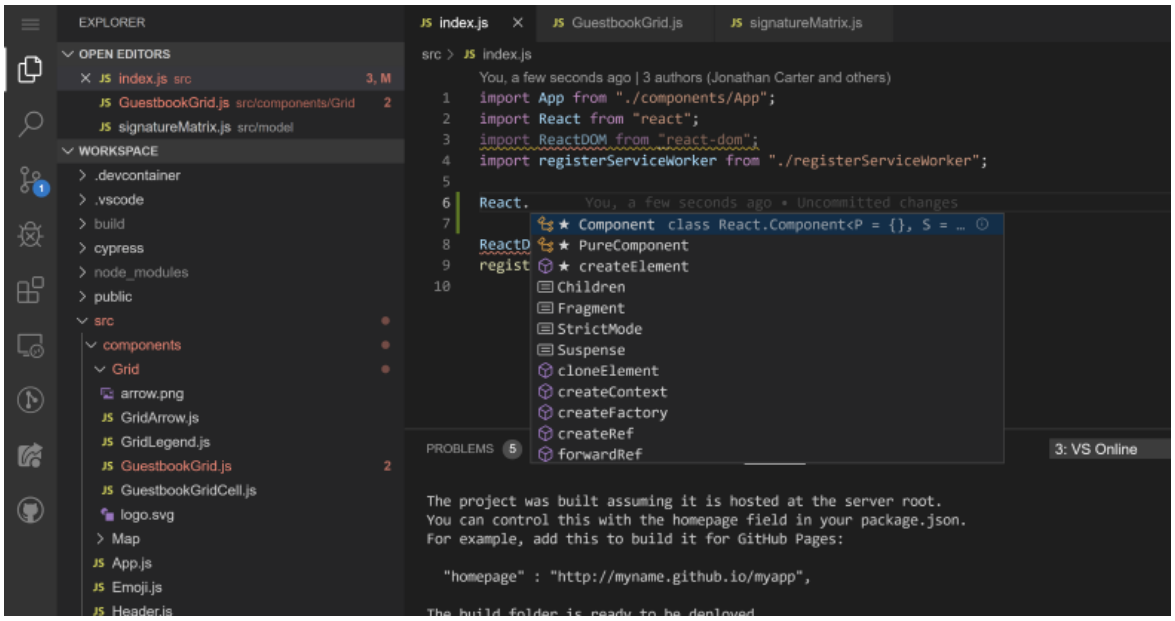


Рисунок 3.3 – Інтерфейс IDE Visual Studio Code

Таблиця 3.2 містить критеріальний аналіз порівняння інтегрованих середовищ розробки.

Таблиця 3.2 – Порівняння засобів розробки

Назва критерію	Microsoft Visual Studio	Jetbrains Rider	Visual Studio Code
Вбудована підтримка C#	1	1	0
Легка інтеграція засобів автоматизованого тестування	1	0	0
Відлагоджування коду з декількох програмних потоків	1	1	0

Продовження таблиці 3.2

Назва критерію	Microsoft Visual Studio	Jetbrains Rider	Visual Studio Code
Легке встановлення сторонніх розширень	1	0	1
Потужні програмні механізми відлагоджування	1	1	0
Вбудована допомога в написанні вихідного коду	1	1	1
Результат	6	4	2

В результаті варіантного аналізу інтегрованих середовищ засобів визначено, що розробку технології доцільно проводити з використанням Microsoft Visual Studio IDE. Це середовище містить усі програмні засоби розробки необхідні для реалізації інформаційної технології автоматизованого тестування.

Отже, проаналізувавши результати варіантного порівняння визначено, що для реалізації інформаційної технології «ISUnit» доцільно використовувати високорівневу мову програмування C# та інтегроване середовище програмування Microsoft Visual Studio. Вони найбільш повно задовольняють потреби та вимоги до розробки технології, визначені в процесі моделювання

3.2 Розробка компоненту двигуна інформаційної технології автоматизованого тестування

В процесі моделювання визначено, що компонент двигуна складається із трьох модулів. Розглянемо особливості реалізації основних складових кожного модулю.

Модуль isunit.engine містить вихідний код, що відповідає за базове виконання ряду функціоналу компоненту двигуна. Серед найголовніших функцій, за котрі відповідає модуль варто виділити:

- Ініціалізація класу «TestEngine» та сервісів, які будуть йому необхідні в процесі роботи.
- Реалізація цих сервісів.
- Реалізація класу «MasterTestRunner», що виконує функцію управління процесом проведення тестування.
- Управління запуском процесу тестування декількох збірок в окремих процесах операційної системи.
- Створення та редагування налаштувань процесу тестування.
- Покращена обробка виключних ситуацій.
- Виконання логування результатів тестування у форматі XML.

Опишемо найважливіші особливості реалізації цих функцій детальніше. Клас «TestEngine» ініціалізується одним з перших після початку запуску процесу тестування. Він містить метод «Initialize» котрий виконує створення усіх сервісів на початку процесу. Цей метод охоплює створення сервісів налаштувань, розширень, фільтрування тестових методів, отримання результату процесу, роботи з файловою системою, з проєктами платформи .NET, доменами, фреймворком виконання, драйверами та сервісом генерації класу запуску тестування. Реалізація методу «Initialize» зображена на рисунку 3.4.

Іншою функцією цього класу є публічний доступ до отримання посилання на основний клас управління процесом тестування «MasterTestRunner».

```
        public void Initialize()
        {
            if(InternalTraceLevel != InternalTraceLevel.Off && !InternalTrace.Initialized)
            {
                var logName = string.Format("InternalTrace.{0}.log", Process.GetCurrentProcess().Id);
                InternalTrace.Initialize(Path.Combine(WorkDirectory, logName), InternalTraceLevel);
            }

            if (Services.ServiceCount == 0)
            {
                Services.Add(new SettingsService(true));
                Services.Add(new RecentFilesService());
                Services.Add(new TestFilterService());
                Services.Add(new ExtensionService());
                Services.Add(new ProjectService());
#if NETFRAMEWORK
                Services.Add(new DomainManager());
                Services.Add(new RuntimeFrameworkService());
                Services.Add(new TestAgency());
#endif
                Services.Add(new DriverService());
                Services.Add(new ResultService());
                Services.Add(new DefaultTestRunnerFactory());
            }

            Services.ServiceManager.StartServices();
        }
```

Рисунок 3.4 – Реалізація методу «Initialize» класу «TestEngine»

Розглянемо детальніше які функції виконує клас «MasterTestRunner». Його задача в організації процесу тестування на найвищому рівні програмної логіки. На початку тестування створюється один об'єкт цього класу. Він містить основні методи «Run» – запуск процесу тестування пошуку тестових методів й тестування із вказаним фільтром, «RunAsync» – асинхронний запуск, «Explore» – пошук тестових методів у збірці по фільтру без їх запуску, «Load» – завантаження та отримання результату запуску групи тестових методів, «StopRun» – зупинка процесу тестування та ряд інших, що включають різноманітні перевірки на високорівневу логіку.

У випадку необхідності проведення одночасного тестування декількох збірок, найкращим рішенням є їх паралельний запуск в різних процесах операційної системи. Основу цієї програмної логіки реалізують клас «AggregatingTestRunner» та його нащадок «MultipleTestProcessRunner».

Частиною функціоналу модуля є покращена обробка помилок. Вона заключається у розширенні інформації та генерації помилки в текстовому форматі зрозумілому користувачеві про виключну ситуацію, що трапилась

протягом роботи інформаційної технології. Реалізація методів, що відповідають за це міститься в класі «ExceptionHelper».

Результати, отримані в результаті роботи інформаційної технології автоматизованого тестування записуються у файл «TestResult.xml» в популярному та зрозумілому форматі XML.

Розширювана мова розмітки (XML) — це простий текстовий формат для представлення структурованої інформації: документів, даних, конфігурації, книг, транзакцій, рахунків-фактур та багато іншого. Він був похідний від старішого стандартного формату під назвою SGML (ISO 8879), щоб бути більш придатним для використання в програмуванні [45].

Модуль isunit.engine виконує часткову генерацію та безпосередньо запис у файл отриману в процесі тестування інформацію про успішність виконання тестових методів збірки.

Наступним розглянемо модуль isunit.engine.api, який виконує роль публічного інтерфейсу до функціоналу компоненту двигуна. Цей модуль використовується компонентами програмного та консольного інтерфейсу, а також, разом з іншими модулями компоненту двигуна є складовою програмного розширення для поширення доступу до інформаційної технології автоматизованого тестування «ISUnit». Окрім доступу до логіки двигуна можна виділити такі функції та класи, що реалізує модуль:

- Реалізація класу «TestPackage».
- Реалізація класу «TestFilter» та «TestFilterBuilder».
- Точка входу до використання модулю isunit.engine.
- Реалізація базових класів виключних ситуацій, що можуть виникнути в процесі тестування.
- Імплементація ряду інтерфейсів, на базі яких можуть створюватись класи їх реалізацій для розширення функціоналу компоненту двигуна.

Усі функції модуля дозволяють програмному та консольному інтерфейсам мати доступ до функціоналу двигуна повною мірою. Розглянемо

деталі реалізації рішення цих задач.

Кожен об’єкт класу «TestPackage» містить інформацію про групу тестових методів, що підлягають тестуванню. Ця інформація складається з ідентифікаційного номера, повного шляху до файлу та налаштувань, що відносяться до цих тестових методів.

Іншим важливим класом в процесі тестування є «TestFilter». Він відповідає за зберігання інформації про фільтрацію тестових методів, якщо були передані відповідні аргументи на початку запуску процесу тестування. Використовуючи фільтри, виконаються лише ті методи, що відповідають вказаному фільтру. Серед них можна виділити фільтрацію по імені тестового метода, атрибутам, категорії, простору імен, в який входить метод та деякі інші. Опрацювання вхідних аргументів та подальше створення об’єктів «TestFilter» з відповідною інформацією виконує клас «TestFilterBuilder».

Найпершою точкою доступу до функціоналу запуску процесу тестування є клас «TestEngineActivator», який своїм методом «CreateInstance» повертає об’єкт класу «TestEngine», котрий в свою чергу виконує логіку ініціалізації системи для початку тестування.

Цей модуль містить ряд класів виключних ситуацій, що є нащадками «System.Exception» та використовуються для покращення процесу опрацювання помилок під час роботи системи шляхом збільшення розуміння причини виникнення проблеми.

Являючись модулем на який посилаються інші компоненти, для вирішення задачі можливості розширення функціоналу інформаційної технології користувачем, розроблено ряд класів інтерфейсів. Використовуючи їх, розробник має можливість написати власний вихідний код з яким буде взаємодіяти компонент двигуна та виконувати задачі актуальні користувачеві технології.

Модуль isunit.engine.core містить програмну логіку найбільш низького рівня та допоміжний вихідний код, яким користується система. Серед основних функцій модуля можна виділити:

- Реалізацію взаємодії між віддаленими тестовими агентами з використанням технології WebSocket.
- Логування інформації в процесі роботи системи.
- Доступ до файлової системи.
- Реалізацію публічного класу «Guard».
- Інші функції описані в процесі моделювання.

Розглянемо особливості реалізації функціоналу модуля isunit.engine.core.

Для компоненту двигуна розроблена можливість паралельного запуску процесу тестування на різних тестових агентах. Тестовим агентом називається віддалений агент, що виконує частину процесу тестування у власному домені або процесі операційної системи рамках одного комп'ютерного пристрою та взаємодіє із інформаційною технологією автоматизованого тестування «ISUnit», використовуючи мережевий протокол TCP.

Transmission Control Protocol (TCP, протокол управління передачею) — один із основних протоколів передачі даних в Інтернеті або локальній мережі. Механізм TCP надає потік даних з попередньою установкою з'єднання, здійснює повторний запит у разі втрати даних та усуває дублювання при отриманні двох копій одного пакета, гарантуючи тим самим (на відміну від UDP) цілісність даних, що передаються, і повідомлення відправника про результати передачі [46]. Реалізацію класу віддаленого тестового агента зображено на рисунку 3.5.

В процесі розробки програмних продуктів для розробників важливим є отримання коректних даних про процеси, що відбуваються під час роботи ПЗ. Запис та форматування такої інформації називається логування. Модуль isunit.engine.core відповідає за цей процес шляхом перехоплення усіх спроб запису потрібної інформації, наприклад, використовуючи запис в консоль, методи System.Diagnostics.Trace або інші менеджери з логування, та записуючи ці дані у відповідний текстовий файл.

```
public class RemoteTestAgent : TestAgent
{
    /// <summary>
    /// Construct a RemoteTestAgent
    /// </summary>
    0 references
    public RemoteTestAgent(Guid agentId, IServiceLocator services)
        : base(agentId, services) { }

    public ITestAgentTransport Transport;

    0 references
    public int ProcessId => System.Diagnostics.Process.GetCurrentProcess().Id;

    3 references
    public override bool Start()
    {
        Guard.OperationValid(Transport != null, "Transport must be set before calling Start().");
        return Transport.Start();
    }

    5 references
    public override void Stop()
    {
        Transport.Stop();
    }

    6 references
    public override ITestEngineRunner CreateRunner(TestPackage package)
    {
        return Services.GetService<ITestRunnerFactory>().MakeTestRunner(package);
    }
}
```

Рисунок 3.5 – Реалізація класу віддаленого тестового агента модуля isunit.engine.core

Базовою функцією, реалізація якої необхідна для коректної роботи інформаційної технології є взаємодія із файловою системою. Модуль, використовуючи класи та методи з простору імен «System» та «System.IO», виконує усі необхідні задачі, що виникають в процесі тестування та пов’язані із файловою системою. Компонент двигуна має доступ до читання та запису інформації в директорії, створення, видалення та запис файлів та інші потрібні методи.

В модулі isunit.engine.core розроблено клас «Guard», що реалізує ряд перевірок, використовуючи які, можливо визначати правильність програмної роботи інформаційної технології в процесі тестування. У випадку невідповідності очікуваному результату методи класу генерують виключні ситуації, котрі можуть бути коректно опрацьовані у вихідному коді, що виконує перевірку. Реалізація класу зображена на рисунку 3.6.

```
public static class Guard
{
    15 references
    public static void ArgumentNotNull(object value, string name)
    {
        if (value == null)
            throw new ArgumentNullException("Argument " + name + " must not be null", name);
    }

    7 references
    public static void ArgumentNotNullOrEmpty(string value, string name)
    {
        ArgumentNotNull(value, name);

        if (value == string.Empty)
            throw new ArgumentException("Argument " + name +" must not be the empty string", name);
    }

    0 references
    public static void ArgumentInRange(bool condition, string message, string paramName)
    {
        if (!condition)
            throw new ArgumentOutOfRangeException(paramName, message);
    }

    13 references
    public static void ArgumentValid(bool condition, string message, string paramName)
    {
        if (!condition)
            throw new ArgumentException(message, paramName);
    }
    9 references
    public static void OperationValid(bool condition, string message)
    {
        if (!condition)
            throw new InvalidOperationException(message);
```

Рисунок 3.6 – Реалізація класу перевірок «Guard»

Отже, в результаті розробки компоненту двигуна інформаційної технології автоматизованого тестування «ISUnit» було реалізовано три модуля, які містять загальну логіку запуску, пошуку тестових методів, проведення та отримання результатів процесу тестування. Розроблено програмні функції визначені в процесі моделювання компонента.

3.3 Розробка компоненту програмного розширення інформаційної технології автоматизованого тестування

Компонент програмного розширення містить програмні засоби, застосовуючи які, користувач технології може використовувати стандартні засоби запуску та отримання результатів тестування інтегрованого середовища програмування Visual Studio «Test Explorer». Для інформаційної

технології автоматизованого тестування наявність такого компонета є важливою, оскільки засіб «TestExplorer» є загальноприйнятим для зручного та швидкого запуску процесу тестування локально безпосередньо з інтегрованого середовища.

Компонент складається із двох складових модулів. Перший «ISUnit.TestAdapter» містить вихідний код, що реалізує зв’язок між технологією та засобами інтеграції із сервісом «TestExplorer». Розглянемо деталі його реалізації.

Вихідний код, що дозволяє реалізувати інтеграцію технології та стандартних засобів тестування Visual Studio Test знаходиться у просторі імен «Microsoft.VisualStudio.TestPlatform.ObjectModel» в бібліотеці класів платформи .NET.

Платформа Visual Studio Test — це механізм, який забезпечує можливості автоматизованого тестування в Visual Studio. Платформа надає можливості розширення для тестових фреймворків, для забезпечення загального набору операцій, таких як фільтрування, запуск та виконання процесу тестування [47].

Першою задачею, що виконує модуль є пошук тестових методів, написаних із використанням програмного інтерфейсу технології. Для її реалізації створено клас, що визначає метод інтерфейсу «ITestDiscoverer» «DiscoverTests». Для пошуку застосовуються відповідні функції компоненту двигуна технології. Зрештою результати пошуку записуються у форматі XML у відповідний файл, який буде використаний інтегрованим середовищем для відображення списку знайдених тестових методів користувачеві.

Для реалізації задачі запуску процесу тестування знайдених тестових методів також використовуються відповідні методи компоненту двигуна. Для їх використання необхідно додати посилання на збірку, що його містить й у випадку розповсюдження компоненту розширення включити їх у спільний програмний пакет. Для запуску процесу тестування користувачу необхідно натиснути відповідну кнопку вікна «TestExplorer». Сигнал про ці дії

повідомляється розширенню, яке в свою чергу викликає власний метод «Run», результатом роботи якого є список тестових методів, що були успішно виконані, не запущені, якщо потрапили в списки фільтрації, або завершились із помилкою або неуспішним результатом. Список у форматі XML використовується середовищем для відображення результатів тестування користувачеві.

Для розповсюдження компонента створено необхідні файли конфігурації збірки для пакетного менеджера «Nuget» [48].

Другий модуль компонента «ISUnitTestAdapterInstall» відповідає за можливість розповсюдження методом вбудованих розширень Visual Studio. Його реалізація заключається в створенні файлу налаштувань розширення з типом .vsixmanifest, що містить необхідну інформацію для генерації розширення, таку як автор, версія, розширений опис, мова, наявність ліцензії, зображення, посилання на усі файли, що повинні бути включені в кінцевий файл розширення та деякі інші. Результатом побудови модуля буде файл із розширенням .vsix, встановлюючи який, Visual Studio отримає доступ до технічних засобів компоненту програмного розширення для роботи з платформою «TestExplorer».

Отже, розроблено компонент програмного розширення, що дозволяє виконувати процес пошуку, запуску, фільтрації та отримання результатів автоматизованого тестування, використовуючи стандартні засоби проведення тестування інтегрованого середовища Visual Studio «TestExplorer». Додана можливість поширення компоненту методами пакетного менеджера «Nuget» та програмних розширень Visual Studio.

3.4 Розробка компоненту консольного інтерфейсу інформаційної технології автоматизованого тестування

Компонент консольного інтерфейсу є таким, що безпосередньо доступний користувачеві технології у вигляді додатку, що визначає певні

деталі його розробки. Реалізація компоненту залежить від специфіки задач, що він вирішує. Встановлено, що компонент повинен розповсюджуватись у форматі додатку з розширенням .exe, приймати вхідні аргументи користувача, опрацьовувати їх та викликати відповідні методи компоненту двигуна.

На основі цього, розглянемо особливості реалізації компонента. Платформа .NET дозволяє будувати додатки двух видів зі своїх модулів. Динамічна бібліотека (DLL) містить набір програмних методів та класів, які можливо повторно використовувати. Виконуваний файл (EXE) запускає додаток, що він містить, на апаратному забезпеченні. Для створення виконуваного файлу компонента консольного інтерфейсу необхідно встановити «<OutputType>Exe</OutputType>» в налаштуваннях збірки модуля «isunit-console». Також необхідно, щоб модуль мав клас із методом «Main», що приймає вхідні аргументи в стрічковому форматі [49]. Цей метод є вхідним для початку роботи додатку.

Розроблена технологія має за мету максимальне задоволення потреб користувачів у проведенні тестування. В зв’язку з цим, для зручності застосування системи вона містить широкий список вхідних параметрів, що дозволяють налаштувати процес пошуку та проведення тестування відповідно потребам. Розроблено 40 можливих вхідних параметрів, що можуть встановлюватись користувачем. Для передачі параметру необхідно викликати додаток з допомогою консольного інтерфейсу, вказуючи відповідні параметри у форматі «--test=FULLNAMES». Базовим є перший параметер, що приймає абсолютний або відносний шлях до збірки, котра містить тестові методи, реалізовані з використанням технології. Важливими параметрами також є «--where», що виконує фільтрацію, «--process», що вказує на необхідність паралельного ізольованого запуску тестування в різних потоках, «--framework» – визначає версію платформи .NET на котрій буде запущений процес тестування, «--timeout» – час у мілісекундах, після котрого процес потрібно зупинити, незалажно від його поточного стану та «--result», що приймає шлях до файлу в який необхідно зберегти результати тестування.

Після отримання списку параметрів та їх значень, модуль їх опрацьовує та встановлює відповідні внутрішні налаштування після чого викликаються методи двигуна для пошуку тестових методів або їх запуску. На рисунку 3.7 зображений вихідний код методу «Execute», що містить програмну логіку рішення про запуск пошуку або початку тестування збірок, що були передані в якості параметрів.

```
/// <summary>
/// Executes tests according to the provided commandline options.
/// </summary>
/// <returns></returns>
1 reference
public int Execute()
{
    if (!VerifyEngineSupport(_options))
        return INVALID_ARG;
    DisplayRuntimeEnvironment(_outWriter);
    if (_options.ListExtensions)
        DisplayExtensionList();
    if (_options.InputFiles.Count == 0)
    {
        if (!_options.ListExtensions)
            using (new ColorConsole(ColorStyle.Error))
                Console.Error.WriteLine("Error: no inputs specified");
        return ConsoleRunner.OK;
    }
    DisplayTestFiles();
    TestPackage package = MakeTestPackage(_options);
    DisplayTestFilters();
    TestFilter filter = CreateTestFilter(_options);
    if (_options.Explore)
        return ExploreTests(package, filter);
    else
        return RunTests(package, filter);
}
```

Рисунок 3.7 – Реалізація методу «Execute»

Отже, розроблено компонент консольного інтерфейсу технології, визначено перелік можливих вхідних параметрів та реалізовано їх опрацювання. Компонент виконує усі задачі, визначені в процесі моделювання.

3.5 Розробка компоненту програмного інтерфейсу інформаційної технології автоматизованого тестування

Компонент програмного інтерфейсу визначає та надає програмні засоби написання тестових методів. Якість та різноманіття цих засобів визначає рівень задоволення користувачів системою, адже, в загальному випадку, можливості надані компонентом програмного інтерфейсу є визначальними при виборі технології автоматизованого тестування.

Основними програмними структурами, що використовуються користувачами є атрибути, стверджуючі та обмежувальні твердження. Розглянемо приклади найчастіше вживаних структур.

У програмному інтерфейсі визначено, шо класи, котрі містять тестові методи повинні позначатись атрибутом «TestFixture», а методи – «Test», для того щоб вони були розпізнані технологією як такі, що повинні бути опрацьовані в процесі автоматизованого тестування. Серед інших розроблених атрибутів варто виділити:

- «TestCase» – для визначення тестових методів, можуть бути визначені з параметрами, котрі при запуску процесу тестування будуть використані в якості вхідних даних методу.
- «Ignore» – для маркування тестового методу як такого, що не повинен бути запущеним в процесі тестування.
- «Parallelizable» – для маркування тестового методу як такого, що може бути запущений паралельно.
- «SetUp» – для визначення методу в якому описані дії, що повинні виконатись перед початком тестування класу, що його містить.
- «TearDown» – для визначення методу в якому описані дії, що повинні виконатись після закінчення тестування усіх тестових методів класу, що його містить.
- «Retry» – вказує, що тестовий метод має бути повторно запущений

у разі помилкового виконання, визначену кількість разів.

Загалом розроблено 45 атрибутів, використання яких дозволяє розширити та налаштувати усі можливі аспекти процесу тестування окремих методів, класів та збірок відповідно потребам користувачів.

Стверджуючі та обмежувальні твердження це структури, що використовуються в секції Assert шаблону написання тестових методів «Arrange-Act-Assert» та відповідно виконують задачі перевірки виконання умови модульного тесту та визначення стану отриманої змінної. Варто зазначити, що в більшості випадків умовою правильності виконання тестового методу є відповідність значення отриманної змінної до очікуваного.

Для уніфікації та покращеного розуміння цілей структури, статичні стверджуючі функції містяться в класі «Assert» та в загальному випадку мають форму «Assert.That(очікуване значення, отримане значення)». У випадку невідповідності отриманому значенню до очікуваного, стверджувальні функції викликають виключну ситуацію з типом «AssertionException», що опрацьовується в компоненті двигуна та свідчить про неуспішність перевірки у тестовому методі. Прикладами стверджуючих функцій класу «Assert» є:

- «That» – ствердження, що отриманий результат відповідає очікуваному.
- «AreEqual» – ствердження, що отримане значення є ідентичним очікуваному.
- «Greater» – ствердження, що отримане значення є більшим за очікуване.
- «IsNotNull» – ствердження, що отриманий результат має якесь значення і не є пустим посиланням (null).
- «IsInstanceOf» – ствердження, що отриманого значення є об’єктом очікуваного класу.
- «Catch» – ствердження, що тестовий метод ініціює виключну ситуацію.

Загалом розроблено 42 унікальних стверджуючі функції, більшість з яких мають декілька варіацій з однаковим ім’ям, але різними вхідними параметрами.

В свою чергу, обмежуючі твердження – це функції, що застосовують до отриманної змінної з метою визначення деталей її стану, або додаткової інформації про її значення. Такими обмежуючими функціями є:

- «AttributeConstraint» – перевіряє, що вказаний атрибут присутній у типу, і що значення атрибута задовольняє визначеним обмеженням.
- «CollectionEquivalentConstraint» – використовується для визначення, чи є дві колекції еквівалентними.
- «EmptyStringConstraint» – перевіряє, чи є змінна із стрічковим типом порожньою.
- «StartsWithConstraint» – перевіряє, чи починається змінна із стрічковим типом з очікуваного підрядка.
- «XmlSerializableConstraint» – перевіряє, чи можливо серіалізувати об’єкт у формат XML.

В великій мірі зручність користування програмним інтерфейсом технології визначається легкістю розуміння вихідного коду, що отримується з його застосуванням. З цією метою, окрім обмежуючих функцій, розроблені статичні класи «Is», «Does», «Has», «Throws» та інші, що містять методи, котрі мають логіку, що семантично відповідає комбінації назв класу та методу. Для прикладу розглянемо запис «Assert.That("Hello", Is.InstanceOf(typeof(string)))». У ньому міститься функція перевірки з параметрами: отримане стрічкове значення – «Hello» та обмежуюча комплексна функція «Is.InstanceOf(typeof(string))», що вказує на перевіку чи є отримане значення об’єктом класу «string». Такий запис є зручним для розуміння, адже уся логіка перевірки очевидна із самого запису. Таким чином можливо побудувати і більш складні структури перевірок залежно від потреб користувачів.

Всього розроблено 12 статичних обмежуючих класів, котрі мають різну кількість семантичних функцій перевірок кожен.

Отже, розроблено компонент програмного інтерфейсу інформаційної технології автоматизованого тестування, визначено його формат та складові програмні структури. Реалізовано 45 атрибутів, 42 унікальні стверджувальні функції та 12 статичних обмежуючих класи, що є основними компонентами інтерфейсу. В результаті, використовуючи компонент системи, користувач отримує потужний, зрозумілий та зручний програмний інтерфейс, що розкриває усі можливості застосування технології та проведення якісного автоматизованого тестування.

3.6 Розробка компоненту динамічних аналізаторів інформаційної технології автоматизованого тестування

В процесі моделювання визначено, що реалізація компоненту динамічних аналізаторів значно залежить від обраних інтегрованого середовища та мови програмування.

Однією з переваг використання технологій Microsoft є те, що платформа програмування .NET надає програмні засоби для реалізації динамічних аналізаторів, котрі в результаті можливо безпосередньо інтегрувати в середовище Visual Studio. Тобто буде використовуватись стандартний та оптимізований для інтегрованого середовища вигляд подання результатів аналізу, що є звичним та зрозумілим для користувачів інформаційної технології.

Попри те, що деталі інтеграції динамічних аналізаторів в середовище виконуються платформою .NET, все ще варто визначити яким чином користувачі отримають доступ до компонента. Проаналізовано, що найкращим методом поширення модулів у керованій платформі програмування .NET є найпопулярніший пакетний менеджер Nuget та менеджер розширення Visual Studio Extension. Вони будуть використані для

надання користувачам доступу до компонента динамічних аналізаторів інформаційної технології.

Для реалізації компоненту необхідно визначити його складові модулі. Основним є модуль, що містить безпосередньо реалізації окремих аналізаторів та допоміжний вихідний код, що необхідний для коректної інтеграції із середовищем Visual Studio. Також необхідно реалізувати модуль, що містить налаштування для пакетного менеджера Nuget, що дозволить поширювати основний модуль користувачам у зручному форматі.

В загальному вигляді один аналізатор являє собою набір текстових файлів, що містять вихідний код. Ці файли охоплюють документ коду аналізатора, що задає правило, визначає частину шаблону Arrange-Act-Assert який відноситься до правила, та котрий потрібно дослідити, аналізує випадки його порушення, вказує на місце в коді де було знайдено невідповідність правилу та виконує інші необхідні дії для максимальної ефективності аналізатора. Файл константних значень містить в собі декілька змінних у текстовому форматі, котрі містять інформацію про порушення правила, визначеного аналізатором й будуть відображені користувачеві у випадку знаходження порушення правила у тестовому методі. Третім документом, що належить до аналізатора є файл виправлень коду, що відповідає за надання можливості виконати запропоновані аналізатором зміни в коді, що будуть відповідати встановленим правилам. Цей документ є необов’язковим. Це зумовлено тим, що не у всіх випадках та не для всіх правил можливо коректно встановити виправлення коду, наприклад, якщо помилка має суто семантичний, а не технічний характер, тобто програмний інтерфейс інформаційної технології використано так, як не було закладено розробником компоненту.

До початку реалізації компоненту необхідно дослідити програмний інтерфейс, що надає платформа .NET для імплементації окремих динамічних аналізаторів. Керована платформа містить простір імен «Microsoft.CodeAnalysis» [50], що відноситься до програмних засобів

реалізації динамічних аналізаторів. Визначено, що для того щоб інтегроване середовище могло коректно використати аналізатор, він повинен бути нащадком базового класу «Microsoft.CodeAnalysis.Diagnostics.DiagnosticAnalyzer», тобто при реалізації класу аналізаторів, його необхідно вказувати як дочірній від «DiagnosticAnalyzer». Таким же чином класи виправлень коду повинні наслідуватись від «Microsoft.CodeAnalysis.CodeFixes.CodeFixProvider». Вказані класи є лише невеликою частиною тих класів, методів та змінних, що необхідні для повноцінної реалізації компоненту динамічних аналізаторів інформаційної технології.

Кращою практикою розробки компоненту є створення відповідної документації про усі аналізатори, що він містить. Така документація допомагає користувачеві розуміти семантику використання програмного інтерфейсу інформаційної технології, адже він матиме повний список та детальні пояснення роботи кожного правила. Розробнику, якщо він почав імплементацію компонента зі створення документації, це дасть можливість ще до початку процесу кодування продумати усі правила, що він хоче вказати для системи та визначити усі вимоги й специфіку реалізації кожного аналізатора, що значно полегшить подальший процес розробки. Визначено, що компонент міститиме три типи аналізаторів: структурні, стверджувальні та приховуючі. Таким чином спершу розроблено директорію, що містить документи для кожного з розроблюваних аналізаторів. Кожен текстовий документ містить назву, правило, що він реалізує, повний опис, приклад і метод усунення порушення та атрибути аналізатора: ідентифікаційний номер, жорсткість, тип, посилання на файл, що містить відповідний вихідний код та наявність автоматичного виправлення коду. В результаті описано 28 структурних, 45 стверджувальних та 2 приховуючих динамічних аналізатори у текстовому форматі. Приклад частини такої документації для одного структурного аналізатора наведено на рисунку 3.8. Додатково був створений документ, що

містить повний перелік усіх реалізованих аналізаторів із коротким описом правил, що вони виконують.

```
# ISUnit2040

## Non-reference types for SameAs constraint

| Topic    | Value
| :--      | :--
| Id       | ISUnit2040
| Severity | Error
| Enabled  | True
| Category | Assertion
| Code     | [SameAsOnValueTypesAnalyzer](SameAsOnValueTypes/SameAsOnValueTypesAnalyzer.cs)

## Description

The SameAs constraint always fails on value types as the actual and the expected value cannot be the same reference.

## Motivation

```csharp
var expected = Guid.Empty;
var actual = expected;

Assert.That(actual, Is.SameAs(expected));
```

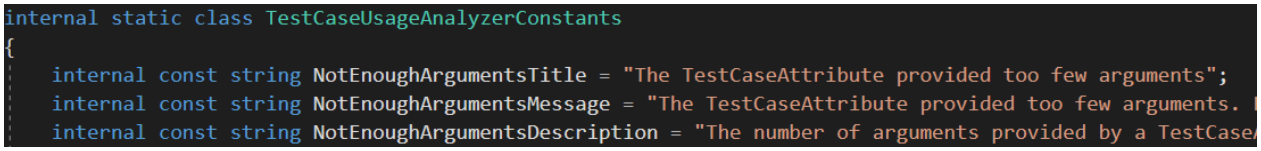
As `Guid` is a `struct`, actual will be a copy of expected but not have the same reference.
```

Рисунок 3.8 – Документація аналізатора «ISUnit2040»

Розглянемо приклад реалізації динамічних аналізаторів кожного типу.

За приклад структурного аналізатора візьмемо «ISUnit1003». Він аналізує правило, котре перевіряє, щоб кількість параметрів тестового методу не була більшою ніж кількість аргументів, що передаються з використанням атрибуту «TestCaseAttribute».

Оскільки повний опис вимог та специфікації аналізатора наведений у відповідній документації для його реалізації скористаємося програмними засобами, що надає простір імен «Microsoft.CodeAnalysis». Спершу необхідно створити файл константних значень, що складається з трьох визначених текстових змінних та містить відповідно заголовок, повідомлення для користувача та детальний опис (рис. 3.9).



```
internal static class TestCaseUsageAnalyzerConstants
{
    internal const string NotEnoughArgumentsTitle = "The TestCaseAttribute provided too few arguments";
    internal const string NotEnoughArgumentsMessage = "The TestCaseAttribute provided too few arguments.
    internal const string NotEnoughArgumentsDescription = "The number of arguments provided by a TestCase
```

Рисунок 3.9 – Клас константних значень аналізатора «ISUnit1003»

Наступним кроком необхідно створити об’єкт класу «DiagnosticDescriptor», що буде містити інформацію про атрибути аналізатора, такі як ідентифікаційний номер, детальний опис, жорсткість, тип та інші (рис. 3.10).

```
private static readonly DiagnosticDescriptor notEnoughArguments = DiagnosticDescriptorCreator.Create(
    id: AnalyzerIdentifiers.TestCaseNotEnoughArgumentsUsage,
    title: TestCaseUsageAnalyzerConstants.NotEnoughArgumentsTitle,
    messageFormat: TestCaseUsageAnalyzerConstants.NotEnoughArgumentsMessage,
    category: Categories.Structure,
    defaultSeverity: DiagnosticSeverity.Error,
    description: TestCaseUsageAnalyzerConstants.NotEnoughArgumentsDescription);
```

Рисунок 3.10 – Створення об’єкту класу «DiagnosticDescriptor» для структурного аналізатора «ISUnit1003»

Для того щоб новий аналізатор можливо було використовувати, його необхідно зареєструвати, використовуючи перевизначений метод «Initialize» батьківського класу «DiagnosticAnalyzer» (рис. 3.11). Таким чином інтегроване середовище автоматично отримає інформацію про необхідність додати описаний клас в список аналізаторів.

```
public override void Initialize(AnalysisContext context)
{
    context.ConfigureGeneratedCodeAnalysis(GeneratedCodeAnalysisFlags.None);
    context.EnableConcurrentExecution();
    context.RegisterSymbolAction(TestCaseUsageAnalyzer.AnalyzeMethod, SymbolKind.Method);
}
```

Рисунок 3.11 – Реалізація перевизначеного методу «Initialize» структурного аналізатора «ISUnit1003»

Ще однією функцією реєстрації є вказання методу класу, що безпосередньо виконує перевірку чи відповідає вихідний код тестового методу встановленому правилу. Для динамічного аналізатора «ISUnit1003» методом перевірки є «AnalyzeMethod». У випадку невідповідності правилу метод, використовуючи програмні засоби простору імен «Microsoft.CodeAnalysis»

(рис. 3.12), повідомляє інтегрованому середовищу про необхідність показати помилку користувачеві.

```
context.ReportDiagnostic(Diagnostic.Create(
    notEnoughArguments,
    attribute.ApplicationSyntaxReference.GetLocation(),
    methodRequiredParameters,
    attributePositionalArguments.Length));
```

Рисунок 3.12 – Виклик методу «ReportDiagnostic» для повідомлення про порушення правила структурного аналізатора «ISUnit1003»

Таким чином було створено 28 структурних динамічних аналізаторів, що охоплюють встановлення перевірок до різноманітних структурних частин тестових методів. Розроблено 8 аналізаторів, що покривають правильність використання паралельного виконання тестових методів. Такі аналізатори відсутні в існуючих реалізаціях систем автоматизованого тестування.

Розглянемо приклад реалізації стверджувального динамічного аналізатора з можливістю автоматичного виправлення вихідного коду на прикладі класу «ISUnit2040». Встановлені вимоги визначають правило, що неправильно використовувати метод «SameAs» у випадку порівняння у блоці Assert шаблону Arrange-Act-Assert двох змінних, що мають тип значень. Правило вказує, що метод «SameAs» повинен використовуватись для порівняння змінних, що мають тип посилання, оскільки метод завжди буде повертати помилку при порівнянні об’єктів із типом значень. Для цього правила можливо встановити і повідомити користувачеві коректне виправлення його коду тестового методу. У випадку порушення правила аналізатор повинен повідомити, що правильно буде застосувати метод порівняння «IsEqualTo» для змінних із типом значень.

Для імплементації автоматичного виправлення вихідного коду необхідно створити документ класу із назвою, що відповідає назві аналізатора,

додавши «CodeFix» в кінці та розмістити його у одну директорією із документом аналізатора.

Для реєстрації виправлення та його зв'язку з аналізаторами необхідно перевизначити змінну «FixableDiagnosticIds» батьківського класу «CodeFixProvider» (рис 3.13). Таким чином інтегроване середовище буде повідомлене про взаємозв'язок цих класів та коректно відобразить можливість виправлення у випадку порушення визначеного правила.

```
public override ImmutableArray<string> FixableDiagnosticIds
    => ImmutableArray.Create(AnalyzerIdentifiers.SameAsOnValueTypes);
```

Рисунок 3.13 –Реєстрація класу виправлення коду стверджувального аналізатора «ISUnit2040»

Імплементація виправлення міститься у перевизначеному методі «RegisterCodeFixesAsync». Цей метод визначає місце та конкретний вихідний код який необхідно замінити із вказанням виправлень, котрі потрібно вставити в тестову функцію.

Таким чином реалізовано 45 стверджувальних динамічних аналізатори серед яких 5 охоплюють область коректного застосування обмежуючих програмних засобів нащадків типу «Constraint». Такі аналізатори відсутні у існуючих реалізаціях систем автоматизованого тестування.

Останнім типом динамічних аналізаторів, що реалізовані в інформаційній технології є приховуючі. Їх існування зумовлене тим, що в Microsoft Visual Studio реалізований величезний набір готових аналізаторів, деякі з котрих можуть невідповідати семантиці програмного інтерфейсу інформаційної технології. Також існують файли з розширенням «.ruleset» в яких визначається котрі з наявних аналізаторів застосовувати при написанні вихідного коду в кожному окремому проєкті [51]. Приховуючі аналізатори розроблюваної інформаційної технології визначають групу існуючих у Visual Studio аналізаторів, результати котрих необхідно приховати в тестових

модулях проєктів, що використовують технологію «ISUnit» для автоматизації тестування. В результаті в файлі з розширенням «.ruleset» у відповідному модулі необхідно відзначити не кожен окремий аналізатор із групи, а лише той, що надає інформаційна технологія.

Розглянемо приклад розробки приховуючого динамічного аналізатора на прикладі класу «ISUnit3001». Він призначений для подавлення групи помилок пов’язаних із відсутністю перевірки змінної на нулове значення (null), у випадку якщо ця перевірка відбувалась раніше з використанням конструкцій «Assert.NotNull», «Assert.IsNotNull» або «Assert.That call», що надаються програмним інтерфейсом інформаційної технології.

Для реалізації такого динамічного аналізатора він повинен бути нащадком класу «DiagnosticSuppressor» з простору імен «Microsoft.CodeAnalysis.Diagnostics». Таким чином вказується, що цей аналізатор є приховуючим. Далі необхідно перевизначити змінну «SupportedSuppressions» (рис. 3.14), вказавши їй у якості значення об’єкт, що містить масив ідентифікаційних номерів аналізаторів Visual Studio, що приховуються та пояснення причини приховування.

```
public static ImmutableDictionary<string, SuppressionDescriptor> SuppressionDescriptors { get; } =
    CreateSuppressionDescriptors(
        "CS8600", // Converting null literal or possible null value to non-nullable type.
        "CS8601", // Possible null reference assignment.
        "CS8602", // Dereference of a possibly null reference.
        "CS8603", // Possible null reference return.
        "CS8604", // Possible null reference argument.
        "CS8605", // Unboxing a possibly null value.
        "CS8606", // Possible null reference assignment to iteration variable.
        "CS8607", // A possible null value may not be passed to a target marked with the [DisallowNull] attribute.
        "CS8629"); // Nullable value type may be null.

0 references | 0 changes | 0 authors, 0 changes
public override ImmutableArray<SuppressionDescriptor> SupportedSuppressions { get; } =
    ImmutableArray.CreateRange(SuppressionDescriptors.Values);
```

Рисунок 3.14 – Вказання групи аналізаторів, що можуть приховатись приховуючим аналізатором «ISUnit3001»

Зрештою потрібно реалізувати метод батьківського класу «ReportSuppressions», в якому виконується перевірка чи варто приховувати повідомлення компілятора у випадку, якщо була зафіксована одна із

визначених невідповідностей. Якщо знайдено, що знайдена в результаті діагностики помилка являється очікуваним результатом використання програмного інтерфейсу інформаційної технології, то результат діагностики не відображається, для чого необхідно викликати метод «ReportSuppression».

Другим модулем, з якого складається компонент динамічних аналізаторів є модуль програмного розширення. Він містить лише документ налаштувань програмного розширення. Цей документ дає можливість розповсюджувати програмний модуль динамічних аналізаторів у вигляді збірки .dll з використанням пакетних менеджерів, наприклад, Nuget.

В документі з розширенням «.vsixmanifest» описана інформація, що необхідна для коректного розповсюдження. У ньому визначена назва пакету, його версія, ідентифікаційний номер, детальний опис вмісту, мову користування, ліцензію, зображення, що буде використовуватись в пакетному менеджері, вимоги до версії Visual Studio на яку розширення буде встановлене й залежності та посилання на усі модулі, що потрібно включити в пакет.

Таким чином, використовуючи додаткову конфігурацію пакетного менеджера можливо поширювати компонент динамічних аналізаторів як частину інформаційної технології автоматизованого тестування «ISUnit».

Отже, результатом розробки компоненту є модуль динамічних аналізаторів та модуль програмного розширення для їх розповсюдження та інтеграції із середовищем розробки. Реалізовано 28 структурних, 45 стверджувальних та 2 приховуючих динамічних аналізатори, котрі своїми визначеними правилами покривають велику частину можливих помилок при імплементації тестових методів з використанням інформаційної технології автоматизованого тестування «ISUnit». Серед них розроблено 13 динамічних аналізатори, аналогів яких немає в існуючих програмних реалізаціях.

3.7 Тестування та аналіз результатів роботи програмного забезпечення

Тестування розробленого програмного продукту інформаційної технології автоматизованого тестування заключається у перевірці коректності роботи кожного окремого її компоненту та засобів взаємодії між ними.

Попередньо усі компоненти системи були сформовані у вигляді окремих збірок для їх зручного розповюдження та передачі з допомогою обраного пакетного менеджера – «Nuget». Для цього на попередніх етапах реалізації були розроблені спеціальні файли конфігурації, що дозволили створити із кожного компоненту, що планується розповсюджуватись – спеціальну збірку, котрі були інтегровані в локальну мережу пакетного менеджера.

Спершу варто виконати тестування компонента програмного інтерфейсу, оскільки користувач взаємодіє із ним найбільше та процес автоматизованого тестування розпочинається із реалізації модульних тестів. Для цього у інтегрованому середовищі Visual Studio 2019 створимо новий проєкт із типом бібліотеки класів, що після його збірки буде сформований у формат .dll. Далі до проєкту необхідно додати посилання на збірку, що містить компонент. Для цього скористаємось вбудованим в інтегроване середовище пакетним менеджером «Nuget» (рис. 3.15).

Отримавши доступ до програмного інтерфейсу створимо новий файл «TestClass.cs» та реалізуємо у ньому клас «TestClass» та метод «TestMethod». Для того шоб ці елементи враховувались в процесі тестування, необхідно їх промаркувати атрибутами, відповідно «TestFixture» та «Test» (рис. 3.16). Таким чином доведено коректність роботи компоненту програмного інтерфейсу технології.

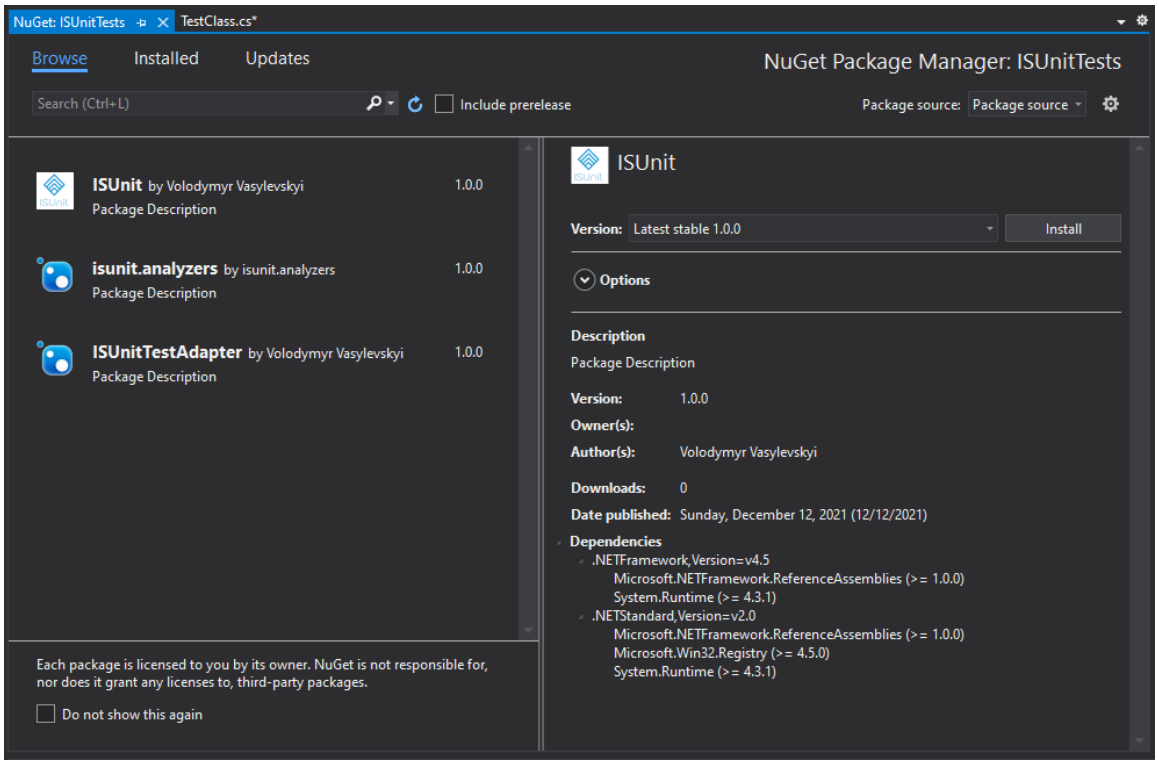


Рисунок 3.15 – Розроблені збірки для пакетного менеджера «Nuget»

```
using ISUnit.Framework;
using System;

namespace ISUnitTests
{
    [TestFixture]
    0 references
    public class TestClass
    {
        [Test]
        0 references
        public void TestMethod()
        {
            // Arrange
            int a = 10;

            // Assert
            Assert.That(a, Is.Positive);
        }
    }
}
```

Рисунок 3.16 – Тестовий метод, реалізований з використанням технології

Далі можливо перевірити правильність виконання компоненту динамічних аналізаторів. Для цього у створений проєкт підключимо відповідну збірку аналізаторів. Реалізуємо новий метод у якому порушимо семантичне правило у секції «assert». В результаті отримуємо повідомлення про порушення від динамічного аналізатора «ISUnit2005» (рис 3.17).

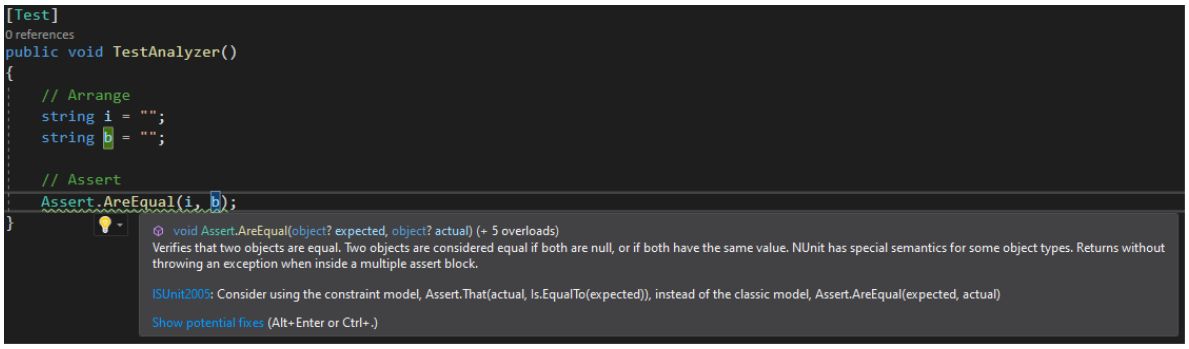


Рисунок 3.17 – Відображення результатів аналізу динамічного аналізатора «ISUnit2005»

Даний аналізатор містить виправлення коду, після його застосування вихідний код, що визначає некоректну семантику автоматично змінюється на коректний (рис. 3.18). Результат свідчить, що компонент динамічних аналізатор визначає, відображає та має можливість виправляти знайдені порушення описаних правил програмного інтерфейсу, а отже працює коректно.

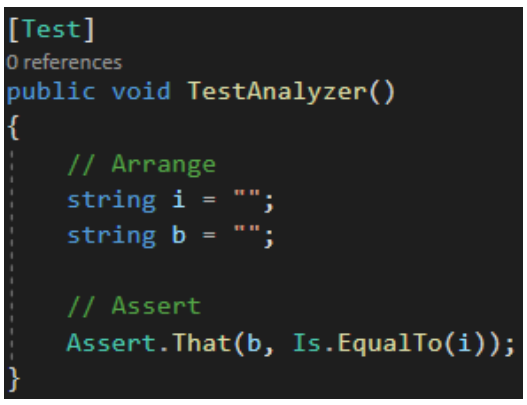


Рисунок 3.18 – Відображення результатів аналізу динамічного аналізатора «ISUnit2015»

Доведено, що компоненти, котрі відповідають за реалізацію тестових методів працюють коректно. На основі цього можливо перевірити компоненти, що відповідають за процес проведення тестування та отримання його результатів. Для цього додамо посилання на компонент програмного розширення та перевіримо чи інтегрується він із стандартною платформою тестування «Test Explorer».

На рисунку 3.19 видно, що компонент коректно знаходить усі тестові методи, котрі написані з використанням розробленої технології та успішно їх запускає і отримує результати тестування, що свідчить про правильність роботи компоненту двигуна, який використовується програмним розширенням.

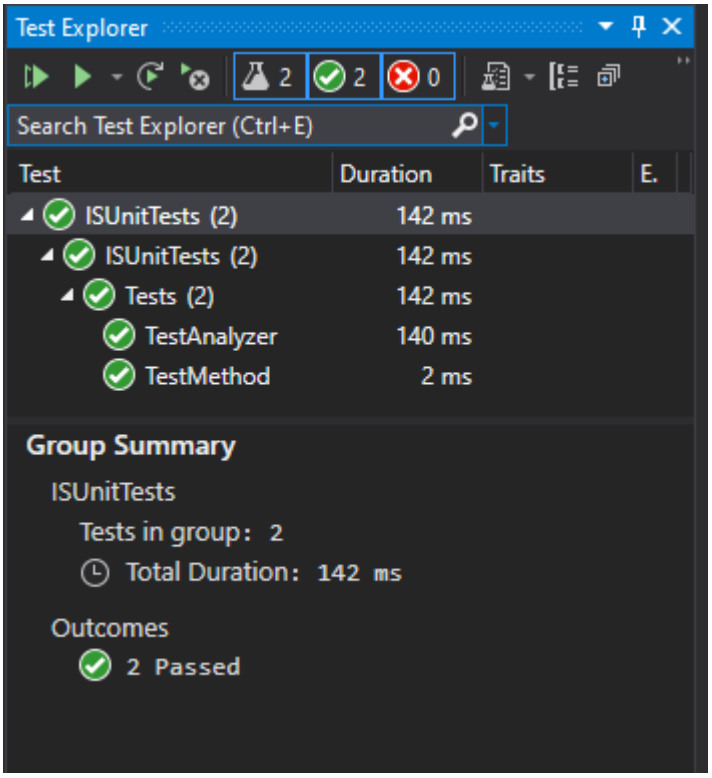


Рисунок 3.19 – Інтеграція технології із стандартними засобами середовища Visual Studio 2019 тестування «Test Explorer»

Також варто виконати перевірку коректності роботи компоненту консольного інтерфейсу. Для цього побудуємо отриманий в результаті

попередніх кроків проєкт. В результаті отримаємо збірку у форматі «.dll», котру й будемо тестувати, використовуючи консольний інтерфейс технології. Для отримання результатів необхідно запустити у програмній стрічці консолі команду «isunit-console ISUnitTests.dll». По завершенню тестування у інтерфейсі прописуються його результати. З рисунку 3.20 видно, що компонент коректно виконує процес запуску, опрацювання та отримання результатів тестування.

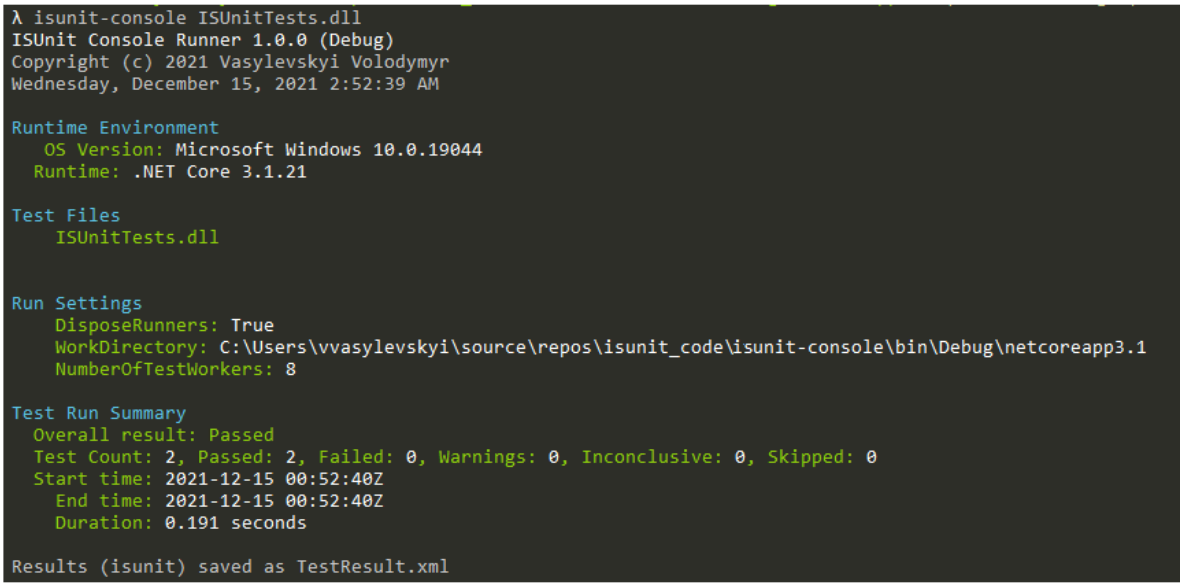


Рисунок 3.20 – Результати виконання процесу тестування з використанням компоненту консольного інтерфейсу технології

Іншим показником якості системи є швидкодія виконання процесу тестування.

Для тестування швидкодії було створено 5 різних тестових збірки, кожна з яких містить 4 класи із 25 методами. Таким чином реалізовано 500 тестових методів, що будуть приймати участь у процесі тестування зі застосуванням опції паралельного виконання. Тестування проводилось у кількості десяти повторень. Середні значення загального часу на проведення процесу тестування в порівнянні із існуючими реалізаціями наведено на рисунку 3.21.

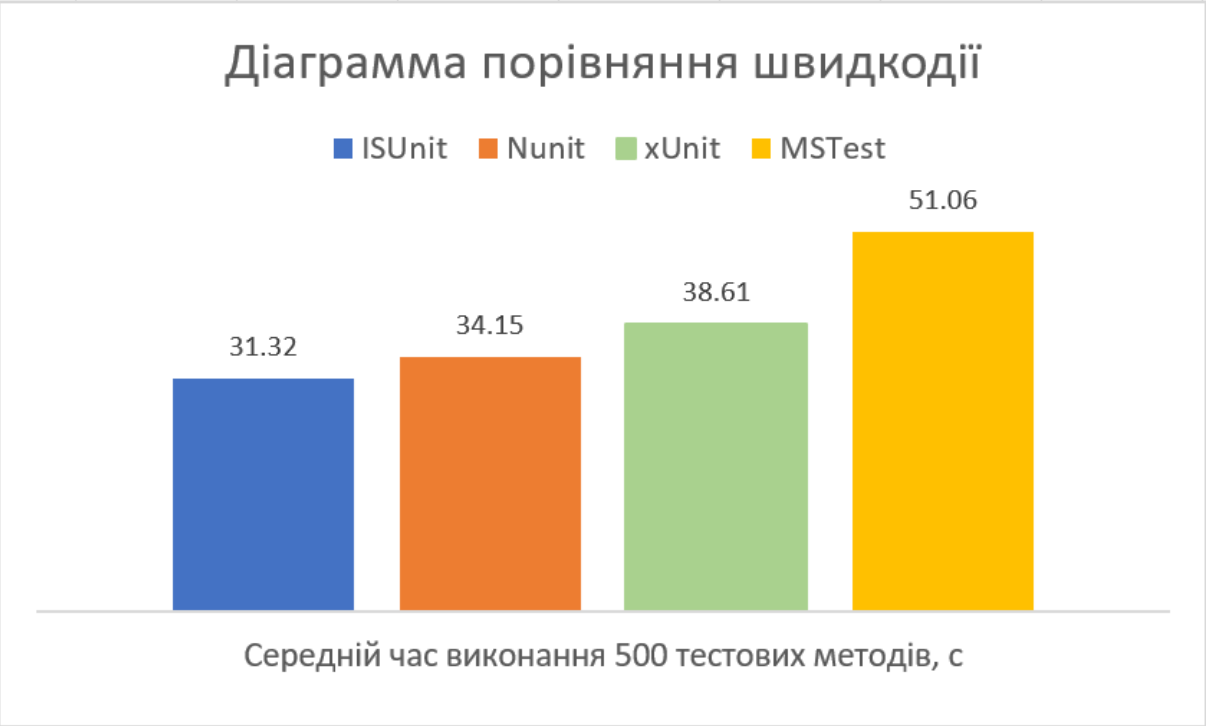


Рисунок 3.21 – Діаграмма порівняння середнього часу виконання тестування 500 тестових методів розробленої технології та аналогів

Отримані результати свідчать, що розроблена технологія має високий рівень швидкодії у порівнянні з аналогами. Середній час на виконання п’ятиста тестових методів з використанням технології «ISUnit» склав – 31.32 секунди. Для визначених аналогів час на виконання тієї ж вибірки є: «NUnit» – 34.15 с., «xUnit» – 38.61 с., «MSTest» – 51.06 с. Тобто показники реалізованого програмного забезпечення виявились кращими на 9%, 23% та 63% відповідно у порівнянні з аналогічними технологіями.

Отже, було визначено, що усі компоненти технології працюють коректно та повнофункціонально, що відповідає вимогам до роботоздатності системи. Також знайдено, що технологія має високий рівень швидкодії у порівнянні з аналогами.

3.8 Висновок

Отже, розроблено усі компоненти інформаційної технології, що були визначені в процесі моделювання. Отриманий програмний продукт відповідає усім вхідним вимогам. Тестування довело, що реалізоване програмне забезпечення має високу швидкодію, повнофункціональне та зручне у користуванні, що в повній мірі відповідає критеріям задоволення інформаційних потреб користувачів у проведенні процесу автоматизованого тестування.

4 ЕКОНОМІЧНА ЧАСТИНА

4.1 Комерційний та технологічний аудит науково-технічної розробки

Метою даного розділу є проведення технологічного аудиту, в даному випадку нового програмного продукту інформаційної технології автоматизованого тестування. Особливістю програми є те, що дана технологія є мультикомпонентною програмною системою модульного тестування «ISUnit», що використовується розробниками (програмістами) з метою покращення процесу проведення автоматизованого тестування на проєкті.

Аналогом розробки є LDRA TESTBed – 42000 грн одна ліцензія або JetBrains Resharper, річна підписка на систему складає 8000 грн.

Для проведення комерційного та технологічного аудиту залучають не менше 3-х незалежних експертів. Оцінювання науково-технічного рівня розробки та її комерційного потенціалу рекомендується здійснювати із застосуванням п’ятибальної системи оцінювання за 12-ма критеріями, у відповідності із табл. 4.1.

Таблиця 4.1 – Рекомендовані критерії оцінювання комерційного потенціалу розробки та їх можлива бальна оцінка

Бали (за 5-ти бальною шкалою)					
Кри-терій	0	1	2	3	4
Технічна здійсненність концепції					
1	Достовірність концепції не підтверджена	Концепція підтверджена експертними висновками	Концепція підтверджена розрахунками	Концепція перевірена на практиці	Перевірено роботоздат-ність продук-ту в реаль-них умовах
2	Багато аналогів на малому ринку	Мало аналогів на малому ринку	Кілька аналогів на великому ринку	Один аналог на великому ринку	Продукт не має аналогів на великому ринку

Продовження табл. 4.1

Ринкові переваги					
3	Ціна продукту значно вища за ціни аналогів	Ціна продукту дещо вища за ціни аналогів	Ціна продукту приблизно дорівнює цінам аналогів	Ціна продукту дещо нижче за ціни аналогів	Ціна продукту значно нижче за ціни аналогів
4	Технічні та споживчі властивості продукту значно гірші, ніж в аналогів	Технічні та споживчі властивості продукту трохи гірші, ніж в аналогів	Технічні та споживчі властивості продукту на рівні аналогів	Технічні та споживчі властивості продукту трохи кращі, ніж в аналогів	Технічні та споживчі властивості продукту значно кращі, ніж в аналогів
5	Експлуатаційні витрати значно вищі, ніж в аналогів	Експлуатаційні витрати дещо вищі, ніж в аналогів	Експлуатаційні витрати на рівні експлуатаційних витрат аналогів	Експлуатаційні витрати трохи нижчі, ніж в аналогів	Експлуатаційні витрати значно нижчі, ніж в аналогів
Ринкові перспективи					
6	Ринок малий і не має позитивної динаміки	Ринок малий, але має позитивну динаміку	Середній ринок з позитивною динамікою	Великий стабільний ринок	Великий ринок з позитивною динамікою
7	Активна конкуренція великих компаній на ринку	Активна конкуренція	Помірна конкуренція	Незначна конкуренція	Конкурентів немає
Практик на здійсненність					
8	Відсутні фахівці як з технічної, так і з комерційної реалізації ідеї	Необхідно наймати фахівців або витрачати значні кошти та час на навчання наявних фахівців	Необхідне незначне навчання фахівців та збільшення їх штату	Необхідне незначне навчання фахівців	Є фахівці з питань як з технічної, так і з комерційної реалізації ідеї
9	Потрібні значні фінансові ресурси, які відсутні. Джерела фінансування ідеї відсутні	Потрібні незначні фінансові ресурси. Джерела фінансування відсутні	Потрібні значні фінансові ресурси. Джерела фінансування є	Потрібні незначні фінансові ресурси. Джерела фінансування є	Не потребує додаткового фінансування
10	Необхідна розробка нових матеріалів	Потрібні матеріали, що використовуються у військово-промисловому комплексі	Потрібні дорогі матеріали	Потрібні досяжні та дешеві матеріали	Всі матеріали для реалізації ідеї відомі та давно використовуються у виробництві

Продовження табл. 4.1

11	Термін реалізації ідеї більший за 10 років	Термін реалізації ідеї більший за 5 років. Термін окупності інвестицій більше 10-ти років	Термін реалізації ідеї від 3-х до 5-ти років. Термін окупності інвестицій більше 5-ти років	Термін реалізації ідеї менше 3-х років. Термін окупності інвестицій від 3-х до 5-ти років	Термін реалізації ідеї менше 3-х років. Термін окупності інвестицій менше 3-х років
12	Необхідна розробка регламентних документів та отримання великої кількості дозвільних документів на виробництво та реалізацію продукту	Необхідно отримання великої кількості дозвільних документів на виробництво та реалізацію продукту, що вимагає значних коштів та часу	Процедура отримання дозвільних документів для виробництва та реалізації продукту вимагає незначних коштів та часу	Необхідно тільки повідомлення відповідним органам про виробництво та реалізацію продукту	Відсутні будь-які регламентні обмеження на виробництво та реалізацію продукту

Усі дані по кожному параметру занесено в таблиці 4.2

Таблиця 4.2 – Результати оцінювання комерційного потенціалу розробки

Критерії оцінювання	ПІБ експертів		
	Експерт 1	Експерт 2	Експерт 3
	Бали		
Технічна здійсненність концепції	4	4	4
Наявність аналогів на ринку	2	4	2
Цінова політика	4	2	4
Технічні та споживчі властивості виробу	4	4	3
Експлуатаційні витрати	4	3	4
Ринок збуту	2	4	3
Конкурентоспроможність	4	3	4
Фахівці з технічної і комерційної реалізації	4	4	3
Фінансування	3	4	4
Матеріально-технічна база	3	3	4
Термін реалізації ідеї	3	4	4
Супровідна документація	4	4	3
Сума	41	43	42
Середньоарифметична сума балів	(41+43+42) / 3 = 42		

За даними таблиці 4.2 можна зробити висновок щодо рівня комерційного потенціалу даної розробки. Для цього доцільно скористатись рекомендаціями, наведеними в таблиці 4.3.

Таблиця 4.3 - Рівні комерційного потенціалу розробки

Середньоарифметична сума балів СБ , розрахована на основі висновків експертів	Рівень комерційного потенціалу розробки
0-10	Низький
11-20	Нижче середнього
21-30	Середній
31-40	Вище середнього
41-48	Високий

Як видно з таблиці, рівень комерційного потенціалу розроблюваного нового програмного продукту є високим, що досягається за рахунок того, що розроблювана технологія, порівняно із існуючими, містить покращений статичний та динамічний аналіз написаних тестових методів. Також перевага розробки полягає в удосконалені інформаційної моделі динамічного аналізу тестових модулів, що відрізняється від існуючих введенням додаткових аналізаторів (близько 20) тестових методів, що забезпечує підвищення ефективності всього процесу модульного тестування.

4.2 Прогнозування витрат на виконання науково-дослідної (дослідно-конструкторської) роботи

Основна заробітна плата розробників розраховується за формулою:

$$З_о = \frac{M}{T_р} \cdot t, \qquad (4.1)$$

де М – місячний посадовий оклад конкретного розробника (дослідника), грн.;

$Т_р$ – число робочих днів в місяці, 23 днів;

t – число днів роботи розробника (дослідника).

Результати розрахунків зведемо до таблиці 4.1.

Таблиця 4.1 – Основна заробітна плата розробників

Найменування посади	Місячний посадовий оклад, грн.	Оплата за робочий день, грн.	Число днів роботи	Витрати на заробітну плату, грн.
Керівник проекту	35000	1521,74	42	63913,043
Інженер	33000	1434,78	42	60260,870
Всього				124173,91

Оскільки в даному випадку розробляється програмний продукт, то розробник виступає одночасно і основним робітником, і тестувальником розроблюваного програмного продукту.

Додаткову заробітна плата прийнято розраховувати як 13 % від основної заробітної плати розробників та робітників:

$$З_д = З_о \cdot 13\ \% / 100\ \% \qquad (4.2)$$

$З_д = (124173,91 \cdot 13\ \% / 100\ \%) = 16142,61$ (грн.)

Згідно діючого законодавства нарахування на заробітну плату складають 22 % від суми основної та додаткової заробітної плати.

$$Н_з = (З_о + З_д) \cdot 22\ \% / 100\% \qquad (4.3)$$

$Н_з = (124173,91 + 16142,61) \cdot 22\ \% / 100\ \% = 30869,63$ (грн.)

Оскільки для розроблювального пристрою не потрібно витрачати матеріали та комплектуючі, то витрати на матеріали і комплектуючі дорівнюють нулю.

Амортизація обладнання, що використовувалось для розробки в спрощеному вигляді амортизація обладнання, що використовувалась для розробки розраховується за формулою:

$$A = \frac{Ц}{Тв} \cdot \frac{t_{вик}}{12} \text{ [грн.].} \quad (4.4)$$

де Ц – балансова вартість обладнання, грн.;

Т – термін корисного використання обладнання згідно податкового законодавства, років;

$t_{вик}$ – термін використання під час розробки, місяців;

Розрахуємо, для прикладу, амортизаційні витрати на комп'ютер балансова вартість якого становить 45000 грн., термін його корисного використання згідно податкового законодавства – 2 роки, а термін його фактичного використання – 1,83 міс.

$$A_{обл} = \frac{45000}{2} \times \frac{1{,}83}{12} = 3423{,}913 \; грн.$$

Аналогічно визначаємо амортизаційні витрати на інше обладнання та приміщення. Розрахунки заносимо до таблиці 4.2. Для розрахунку амортизації нематеріальних ресурсів використовується формула:

$$A_{н.р.} = Ц_{н.р.} * Н_а * \frac{t_{вик.}}{12} \quad (4.5)$$

Норму амортизації $Н_а$ приймемо за 11 %.

Таблиця 4.2 – Амортизаційні відрахування матеріальних і нематеріальних ресурсів для розробників

Найменування обладнання	Балансова вартість, грн.	Строк корисного використання, років	Термін використання обладнання, місяців	Амортиза-ційні відрахуван-ня, грн.
Комп'ютер (Lenovo ThinkPad T470P, Dell 27)	45000	2	1,83	3423,913
Приміщення	550000	20	1,83	4184,783
Ліцензійна ОС, та спеціалізовані ліцензійні нематеріальні ресурси (Visual Studio Professional Edition)	9050	-	1,83	151,489
Всього				7760,18

Тарифи на електроенергію для непобутових споживачів (промислових підприємств) відрізняються від тарифів на електроенергію для населення. При цьому тарифи на розподіл електроенергії у різних постачальників (енергорозподільних компаній), будуть різними. Крім того, розмір тарифу залежить від класу напруги (1-й або 2-й клас). Тарифи на розподіл електроенергії для всіх енергорозподільних компаній встановлює Національна комісія з регулювання енергетики і комунальних послуг (НКРЕКП). Витрати на силову електроенергію розраховуються за формулою:

$$В_е = В \cdot П \cdot Ф \cdot К_п, \quad (4.6)$$

де В – вартість 1 кВт-години електроенергії з ПДВ для 1 класу підприємства, В = 2,01 грн./кВт;

П – встановлена потужність обладнання, кВт. П = 0,7 кВт;

Ф – фактична кількість годин роботи обладнання, годин;

$К_п$ – коефіцієнт використання потужності, $К_п$ = 0,9;

$В_е = 0,9 \cdot 0,7 \cdot 8 \cdot 42 \cdot 2,01 = 425,4768$ (грн.)

До статті «Інші витрати» належать витрати, які не знайшли відображення у зазначених статтях витрат і можуть бути віднесені безпосередньо на собівартість досліджень за прямими ознаками. Витрати за статтею «Інші витрати» розраховуються як 50…100% від суми основної заробітної плати дослідників:

$$I_в = (З_о + З_р) \cdot \frac{Н_{ів}}{100\%}, \quad (4.7)$$

де $Н_{ів}$ – норма нарахування за статтею «Інші витрати».

$I_в$ = 124173,91 * 115% / 100% = 142800 (грн.)

До статті «Накладні (загальновиробничі) витрати» належать: витрати, пов'язані з управлінням організацією; витрати на винахідництво та раціоналізацію; витрати на підготовку (перепідготовку) та навчання кадрів; витрати, пов'язані з набором робочої сили; витрати на оплату послуг банків; витрати, пов'язані з освоєнням виробництва продукції; витрати на науково-технічну інформацію та рекламу та ін. Витрати за статтею «Накладні (загальновиробничі) витрати» розраховуються як 100…150% від суми основної заробітної плати дослідників:

$$Н_{нзв} = (З_о + З_р) \cdot \frac{Н_{нзв}}{100\%}, \quad (4.8)$$

де $Н_{нзв}$ – норма нарахування за статтею «Накладні (загальновиробничі) витрати».

$Н_{нзв}$ = 124173,91 * 128 % / 100 % = 158943 (грн.)

Сума всіх попередніх статей витрат дає загальні витрати на проведення науково-дослідної роботи:

$B_{заг}$ = 124173,91+16142,61+30869,63+7760,18+425,48+142800+158943 =
= 481114,43 грн.

Загальні витрати на завершення науково-дослідної (науково-технічної) роботи та оформлення її результатів розраховуються *ЗВ*, визначається за формулою:

$$ЗВ = \frac{B_{заг}}{\eta} \text{ (грн)}, \qquad (5.9)$$

де η – коефіцієнт, який характеризує етап (стадію) виконання науково-дослідної роботи.

Так, якщо науково-технічна розробка знаходиться на стадії: науково-дослідних робіт, то η=0,1; технічного проектування, то η=0,2; розробки конструкторської документації, то η=0,3; розробки технологій, то η=0,4; розробки дослідного зразка, то η=0,5; розробки промислового зразка, то η=0,7; впровадження, то η=0,9. Оберемо η = 0,7, так як розробка, на даний момент, знаходиться на стадії промислового зразка:

ЗВ = 481114,43 / 0,7 = 687306 грн.

4.3 Розрахунок економічної ефективності науково-технічної розробки за її можливої комерціалізації потенційним інвестором

В ринкових умовах узагальнювальним позитивним результатом, що його може отримати потенційний інвестор від можливого впровадження результатів тієї чи іншої науково-технічної розробки, є збільшення у потенційного інвестора величини чистого прибутку. Саме зростання чистого

прибутку забезпечить потенційному інвестору надходження додаткових коштів, дозволить покращити фінансові результати його діяльності, підвищить конкурентоспроможність та може позитивно вплинути на ухвалення рішення щодо комерціалізації цієї розробки.

Для того, щоб розрахувати можливе зростання чистого прибутку у потенційного інвестора від можливого впровадження науково-технічної розробки необхідно:

а) вказати, з якого часу можуть бути впроваджені результати науково-технічної розробки;

б) зазначити, протягом скількох років після впровадження цієї науково-технічної розробки очікуються основні позитивні результати для потенційного інвестора (наприклад, протягом 3-х років після її впровадження);

в) кількісно оцінити величину існуючого та майбутнього попиту на цю або аналогічні чи подібні науково-технічні розробки та назвати основних суб'єктів (зацікавлених осіб) цього попиту;

г) визначити ціну реалізації на ринку науково-технічних розробок з аналогічними чи подібними функціями.

При розрахунку економічної ефективності потрібно обов'язково враховувати зміну вартості грошей у часі, оскільки від вкладення інвестицій до отримання прибутку минає чимало часу. При оцінюванні ефективності інноваційних проектів передбачається розрахунок таких важливих показників:

- абсолютного економічного ефекту (чистого дисконтованого доходу);
- внутрішньої економічної дохідності (внутрішньої норми дохідності);
- терміну окупності (дисконтованого терміну окупності).

Аналізуючи напрямки проведення науково-технічних розробок, розрахунок економічної ефективності науково-технічної розробки за її можливої комерціалізації потенційним інвестором можна об'єднати, враховуючи визначені ситуації з відповідними умовами.

В випадку розробки чи суттєвого вдосконалення програмного засобу (програмного забезпечення, програмного продукту) для використання масовим споживачем майбутній економічний ефект буде формуватися на основі таких даних:

$$\Delta П_i = (\pm \Delta Ц_o \cdot N + Ц_o \cdot \Delta N)_i \cdot \lambda \cdot \rho \cdot (1 - \frac{\vartheta}{100}), \quad (4.10)$$

де $\pm\Delta Ц_o$ – зміна вартості програмного продукту (зростання чи зниження) від впровадження результатів науково-технічної розробки в аналізовані періоди часу;

N – кількість споживачів які використовували аналогічний продукт у році до впровадження результатів нової науково-технічної розробки;

$Ц_o$ – основний оціночний показник, який визначає діяльність підприємства у даному році після впровадження результатів наукової розробки, $Ц_o = Ц_б \pm \Delta Ц_o$;

Цб – вартість програмного продукту у році до впровадження результатів розробки;

ΔN – збільшення кількості споживачів продукту, в аналізовані періоди часу, від покращення його певних характеристик;

λ – коефіцієнт, який враховує сплату податку на додану вартість. Ставка податку на додану вартість дорівнює 20%, а коефіцієнт $\lambda = 0{,}8333$.

p – коефіцієнт, який враховує рентабельність продукту;

ϑ – ставка податку на прибуток, у 2021 році ϑ =18%.

Припустимо, що при прогнозованій ціні 6000 грн. за одиницю виробу, термін збільшення прибутку складе 3 роки. Після завершення розробки і її вдосконалення, можна буде підняти її ціну на *750* грн. Кількість одиниць реалізованої продукції також збільшиться: протягом першого року – на 1200 шт., протягом другого року – на 1500 шт., протягом третього року на 1700 шт. До моменту впровадження результатів наукової розробки реалізації продукту не було:

*$\Delta П_1$ = (0*750 + (6000 + 750)*1200)* 0,8333* 0,37) * (1 - 0,18) = 1820399,927 грн.*

*$\Delta П_2$ = (0*750 + (6000 + 750)*(1200+1500)* 0,8333* 0,37) * (1 - 0,18) = 4607887,316 грн.*

*$\Delta П_3$ = (0*750 + (6000 + 750)*(1200+1500+1700)* 0,8333* 0,37) * (1 - 0,18) = 7509149,700 грн.*

Отже, комерційний ефект від реалізації результатів розробки за три роки складе 13937436,94 грн.

Розраховуємо приведену вартість збільшення всіх чистих прибутків *ПП*, що їх може отримати потенційний інвестор від можливого впровадження та комерціалізації науково-технічної розробки:

$$ПП = \sum_{1}^{T} \frac{\Delta П_i}{(1+\tau)^t}, \qquad (5.11)$$

де $\Delta П_i$ – збільшення чистого прибутку у кожному із років, протягом яких виявляються результати виконаної та впровадженої науково-дослідної (науково-технічної) роботи, грн;

Т – період часу, протягом якою виявляються результати впровадженої науково-дослідної (науково-технічної) роботи, роки;

τ – ставка дисконтування, за яку можна взяти щорічний прогнозований рівень інфляції в країні, $\tau = 0,05...0,15$;

t – період часу (в роках).

Збільшення прибутку ми отримаємо починаючи з першого року:

ПП = (1820399,927/(1+0,1)^1)+(4607887,316/(1+0,1)^2)+(7509149,700/(1+0,1)^3) = 1654909,02 + 3808171,34 + 5641735,31 = 11104815,67 грн.

Далі розраховують величину початкових інвестицій *PV*, які потенційний інвестор має вкласти для впровадження і комерціалізації науково-технічної розробки. Для цього можна використати формулу:

$$PV = k_{інв} * ЗВ, \quad (4.12)$$

де $k_{інв}$ – коефіцієнт, що враховує витрати інвестора на впровадження науково-технічної розробки та її комерціалізацію. Це можуть бути витрати на підготовку приміщень, розробку технологій, навчання персоналу, маркетингові заходи тощо; зазвичай $k_{інв}=2...5$, але може бути і більшим;

ЗВ – загальні витрати на проведення науково-технічної розробки та оформлення її результатів, грн.

$PV = 2 * 687306 = 1374612{,}65$ *грн.*

Тоді абсолютний економічний ефект $E_{абс}$ або чистий приведений дохід *(NPV, Net Present Value)* для потенційного інвестора від можливого впровадження та комерціалізації науково-технічної розробки становитиме:

$$E_{абс} = ПП - PV, \quad (4.13)$$

$E_{абс} = 11104815{,}67 - 1374612{,}65 = 9730203{,}02$ грн.

Оскільки $E_{абс} > 0$ то вкладання коштів на виконання та впровадження результатів даної науково-дослідної (науково-технічної) роботи може бути доцільним.

Для остаточного прийняття рішення з цього питання необхідно розрахувати внутрішню економічну дохідність або показник внутрішньої но-рми дохідності *(IRR, Internal Rate of Return)* вкладених інвестицій та порівняти її з так званою бар'єрною ставкою дисконтування, яка визначає ту мінімальну

внутрішню економічну дохідність, нижче якої інвестиції в будь-яку науково-технічну розробку вкладати буде економічно недоцільно.

Розрахуємо відносну (щорічну) ефективність вкладених в наукову розробку інвестицій $E_в$. Для цього використаємо формулу:

$$E_в = \sqrt[T_ж]{1 + \frac{E_{абс}}{PV}} - 1, \tag{4.14}$$

$T_ж$ – життєвий цикл наукової розробки, роки.

$$E_в = \sqrt[3]{(1 + 9730203{,}02/1374612{,}65} - 1 = 1{,}007$$

Визначимо мінімальну ставку дисконтування, яка у загальному вигляді визначається за формулою:

$$\tau = d + f, \tag{4.15}$$

де d – середньозважена ставка за депозитними операціями в комерційних банках; в 2021 році в Україні d = (0,09…0,14);

f – показник, що характеризує ризикованість вкладень; зазвичай, величина f = (0,05...0,5).

$$\tau_{\min} = 0{,}14 + 0{,}05 = 0{,}19.$$

Так як $E_в > \tau_{\min}$, то інвестор може бути зацікавлений у фінансуванні даної наукової розробки.

Розрахуємо термін окупності вкладених у реалізацію наукового проекту інвестицій за формулою:

$$T_{ок} = \frac{1}{E_в}, \tag{4.16}$$

$T_{ок}$ = 1 / 1,007 = 0,99 р.

Оскільки $T_{ок}$ < 3-х років, а саме термін окупності рівний 0,99 роки, то фінансування даної наукової розробки є доцільним.

Висновки до розділу: економічна частина даної роботи містить розрахунок витрат на розробку нового програмного продукту сума яких складає 687306 гривень. Було спрогнозовано орієнтовану величину витрат по кожній з статей витрат. Також розраховано чистий прибуток, який може отримати виробник від реалізації нового технічного рішення, розраховано період окупності витрат для інвестора та економічний ефект при використанні даної розробки. В результаті аналізу розрахунків можна зробити висновок, що розроблений програмний продукт за ціною дешевший за аналог і є висококонкурентоспроможним. Період окупності складе близько 0,99 роки.

4.4 Висновок

В процесі виконання розрахунків економічної частини магістерської кваліфікаційної роботи було проведено відповідні обчислення та доведено, що розробка інформаційної технології автоматизованого тестування є економічно доцільною.

Спершу проведено комерційний та технологічний аудит розробленої технології. Отримані результати оцінювання довели високий комерційний потенціал розробки.

Також проведені розрахунки витрат на розробку нового програмного продукту, котрі склали 687306 гривень. Прогнозовано орієнтовану величину по кожній з статей витрат. Розраховано чистий прибуток, який може отримати виробник від реалізації технічного рішення, період окупності витрат для інвестора та економічний ефект використання розробленої технології.

Загальні результати обрахунків економічної частини розробки довели висококонкурентоспроможність технології, період окупності реалізації котрої складе близько 0,99 років, що свідчить про доцільність її фінансування.

ВИСНОВКИ

З метою зменшення витрат часу та коштів в процесі реалізації програмних продуктів, розробники застосовують системи автоматизованого тестування. Сьогодні ці технології є важливою частиною розробки будь-якого програмного продукту.

У ході виконання магістерської кваліфікаційної роботи розглянуто питання актуальності розробки інформаційної технології автоматизованого тестування.

На основі аналізу стану питання сучасного розвитку технологій автоматизованого тестування доведено актуальність розробки. Проведено аналіз методів та засобів вирішення задачі автоматизованого тестування. Розглянуто аналоги технології та на основі їх порівняння встановлено вимоги та завдання для власної розробки.

В процесі моделювання визначено, що інформаційна технологія буде представлена у формі мультикомпонентної системи автоматизованого тестування. На основі аналізу складових побудовано структурну схему компонентів технології. Описано деталі реалізації складових модулів програмного продукту. Обґрунтовано використання шаблону написання тестових методів «Arrange-Act-Assert». Розроблено високоефективний алгоритм пошуку тестових методів. У компоненті динамічних аналізаторів розроблено загалом 75 аналізатори, серед котрих реалізовано 13, аналогів яких немає в існуючих програмних реалізаціях

На основі проведеного моделювання обґрунтовано застосування мови програмування C# та інтегрованого середовища Visual Studio 2019. В результаті розроблено усі частини мультикомпонентної технології, доведено їх роботоздатність та ефективність. Встановлено, що показники швидкодії реалізованого програмного забезпечення є кращими на 9%, 23% та 63% у порівнянні з визначеними аналогічними технологіями.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. The True Cost of a Software Bug: Part One. [Електронний ресурс]. Режим доступу до ресурсу: https://www.celerity.com/the-truecost-of-a-software-bug.

2. Software Bugs Cost U.S. Economy $59.6 Billion Annually, RTI Study Finds, 2021. [Електронний ресурс]. Режим доступу до ресурсу: http://www.nist.gov/director/prog-ofc/report02-3.pdf.

3. Хант А. Pragmatic Unit Testing in C# with Nunit – Даллас, Техас: The Pragmatic Bookshelf, 2007. – 45 с.

4. V. O. Vasylevskyi, A. A. Yarovyi, «Features of providing access to the software tools of the automat-ed testing information system», Матеріали V міжнародної науково-практичної конференції «Modern directions of scientific research development», с. 230 – 234, 2021. [Електронний ресурс]. Режим доступу: https://sci-conf.com.ua/wp-content/uploads/2021/11/MODERN-DIRECTIONS-OF-SCIENTIFIC-RESEARCH-DEVELOPMENT-28-30.10.21.pdf.

5. Василевський В. О., Яровий А. А., «Створення інтелектуальної системи автоматизованого тестування на основі фреймворку юніт-тестування», Матеріали L Науково-технічної конференції факультету інформаційних технологій та комп'ютерної інженерії (2021) [Електронний ресурс]. – Режим доступу: https://conferences.vntu.edu.ua/index.php/all-fitki/all-fitki-2021/paper/view/12406/10363.

6. Василевський В. О., Яровий А. А., «Аналіз структури компонентів інформаційної технології автоматизованого тестування», Матеріали всеукраїнської науково-практичної конференції Молодь в науці: дослідження, проблеми, перспективи (МН-2022) [Електронний ресурс]. – Режим доступу: https://conferences.vntu.edu.ua/index.php/mn/mn2022/paper/viewFile/14166/11977.

7. Ошеров Р. The art of unit testing with examples in .NET – Гринвіч: Manning Publication Co, 2009. – 20 с.

8. Майерс Г. Искусство тестирования программ, 3-е издание / Г. Майерс, Т. Баджетт, К. Сандлер [Електронний ресурс] – Режим доступу: http://www.dialektika.com/books/978-5-8459-1796-6.html.

9. Лангр Д. Modern C++ Programming with Test-Driven Development: Code Better, Sleep Better – Манчестер: Pragmatic Bookshelf, 2013. – 18 с.

10. Фрімен С., Growing Object-Oriented Software Guided by Tests – Кроудфордсвіль, Індіана: Donnel-ley, 2012. – с. 28-29.

11. Декостер К. Pro NuGet (Expert's Voice in Microsoft) 1st ed. Edition – Нью-Йорк: Apress, 2013 – 41 с.

12. How to Run and Interpret Unit Tests with Visual Studio Test Explorer [Електронний ресурс]. Режим доступу до ресурсу: https://adamtheautomator.com/visual-studio-test-explorer/.

13. Бек К. Test Driven Development: By Example – Лондон: Addison-Wesley Professional, 2014. – 63 с.

14. Хоріков В. Unit Testing Principles, Practices, and Patterns – Гринвіч: Manning Publication Co, 2020. – 140 с.

15. Сравнение xUnit.net, NUnit и MSTest [Електронний ресурс] – Режим доступу до ресурсу: https://qaat.ru/sravnenie-xunit-net-nunit-i-mstest/.

16. Вступ до NUnit [Електронний ресурс] – Режим доступу до ресурсу: https://itvdn.com/ru/blog/article/entry-into-nunit.

17. Why Should You Use xUnit? A Unit Testing Framework For .Net [Електронний ресурс] – Режим доступу до ресурсу: https://www.clariontech.com/blog/why-should-you-use-xunit-a-unit-testing-framework-for-.net.

18. Юніт-тестування з MSTest [Електронний ресурс] – Режим доступу до ресурсу: https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-with-mstest.

19. How to choose the best test framework [Електронний ресурс] –

Режим доступу до ресурсу: https://techbeacon.com/app-dev-testing/how-choose-best-test-framework.

20. Що таке інформаційні технології [Електронний ресурс] – Режим доступу до ресурсу: http://apeps.kpi.ua/shcho-take-informatsiini-technologii/en.

21. Мартін Р. Чистий код: створення і рефакторинг за допомогою Agile – Харків: Видавництво «Фабула», 2019 – 98 с.

22. Application program interface (API) [Електронний ресурс] – Режим доступу до ресурсу: https://searchapparchitecture.techtarget.com/definition/application-program-interface-API.

23. Грегорі Д. Game Engine Architecture – Лондон: CRC Press, 2013 – 183 с.

24. Тідвелл Д. Разработка пользовательских интерфейсов / Санкт-Петербург.: Питер, 2008. – 108 с.

25. Extend Visual Studio IDE [Електронний ресурс] – Режим доступу до ресурсу: https://visualstudio.microsoft.com/vs/features/extend/.

26. What is a Distributed System, and How Does it Work? [Електронний ресурс] – Режим доступу до ресурсу: https://www.confluent.io/learn/distributed-systems/.

27. Code analysis [Електронний ресурс] – Режим доступу до ресурсу: https://docs.microsoft.com/en-us/dotnet/framework/code-analyzers.

28. Gomes P. Unit Testing and the Arrange, Act and Assert (AAA) Pattern [Електронний ресурс] – Режим доступу до ресурсу: https://medium.com/@pjbgf/title-testing-code-ocd-and-the-aaa-pattern-df453975ab80.

29. Making Better Unit Tests: part 1, the AAA pattern [Електронний ресурс] – Режим доступу до ресурсу: https://freecontent.manning.com/making-better-unit-tests-part-1-the-aaa-pattern/.

30. Write your first analyzer and code fix [Електронний ресурс] – Режим доступу до ресурсу: https://docs.microsoft.com/en-us/dotnet/csharp/roslyn-

sdk/tutorials/how-to-write-csharp-analyzer-code-fix.

31. Троельсен А. C# 6.0 and the .NET 4.6 Framework – Нью-Йорк: Apress, 2015 – 1018 с.

32. IntelliJ IDEA overview [Електронний ресурс] – Режим доступу до ресурсу:https://www.jetbrains.com/help/idea/discover-intellij-idea.html.

33. Процес розробки програмного забезпечення [Електронний ресурс] – Режим доступу до ресурсу: – https://en.wikipedia.org/wiki/Software_development_process.

34. Нейгел К. C# 4.0 и платформа .NET 4 для профессионалов. – Москва: Диалектика, 2011. – 358 с.

35. Ріхтер Д. CLR via C#. – Редмонд, Вашингтон: Microsoft Press, 2012. – 214 с.

36. Страуструп Б. Programming: Principles and Practice Using C++ – Бостон: Addison-Wesley, 2014. – 26 с.

37. Блох Д. Effective Java – Бостон: Pearson Education Inc., 2018. – 32 с.

38. Шилдт Г. Java 8. Руководство для начинающих, 6-е издание – Москва: «Диалектика», 2017 – 418 ст.

39. Лутц М. Изучаем Python. 4-е издание – Санкт-Петербург, 2011 – 67 ст.

40. Бубнов И. Лучшие IDE для разработки на C# Один очевидный вариант и несколько других. [Електронний ресурс] – Режим доступу до ресурсу: https://geekbrains.ru/posts/c_sharp_ides.

41. Visual Studio Лучшие в своем классе средства для разработчиков [Електронний ресурс] – Режим доступу до ресурсу: https://visualstudio.microsoft.com/en/.

42. Быстрая и мощная кросс-платформенная IDE для .NET [Електронний ресурс] – Режим доступу до ресурсу: https://www.jetbrains.com/ru-ru/rider/.

43. ReSharper [Електронний ресурс] – Режим доступу до ресурсу:

https://ru.wikipedia.org/wiki/ReSharper.

44. Code editing. Redefined. [Електронний ресурс] – Режим доступу до ресурсу: https://code.visualstudio.com/.

45. Xml essentials [Електронний ресурс] – Режим доступу до ресурсу: https://www.w3.org/standards/xml/core.

46. Хант К., TCP/IP Network Administration – Себастополь, Каліфорнія: O`Reilly Media, 2002. – 119 с.

47. Човдгурі К., Mastering Visual Studio 2019 - Second Edition – Бостон: O`Reilly Media, 2019. – 371 с.

48. Адеволе А., C# and .NET Core Test Driven Development – Бірмінгем: Packt Publishing, 2018. – 90 с.

49. Троельсен А. Pro C# 7: With .NET and .NET Core – Нью-Йорк: Apress, 2017 – 618 с.

50. Bacani M. Roslyn Cookbook: Compiler as a Service, Code Analysis, Code Quality and more – Бірмінгем: Packt Publishing, 2017. – 214 с.

51. Customize a rule set [Електронний ресурс] – Режим доступу до ресурсу: https://docs.microsoft.com/en-us/visualstudio/code-quality/how-to-create-a-custom-rule-set?view=vs-2022.