**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ**

**ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ В.Н. КАРАЗІНА**

**ФАКУЛЬТЕТ КОМП’ЮТЕРНИХ НАУК**

# **КУРСОВА РОБОТА**

з дисципліни «Математичні методи тестування і верифікації програмного забезпечення»

Тема «Тестування на відповідність протоколам»

Виконав студент 2 курсу

групи КБ-21

Кравченко Євгеній Миколайович

Перевірив: доц. Нарєжній О. П.

Харків – 2018

**ЗМІСТ**

[ВСТУП 4](#_Toc9492359)

[1 ТЕОРЕТИЧНА ЧАСТИНА 5](#_Toc9492360)

[1.1 Історія розвитку програмного забезпечення 5](#_Toc9492361)

[1.2 Основні поняття та визначення 7](#_Toc9492362)

[1.3 Методи тестування 8](#_Toc9492363)

[Статичне та динамічне тестування 8](#_Toc9492364)

[Тестування «білої скриньки» 9](#_Toc9492365)

[Тестування «чорної скриньки 10](#_Toc9492366)

[1.4 Види тестування програмного забезпечення 12](#_Toc9492367)

[Класифікація за ознаками 12](#_Toc9492368)

[Опис видів тестування 13](#_Toc9492369)

[1.5 Рівні тестування 15](#_Toc9492370)

[2 ПРАКТИЧНА ЧАСТИНА 19](#_Toc9492376)

[2.1 Формування вимог до програмного забезпечення 19](#_Toc9492377)

[2.2 Тестування програмного забезпечення 19](#_Toc9492378)

[ВИСНОВКИ 22](#_Toc9492379)

[ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ 24](#_Toc9492380)

[ДОДАТОК А 25](#_Toc9492381)

# **ВСТУП**

Тестування програмного забезпечення — техніка контролю якості, що перевіряє відповідність між реальною і очікуваною поведінкою програми завдяки кінцевому набору тестів, які обираються певним чином.

Якість не є абсолютною, це суб'єктивне поняття. Тому тестування, як процес своєчасного виявлення помилок та дефектів, не може повністю забезпечити коректність програмного забезпечення. Воно тільки порівнює стан і поведінку продукту зі специфікацією. При цьому треба розрізняти тестування програмного забезпечення й забезпечення якості програмного забезпечення, до якого належать всі складові ділового процесу, а не тільки тестування.

Зазвичай, поняття якості обмежується такими поняттями як коректність, надійність, практичність, безпечність, але може містити більше технічних вимог, котрі описані у стандарті ISO 9126. Склад та зміст супутньої документації процесу тестування визначається стандартом IEEE 829—1998 Standard for Software Test Documentation. Існує багато підходів до тестування програмного забезпечення, але ефективне тестування складних продуктів — це по суті дослідницький та творчий процес, а не тільки створення та виконання рутинної процедури.

# **ТЕОРЕТИЧНА ЧАСТИНА**

## **Історія розвитку програмного забезпечення**

Перші програмні системи розроблялися в рамках програм наукових досліджень або програм для потреб міністерств оборони. Тестування таких продуктів проводилося строго формалізовано із записом всіх тестових процедур, тестових даних, отриманих результатів. Тестування виділялося в окремий процес, який починався після завершення кодування, але при цьому, як правило, виконувалося тим же персоналом.

У 1960-х багато уваги приділялося «вичерпному» тестуванню, яке повинно проводитися з використанням усіх шляхів у коді або всіх можливих вхідних даних. Було відзначено, що в цих умовах повне тестування ПЗ неможливе, тому що, по-перше, кількість можливих вхідних даних дуже велика, по-друге, існує безліч шляхів, по-третє, складно знайти проблеми в архітектурі та специфікаціях. З цих причин «вичерпне» тестування було відхилено й визнано теоретично неможливим.

На початку 1970-х тестування ПЗ розглядалося як «процес, спрямований на демонстрацію коректності продукту» або як «діяльність з підтвердження правильності роботи ПЗ». У програмній інженерії, яка в той час зароджувалася, верифікація ПЗ визначалася як «доказ правильності». Хоча концепція була теоретично перспективною, на практиці вона вимагала багато часу й не охоплювала всі аспекти тестування. Було вирішено, що доказ правильності — неефективний метод тестування ПЗ. Однак, у деяких випадках демонстрація правильної роботи використовується і в наші дні, наприклад, приймально-здавальні випробування. У другій половині 1970-х тестування представлялося як виконання програми з наміром знайти помилки, а не довести, що вона працює. Успішний тест — це тест, який виявляє раніше невідомі проблеми. Даний підхід цілком протилежний попередньому. Зазначені два визначення являють собою «парадокс тестування», в основі якого лежать два протилежних твердження: з одного боку, тестування дозволяє переконатися, що продукт працює добре, а з іншого — виявляє помилки у ПЗ, показуючи, що продукт не працює. Друга мета тестування є більш продуктивною з точки зору поліпшення якості, оскільки не дозволяє ігнорувати недоліки ПЗ.

У 1980-х тестування розширилося таким поняттям як запобіганням дефектам. Проектування тестів — найбільш ефективний з відомих методів запобігання помилок. В цей же час почали висловлюватися думки, що необхідна методологія тестування, зокрема, що тестування повинно включати перевірки впродовж усього циклу розроблення, при цьому це має бути керований процес. В ході тестування треба перевірити не тільки зібрану програму, але й вимоги, код, архітектуру, самі тести. «Традиційне» тестування, яке існувало до початку 1980-х, відносилося тільки до скомпілювати, готової системі (зараз це зазвичай називається системне тестування), але надалі тестувальники стали залучатися в усі аспекти життєвого циклу розроблення. Це дозволяло раніше знаходити проблеми у вимогах та архітектурі й тим самим скорочувати терміни та бюджет розроблення. У середині 1980-х з'явилися перші інструменти для автоматизованого тестування. Передбачалося, що комп'ютер зможе виконати більше тестів, ніж людина, причому зробить це більш надійно. Спочатку ці інструменти були вкрай простими й не мали можливості написання сценаріїв на скриптових мовах.

На початку 1990-х у поняття «тестування» стали включати планування, проектування, створення, підтримку й виконання тестів та тестових оточень, а це означало перехід від тестування до забезпечення якості, що охоплює весь цикл розроблення ПЗ. У цей час починають з'являтися різні програмні інструменти для підтримки процесу тестування: більш просунуті середовища для автоматизації з можливістю створення скриптів і генерації звітів, системи управління тестами, ПЗ для проведення навантажувального тестування. У середині 1990-х з розвитком Інтернету й розробленням великої кількості веб-застосунків особливої популярності стало набувати «гнучке тестування» (за аналогією з гнучкими методологіями програмування).

У 2000-х з'явилося ще більш широке визначення тестування, коли в нього було додано поняття «оптимізація бізнес-технологій». BTO направляє розвиток інформаційних технологій згідно з цілями бізнесу. Основний підхід полягає в оцінці та максимізації значущості всіх етапів життєвого циклу розроблення ПЗ для досягнення необхідного рівня якості, продуктивності, доступності.

## **Основні поняття та визначення**

Тестування — це одна з технік контролю якості, що включає в себе

* + Планування робіт (Test Management)
  + Проектування тестів (Test Design)
  + Виконання тестування (Test Execution)
  + Аналіз отриманих результатів (Test Analysis).

Верифікація (Verification) — це процес оцінки системи або її компонентів з метою визначити чи задовольняють результати поточного етапу розробки умовам, сформованим на початку цього етапу. Тобто чи виконуються цілі, терміни, завдання з розробки проекту, визначені на початку поточної фази. Валідація (Validation) — це визначення відповідності розроблюваного програмного забезпечення між очікуваннями і потребами користувача, вимогам до системи.

План Тестування (Test Plan) — це документ, що описує весь обсяг робіт з тестування, починаючи з опису об'єкта, стратегії, розкладу, критеріїв початку і закінчення тестування, до необхідного в процесі роботи обладнання, спеціальних знань, а також оцінки ризиків з варіантами їх вирішення.

Тест дизайн (Test Design) — це етап процесу тестування програмного забезпечення, на якому проектуються і створюються тестові випадки (тест кейси), відповідно до визначених раніше критеріями якості та цілями тестування.

Тестовий випадок (Тест кейс/Test Case) — це документ, що описує сукупність кроків, конкретних умов і параметрів, необхідних для перевірки реалізації тестованої функції або її частини.

Баг/Дефект Репорт (Bug Report) — це документ, що описує ситуацію або послідовність дій (Steps), що призвела до некоректної роботи об'єкта тестування (Misbehavior), із зазначенням причин та очікуваного результату (Expected Result).

Тестове Покриття (Test Coverage) — це одна з метрик оцінки якості тестування, що представляє із себе щільність покриття тестами вимог або коду, що виконується.

Деталізація Тест Кейсів (Test Case Specification) — це рівень деталізації опису тестових кроків і необхідного результату, при якому забезпечується розумне співвідношення часу проходження до тестового покриття.

Час Проходження Тест Кейса (Test Case Pass Time) — це час від початку проходження кроків тест кейса до отримання результату тесту.

## **Методи тестування**

### **Статичне та динамічне тестування**

Тестова діяльність, що пов'язана з аналізом результатів розробки програмного забезпечення, називається статичним тестуванням. Воно передбачає перевірку програмних кодів, контроль та перевірку програми без запуску на комп'ютері. Тестова діяльність, що передбачає експлуатацію програмного продукту, називається динамічним тестуванням. Динамічне та статичне тестування доповнюють одне одного.

На етапі статичного тестування перевіряється вся документація, отримана як результат життєвого циклу програми. Це і технічне завдання, і специфікація, і вихідний текст програми на мові програмування. Вся документація аналізується на предмет дотримання стандартів програмування. У результаті статичної перевірки встановлюється, наскільки програма відповідає заданим критеріям та вимогам замовника. Усунення неточностей та помилок у документації — запорука того, що створюваний програмний засіб має високу якість.

Динамічні методи застосовуються в процесі безпосереднього виконання програми. Коректність програмного засобу перевіряється на безлічі тестів або наборів підготовлених вхідних даних. При прогоні кожного тесту збираються та аналізуються дані про відмови та збої в роботі програми.

### **Тестування «білої скриньки»**

Відома: внутрішня структура програми.

Досліджуються: внутрішні елементи програми і зв'язки між ними.

Об'єктом тестування тут є не зовнішня, а внутрішня поведінка програми. Перевіряється коректність побудови всіх елементів програми та правильність їхньої взаємодії один з одним. Зазвичай аналізуються керуючі зв'язки елементів, рідше — інформаційні зв'язки. Тестування за принципом «білого ящика» характеризується ступенем, в якому тести виконують або покривають логіку (вихідний текст) програми.

Особливості тестування «білої скриньки»

Зазвичай тестування «білої скриньки» засноване на аналізі керуючої структури програми. Програма вважається повністю перевіреною, якщо проведено вичерпне тестування маршрутів (шляхів) її графа управління.

У цьому випадку формуються тестові варіанти, в яких:

Гарантується перевірка всіх незалежних маршрутів програми.

Знаходяться гілки True, False для всіх логічних рішень.

Виконуються всі цикли (у межах їхніх кордонів та діапазонів).

Аналізується правильність внутрішніх структур даних.

Недоліки тестування «білої скриньки»:

Кількість незалежних маршрутів може бути дуже велика.

Повне тестування маршрутів не гарантує відповідності програми вихідним вимогам до неї.

У програмі можуть бути пропущені деякі маршрути.

Не можна виявити помилки, поява яких залежить від даних.

Переваги тестування «білої скриньки» пов'язані з тим, що принцип «білої скриньки» дозволяє врахувати особливості програмних помилок:

Кількість помилок мінімально в «центрі» і максимально на «периферії» програми.

Попередні припущення про ймовірність потоку керування або даних у програмі часто бувають некоректними. У результаті типовим може стати маршрут, модель обчислень за яким опрацьована слабо.

При записі алгоритму програмного забезпечення у вигляді тексту на мові програмування можливе внесення типових помилок трансляції (синтаксичних та семантичних).

Деякі результати в програмі залежать не від вихідних даних, а від внутрішніх станів програми.

Кожна з цих причин є аргументом для проведення тестування за принципом «білої скриньки». Тести «чорної скриньки» не зможуть реагувати на помилки таких типів.

### **Тестування «чорної скриньки**

Відомі: функції програми.

Досліджується: робота кожної функції на всій області визначення.

Основне місце програми тестів «чорної скриньки» — інтерфейс ПЗ.

Ці тести демонструють:

* Як виконуються функції програми.
* Як приймаються вихідні дані.
* Як виробляються результати.
* Як зберігається цілісність зовнішньої інформації.

При тестуванні «чорної скриньки» розглядаються системні характеристики програм, ігнорується їхня внутрішня логічна структура. Вичерпне тестування, як правило, неможливе. Наприклад, якщо в програмі 10 вхідних величин і кожна приймає по 10 значень, то кількість тестових варіантів становитиме 10^10 . Тестування «чорної скриньки» не реагує на багато особливостей програмних помилок.

Тестування «чорної скриньки» (функціональне тестування) дозволяє отримати комбінації вхідних даних, які забезпечують повну перевірку всіх функціональних вимог до програми. Програмний виріб тут розглядається як «чорна скринька», чию поведінку можна визначити тільки дослідженням його входів та відповідних виходів. При такому підході бажано мати: набір, утворений такими вхідними даними, які призводять до аномалій у поведінці програми (назвемо його IT); набір, утворений такими вхідними даними, які демонструють дефекти програми (назвемо його OT).

Будь-який спосіб тестування «чорної скриньки» повинен: виявити такі вхідні дані, які з високою ймовірністю належать набору IT; сформулювати такі очікувані результати, які з високою імовірністю є елементами набору OT. Принцип «чорної скриньки» не альтернативний принципу «білої скриньки». Скоріше це доповнює підхід, який виявляє інший клас помилок. Тестування «чорної скриньки» забезпечує пошук наступних категорій помилок: некоректних чи відсутніх функцій; помилок інтерфейсу; помилок у зовнішніх структурах даних або в доступі до зовнішньої бази даних; помилок характеристик (необхідна ємність пам'яті і т. д.); помилок ініціалізації та завершення. подібні категорії помилок способами «білої скриньки» не виявляються.

## **Види тестування програмного забезпечення**

### **Класифікація за ознаками**

За ступенем автоматизації: ручне тестування (manual testing), автоматизоване тестування (automated testing), напівавтоматизоване тестування (semiautomated testing).

За ступенем підготовленості до тестування: тестування по документації (formal testing), тестування ad hoc або інтуїтивне тестування (ad hoc testing) — тестування без тест плану та документації, що базується на методиці передбачення помилки на власному досвіді тестувальника.

За знанням системи: тестування чорної скриньки (black box), тестування білої скриньки (white box), тестування сірої скриньки (grey box)

За ступенем ізольованості компонентів: компонентне (модульне) тестування (component/unit testing), інтеграційне тестування (integration testing), сиистемне тестування (system/end-to-end testing).

За часом проведення тестування: альфа-тестування (alpha testing), тестування при прийманні або Димове тестування (smoke testing), тестування нової функціональності (new feature testing), регресивне тестування (regression testing), тестування при здачі (acceptance testing) – випадок описаний в даній курсові роботі, Бета-тестування (beta testing).

За об'єктом тестування: функціональне тестування (functional testing), тестування продуктивності (performance testing), навантажувальне тестування (load testing), стрес-тестування (stress testing), тестування стабільності (stability/endurance/soak testing), тестування зручності використання або Юзабіліті-тестування (usability testing), тестування інтерфейсу користувача (UI testing), тестування безпеки (security testing), тестування локалізації (localization testing), естування сумісності (compatibility testing).

За ознакою позитивності сценаріїв: позитивне тестування (positive testing), негативне тестування (negative testing).

### **Опис видів тестування**

Інсталяційне тестування запевняє, що система встановлена ​​правильно і коректно працює на апаратному забезпеченні конкретного клієнта.

Тестування сумісності, основною метою якого є перевірка коректної роботи продукту в певному середовищі. Середовище може включати в себе наступні елементи:

* Апаратна платформа
* Мережеві пристрої
* Периферія (принтери, CD/DVD-приводи, веб-камери та ін.);
* Операційна система (Unix, Windows, MacOS, …)
* Бази даних (Oracle, MS SQL, MySQL, …)
* Системне програмне забезпечення (веб-сервер, фаєрвол, антивірус, …)
* Браузери (Internet Explorer, Firefox, Opera, Chrome, Safari)
* Смоук тестування (Smoke testing)

Мінімальний набір тестів на явні помилки. Цей тест зазвичай виконується самим програмістом. Програму, що не пройшла такий тест, не має сенсу передавати на глибше тестування.

Регресивне тестування виявляє помилки у вже протестованих ділянках початкового коду. Такі помилки — коли після внесення змін до програми перестає працювати те, що мало б працювати, — називають регресивними помилками.

Регресивне тестування (за деякими джерелами) включає:

* new bug-fix — перевірка виправлення знайдених дефектів;
* old bug-fix — перевірка, що виявлені раніше й виправлені дефекти не

відтворюються в системі знову;

* side-effect — перевірка того, що не порушилася працездатність

працюючої раніше функціональності, якщо її код міг бути зачеплений під час виправлення деяких дефектів в іншій функціональності.

* Функціональне тестування перевіряє, чи реалізовані функціональні

вимоги, тобто можливості ПЗ в певних умовах вирішувати завдання, потрібні користувачам. Функціональні вимоги визначають, що саме робить продукт, які завдання вирішує.

Функціональні вимоги включають в себе: функціональна придатність, точність, можливість до взаємодії, відповідність стандартам та правилам, захищеність.

Нефункціональне тестування описує тести, необхідні для визначення характеристик ПЗ, які можуть бути виміряні різними величинами. В цілому, це тестування того, «як» система працює. Далі перелічені основні види нефункціональних тестів:

Всі види тестування продуктивності: навантажувальне тестування, стресове тестування, тестування стабільності та надійності, об'ємне тестування, інсталяційне тестування, тестування зручності користування, тестування на «відмову» та відновлення, конфігураційне тестування, деструктивне тестування намагається привести ПЗ чи підсистему до збою. Воно перевіряє, чи ПЗ продовжує функціонувати навіть при отриманні неправильних або неочікуваних вхідних даних, встановлюючи тим самим надійність перевірки вхідних даних і управління помилками підпрограм. Тестування швидкодії проводиться з метою встановлення, як швидко працює система або її частина, під певним навантаженням. Також може слугувати для перевірки й підтвердження інших атрибутів якості системи, таких як масштабування, надійність та використання ресурсів.

В тестуванні швидкодії виділяють такі напрямки: навантажувальне, стрес тестування стабільності, конфігураційне, тестування зручності використання, виконується з метою визначення зручності використання ПЗ для його подальшого застосування. Це метод оцінки зручності продукту у використанні, заснований на залученні користувачів як тестувальників, випробувачів і підсумовуванні отриманих від них висновків.

## **Рівні тестування**

## Модульне тестування ідноситься до тестів, які перевіряють функціональність певного розділу коду, зазвичай на функціональному рівні. В об'єктно-орієнтованому середовищі, це, як правило, тестування на рівні класу, а мінімальні модульні тести містять у собі конструктори та деструктори.

Такі типи тестів зазвичай пишуться розробниками під час роботи над кодом (стиль «білої скриньки»), щоб впевнитись, що дана функція працює так, як очікувалося. Одна функція може мати кілька тестів, щоб переглянути всі випадки використання коду. Модульне тестування саме по собі не може перевірити функціонування частини ПЗ, а використовується щоб гарантувати, що основні блоки ПЗ працюють незалежно один від одного.

Модульне тестування — це процес розробки ПЗ, що включає в себе синхронізовані застосування широкого спектру для запобігання дефектів та для виявлення стратегій з метою зниження ризиків розробки ПЗ, часу та витрат. Воно виконується розробником ПЗ або інженером, під час будівельної фази життєвого циклу розробки ПЗ. Модульне тестування спрямоване на усунення помилок проектування. Ця стратегія спрямована на підвищення якості одержуваного ПЗ, до такого рівня, як вимагає процес контролю якості.

Залежно від очікуваної організації розробки ПЗ, модульне тестування може включати статичний аналіз коду, аналіз потоку даних аналізу метрик, експертні оцінки коду, аналізу покриття коду та інші методи перевірки ПЗ.

### Інтеграційне тестування є типом тестування ПЗ, яке прагне перевірити інтерфейси між компонентами від програмного дизайну. Програмні компоненти можуть бути інтегровані як в рамках ітеративного підходу, так і всі разом.

Інтеграційне тестування працює над виявленням дефектів у інтерфейсах та взаємодії інтегрованих компонентів (модулів). Воно проводиться до тих пір, поки великі групи тестованих компонентів ПЗ, які відповідають потрібній архітектурі, починають працювати як система.

Рівні інтеграційного тестування: компонентний інтеграційний рівень (Component Integration testing). Перевіряється взаємодія між компонентами системи після проведення компонентного тестування; системний інтеграційний рівень (System Integration Testing). Перевіряється взаємодія між різними системами після проведення системного тестування.

Підходи до інтеграційного тестування: знизу вгору (Bottom Up Integration). Усі низькорівневі модулі, процедури або функції збираються воєдино і потім тестуються. Після чого збирається наступний рівень модулів для проведення інтеграційного тестування. Даний підхід вважається корисним, якщо всі або практично всі модулі розроблюваного рівня готові. Також даний підхід допомагає визначити за результатами тестування рівень готовності додатків; зверху вниз (Top Down Integration). У першу чергу тестуються компоненти верхнього рівня ієрархії об'єктів з використанням заглушок замість компонентів більш низького рівня; великий вибух («Big Bang» Integration). Усі або практично усі розроблені модулі збираються разом у вигляді закінченої системи або її основної частини й потім проводиться інтеграційне тестування. Такий підхід дуже хороший для збереження часу. Проте, якщо тест кейси та їхні результати записані неправильно, то сам процес інтеграції дуже ускладниться, що стане перепоною для команди тестування при досягненні основної мети інтеграційного тестування.

### Системне тестування тестує інтегровану систему для перевірки відповідності всім вимогам. Крім того, системне тестування ПЗ повинно гарантувати, що програма працює так, як очікувалося, а також, що її не можна знищити або пошкодити її робоче середовище, яке викличе процеси в цьому середовищі, що переведуть систему в неробочий стан. Системне інтеграційне тестування перевіряє, чи система інтегрується в будь-яку зовнішню систему (або системи) відповідно до системних вимог.

### Альфа-тестування — імітація реальної роботи з системою штатними розробниками або реальна робота з системою потенційними користувачами/замовником. Найчастіше альфа-тестування проводиться на ранній стадії розробки продукту, але у деяких випадках може застосовуватися для закінченого продукту як внутрішнього приймального тестування. Іноді альфа-тестування виконується під відлагоджувачем або з використанням середовища, яке допомагає швидко виявляти знайдені помилки. Виявлені помилки можуть бути передані тестувальникам для додаткового дослідження у середовищі, подібному тому, в якому буде використовуватися програма.

### Бета-тестування — у деяких випадках виконується поширення версії з обмеженнями (за функціональністю або часом роботи) для певної групи осіб, з тим щоб переконатися, що продукт містить достатньо мало помилок. Іноді бета-тестування виконується для того, щоб отримати зворотній зв'язок про продукт від його майбутніх користувачів. Часто для вільного/відкритого ПЗ стадія альфа-тестування характеризує функціональне наповнення коду, а бета-тестування — стадію виправлення помилок. При цьому, як правило, на кожному етапі розробки проміжні результати роботи доступні кінцевим користувачам.

Покриття коду, за своєю суттю, є тестуванням методом білого ящика. Тестоване ПЗ збирається зі спеціальними налаштуваннями або бібліотеками й/або запускається в особливому середовищі, в результаті чого для кожної використовуваної (виконуваної) функції програми визначається місцезнаходження цієї функції у вихідному коді. Цей процес дозволяє розробникам та фахівцям із забезпечення якості визначити частини системи, які, при нормальній роботі, використовуються дуже рідко або ніколи не використовуються (такі як код обробки помилок тощо). Це дозволяє зорієнтувати тестувальників на тестування найбільш важливих режимів.

Як правило, інструменти та бібліотеки, які використовуються для отримання покриття коду, вимагають значних витрат продуктивності та/або пам'яті, неприпустимих при нормальному функціонуванні ПЗ. Тому вони можуть використовуватися тільки в лабораторних умовах.

Приймальне тестування нормальний процес тестування, який перевіряє відповідність системи вимогам і проводиться з метою: визначення чи задовольняє система приймальним критеріям; винесення рішення замовником або іншою уповноваженою особою приймається додаток чи ні.

Приймальне тестування виконується на основі набору типових тестових випадків та сценаріїв, розроблених на основі вимог до даного додатку. Рішення про проведення приймального тестування приймається тоді, коли: продукт досяг необхідного рівня якості; замовник ознайомлений з Планом приймальних Робіт (Product Acceptance Plan) або іншим документом, де описаний набір дій, пов'язаних з проведенням приймального тестування, дата проведення, відповідальні тощо.

Фаза приймального тестування триває до тих пір, доки замовник не виносить рішення про відправлення програми на доопрацювання або видачі додатка. Самец ей вид тестування буде описано у практичній частині. Для тестування програми написаної за допомогою стандартної бібліотеки Java Swing було викортистано фреймворки Jemmy і JUnit.

# **ПРАКТИЧНА ЧАСТИНА**

## **Формування вимог до програмного забезпечення**

Функціональні вимоги:

1. Можливість створити новий документ з заданим розміром;
2. Можливість загрузки зображення (.png, .jpeg) з файлової системи;
3. Можливість зберігати створені зображення;
4. Можливість малювати довільну криву, лінію, прямокутник, еліпс;
5. Можливість створювати текстове поле з текстом різного шрифту і розміру;
6. Можливість затирати окремі елементи малюнку та очищати все полотно;
7. Можливість відміняти 5 останніх дій та повертати їх назад;
8. Можливість редагувати товщину та колір будь-якого елементу малюнка полотна;
9. Можливість редагувати колір заливки та контуру фігур на полотні.

Вимоги до зовнішнього інтерфейсу:

1. Інтерфейс повинен бути виконаний у світлих та яскравих тонах;
2. Управління елементами графічного інтерфейсу повинно бути інтуїтивно зрозумілою;
3. Всі елементи управління повинні супроводжуватися тематичною картинкою;
4. Головне вікно повинно легко адаптуватися під будь-який розмір екрану;
5. Мова інтерфейсу англійська.

## **Тестування програмного забезпечення**

Для тестування програми Paint було написано тест кейс з використанням фреймворків Jemmy та Junit. Jemmy використовувався для доступу до елементів інтерфейсу користувача. Junit ­­­– для перевірки функціональності програми.

Спочатку розглянемо зовнішній вигляд програми що тестується:

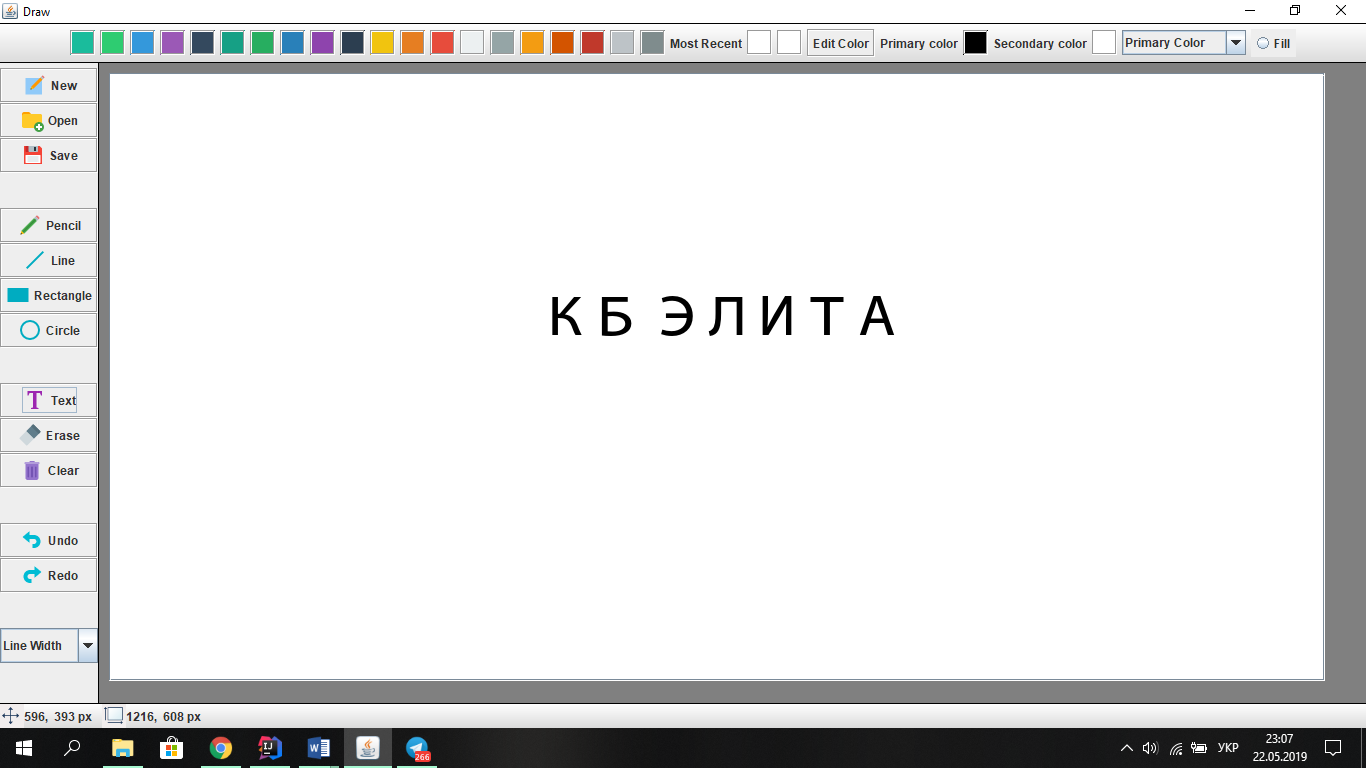


Рисунок 2.1 – Зовнішній вигляд програми

Як видно на рисунку 2.1, програма виконана у світлих тонах та зі зручним інтерфейсом, тобто вимогу щодо зовнішнього вигляду виконано.

Після перевірки відповідності вимог до зовнішнього вигляду програми було створено тест кейс. В якому було протестовано функціональні можливості програми. Junit тести використовують фреймворк Jemmy для звернення до елементів Java Swing програми і перевіряють функціональність.

На рисунку 2.3 можемо спостерігати зображення успішного проходження Junit тестів, що свідчить про коректну роботу програми. В цих тестах перевірялися наявність вказаних в вимогах елементів та їх функціонал.

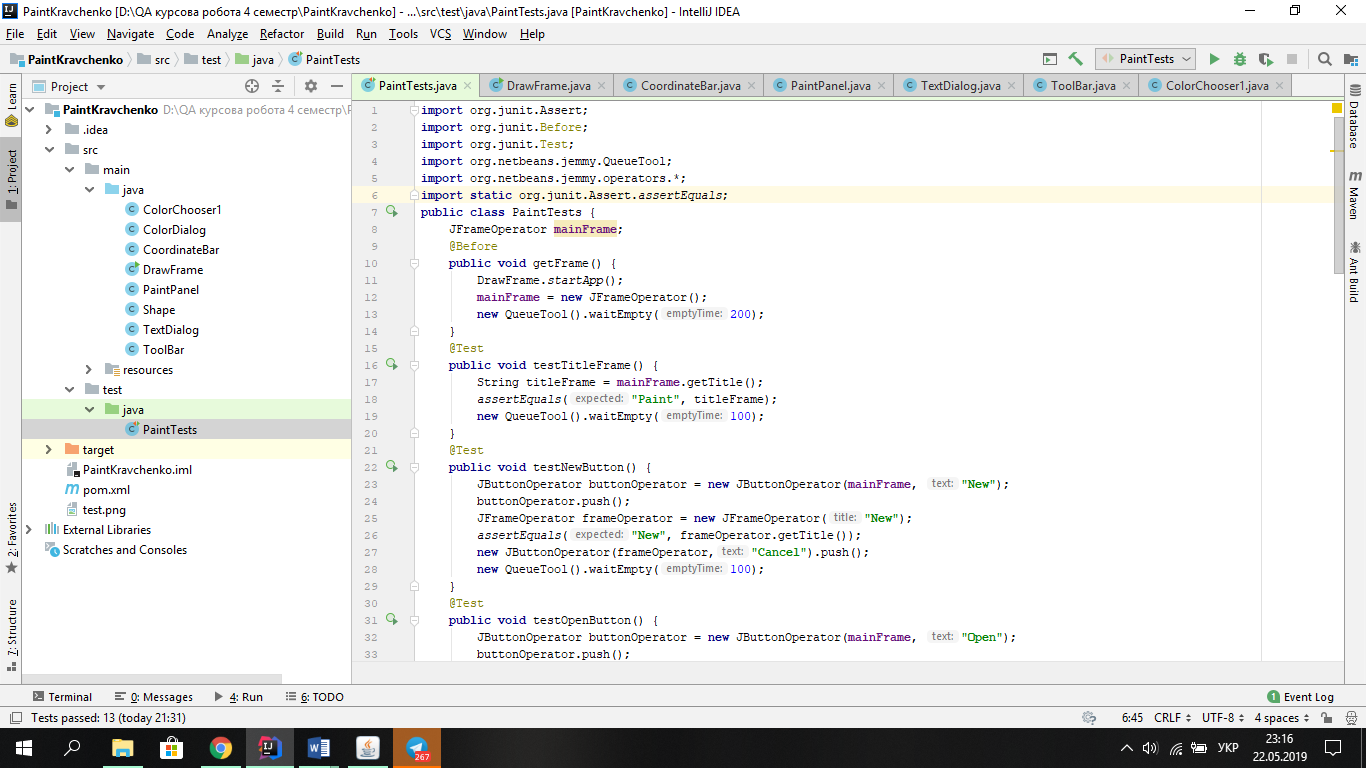


Рисунок 2.2 – Приклади тестів Junit

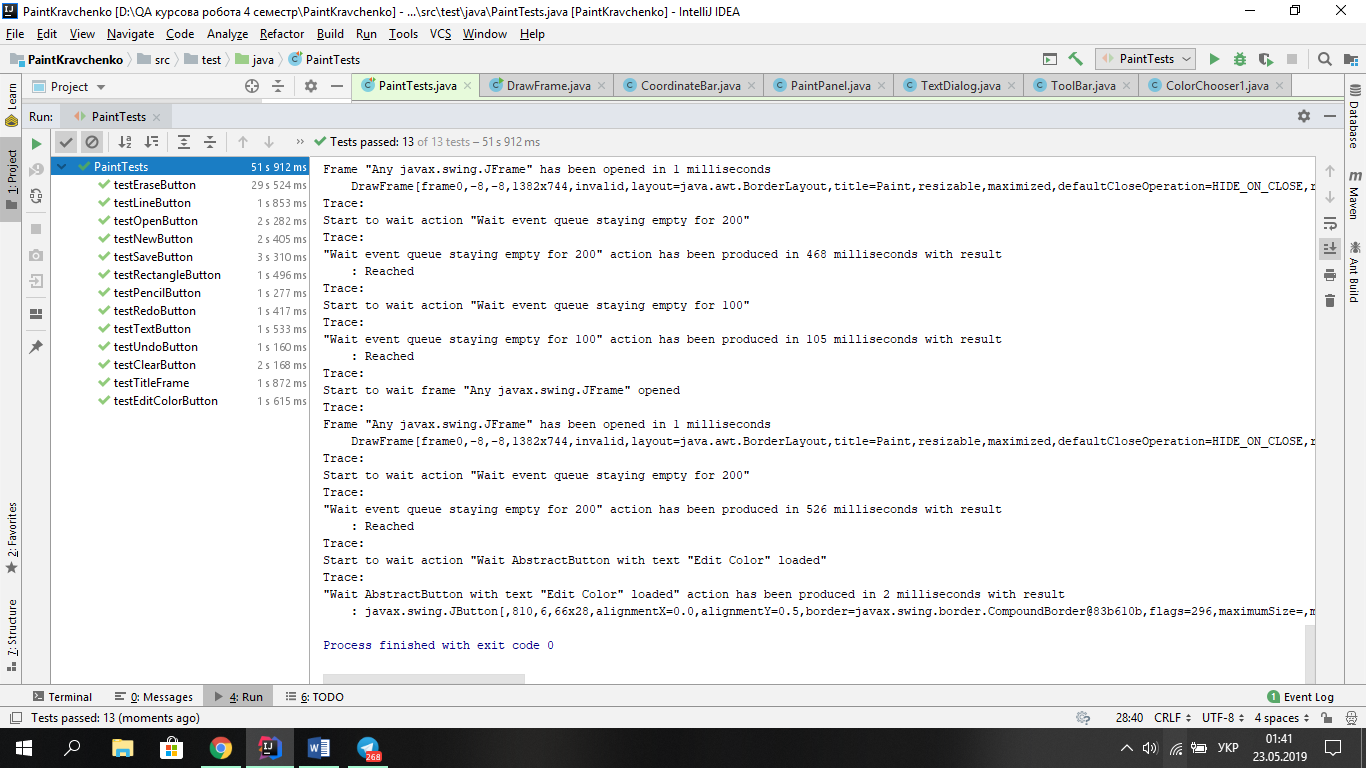


Рисунок 2.3 – Результати тестів

# **ВИСНОВКИ**

Тестування прийнятності для користувача (ТПК) являє собою процес, за допомогою якого отримується підтвердження того, що система відповідає взаємно узгодженим вимогам. Експерт предметної області, переважно власник або клієнт об'єкта який тестується, забезпечує таке підтвердження після перегляду. При розробці програмного забезпечення ТПК, є одним із заключних етапів проекту і часто відбувається, перш ніж клієнт отримує готову систему.

Користувачі системи виконують тести, які обумовлені розробниками у договорі або специфікації. Дизайнери тестів розробляють формальні тести, а також ряд рівнів серйозності. В ідеалі, приймальні тести для користувача та їх формальна інтеграція і системні тест кейси не повинні бути складені однією і тією ж особою. ТПК виступає як остаточна перевірка необхідної бізнес-функції та належного функціонування системи, емулюючи умови реального використання від імені клієнта або конкретного покупця. Якщо програмне забезпечення працює як належить і не виникає проблем під час нормального використання, можна розумно екстраполювати той же рівень стабільності виробництва.

Тести для користувачів, які зазвичай виконуються клієнтами або кінцевими користувачами, зазвичай не націлені на виявлення простих проблем, таких як орфографічні помилки і косметичні проблеми. У промисловому секторі, загальне ТПК є "заводським приймальним тестом " (ЗПТ). Цей тест виконується перед встановленням відповідного устаткування. Велику частину часу перевіряється не тільки, чи дане обладнання відповідає заданій специфікації, а й, чи є устаткування повністю функціональним. ЗПТ зазвичай включає в себе перевірку повноти, перевірку за договірними вимогами, доказ функціональності (або шляхом моделювання або звичайним тестуванням функцій) і заключний огляд.

Результати цих тестів вселяють впевненість у клієнтів про те, як система буде працювати у виробництві. Там також можуть бути юридичні або договірні вимоги для прийняття системи.

# **ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ**

1. Kaner C. Lessons Learned in Software Testing / C. Kaner, J. Bach, B. Pettichord. – John Wiley & Sons, 2002 – 580 p.
2. Методологія тестування ПЗ. Частина 1 [Електронний ресурс]. – Режим доступу до ресурсу: http://www.it4business.ru/lib/705/.
3. Методологія тестування ПЗ. Частина 2 [Електронний ресурс]. – Режим доступу до ресурсу: <http://www.it4business.ru/lib/717/>.
4. Методологія тестування ПЗ. Частина 3 [Електронний ресурс]. – Режим доступу до ресурсу: http://www.it4business.ru/lib/802/. 75
5. Офіційний сайт системи Redmine [Електронний ресурс]. – Режим доступу до ресурсу: <http://www.redmine.org>.
6. Тестування програмного забезпечення та концепції менеджменту програмних продуктів [Електронний ресурс]. – Режим доступу до ресурсу: http://www.slideshare.net/.
7. Портал спеціалістів по тестуванню [Електронний ресурс]. – Режим доступу до ресурсу: <http://software-testing.ru/>.
8. Маніфест Agile [Електронний ресурс]. – Режим доступу до ресурсу: <http://agilemanifesto.org/>.
9. Тестування програмного забезпечення [Електронний ресурс]. – Режим доступу до ресурсу: <http://www.protesting.ru/>.

# **ДОДАТОК А**

import org.junit.Assert;

import org.junit.Before;

import org.junit.Test;

import org.netbeans.jemmy.QueueTool;

import org.netbeans.jemmy.operators.\*;

import static org.junit.Assert.assertEquals;

public class PaintTests {

JFrameOperator mainFrame;

@Before

public void getFrame() {

DrawFrame.startApp();

mainFrame = new JFrameOperator();

new QueueTool().waitEmpty(200);

}

@Test

public void testTitleFrame() {

String titleFrame = mainFrame.getTitle();

assertEquals("Paint", titleFrame);

new QueueTool().waitEmpty(100);

}

@Test

public void testNewButton() {

JButtonOperator buttonOperator = new JButtonOperator(mainFrame, "New");

buttonOperator.push();

JFrameOperator frameOperator = new JFrameOperator("New");

assertEquals("New", frameOperator.getTitle());

new JButtonOperator(frameOperator,"Cancel").push();

new QueueTool().waitEmpty(100);

}

@Test

public void testOpenButton() {

JButtonOperator buttonOperator = new JButtonOperator(mainFrame, "Open");

buttonOperator.push();

JFileChooserOperator fileChooserOperator = new JFileChooserOperator();

JButtonOperator openOperator = new JButtonOperator(fileChooserOperator, "Open");

assertEquals("Open", openOperator.getText());

fileChooserOperator.cancel();

}

@Test

public void testSaveButton() {

JButtonOperator buttonOperator = new JButtonOperator(mainFrame, "Save");

buttonOperator.push();

JFileChooserOperator fileChooserOperator = new JFileChooserOperator();

JTextFieldOperator textFieldOperator = new JTextFieldOperator(fileChooserOperator);

textFieldOperator.setText("test");

JButtonOperator jButtonOperator = new JButtonOperator(fileChooserOperator, "Save");

jButtonOperator.push();

Assert.assertNotNull(buttonOperator);

}

@Test

public void testPencilButton() {

JButtonOperator buttonOperator = new JButtonOperator(mainFrame, "Pencil");

Assert.assertNotNull(buttonOperator);

Assert.assertNotNull(buttonOperator.getIcon());

}

@Test

public void testLineButton() {

JButtonOperator buttonOperator = new JButtonOperator(mainFrame, "Line");

Assert.assertNotNull(buttonOperator);

Assert.assertNotNull(buttonOperator.getIcon());

}

@Test

public void testRectangleButton() {

JButtonOperator buttonOperator = new JButtonOperator(mainFrame, "Rectangle");

Assert.assertNotNull(buttonOperator);

Assert.assertNotNull(buttonOperator.getIcon());

}

@Test

public void testTextButton() {

JButtonOperator buttonOperator = new JButtonOperator(mainFrame, "Text");

Assert.assertNotNull(buttonOperator);

Assert.assertNotNull(buttonOperator.getIcon());

}

@Test

public void testEraseButton() {

JButtonOperator buttonOperator = new JButtonOperator(mainFrame, "Erase");

Assert.assertNotNull(buttonOperator);

Assert.assertNotNull(buttonOperator.getIcon());

}

@Test

public void testClearButton() {

JButtonOperator buttonOperator = new JButtonOperator(mainFrame, "Clear");

Assert.assertNotNull(buttonOperator);

Assert.assertNotNull(buttonOperator.getIcon());

}

@Test

public void testUndoButton() {

JButtonOperator buttonOperator = new JButtonOperator(mainFrame, "Undo");

Assert.assertNotNull(buttonOperator);

Assert.assertNotNull(buttonOperator.getIcon());

}

@Test

public void testRedoButton() {

JButtonOperator buttonOperator = new JButtonOperator(mainFrame, "Redo");

Assert.assertNotNull(buttonOperator);

Assert.assertNotNull(buttonOperator.getIcon());

}

@Test

public void testEditColorButton() {

JButtonOperator buttonOperator = new JButtonOperator(mainFrame, "Edit Color");

Assert.assertNotNull(buttonOperator);

}

}