Міністерство освіти і науки України

Харківський національний університет імені В.Н. Каразіна

Факультет комп’ютерних наук

**КУРСОВА РОБОТА**

з дисципліни «Математичні методи та технології тестування і верифікації програмного забезпечення»

Тема «Припущення помилок»

Виконала:

студентка 2 курсу, групи КС-21

Спеціальності:

122 - Компьютерні науки

Дібцева Анна Миколаївна

Перевірив:   
доц. Нарєжній О. П. ..............

Мелкозьорова О.М. ..............

Харків – 2020

**ЗМІСТ**

[ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ 3](#_Toc42261126)

[ВСТУП 4](#_Toc42261127)

[РОЗДІЛ 1 ОГЛЯД ЗАДАЧІ 6](#_Toc42261128)

[РОЗДІЛ 2 КЛАСИФІКАЦІЯ ПОМИЛОК ТА ТЕСТУВАННЯ 7](#_Toc42261129)

[2.1 Класифікація помилок 7](#_Toc42261130)

[2.2 Тестування 10](#_Toc42261131)

[2.3 Рівні тестування 12](#_Toc42261132)

[2.4 Види тестування 13](#_Toc42261133)

[2.4.1 Функціональні види тестування 13](#_Toc42261134)

[2.4.2 Нефункціональні види тестування 14](#_Toc42261135)

[2.4.3 Пов’язані зі змінами види тестування 16](#_Toc42261136)

[2.5 Життєвий цикл тестування 17](#_Toc42261137)

[РОЗДІЛ 3 ТЕСТУВАННЯ МЕТОДОМ ПРИПУЩЕННЯ ПОМИЛОК 19](#_Toc42261138)

[3.1 Тестування методом припущення помилок 19](#_Toc42261139)

[3.2 Приклад створення тесту із використанням методу припущення помилок 21](#_Toc42261140)

[ВИСНОВКИ 26](#_Toc42261141)

[ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 27](#_Toc42261142)

[ДОДАТОК А ПОВНА ВЕРСІЯ РЕАЛІЗАЦІЇ 28](#_Toc42261143)

# ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ

Програмна помилка – помилка у програмі чи системі, через яку програма демонструє неочікувану поведінку та, як наслідок, результат [3].

Специфікація - це шаблон проектування, за допомогою якого представлення правил бізнес логіки може бути перетворено у вигляді ланцюжка об'єктів, пов'язаних операціями булевої логіки [4].

База даних (БД) — це організована структура, яка призначена для зберігання, зміни та обробки взаємозалежної інформації, переважно великих обсягів [5].

Валідація(англ. **validation**) - це визначення відповідності розроблюваного за очікуваннями і потребами користувача, вимогам до системи[2].

Верифікація(англ. **verification**) – це процес визначення, чи виконують програмні засоби та їх компоненти вимоги, накладені на них на послідовних етапах життевого цикла розроблюваної системи.

XPath - XML path language.

# ВСТУП

Припущення помилок(англ. Error quessing) - це метод тестування програмного забезпечення на припущені помилки, які можуть переважати в коді. Цей метод допоможе знайти численні недоліки, котрі систематичні методи можуть не виявити.

Це заснована на досвіді методика тестування, в якій тестовий аналітик використовує свій досвід, щоб припускати проблемні області програми. Ця методика обов'язково вимагає кваліфікованих і досвідчених тестувальників [1].

На теперішній час велику частину життя людини займають електронні гаджети з додатками та соціальними мережами, інтернет із веб-сторінками і т.п. І ніша інтернету постійно розвивається.

При створенні додатків та інтернет ресурсів минаються такі етапи: розробка проекту, дизайн, програмування, тестування та реліз(випуск до публічного користування). Причому етап тестування надзвичайно важливий, адже він виявляє недоліки та проблемні зони продукту, допомагає у створенні ліпшої архітектури та полегшує подальшу підтримку продукта в довгостроковій перспективі, а також робить рефакторінг більш простим та безпечним.

Тестування програмного забезпечення(англ. Software Testing) - перевірка відповідності між реальним і очікуваним поведінкою програми, здійснювана на кінцевому наборі тестів, обраному певним чином. У більш широкому сенсі, тестування - це одна з технік контролю якості, що включає в себе активності з планування робіт (англ. Test Management), проектування тестів (англ. Test Design), виконання тестування (англ. Test Execution) і аналізу отриманих результатів (англ. Test Analysis) [2].

Тож у цій роботі буде розглянуто такий метод тестування програмного забезпечення, як «припущення помилок», буде зроблений докладний опис його класифікації та особливостей, а також розглянуте загальне поняття помилки та загальні етапи будування тестових сценарієв для програмного забезпечення.

# РОЗДІЛ 1 ОГЛЯД ЗАДАЧІ

У цій роботі проводяться дослідження методу припущення помилок для тестування програмного забезпечення.

Виходячи з теми, об'єкт дослідження це основні методи та етапи тестування програмного забезпечення.

Предметом дослідження даної курсової роботи є метод «припущення помилок», що використовується у тестуванні.

Виходячи з вищевикладеного, метою дослідження цієї роботи є розгляд методики тестування, що базується на створенні тестів, що перевірятимуть ті аспекти продукту, що на припущення тестувальника, можуть викликати помилки.

Тож у результаті ми маємо отримати докладний аналіз різних помилок, аспектів, недоліків та переваг даного способу тестування.

# РОЗДІЛ 2 КЛАСИФІКАЦІЯ ПОМИЛОК ТА ТЕСТУВАННЯ

Комп'ютерні технології все глибше проникають в повсякденне життя сучасної людини. Програмне забезпечення керує роботою безлічі оточуючих речей - від мобільних телефонів і комп'ютерів до пральних машин і кредитних карт. У будь – якому випадку, всі люди стикаються з тими чи іншими помилками в програмах. Задача ж тестувальника у тому, щоб запобігти помилкам у програмному продукті ще на етапі розробки. Завдяки тестуванню запобігаються безліч системних паламок і структура програми стає більш доступною для розгортування, рефакторінгу та підтримки.

## 2.1 Класифікація помилок

Різноманіття помилок, що зустрічаються в програмних виробах, породжує безліч їх класифікацій, необхідних для систематизації великого статистичного матеріалу, накопиченого при створенні і експлуатації програмного забезпечення.

Їх можна систематизувати по етапах розробки програмного продукту:

1. При описі вимог до програмного продукту з'являються помилки в специфікаціях, обумовлені неповним або некоректним розумінням вимог користувача розробником. Неякісно сформульовані користувачем вимоги також породжують помилки у вимогах до програмного виробу. Це найбільш серйозні помилки, виправлення яких на наступних етапах проектування, і особливо в процесі експлуатації, може зажадати величезних трудовитрат.

2. На етапі системного аналізу розв'язуваної проблеми можуть також з'являтися неточні формулювання вимог до програмного забезпечення, а також пропуск окремих вимог.

3. Архітектурне проектування програмного виробу може породжувати помилки, пов'язані з неточностями у функціональній декомпозиції і в інтерфейсах між модулями.

4. При алгоритмізації окремих функціональних задач можуть бути допущені помилки, обумовлені вибором невірного або неефективного алгоритму, неправильного математичного методу вирішення, невірною логікою алгоритму.

5. Серйозні помилки можуть бути допущені при проектуванні бази даних(БД) (в описі моделі, визначенні типів елементів і структур даних, а також зв'язків між ними, способах звернень до даних і т. п.)-

6. Найбільше число помилок відноситься до етапу кодування і тестування. Помилки цієї групи досліджені найбільш повно, і для них зібраний великий статистичний матеріал - помилки при програмуванні можна просто розділити на синтаксичні і логічні. Багато синтаксичні помилок виявляє транслятор, але чимало синтаксичних помилок при трансляції не виявляється.

Так в ряді мов програмування не виявляються пропуски операторів в програмі, порушення форматів введення/виведення, правил індексації масивів, відсутність початкових значень змінних і т.п. не виявляються, як правило, і помилки в вказівці типів переданих параметрів при викликах процедур.

Найбільш поширені такі помилки, зафіксовані при розробці програмного забезпечення:

1. Помилки в специфікаціях:

* неповна або неоднозначна специфікація;
* некоректне визначення проблеми.

2. Помилки проектування:

* нерозуміння специфікацій;
* некоректний алгоритм вирішення задачі;
* пропуск окремих кроків і варіантів алгоритму;
* помилки ініціалізації змінних;
* помилки в управлінні циклом;
* пропуски окремих типів даних.

3. Помилки кодування:

* неправильне розуміння проектних документів;
* помилки в керуючих структурах;
* помилки форматів вводу/виводу;
* помилки індексування;
* помилки ініціалізації та повторна ініціалізація;
* суперечливі найменування змінних;
* помилки міжпрограмних інтерфейсів;
* помилки в записі математичних виразів;
* переповнення, втрата значущості або точності;
* логічна помилка;
* відсутність перевірки прапорця і контролю установки початкових значень;
* помилки в операціях маніпулювання даними;
* помилки в інтерфейсі користувача і в сполученнях з системним програмним забезпеченням.

4. Помилки тестування і налагодження:

* неадекватні тестові набори даних;
* недостатнє або некоректне використання тестових варіантів і даних;
* неправильна інтерпретація результатів тестування;
* неправильні висновки про причини помилки і се джерелі;
* неправильне розуміння програмної специфікації при виборі тестових наборів даних.

5. Помилки в описі бази даних:

* помилки в сполученнях з базою даних;
* помилки в словниковій базі метаданих;
* помилки в ініціалізації бази даних.

6. Канцелярські помилки:

* описка;
* неправильне використання клавіш;
* пропуск або перестановка рядків програми.

7. Зовнішні помилки:

* відмови технічних пристроїв;
* реакція програмного забезпечення на збої в роботі технічних пристроїв;
* помилки через аварійні ситуації, що виникають в інших системах, з якими взаємодіє розглядається програмне виріб;
* помилки в документації.

Імовірність появи помилок перерахованих категорій залежить від багатьох факторів, але найбільш поширені логічні помилки (20-35%), помилки маніпулювання даними, помилки зовнішніх і внутрішніх інтерфейсів, описів даних (кожна категорія від 5 до 20%).

## 2.2 Тестування

Як було вже оговорено вище, тестуванням програмного забезпечення є перевірка відповідності між реальним і очікуваним поведінкою програми, здійснювана на кінцевому наборі тестів, обраному певним чином.

Тестування проводиться відповідно до певних цілей(які можуть бути задані явно або неявно) і різним рівнем точності. Визначення мети точним чином, що виражається кількісно, дозволяє забезпечити контроль результатів тестування. Тестові сценарії можуть розроблятися як для перевірки функціональних вимог(відомі як функціональні тести), так і для оцінки нефункціональних вимог. При цьому, існують такі тести, коли кількісні параметри і результати тестів можуть лише опосередковано говорити про задоволення цілям тестування(наприклад, «usability» — легкість, простота використання, в більшості випадків, не може бути явно описана кількісними характеристиками).

Можна виділити наступні найбільш розповсюджені та обосновані цілі тестування:

1. Приймальне тестування(англ. Acceptance Testing) – це формальний процес тестування(за документацією), 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.

Це можливо в тому випадку, якщо замовник бере на себе відповідальність, пов'язану з проведенням таких робіт, як сторона «приймаюча», або специфіковані типові завдання, успішна перевірка(тестування) яких дозволяє говорити про задоволення вимог замовника. Такі тести можуть проводитися як із залученням розробників системи, так і без них.

2. Підвищення вірогідності того, що додаток, призначений для тестування, буде працювати коректно при будь-яких обставинах.

3. Підвищення вірогідності того, що додаток, призначений для тестування, буде відповідати всім описаним вимогам.

4. Надання актуальної інформації щодо стану продукта на даний момент.

Також існує поняття альфа-і бета-тестування (Alpha and beta testing). Перед тим, як випускається програмне забезпечення, як мінімум, воно повинно проходити стадії альфа (внутрішнє пробне використання) і бета (пробне використання із залученням відібраних зовнішніх користувачів) версій. Звіти про помилки, що надходять від користувачів цих версій продукту, обробляються відповідно до певних процедур, що включають підтверджуючі тести (будь-якого рівня), що проводяться фахівцями групи розробки. Даний вид тестування не може бути заздалегідь спланований.

Етапи тестування такі:

1. Аналіз продукту;
2. Робота з вимогами;
3. Розробка стратегії тестування та планування процедур контроля якості;
4. Створення тестової документації;
5. Тестування прототипу;
6. Основне тестування;
7. Стабілізація;
8. Експлуатація.

## 2.3 Рівні тестування

Тестування розділяються на чотири рівні: модульне, інтеграційне, системне та операційне тестування. Розглянемо кожне з них окремо:

1. Модульне або компонентне тестування(англ. Unit Testing) перевіряє функціональність і шукає дефекти в частинах програми, які доступні і можуть бути протестовані окремо (модулі програм, об'єкти, класи, функції і т.д.).

**2. Інтеграційне тестування(англ. Integration Testing)** перевіряє взаємодію між компонентами системи після проведення компонентного тестування.

3. Системне тестування(англ. System Testing) перевіряє функціональні, так і нефункціональні вимоги в системі в цілому. При цьому виявляються дефекти, такі як невірне використання ресурсів системи, непередбачені комбінації даних даних рівня користувача, несумісність з оточенням, непередбачені сценарії використання, відсутня або невірна функціональність, незручність використання і т. д.

4. Операційне тестування(англ. Release Testing) перевіряє, що система задовольняє потребам користувача і виконує свою роль в середовищі своєї експлуатації, як це було визначено в бізнес моделі системи. Слід врахувати, що і бізнес модель може містити помилки. Тому так важливо провести операційне тестування як фінальний крок валідації. Крім цього, тестування в середовищі експлуатації дозволяє виявити і нефункціональні проблеми, такі як: конфлікт з іншими системами, суміжними в області бізнесу або в програмних і електронних оточеннях; недостатня продуктивність системи в середовищі експлуатації та ін. Очевидно, що знаходження подібних речей на стадії впровадження — критична і дорога проблема. Тому так важливо проведення не тільки верифікації, а й валідації, з самих ранніх етапів розробки програмного забезпечення.

## 2.4 Види тестування

В цілому у тестуванні розрізняють три види: функціональні, нефункціональні та пов’язані з зміненнями види тестування. Розглянемо кожне з них окремо.

### 2.4.1 Функціональні види тестування

Функціональні тести базуються на функціях і особливостях, а також взаємодії з іншими системами, і можуть бути представлені на всіх рівнях тестування.

Функціональні види тестування розглядають зовнішню поведінку системи.

Далі перелічені одні з найрозповсюджених видів функціональних тестів:

* функціональне тестування(англ. Functional testing);
* тестування інтерфейсу користувача(англ. GUI Testing);
* тестування безпеки(англ. Security and Access Control Testing);
* тестування взаємодії(англ. Interoperability Testing).

Функціональне тестування розглядає заздалегідь зазначену поведінку і ґрунтується на аналізі специфікацій функціональності компонента або системи в цілому.

Тестування інтерфейсу користувача (англ. GUI Testing) — це функціональна перевірка інтерфейсу на відповідність вимогам - розмір, шрифт, колір, послідовну поведінку.

Тестування безпеки(англ. Security and Access Control Testing) — це стратегія тестування, яка використовується для перевірки безпеки системи, а також для аналізу ризиків, пов'язаних із забезпеченням цілісного підходу до захисту програми, атак хакерів, вірусів, несанкціонованого доступу до конфіденційних даних.

Тестування взаємодії (англ. Interoperability Testing) - це функціональне тестування, яке перевіряє здатність програми взаємодіяти з одним і більше компонентами або системами і включає в себе тестування сумісності (англ. compatibility testing) і інтеграційне тестування

### 2.4.2 Нефункціональні види тестування

Нефункціональне тестування описує тести, необхідні для визначення характеристик програмного забезпеченню, котрі можуть бути виміряні різноманітними величинами. В цілому, це тестування того, «як» працює система.

Далі перелічені одні з найрозповсюджених видів функціональних тестів:

* всі види тестування продуктивності: навантажувальне тестування(англ. Performance and Load Testing), стресове тестування(англ. Stress Testing), тестування стабільності або надійності(англ. stability/Reliability Testing) та об'ємне тестування(англ. Volume Testing);
* тестування установки(англ. Installation testing);
* тестування зручності користування(англ. Usability Testing);
* тестування на відмову та відновлення(англ. Failover and Recovery Testing);
* конфігураційне тестування(англ. Configuration Testing).

Навантажувальне тестування(англ. Performance and Load Testing) - це автоматизоване тестування, що імітує роботу певної кількості бізнес користувачів на будь-якому загальному(поділюваному ними) ресурсі.

Стресове тестування(англ. Stress Testing) дозволяє перевірити наскільки додаток і система в цілому працездатні в умовах стресу і також оцінити здатність системи до регенерації, тобто до повернення до нормального стану після припинення впливу стресу. Стресом в даному контексті може бути підвищення інтенсивності виконання операцій до дуже високих значень або аварійна зміна конфігурації сервера. Також одним із завдань при стресовому тестуванні може бути оцінка деградації продуктивності, таким чином цілі стресового тестування можуть перетинатися з цілями тестування продуктивності.

Тестування стабільності або надійності(англ. stability/Reliability Testing). Завданням тестування стабільності(надійності) є перевірка працездатності програми при тривалому(багатогодинному) тестуванні із середнім рівнем навантаження.

Об'ємне тестування(англ. Volume Testing). Завданням об'ємного тестування є отримання оцінки продуктивності при збільшенні обсягів даних в базі даних додатка.

Тестування установки(англ. Installation testing) направлено на перевірку успішної інсталяції та налаштування, а також оновлення або видалення програмного забезпечення.

Тестування зручності користування - це метод тестування, спрямований на встановлення ступеня зручності використання, здатності до навчання, зрозумілості та привабливості для користувачів розроблюваного продукту в контексті заданих умов. Сюди також входить User eXperience - відчуття, що випробовується користувачем під час використання цифрового продукту, в той час як User interface — це інструмент, що дозволяє здійснювати інтеракцію «користувач—веб-ресурс».

Тестування на відмову і відновлення(англ. Failover and Recovery Testing) перевіряє тестований продукт з точки зору здатності протистояти і успішно відновлюватися після можливих збоїв, що виникли у зв'язку з помилками програмного забезпечення, відмовами обладнання або проблемами зв'язку(наприклад, відмова мережі). Метою даного виду тестування є перевірка систем відновлення(або дублюючих основний функціонал систем), які, в разі виникнення збоїв, забезпечать збереження і цілісність даних тестованого продукту.

Конфігураційне тестування(англ. Configuration Testing) - спеціальний вид тестування, спрямований на перевірку роботи програмного забезпечення при різних конфігураціях системи(заявлених платформах, підтримуваних драйверах, при різних конфігураціях комп'ютерів і т. д.)

### 2.4.3 Пов’язані зі змінами види тестування

Після проведення необхідних змін, таких як виправлення бага/дефекту, програмне забезпечення повинно бути пере тестовано для підтвердження того факту, що проблема була дійсно вирішена. Нижче перераховані види тестування, які необхідно проводити після установки програмного забезпечення, для підтвердження працездатності програми або правильності здійсненого виправлення дефекту:

* димове тестування(англ. Smoke Testing);
* регресійне тестування(англ. Regression Testing);
* повторне тестування(англ. Re-testing);
* тестування збірки(англ. Build Verification Test);
* санітарне тестування або перевірка узгодженості/справності(англ. Sanity Testing).

Димове (англ. Smoke Testing) тестування розглядається як короткий цикл тестів, що виконується для підтвердження того, що після складання коду(нового або виправленого) встановлюється додаток, стартує і виконує основні функції.

Регресійне тестування(англ. Regression Testing) - це вид тестування спрямований на перевірку змін, зроблених в додатку або навколишньому середовищі(лагодження дефекту, злиття коду, міграція на іншу операційну систему, базу даних, веб сервер або сервер додатка), для підтвердження того факту, що існуюча раніше функціональність працює як і раніше. Регресійними можуть бути як функціональні, так і нефункціональні тести.

Повторне тестування(англ. Re-testing) - тестування, під час якого виконуються тестові сценарії, що виявили помилки під час останнього запуску, для підтвердження успішності виправлення цих помилок.

Різниця між регресивним та повторним тестуванням у тому, що повторне тестування перевіряє виправлення багів, а регресивне тестування перевіряє, що виправлення багів, а також будь-які зміни в коді програми, не вплинули на інші модулі програмного забезпечення і не викликало нових багів.

Тестування збірки(англ. Build Verification Test) — тестування спрямоване на визначення відповідності, випущеної версії, критеріям якості для початку тестування. За своїми цілями є аналогом димового тестування, спрямованого на приймання нової версії в подальше тестування або експлуатацію. Вглиб воно може проникати далі, в залежності від вимог до якості випущеної версії.

Санітарне тестування(англ. Sanity Testing) - це вузьконаправлене тестування достатнє для доказу того, що конкретна функція працює відповідно до заявлених у специфікації вимог. Є підмножиною регресійного тестування. Використовується для визначення працездатності певної частини програми після змін вироблених в ній або навколишньому середовищі. Зазвичай виконується вручну.

## 2.5 Життєвий цикл тестування

Життєвий цикл тестування ділять на сім етапів:

1. Перша стадія це загальне планування і аналіз вимог. Вона об'єктивно необхідна як мінімум для того, щоб мати уявлення про предмет тестування, об’єм роботи, складність та ресурси, необхідні для цієї роботи. Як правило, отримати відповіді на ці питання неможливо без аналізу вимог, тому що саме вимоги є первинним джерелом відповідей.

2. Друга стадія - уточнення критеріїв приймання. Вона дозволяє сформулювати або уточнити метрики і ознаки можливості або необхідності початку тестування, припинення і відновлення тестування, завершення або припинення тестування.

3. Третя стадія - уточнення стратегії тестування. Вона являє собою ще одне звернення до планування, але на локальному рівні: розглядаються і уточнюються ті частини стратегії тестування, які актуальні для поточної ітерації.

4. На четвертій стадії виконується розробка тест-кейсів. Вона присвячена розробці, перегляду, уточненню, доопрацюванню, переробці та іншим діям з тест-кейсами, наборами тест-кейсів, тестовими сценаріями та іншими артефактами, які будуть використовуватися при безпосередньому виконанні тестування.

5. На п’ятій стадії тест-кейси виконуються.

6. На шостій стадії виконується фіксація знайдених дефектів. П’ята та шоста стадії тісно пов'язані між собою і фактично виконуються паралельно: дефекти фіксуються відразу за фактом їх виявлення в процесі виконання тест-кейсів. Однак найчастіше після виконання всіх тест-кейсів і написання всіх звітів про знайдені дефекти проводиться явно виділена стадія уточнення, на якій всі звіти про дефекти розглядаються повторно з метою формування єдиного розуміння проблеми і уточнення таких характеристик дефекту, як важливість і терміновість.

7. Під час сьомої стадії виконується аналіз результатів тестування.

8. І на восьмій стадії - звітність. Сьома та восьма стадії також тісно пов'язані між собою і виконуються практично паралельно. Сформульовані на стадії аналізу результатів висновки безпосередньо залежать від плану тестування, критеріїв приймання та уточненої стратегії, отриманих на стадіях 1, 2 і 3. Отримані висновки оформляються на стадії 8 і служать основною для стадій 1, 2 і 3 наступної ітерації тестування. Таким чином, цикл замикається.

# РОЗДІЛ 3 ТЕСТУВАННЯ МЕТОДОМ ПРИПУЩЕННЯ ПОМИЛОК

## 3.1 Тестування методом припущення помилок

Метод припущення помилок – це метод тестування програмного забезпечення на припущені помилки, які можуть переважати в коді. Цей метод допоможе знайти численні недоліки, котрі систематичні методи можуть не виявити. І, як правило, він має буди застосований після того, як більшість формальних методів тестування(за документацією) були проведені, тому що він виконує «зачищення» і йде як доповнення до систематичних методів[7].

Ця методика заснована на досвіді тестування, в якій тестовий аналітик використовує свій досвід, щоб припускати проблемні області програми. Вона обов'язково вимагає кваліфікованих і досвідчених тестувальників.

Тестові випадки для пошуку проблем в програмному забезпеченні написані на основі попереднього досвіду тестування з аналогічними додатками. Таким чином, обсяг тестових випадків, як правило, залежить від того, який тип тестування тестовий аналітик був задіяний в минулому. Техніка вгадування помилок не слідує яким-небудь певним правилам. Тестові випадки можуть бути розроблені в залежності від ситуації, або на основі функціональних документів, або при виявленні несподіваної/недокументованої помилки під час тестування операцій [8].

Наприклад, якщо аналітик припускає, що сторінка входу схильна до помилок, то тестувальники будуть писати докладні тестові випадки, концентруючись на сторінці входу. Вони можуть придумати різні комбінації даних для перевірки сторінки входу в систему.

Для розробки тестових випадків, заснованих на методі вгадування помилок, аналітик може використовувати минулий досвід для визначення умов.

Цей метод може бути використаний на будь-якому рівні тестування і для перевірки поширених помилок, таких як:

* ділення на нуль;
* уведення порожніх місць у текстові поля;
* натиск кнопки відправки без уведення значень;
* загрузка файлі, перевищуючих максимальне обмеження;
* виключення нульового вказівника;
* неприпустимий параметр.

Рівень досягнення цієї техніки в основному залежить від здібностей тестувальників.

Основна мета цієї методики - вгадати можливі помилки в тих областях, де формальне тестування не працюватиме [1].

В результаті повинен зібратися повний набір тестів без будь-яких пропущених областей і без створення надлишкових тестів.

Вгадування помилок - це в основному інтуїтивний і спеціальний процес, тому дуже важко дати чітко визначену процедуру цього методу. Основний спосіб полягає в тому, щоб спочатку перерахувати всі можливі помилки або області, схильні до помилок в додатку, а потім створити тестові випадки на основі цього списку.

Для припущення помилок використовують наступні фактори:

* уроки, витягнуті з минулих релізів;
* інтуїція тестувальника;
* вивчення історії;
* попередні дефекти;
* виробництво квитків;
* перегляньте контрольний список;
* користувальницький інтерфейс програми;
* попередні результати випробувань;
* звіти про ризики програми;
* різноманітність даних, що використовуються для тестування;
* загальні правила тестування.

Тестування программного продукту по доступу до коду розділяють на:

* тестування «білого ящика» - тестування программного продукта з доступом до коду [6];
* тестування «чорного ящика» - тестування программного продукта без доступа до коду[6];
* тестування «сірого ящика» - тестування программного продукта з обмеженим доступом до коду, коли тестувальник не працює з кодом продукту, але ознайомлений із внутрішньою структурою програми та взаємодією між компонентами[6].

Метод припущення помилок це один з видів тестування чорного ящика, і його можна розглядати як неструктурований підхід до тестування програмного забезпечення, бо тестувальник не має уявлення про внутрішню структуру програмного продукту.

Перевагами методу припущення помилок те, що він виявляється дуже ефективним при використанні в поєднанні з іншими формальними методами тестування. А також він розкриває ті дефекти, які інакше було б неможливо виявити за допомогою формального тестування. Таким чином, досвід роботи тестувальника економить масу часу і сил.

В свою чергу, головний недолік цієї методики полягає в тому, що вона залежить від особистості тестувальника і, таким чином, його досвід контролює якість тестових випадків. Тому тільки досвідчені тестувальники можуть виконати це тестування. Також він не може гарантувати, що програмне забезпечення досягло очікуваного рівня якості.

## 3.2 Приклад створення тесту із використанням методу припущення помилок

Тепер розглянемо тест написаний із використанням методу припущення помилок. Завдання тесту буде перевірити коректність роботи крайніх кнопок переходу між сторінками каталогу. Для цього створен проект Maven в IntelliJ IDEA. В файлі ­pom.xml прописані залежності Selenium та JUnit, необхідні для роботы. Після чого був створен клас Error\_guessing\_test, де написан сам тест.

Для початку прописуємо необхідний елемент - об'єкт класу WebDriver, за допомогою якого буде проводитися взаємодія з браузером. Та поле типу String для збереження посилання на сторінку, з якою буде створена взаємодія. Далі в методі з анотацією @BeforeClass вказуємо розташування веб драйвера і створюємо його об'єкт.

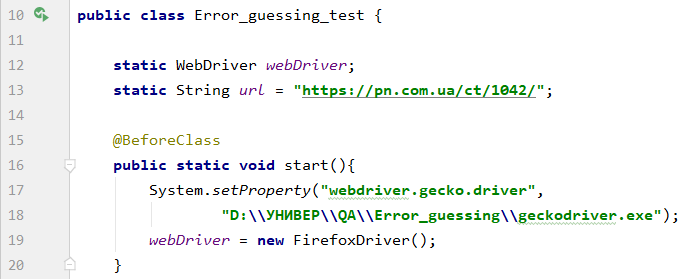


Рисунок 3.1 – Початок створення класу для тестування

У методі з анотацією @AfterClass вказуємо, що необхідно закрити веб драйвер після закінчення тесту. В результаті виконання написаних методів буде автоматично створюватися об'єкт класу WebDriver і по закінченню всіх тестів він буде автоматично закриватися.

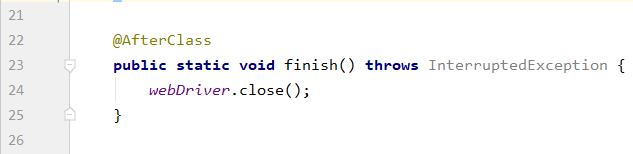


Рисунок 3.2 – Метод finish() класу тестування

Далі створюємо метод errorGuessingTest(), де буде проходити основне тестування.

На початку тесту прописане відкриття веб-сторінки pn.com.ua. Далі за XML path language(XPath) знаходимо назву першого товару у списку, та зберігаємо її у поле, для подальшого використання у тестуванні. Далі, так само, за XPath знаходимо кнопку переходу на попередню сторінку і натискаємо на неї. Робиться це, для перевірки коректності реагування сторінки. Вона повинна залишитися на першій сторінці каталогу, а не переходити на останню. Тож, щоб перевірити реакцію веб-сторінки ми послідовно зрівнюємо очікуваний результат з отриманим. Перевіряємо ми такі аспекти: назву першого товару на сторінці з записаною раніше, та очікуване розташування номерів сторінок каталогу. Якщо хочаб один із цих аспектів не виконається, це буде значити, что перехід на іншу сторінку переліку відбувся. Описані дії зображені на малюнку 3.3:

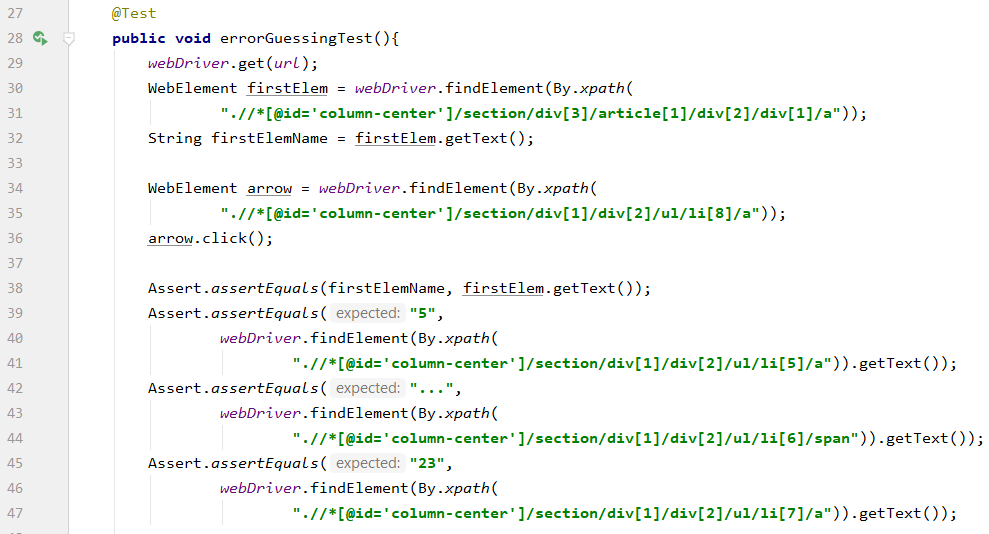


Рисунок 3.3 – Метод errorGuessingTest(), частина 1

Далі за XPath знаходимо кнопку переходу на останню – 23 сторінку, і натискаємо на неї. Тут перевіряємо, що ми не на першій сторінці каталогу, використавши метод Assert.assertNotSame(String arg0, String arg1), що будет пройдений, якщо назва першого товару першої сторінки не співпаде з назвою першого товару теперішньої сторінки. І так само, як і в перший раз перевіряємо очікуване розположення нумерації сторінок. Описані дії зображені на малюнку 3.4:

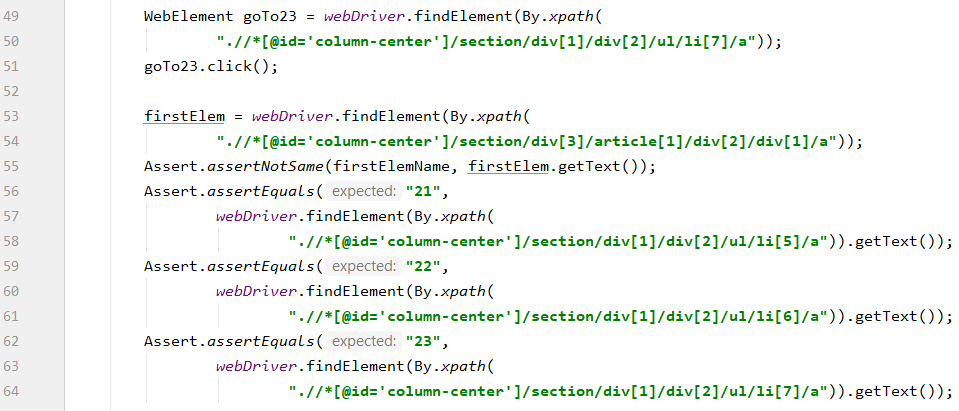


Рисунок 3.4 – Метод errorGuessingTest(), частина 2

Далі, як і на початку тесту, знаходимо та натискаємо кнопку переходу на іншу сторінку, але на цей раз, на наступну. І виконуємо ті ж самі кроки перевірки, як на рисунку 3.4.

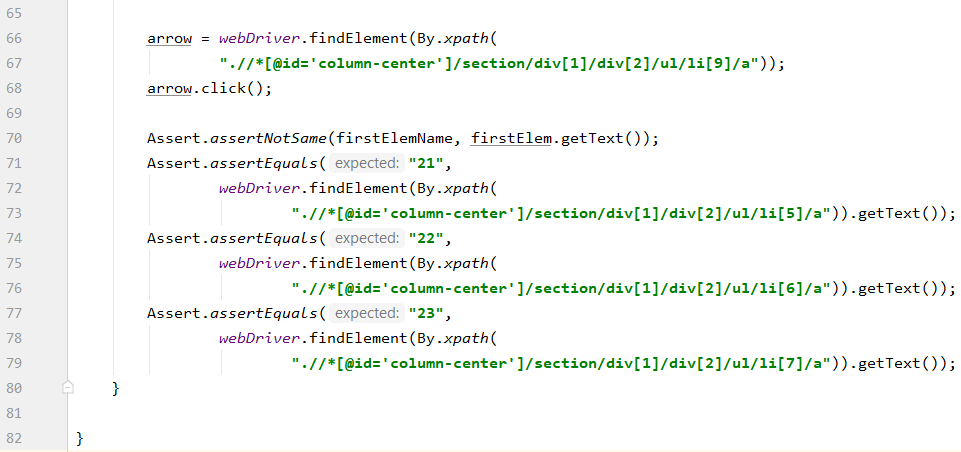


Рисунок 3.5 – Метод errorGuessingTest(), частина 3

Таким чином тест із використанням методу припущення помилок повністю оформлений, та виконаний успішно, що виявляє, що дефекту у цій частині програми немає. Повний код реалізації наведений у Додатку А.І

# ВИСНОВКИ

На сьогодні, в нашому сучасному житті велику частину займають технології. І тестування програмного забезпечення надзвичайно важливе, адже воно виявляє недоліки та проблемні зони продукту, допомагає у створенні ліпшої архітектури та полегшує подальшу підтримку продукта в довгостроковій перспективі, а також робить рефакторінг більш простим та безпечним.

У цій роботі було розглянуто поняття помилки, тестування, його рівнів та видів. Докладно був описаний метод припущення помилок у тестуванні. Були розглянуто його визначення, суть, мету, переваги та недоліки цього методу. Також із його використанням був створений невеликий тест програмних компонентів веб-сторінки, що був успішно пройдений.

І хоча припущення помилок є одним з ключових методів тестування, воно не забезпечує повного охоплення програми. Метод також не може гарантувати, що програмне забезпечення досягло очікуваного рівня якості. Ця техніка повинна бути об'єднана з іншими методами, щоб дати кращі результати. А для проведення такого тестування необхідно мати кваліфікованих і досвідчених тестувальників.

Істотною перевагою цього методу тестування є те, що він виявляє дефекти в тих областях, які в іншому випадку залишаються непоміченими іншими формальними методами тестування.

Таким чином, можна зробити висновок, що припущення помилок це корисний метод для професійних тестувальників, бо їх досвід формує основу для цієї техніки тестування разом із переважаючими правилами та помилками, котрі ми виділили у цій роботі.

# ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. What is Error Guessing Technique? - Software Testing Help // https://  
   www.softwaretestinghelp.com/error-guessing-technique/. Дата звернення: 02.06.2020.
2. Тестирование. Фундаментальная теория – Хабр // [https://habr.com/ru/  
   post/279535/](https://habr.com/ru/post/279535/). Дата звернення: 02.06.2020.
3. Программная ошибка – Википедия // https://ru.wikipedia.org/wiki/  
   Программная\_ошибка
4. Спецификация // https://kartaslov.ru/значение-слова/спецификация
5. Герасимик Іван Що таке база даних? // <http://apeps.kpi.ua/shco-take-basa-danykh>. Дата звернення: 03.06.2020.
6. Классификация видов тестирования // https://qa-academy.by/qaacademy/  
   news/klassifikaciya-vidov-testirovaniya/. Дата звернення: 04.06.2020.
7. Error Guessing in Software Testing // https://qatestlab.com/resources/  
   knowledge-center/error-guessing-in-software-testing/. Дата звернення: 05.06.2020.
8. Error quessing – Wikipedia // <https://en.wikipedia.org/wiki/Error_guessing>. Дата звернення: 04.06.2020.

# ДОДАТОК А ПОВНА ВЕРСІЯ РЕАЛІЗАЦІЇ

Лістинг 1 – Реалізація тесту з використанням методу припущення помилок

import org.junit.AfterClass;  
import org.junit.Assert;  
import org.junit.BeforeClass;  
import org.junit.Test;  
import org.openqa.selenium.By;  
import org.openqa.selenium.WebDriver;  
import org.openqa.selenium.WebElement;  
import org.openqa.selenium.firefox.FirefoxDriver;  
  
public class Error\_guessing\_test {  
 static WebDriver *webDriver*;  
 static String *url* = "https://pn.com.ua/ct/1042/";  
 @BeforeClass  
 public static void start(){  
 System.*setProperty*("webdriver.gecko.driver", "D:\\УНИВЕР\\QA\\Error\_guessing\\geckodriver.exe");  
 *webDriver* = new FirefoxDriver();  
 }  
  
 @Test  
 public void errorGuessingTest(){  
 *webDriver*.get(*url*);  
 WebElement firstElem = *webDriver*.findElement(By.*xpath*(".//\*[@id='column-center']/section/div[3]/article[1]/div[2]/div[1]/a"));  
 String firstElemName = firstElem.getText();  
  
 WebElement arrow = *webDriver*.findElement(By.*xpath*(".//\*[@id='column-center']/section/div[1]/div[2]/ul/li[8]/a"));  
 arrow.click();  
  
 Assert.*assertEquals*(firstElemName, firstElem.getText());  
 Assert.*assertEquals*("5",  
 *webDriver*.findElement(By.*xpath*(".//\*[@id='column-center']/section/div[1]/div[2]/ul/li[5]/a")).getText());  
 Assert.*assertEquals*("...",  
 *webDriver*.findElement(By.*xpath*(".//\*[@id='column-center']/section/div[1]/div[2]/ul/li[6]/span")).getText());  
 Assert.*assertEquals*("23",  
 *webDriver*.findElement(By.*xpath*(".//\*[@id='column-center']/section/div[1]/div[2]/ul/li[7]/a")).getText());  
  
 WebElement goTo23 = *webDriver*.findElement(By.*xpath*(".//\*[@id='column-center']/section/div[1]/div[2]/ul/li[7]/a"));  
 goTo23.click();  
  
 firstElem = *webDriver*.findElement(By.*xpath*(".//\*[@id='column-center']/section/div[3]/article[1]/div[2]/div[1]/a"));  
 Assert.*assertNotSame*(firstElemName, firstElem.getText());  
 Assert.*assertEquals*("21",  
 *webDriver*.findElement(By.*xpath*(".//\*[@id='column-center']/section/div[1]/div[2]/ul/li[5]/a")).getText());  
 Assert.*assertEquals*("22",  
 *webDriver*.findElement(By.*xpath*(".//\*[@id='column-center']/section/div[1]/div[2]/ul/li[6]/a")).getText());  
 Assert.*assertEquals*("23",  
 *webDriver*.findElement(By.*xpath*(".//\*[@id='column-center']/section/div[1]/div[2]/ul/li[7]/a")).getText());  
  
 arrow = *webDriver*.findElement(By.*xpath*(".//\*[@id='column-center']/section/div[1]/div[2]/ul/li[9]/a"));  
 arrow.click();  
  
 Assert.*assertNotSame*(firstElemName, firstElem.getText());  
 Assert.*assertEquals*("21",  
 *webDriver*.findElement(By.*xpath*(".//\*[@id='column-center']/section/div[1]/div[2]/ul/li[5]/a")).getText());  
 Assert.*assertEquals*("22",  
 *webDriver*.findElement(By.*xpath*(".//\*[@id='column-center']/section/div[1]/div[2]/ul/li[6]/a")).getText());  
 Assert.*assertEquals*("23",  
 *webDriver*.findElement(By.*xpath*(".//\*[@id='column-center']/section/div[1]/div[2]/ul/li[7]/a")).getText());  
 }  
 @AfterClass  
 public static void finish() throws InterruptedException {  
 *webDriver*.close();  
 }  
}