Міністерство освіти і науки України

Харківський національний університет імені В.Н.Каразіна

Факультет комп’ютерних наук

Курсова робота на тему:

**«Випадкове тестування»**

Виконав:

студент групи КБ-21

Дерев’янко Ярослав

Перевірив:

доц. Нарєжній О.П.

Харків

2019

Зміст

[Вступ 3](#_Toc296355562)

[1 Теоретичні основи 7](#_Toc296355564)

[1.1 Призначення тестування 7](#_Toc296355565)

[1.2 Класифікація життєвих циклів ПЗ та їх вплив на процес тестування 8](#_Toc296355566)

[1.3 Тестування і якість програмного продукту 12](#_Toc296355567)

[1.4 Аналіз процесу тестування ПЗ 13](#_Toc296355568)

[1.5 Автоматизація тестування 15](#_Toc296355569)

[1.6 Переваги автоматичного тестування 15](#_Toc296355570)

[1.7 Недоліки автоматизованого тестування 16](#_Toc296355571)

[1.8 Класифікація видів тестування 16](#_Toc296355570)

[1.9 Випадкове тестування та його історія 17](#_Toc296355571)

[2 Фреймворк тестування JUnit 20](#_Toc296355564)

[2.1 Інформація про JUnit 20](#_Toc296355565)

[2.2 Початок роботи з JUnit 20](#_Toc296355566)

[2.3 Синтаксис JUnit 21](#_Toc296355567)

[3 Програмна реалізація 24](#_Toc296355564)

[3.1 Імпорт залежностей Maven до створеного проекту 24](#_Toc296355565)

[3.2 Створення та реалізація тестів за допомогою JUnit 25](#_Toc296355566)

[Висновки 33](#_Toc296355567)

[Перелік джерел посилання 36](#_Toc296355567)

[Додаток А Код програми 37](#_Toc296355567)

[Додаток Б Вміст pom.xml 42](#_Toc296355567)

Вступ

Тестування – важлива частина будь-якої програми контролю якості. Якість програмного продукту характеризується набором властивостей, які визначають наскільки продукт «гарний» з огляду зацікавлених сторін, таких як: замовник продукту, користувач, який в решті-решт буде працювати з нашою програмою, розробники і тестувальники продукту, співробітники відділів маркетингу та продажу та інші. Кожен з учасників може мати своє, відмінне від усіх інших, уявлення про продукт і про те, наскільки високою є його якість.

Автори можуть дотримуватися різних методів розробки, сперечатися про переваги того чи іншого підходу в плануванні процесів або документуванні процедур, а також гнучкості останніх, проте загальна схема створення інформаційних систем досить проста і складається в основному з одних і тих же модулів і процесів:

* управління проектом у вигляді координації зусиль проектної команди, спрямованих на досягнення цілей проекту оптимальним шляхом;
* постановка задачі у вигляді визначення вимог і подальших робіт з ними;
* управління змінами в проекті: зміна може стосуватися як безпосередньо самих вимог до системи, так і торкатися організаційної схеми процесу, і може виникати через бажання Замовника або бути наслідком виявлених в системі дефектів;
* розробка ПЗ, як безпосереднє створення програмної реалізації функціональних вимог і проектування схем зберігання і виконання дій над інформацією в системі;
* тестування ПЗ - процес, що вирішує завдання верифікації відповідності вимог висунутих до системи і їх програмної реалізації;
* експлуатація ПО як сума завдань, спрямована на забезпечення технічної і технологічної підтримки процесу роботи системи, яка включає підтримку і необхідний системне адміністрування.

Як бачимо, процес розробки системи складається з декількох взаємопов'язаних модулів, якими вже в свою чергу і оперують автори методологій і підходів, зміщуючи пріоритети між напрямками або змішуючи завдання декількох напрямків (пропонуючи, наприклад, здійснення завдань тестування в рамках діяльності по безпосередній розробці програмної реалізації і т.д.). Суть залишається незмінною - є технологічний ланцюжок процесів розробки інформаційних систем, модулі якого взаємозалежні і не можуть функціонувати у відриві один від одного.

Таким чином, постановка задачі забезпечення якості продукту в першу чергу складається з визначення зацікавлених осіб, їх критеріїв якості і потім знаходження оптимального рішення, що задовольняє цим критеріям. Тестування є одним з найбільш поширених способів забезпечення якості розробки програмного забезпечення і входить в набір ефективних засобів сучасної системи забезпечення якості програмного продукту.

З технічної точки зору тестування полягає у виконанні програми на деякій множині вихідних даних та порівнянні одержуваних результатів із заздалегідь відомими (еталонними) з метою встановити відповідність різних властивостей і характеристик програми замовленим властивостям.

Існує декілька визначень та навіть понять, якими характеризується процес тестування:

Налагодження (Debug, Debugging) - процес пошуку, локалізації та виправлення помилок в програмі.

Тестування - це процес виконання ПЗ системи або компонента в умовах аналізу або запису отриманих результатів з метою перевірки (оцінки) деяких властивостей об'єкта, який піддають тестуванню.

Тестування - це процес аналізу пункту вимог до ПЗ з метою фіксації відмінностей між існуючим станом ПЗ і необхідним (що свідчить про виникнення помилки або декількох(багатьох) помилок) при експериментальній перевірці відповідного пункту вимог.

Тестування - це контрольоване виконання програми на кінцевій множині тестових даних і аналіз результатів цього виконання для пошуку помилок.

Іноді терміни «тестування» і «налагодження» використовують як взаємозамінні, але, як правило, ці два процеси розрізняють за рахунок деяких відмінностей. Тестування - це засіб виявлення помилок, тоді як налагодження є засобом пошуку та усунення причин вже виявлених помилок.

Хорошим вважають тестовий варіант з високою ймовірністю виявлення ще не розкритої помилки. Успішним можна назвати тест, який виявляє помилку, яка не давала про себе знати до цього моменту.

Метою проектування тестових варіантів є систематичне виявлення різних класів помилок при мінімальних витратах часу і вартості.

Тестування забезпечує:

1.Виявлення помилок.

2.Демонстрацію відповідності функцій програми її призначенням.

3.Демонстрацію реалізації вимог до характеристик програми.

4.Відображення надійності як індикатора якості програми.

Тестування не може показати відсутність дефектів (воно може показувати тільки присутність дефектів). Важливо пам'ятати про це під час проведення тестування.

Існує безліч технік та методів тестування. Їх ми розглянемо далі. Метою цієї курсової роботи є детальний розгляд процесу тестування як складової процесу забезпечення якості ПЗ на прикладі Випадкового тестування (Random testing), а також теоретичне обґрунтування основних положень даного процесу і перевірка практично на основі розробленої програми.

Тестування на основі випадкових даних (Random testing) - техніка тестування, в якій вхідні дані, дії або навіть самі тест-кейси вибираються на основі (псевдо)випадкових значень так, щоб відповідати операційному профілю (Operational profile) - підмножині дій, які відповідають певній ситуації або сценарію роботи з проектом.

1 Теоретичні основи

**1.1 Призначення тестування**

Причиною неправильної поведінки програми є людські помилки.. Причини помилок досить різноманітні: відсутність достатніх знань та досвіду, висока складність розроблюваної системи або просто фізична втома. Щоб запобігти виникненню помилок у результатах нашої роботи, ми мусимо постійно її перевіряти: самостійно або залучивши іншу особу до перевірки, що часто є більш ефективним з ряду причин, таких, наприклад, як психологічних - ми не схильні критично відноситись до власних результатів, або ми просто можемо не помітити помилок з причини нестачі інформації або попередньо зроблених неправих припущень.

Подібну перевірку зазвичай називають тестуванням. Якщо говорити простою мовою: «тестування - це перевірка, що програма робить те, що вона має робити, і не робить того, чого робити не повинна». Тестування потрібне, щоб знизити ризики появи проблем шляхом раннього виявлення помилок, і як результат, знизити ймовірність настання неочікуваних негативних явищ та ситуацій під час використанні програмного забезпечення користувачем, що загрожують прибутку, досягненню мети або навіть життю.

Однією з найсуттєвіших причин вдаватися до тестування є наступне: сьогодні у епоху новітніх гаджетів та швидкісного Інтернету користувач очікує зручного встановлення програми за одним дотиком, без переходу по безлічі сторінок та завантажування зайвого. Якщо ви не можете надати користувачу таку можливість - він дуже просто може завантажити все необхідне з ресурсу конкурента. Зросли також очікування користувачів і щодо зручності інтерфейсу програмного забезпечення. Інтерфейс повинен бути зручним, зрозумілим для користувача, простим у використанні та за своїми функціями, має підтримувати різні операційні системи, браузери та конфігурації користувачів. Все це має підлягати тестуванню настільки ж ретельному як і правильність функціонування, адже якість нашого програмного забезпечення в основному визначається задоволенням очікувань замовника, тобто потенційного користувача.

В наш час, тестування - процес, що включає у себе планування, підготовку, проектування, створення та виконання тестів та екземплярів тестового середовища, документування та аналіз результатів.

Навіть беручи до уваги величезну кількість методів і прийомів тестування, зазвичай, велика частина помилок залишається не виявленою до моменту початку реального вжитку системи або програмного забезпечення. Основною причиною цього є недолік часу та ресурсів на тестування.

Автоматизація тестування програмного забезпечення має суттєво допомогти інженерам у перевірках з метою збільшити об'єми тестування, необхідного для сучасних продуктів. Автоматизація здатна значно скоротити витрати часу на тестування та навіть практично прирівняти до нуля його трудомісткість.

**1.2 Класифікація життєвих циклів ПЗ та їх вплив на процес тестування**

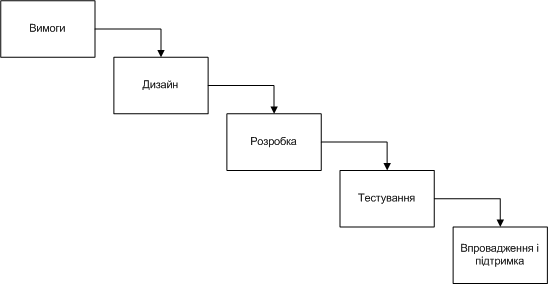
Наразі програмні продукти мають схильність до значного ускладнення, як в розумінні, так і в реалізації. Це зумовлено як розвитком комп’ютерної індустрії, так і людства в цілому. З’являються нові методи, програмні продукти розвиваються, в них з'являються нові функції і можливості. При цьому зростає складність цих продуктів, взаємозв'язків їхніх компонентів і підсистем, інтеграція та взаємодія з іншими додатками. Всі вищеперераховані процеси дуже сильно збільшують ймовірність виникнення дефектів та несподіваних помилок в програмному забезпеченні.

Тому зараз тестування є не просто необхідною, а просто неодмінною умовою безперебійного та успішного функціонування як для гігантів індустрії, так і для дрібних розробників програмного забезпечення "під замовлення".

Напевно головною особливістю тестування є те, що воно не може бути самостійною активністю. Тестування займає своє визначене місце у моделі життєвого циклу розробки продукту і саме ця модель має найбільший вплив на організацію процесу тестування. Саме від моделі залежить що саме і коли саме повинно бути перевірено, об’єм регресійного тестування, модель визначає які техніки тестування буде використано та багато інших значних та незначних аспектів. Класичними вважаються – каскадна, V-модель та ітеративна. Найпопулярнішими ж сьогодні є гнучкі модель розробки. Розглянемо детально кожну з них.

**Каскадна модель або «водоспад»**

Одна з перших моделей розробки ПЗ, яку запропонував 1970 році Уінстон Ройс. Ця модель передбачає послідовне виконання всіх етапів проекту в порядку, який чітко фіксується ще на початку розробки ідеї (рис.1.1). Перехід на наступний етап означає повне завершення робіт на попередньому. Все, що визначено на стадії формування вимог, суворо документується у вигляді технічного завдання аж до повного закінчення розробки проекту. Кожна стадія завершується створенням повного комплекту документації, щоб розробка могла бути продовжена іншою командою розробників.

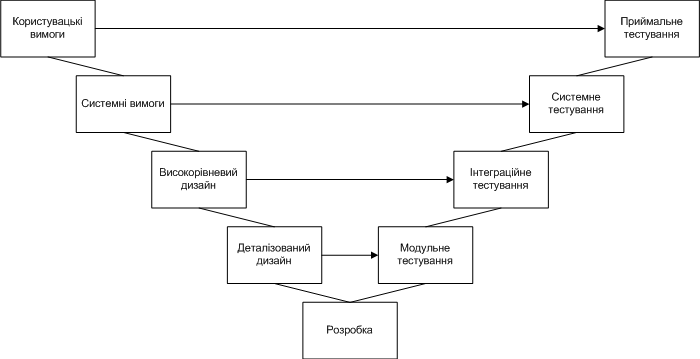


**Рис. 1.1 Каскадна модель або «водоспад»**

*Тестування у каскадній моделі розробки*

Тестування у «водоспадній» моделі є майже фінальною стадією перед завершенням життєвого циклу проекту. Всі дефекти виявляються незадовго до дати випуску програмного забезпечення на ринок або здачі замовнику. Вартість виправлення дефектів знайдених у кінці проекту набагато більша, ніж на ранніх етапах. Найкритичнішою є ситуація, коли дефект знайдений у специфікації, що змушує починати роботу над проектом з самого початку. Частіше за все такі проекти вигідніше закрити.

**V-модель**



**Рис. 1.3 V- модель**

V-модель була розроблена для вирішення проблем традиційного «водоспаду», а саме: виявлення дефектів в кінці життєвого циклу розробки. V-модель надає можливість залучення тестування якомога раніше у життєвому циклі.

V-модель демонструє як верифікація і валідація можуь бути інтегровані у кожний з етапів життєвого циклу (рис.1.3).

Є цілий ряд заходів, які повинні бути виконані до кінця написання коду програмного продукту. Ці заходи повинні проводитися одночасно з кодуванням, і тестувальники повинні взаємодіяти та співпрацювати на цих етапах із розробниками і бізнес-аналітиками. Даний підхід дуже ефективний, тому що тестувальники виявляють дефекти у специфікаціях та документах, тим самим запобігаючи перенесенню цих помилок у код.

**Ітераційна (спіральна) модель**

Ітераційна модель означає розробку малого ядра, на яке пізніше нарощуються все нові і нові функціональності. Замість тривалої послідовної розробки, як в першому випадку, існує цикл декількох самостійних ітерацій (рис.1.4). Результатом виконання кожної ітерації є продукт, що містить невелику частину функціональності кінцевого продукту.

Перевагами такого підходу є більш короткий час до здачі в експлуатацію замовнику, або до виходу продукту на ринок, а також можливість швидко змінити необхідну частину, якщо змінилися потреби замовника або специфікація ПЗ, яке розробляється.



**Рис. 1.4 Ітераційна (спіральна) модель**

*Тестування у спіральній моделі*

Ітераційність зумовлює тестування нового функціоналу, регресійне тестування існуючого (перевірка, що внесені зміни не викликали ніяких помилок та дефектів в інші областях програми) та інтеграційне тестування між новою та існуючою частинами.

Регресійне тестування є важливим для кожної ітерації після першої. Це означає, що об’єм тестування буде збільшуватись з кожною новою доданою ітерацією.

**Гнучка розробка**

Гнучка розробка бере за основу ітеративний підхід, але організовує більш «легкий» та орієнтований на людські ресурси механізм. Процеси використовують зворотній зв'язок замість планування як основний механізм контролю. Зворотній зв'язок гарантується частими перевірками та тестами і раннім релізом продукту, що є в розробці. Найпопулярнішими варіаціями гнучких моделей є *Екстремальне програмування(XP) і Scrum.*

*Тестування у гнучкій розробці*

Гнучка розробка часто використовує підхід *розробки через тести* - це підхід, який складається спочатку з написання автоматичних модульних тестів для перевірки всіх дефектів, а потім написання коду безпосередньо.

Судячи з всього вищесказаного, дії з тестуванню можуть бути найрізноманітнішими і проходити на різних стадіях життєвого циклу розробки програмного продукту. Моделі, що широко використовуються на даний момент передбачають залучення тестування якомога раніше і все частіше вимагають на кожну дію команди розробки відповідну дію команди тестування.

**1.3 Тестування і якість програмного продукту**

Тестування проводить контроль якості продукту, який розробляється. Воно забезпечує розробника, інженера, замовника та інших зацікавлених в продукті людей своєчасним зворотнім зв’язком – надає у вигляді певних метрик (кількість виявлених дефектів, їх критичність, кількість успішних/провалених тестів), дані про стан продукту, на основі яких приймаються рішення.

Тестування може надавати тільки інформацію. Воно вимірює та оцінює якість продукту, однак ніяк не може гарантувати цю саму якість.

Тестування і забезпечення якості часто помилково тлумачать як щось однакове, невідривне один від одного. Однак вони мають різні цілі: ціль тестування полягає у знаходженні дефектів, ціллю дій з забезпеченню якості є виправлення старих та запобігання утворенню нових помилок та дефектів.

Тобто тестування - це процес перевірки визначених вимог до якості. Для програмного забезпечення функції тестування можуть включати перевірку програмного забезпечення на відповідність набору вимог (верифікація) і перевірки того, що програмне забезпечення задовольняє цілям, що перед ним було визначено (валідація). Забезпечення якості ж набагато більше спрямоване на послідовне поліпшення процесу, який дозволяє розробляти якісне програмне забезпечення.

**1.4 Аналіз процесу тестування ПЗ**

В основному процес тестування складається із таких головних дій як:

• Планування і контроль

• Аналіз і дизайн

• Розробка та виконання тестів

• Оцінка отриманих результатів

• Дії після завершення тестування

Ці пункти знаходяться у логічній послідовності, однак можуть виконуватись і паралельно.

**Планування і контроль**

На цьому етапі визначаються цілі тестування, наприклад: пошук дефектів у програмному забезпеченні або вимірювання продуктивності чи надійність даної системи, тощо.

Дії з забезпечення контролю відбуваються протягом усього життєвого циклу розробки програмного забезпечення і включають в себе порівняння отриманих в наслідок тестування результатів з очікуваними, звітування про стан тестування і його результати, у тому числі відхилення від плану або несправності програми. Вони також включають дії необхідні для досягнення запланованих цілей проекту.

**Аналіз і дизайн**

На даному етапі загальні цілі, які переслідуються специфікацією проекту, транслюються у конкретні умови для тестування і тестові випадки.

Задачами даної фази основного процесу тестування можуть бути: перевірка специфікації з вимогами, оцінка об’єктів тестування на можливість перевірки, визначення пріоритетів тестових випадків, визначення необхідних тестових даних, тощо.

**Розробка та виконання тестів**

Етап розробки і виконання тестів є сукупністю дій, під час яких тестові випадки перетворюються у конкретні тести та їх набори, які включають в себе інформацію, необхідну для їх виконання.

Серед задач етапу можна виділити:

• Розробка і визначення пріоритетів тестових випадків

• Розробка процедури для кожного з тестових випадків, створення для них правильних тестових даних, написання для нього автоматичного коду

• Створення тестових наборів для ефективного виконання тестів

• Перевірка, що обладнання для тестування налаштовано правильно

• Виконання тестових процедур

• Отримання та запис результатів виконання тестів

• Порівняння отриманих результатів з очікуваними

• Пошук помилок та відхилень, їх аналіз з метою пошуку причин цих відхилень.

• Повтор всіх попередніх дій з тестування, щоб перевірити чи знайдені дефекти та помилки програмного забезпечення дійсно виправлено

**Оцінка отриманих результатів**

Оцінка вихідного критерію – це дії, під час яких результати виконання тестів оцінюються на предмет відповідності поставленим перед ними цілям.

Для етапу характерні:

• Оцінка результатів тестування на відповідність їх до встановленого вихідного критерію

• Оцінка необхідності проведення додаткових тестів

• Написання звіту з отриманих результатів для зацікавлених осіб, таких як: замовник чи потенційний користувач.

**Дії після завершення тестування**

Цей етап включає в себе збір даних про вже завершене тестування. Це відбувається вже в основному на етапах релізу продукту, завершення проекту або його відміні.

Перелік задач:

• Перевірка які з запланованих результатів отримані в наслідок тестування

• Написання та закриття звітів про труднощі та непередбачувану поведінку програмного забезпечення

• Документування результатів та інше

**1.5 Автоматизація тестування**

Під автоматизацією тестування мають на увазі підключення інструментів для перевірки програми під час її виконання – тобто для динамічного тестування. Подібні інструменти моделюють дії потенційного користувача за допомогою спеціальних тестових фреймворків.

Найбільш розповсюдженою формою автоматизації є тестування додатку через графічний користувацький інтерфейс. Популярність такого підходу зумовлюється по-перше тим, що додаток перевіряється у той самий спосіб, котрим його буде використовувати людина, по-друге, можна тестувати додаток, на маючи при чому доступу до програмного коду. Так як в нашому випадку тестування буде відбуватися методом «білого ящика», тобто ми маємо доступ до коду програми, ми будемо використовувати фреймворк JUnit та проводити тести не за допомогою графічного інтерфейсу, а за допомогою коду в програмному продукті.

**1.6 Переваги автоматичного тестування**

1. Повторюваність - усі написані тести завжди будуть виконуватися одноманітно і точно.

2. Запрограмованість - ви можете запрограмувати складні тести, які спеціаліст з тестування не може створити вручну.

3. Швидке виконання - автоматизованому тесту не потрібно звірятися з інструкціями та документацією, що сильно скорочує час, відведений на тестування.

4. Звіти – можна налаштувати автоматичне зберігання та обробку отриманих результатів.

5. Виконання без втручання - тести можуть виконуватися без присутності спеціаліста, та навіть в неробочий час.

Аналізуючи переваги, можна сказати, що автоматизація тестування надає безліч переваг. Найбільш значущою є можливість позбавити спеціаліста з тестування рутинного повторення однакових тестів, зменшивши цим самим кількість помилок, іншою ж причиною переваги автоматизації тестування можна вважати можливість реалізувати такі впливи на програму, які неможливо зробити вручну – тестування продуктивності і навантаження.

**1.7 Недоліки автоматизованого тестування**

Не зважаючи на те, що автоматизація тестування значно полегшує цей процес, існують і суттєві недоліки. Автоматичні функціональні тести орієнтовані тільки на підтвердження, що все працює правильно, що нічого не зламалося (регресійне тестування), а не на знаходження нових дефектів, які могли бути просто на просто пропущені тестами .

Автоматичні тести є досить чутливим до його змін. Вони можуть бути написані для практично завершеного додатку. І є чуттєвими до будь-яких змін у ньому, що може привести до виникнення нових і нових дефектів.

У випадку невдачі, пошук причин неуспішності тесту потребує затрат часу як правило більших ніж під час ручного тестування.

**1.8 Класифікація видів тестування**

Існує безліч видів тестування: одні зазвичай виконують самі розробники, а інші - спеціалізовані групи. Види тестування поділяють на:

**Блокове тестування** - тестування повного класу, методу, маленької програми, написаної одним програмістом або групою, що виконується окремо від інших частин системи.

**Тестування компонента** - це тестування класу, пакету, невеликого додатка або іншого елемента системи, розробленого кількома програмістами або групами, що виконується в ізоляції від інших частин системи.

**Інтеграційне тестування** - це спільне виконання двох або більше класів, пакетів або підсистем, створених кількома програмістами або групами.

**Регресивне тестування** - повторне виконання тестів, яке спрямоване на виявлення дефектів у програмі, що вже пройшла цей набір тестів.

**Тестування системи** - це виконання ПЗ в його остаточній конфігурації, інтегрованого з іншими програмними та апаратними системами.

**1.9 Випадкове тестування та його історія**

Випадкове тестування - це метод тестування програмного забезпечення (в основному методом чорного ящика, але в нашому випадку білого, бо ми маємо доступ до коду програми), при якому програми тестуються шляхом генерування випадкових незалежних вхідних даних. Результати виведення порівнюються зі специфікаціями програмного забезпечення, щоб переконатися, що результати тесту пройдені або не пройдено. У разі відсутності специфікацій використовуються винятку мови програмування: якщо виключення виникає під час виконання тесту, то це означає, що в програмі є помилка.

**Характеристики випадкового тестування:**

* Випадкове тестування проводиться, коли дефекти НЕ ідентифікуються через рівні проміжки часу.
* Випадкове введення використовується для перевірки надійності та продуктивності системи.
* Економить час і зусилля.
* Інші методи тестування не можуть бути використані для даної задачі.

**Кроки випадкового тестування:**

* Випадкові вхідні дані ідентифікуються для оцінки системою.
* Тестові вхідні дані вибираються незалежно від тестової специфікації.
* Тести виконуються з використанням цих випадкових вхідних даних.
* Результати записуються і порівнюються з очікуваними результатами.
* Після цього відтворити / повторити тести, виявити дефекти, виправити і повторно протестувати.

Одним з поширених прикладів є використання випадкових цілих чисел для перевірки програмної функції, яка повертає результати на основі цих цілих чисел. Ці функції можуть включати в себе цикли «For» або інші алгоритми для видачі результату, де набір випадкових тестових випадків моделює або апроксимує більш широкий набір випадків, введених можливими користувачами, які моделюються за допомогою комп’ютера.

Інші види випадкового тестування можуть включати використання евристики, яка керує використанням випадкових вхідних даних. В цілому, і особливо при роботі з цілими числами або іншими типами змінних, випадкове тестування є настільки ж випадковим, що і набір використовуваних випадкових вхідних даних - іншими словами, тестери часто вважають за краще використовувати обмежений набір цілих чисел, а не нескінченну послідовність. Спеціальні методи випадкового тестування побудовані на механіці того, що ми маємо на увазі під випадковим, і на тому як розробники отримують, псевдовипадковий набір вхідних даних для тесту.

Обговорення випадкового тестування також можуть обертатися навколо ефективності його використання. Одна ідея полягає в тому, що оскільки довільне тестування може виконуватися автоматизованими системами, а не фахівцями-людьми, воно може мати переваги перед спрямованим тестуванням. Проте, спрямоване тестування може бути більш ефективним з точки зору обсягу необхідного тестування. Деякі розробники і експерти також використовують термін «випадкове тестування» для позначення неефективного або навіть некомпетентного тестування, де спрямоване тестування розглядається як переважаючий метод.

Випадкове тестування апаратних засобів було вперше досліджено Мелвіном Бройером в 1971 році, а перші спроби оцінити його ефективність було вжито Пратімою і Вішвані Агравалом в 1975 році.

В області програмного забезпечення Дюран і Нтафос досліджували випадкове тестування в 1984 році.

Згідно з основоположною статтею про випадкове тестування Д. Гамлета: «Технічне, математичне значення «випадкового тестування» відноситься до явної відсутності «системи» у виборі даних випробувань, так що немає ніякої кореляції між різними тестами.»

2 Фреймворк тестування JUnit

Для виконання даної курсової роботи нам потрібно створити програму, в якій буде декілька класів з різними функціями, які ми і будемо тестувати за допомогою випадкових вхідних даних. Для створення тестів в даному випадку я буду використовувати один з найпростіших у використанні та розумінні фреймворків – JUnit.

**2.1 Інформація про JUnit**

Пропоную детальніше ознайомитися з цим фреймворком.

JUnit - бібліотека для модульного тестування програмного забезпечення на мові Java.

Створений Кентом Беком і Еріком Гамою, JUnit належить родині фреймворків xUnit для різних мов програмування, що бере початок в SUnit Кента Бека для Smalltalk. JUnit породив екосистему розширень - JMock, EasyMock, DbUnit, HttpUnit і т. Д.

JUnit був портований на інші мови, включаючи PHP (PHPUnit), C # (NUnit), Python (PyUnit), Fortran (fUnit), Delphi (DUnit), Free Pascal (FPCUnit), Perl (Test :: Unit), C ++ (CPPUnit ), Flex (FlexUnit), JavaScript (JSUnit), COS (COSUnit).

Досвід, отриманий під час роботи з JUnit, важливий в розробці концепцій тестування програмного забезпечення.

**2.2 Початок роботи з JUnit**

Для початку нам треба створити Maven Project в Intellij IDEA. Саме це середовище програмування буде використовуватися в нашій курсовій роботі.

Після створення проекту нам треба підключити залежності для імпорту JUnit’а без ручного імпорту бібліотеки в папку з проектом. В pom.xml копіюємо наступне:

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.13-beta-2</version>

</dependency>

Завдяки цьому бібліотеки JUnit автоматично імпортуються до нашого проекту.

**2.3 Синтаксис JUnit**

Синтаксис фреймворку буде представлено в таблиці 1.

Таблиця 1 – анотації, які використовуються в JUnit

|  |  |
| --- | --- |
| **Анотація** | **Опис** |
| **@Test** public void method() | Анотація **@Test** визначає, що метод method() є тестовим |
| **@Before** public void method() | Анотація **@Before вказує на те**,  що метод буде виконуватися перед кожним методом **@Test** |
| **@After** public void method() | Анотація **@After** вказує на те що метод буде виконуватися після кожного методу **@Test** |
| **@BeforeClass** public static void method() | Анотація **@BeforeClass** вказує на те, що метод буде виконуватися на початку всіх тестів, а точніше в момент запуску тестів (перед усіма тестами **@Test**). |
| **@AfterClass** public static void method() | Анотація **@AfterClass** вказує на те, що метод буде виконуватися після всіх тестів. |
| **@Ignore** | Анотація **@Ignore** каже, що метод буде проігнорований в момент проведення тестування. |
| @Test (**expected** = Exception.class) | (**expected** = Exception.class) - вказує на те, що в даному тестовому методі ви навмисно очікується Exception. |
| @Test (**timeout**=100) | (**timeout**=100) – вказує, що тестований метод не повинен займати більше ніж 100 мілісекунд. |

Методи, які використовуються під час тестування за допомогою JUnit буде приведено в таблиці 2.

Таблиця 2 – методи

|  |  |
| --- | --- |
| **Метод** | **Опис** |
| **fail(String)** | Вказує на те що б тестовий метод завалився при цьому виводячи текстове повідомлення. |
| **assertTrue([message], boolean condition)** | Перевіряє, що логічна умова істина. |
| **assertsEquals([String message], expected, actual)** | Перевіряє, що два значення збігаються. |
| **assertNull([message], object)** | Перевіряє, що об'єкт є порожнім null. |
| **assertNotNull([message], object)** | Перевіряє, що об'єкт не є порожнім null. |
| **assertSame([String], expected, actual)** | Перевіряє, що обидві змінні стосуються одного об'єкта. |
| **assertNotSame([String], expected, actual)** | Перевіряє, що обидві змінні стосуються різних об'єктів. |

Тепер, коли ми познайомилися з синтаксисом та основними можливостями фреймворку, ми можемо використовувати його для виконання нашого завдання курсової роботи. В нашому проекті будуть використовуватися і інші інструменти, але основне тестування буде відбуватися саме за допомогою JUnit.

3 Програмна реалізація

**3.1 Імпорт залежностей Maven до створеного проекту**

В першу чергу після створення проекту ми імпортуємо Maven залежності, щоб підключити бібліотеки JUnit до проекту. Для цього в нашому випадку заходимо на сайт з описом JUnit та копіюємо звідти все необхідне. Під необхідним мається на увазі наступне:

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.13-beta-2</version>

<scope>compile</scope>

</dependency>

Після цих дій нам потрібно вручну завантажити залежності в наш проект. Для цього переходимо за наступним шляхом (рис. 3.1) :

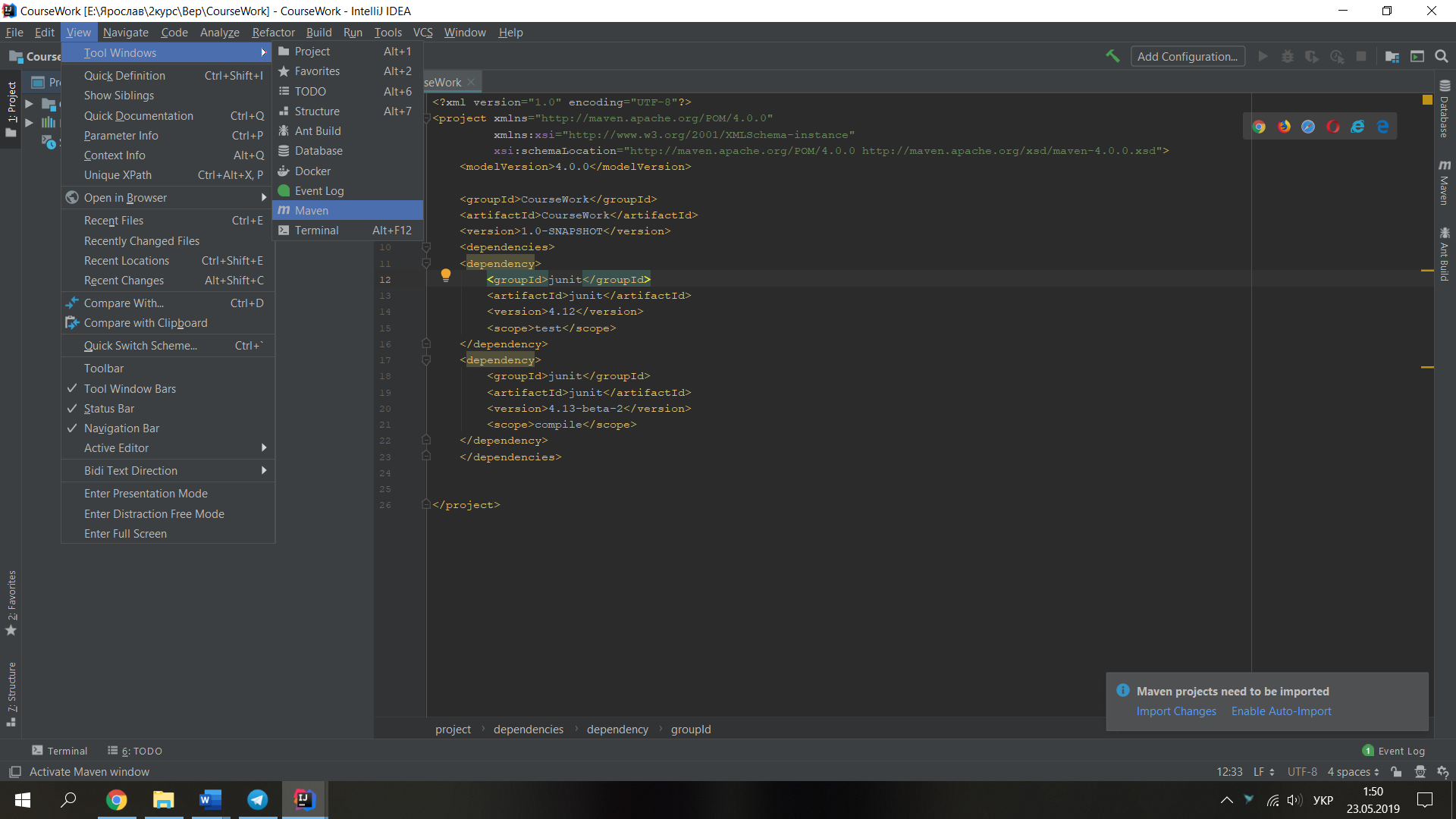


Рисунок 3.1 – Шлях для імпорту залежностей

Тепер у відкритому вікні натискаємо на кругову стрілку зліва вгорі, це можна спостерігати на рисунку 3.2:

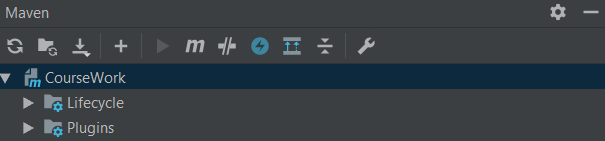


Рисунок 3.2 – Проект до імпорту

Після натискання в нашому проекті з’являються залежності, це ми можемо спостерігати на рисунку 3.3:

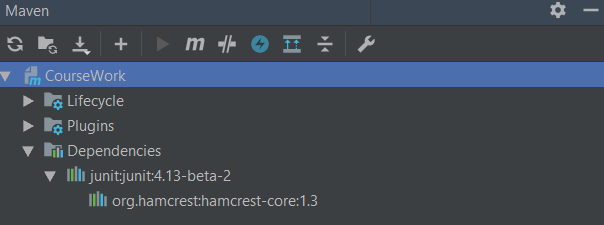


Рисунок 3.3 – Проект після імпорту

На цьому роботу з імпортом завершено. Можна переходити до самої реалізації тестів.

**3.2 Створення та реалізація тестів за допомогою JUnit**

Переходимо до створення першого класу, на якому ми будемо виконувати випадкове тестування. Для цього ми будемо використовувати генератор псевдовипадкових чисел, вбудований в мову програмування Java та Java-машину. Сутність тесту полягатиме в тому, що генерація випадкових чисел буде імітувати користувача, який водить мишкою по екрану у випадковому порядку і в певній точці натискає на мишку. При цьому ми будемо перевіряти, чи влучить цим натисканням користувач на якусь із кнопок нібито розташованих на неіснуючому сайті. Обмеженнями, які будуть імітувати кнопки, будуть умови нашого тесту, в яких будуть задані координати, при яких тест буде пройдено. Якщо тест пройдено, то користувач натиснув на кнопку, якщо ж ні, то користувач натиснув десь повз кнопку. Реалізацію цього тесту було виконано наступним чином:

Створюємо клас, в якому будуть сетери та гетери для нашої кнопки. Більше в першому випадку нам нічого не потрібно. Потім створюємо другий клас, в якому буде розміщуватися наш тест: він буде спрацьовувати коли користувач влучив на кнопку і буде не проходити, коли користувач натискає повз. Координати будуть випадково моделюватися перед кожним тестом за допомогою функції (Math.random() \* 200); Структуру проекту для першого тесту можна спостерігати на рисунку 3.4:

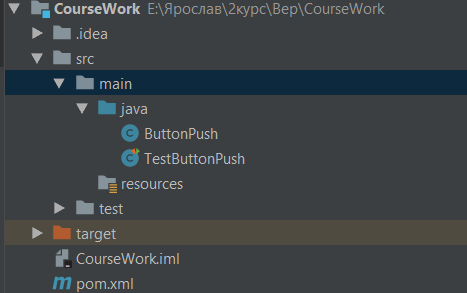


Рисунок 3.4 – Структура проекту

Тест буде проходити, якщо курсор користувача в момент натискання відповідав умовам, передбаченим в тесті. Ці умови можна спостерігати на рисунку 3.5:



Рисунок 3.5 – Умови проходження тесту

Тепер спробуємо запускати тест, і подивимося з якої спроби змодельований користувач таки зможе влучити у кнопку своїм курсором та пройти тест.

Перший тест було провалено, бо місце, куди натиснули не відповідало умовам тест, тобто – не було нашою кнопкою. Результат можемо бачити на рисунку 3.6:

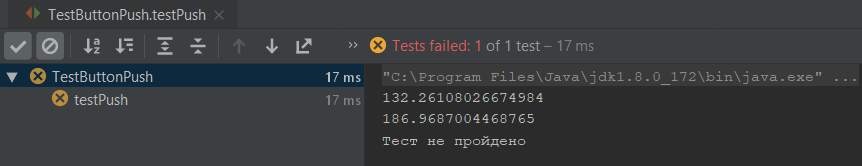


Рисунок 3.6 – Невдале проходження тесту, через невідповідність даних

Другий тест також пройшов невдало. Це можна побачити на рисунку 3.7:

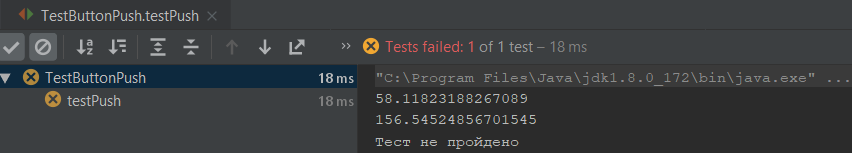


Рисунок 3.7 – Невдале проходження другої спроби тесту

Третій та четвертий тести також були невдалими, а ось п’ятий раз тест таки пройшов, а отже курсор користувача все таки влучив по кнопці. Вдалий варіант проходження тесту ми можемо спостерігати на рисунку 3.8:

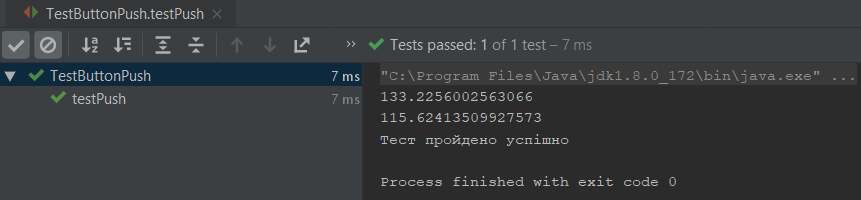


Рисунок 3.8 – Вдале проходження тесту

Провівши тест 100 разів, можна сказати, що ймовірність проходження даного тесту за даних умов можна оцінити як 1/3, зі ста тестів було пройдено 32 і не пройдено 68. Цей показник значно зменшиться при зміні умов тесту. Одне можна сказати точно – робота з випадковими даними дуже незручна, ймовірність правильного виконання тестів дуже низька. Дані які використовуються тут не були б робочими для багатьох інших програм.

Тепер перейдемо до створення другого класу для тестування інших випадкових даних. В нашому другому тесті ми будемо моделювати погоду за допомогою генератора випадкових чисел Java. Створюємо 2 класи як в минулому варіанті тестування. Структура програми після додавання цих двох класів буде виглядати так, як на рисунку 3.9:

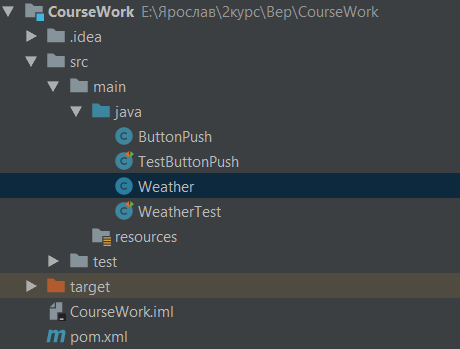


Рисунок 3.4 – Структура проекту після створення другого тесту

Робота другого нашого тесту буде полягати в тому, щоб проходити, коли генератор згенерує «Сонячну» погоду і не проходити за всіх інших умов. Для цього ми створюємо клас Weather в якому мається тільки гетер і сетер для рядка, в якому і буде записаний стан погоди. Другий клас з тестом буде моделювати випадкове число від 1 до 5, за допомогою якого через оператор «if» рядку буде присвоюватись одне із значень: "Sunny", "Rainy", "Windy", "Snowy" та "HOT". Наш тетс буде проходити тільки при першому значенні. Умова проходження тесту зображена на рисунку 3.10:



Рисунок 3.10 – Умова проходження другого тесту

Тепер, коли все необхідне встановлено, ми можемо почати запуск нашого тесту та перевірити ймовірність його проходження.

Перший тест не було пройдено через невідповідність даних. Це можна побачити на рисунку 3.11:

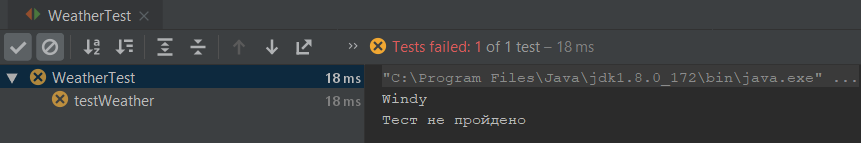


Рисунок 3.11 – Невдале проходження 2 тесту, через невідповідність даних

Другий тестовий випадок також закінчився безуспішно. Результат на рисунку 3.12:

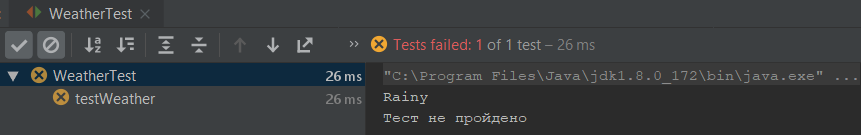


Рисунок 3.11 – Невдале проходження 2 спроби 2 тесту

Після п’ятого запуску тест нарешті пройдено успішно, так як дані відповідають вимогам. Результат правильного проходження тесту можна спостерігати на рисунку 3.13:

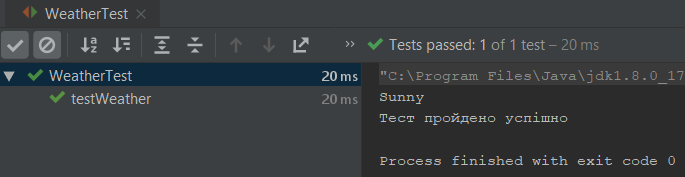


Рисунок 3.13 – Успішне проходження 2 тесту

Після запуску нашого тесту 100 разів, можна сказати, що ймовірність проходження даного тесту за даних умов можна оцінити як 1/5, зі ста тестів було пройдено 21 і не пройдено 79. Цей показник може змінитися як в одну, так і в іншу сторону при зміні умов тесту. Ймовірність правильного виконання тестів ще нижча, ніж в першому випадку. Дані які використовуються тут не були б робочими для багатьох інших програм.

Почнемо створення третього і останнього в нашій курсовій роботі тесту. Даний тест буде перевіряти перейшов учень поріг ЗНО чи все ж не склав на основі генерації випадкового числа від 95 до 200. Умовою проходження тесту є бал більший за 100. Для цього як і в минулих тестах створимо два класи. Як виглядатиме структура програми після додавання 3 тесту можна бачити на рисунку 3.14:

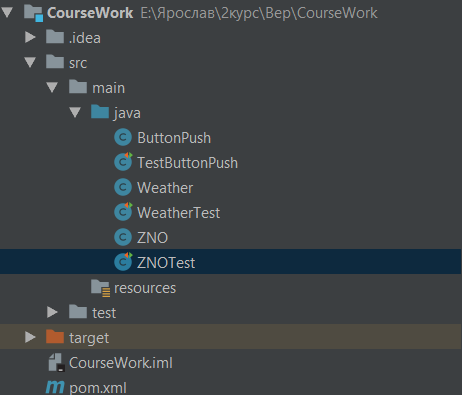


Рисунок 3.14 – Структура проекту після створення третього тесту

В першому нашому класі з третього тесту, як і в попередніх буде лише гетер та сетер для поля з балом. В класі, який буде проводити тестування, буде задана умова проходження тесту, яку можна спостерігати на рисунку 3.15:



Рисунок 3.15 – Умова проходження третього тесту

Дані, які будуть тестуватися будуть випадковим чином генеруватися за допомогою псевдогенератора (int)(Math.random() \* 100); Після цього буде перевірятися чи згенероване число відповідає умовам тесту.

Після створення тесту та виконання всіх умов можемо нарешті його запускати.

Перший тест пройшов успішно, згенероване число дорівнювало 153 і відповідало всім умовам. Результат проходження на рисунку 3.16:

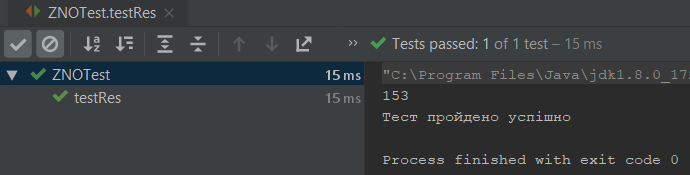


Рисунок 3.16 – Успішне проходження 3 тесту

Другий тест також був успішним, а от третій не пройшов через невідповідність даним. Результат на рисунку 3.17:

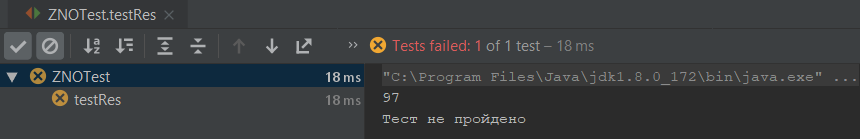


Рисунок 3.17 – Невдале проходження 3 тесту, через невідповідність умовам

Після проведення нашого тесту 100 разів, можна сказати, що ймовірність проходження даного тесту за даних умов можна оцінити як 9/10, зі ста тестів було пройдено 89 і не пройдено 11. Цей показник може суттєво змінитися як в одну, так і в іншу сторону при зміні умов тесту. Ймовірність правильного виконання дуже висока тільки через малу ймовірність даних вийти за межі умов.

Після проходження всіх тестів можемо скласти певну таблицю за результатами. Дані тестів занесені до таблиці 3:

Таблиця 3 – Дані проведеного тестування

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № тесту | Кількість  випробувань | Пройшли | Не пройшли | Ймовірність |
| Тест 1 | 100 | 32 | 68 | 1/3 |
| Тест 2 | 100 | 21 | 79 | 1/5 |
| Тест 3 | 100 | 89 | 11 | 9/10 |

Висновки

Підводячи підсумки даної курсової роботи можна сказати, що поставлена перед нами ціль була виконана. По-перше, ми ознайомилися з тестуванням в цілому і тепер маємо більш глибоке розуміння цього процесу, його відмінності від інших процесів в життєвому циклі програмного забезпечення, його величезну значущість в життєвому циклі кожного проекту, який розробляється. По-друге, ознайомилися з різними життєвими циклами проектів та побачили як вони впливають на процес тестування. По-третє, дізналися про класифікацію та види тестування. Детально ознайомилися з автоматизацією тестування, її перевагами та недоліками.

Автоматизація емуляції поведінки користувача має ряд безперечних переваг, однак не може повністю замінити ручне тестування через наступні причини:

* реалізація користувацького інтерфейсу не передбачає або робить економічно невигідним емуляцію впливів на нього (наприклад, у випадку, коли не має можливості отримати доступ до елементів графічного інтерфейсу та їх атрибутів);
* короткостроковий проект (написання тестів для автоматизації дії користувача по вартості дорівнюватиме мінімум трьом циклам ручного тестування і для короткострокових проектів не має економічного сенсу);
* лише тестові набори для ручного виконання можуть слугувати оцінці таких характеристик як практичність графічного інтерфейсу користувача, перевірці його відповідності певним стандартам або для проведення приймального тестування.

Що стосується безпосередньо теми курсової роботи, то можна сказати, що ми детально ознайомилися з властивостями та реалізацією процесу випадкового тестування. Тестування на основі випадкових даних (Random testing) - техніка тестування, в якій вхідні дані, дії або навіть самі тест-кейси вибираються на основі (псевдо)випадкових значень. В нашому випадку було перевірено випадкові вхідні дані, які генерувалися за допомогою генератора псевдовипадкових чисел та перетворювалися потім в необхідні за специфікацією нашої програми.

В ході роботи над проектом було створено три тестових випадки. Всі випадки перевірялися за допомогою тестів JUnit. Під час тестування ми детально ознайомилися з функціональністю цього фреймворку для тестування. Основними його перевагами є простота у використанні та зрозумілість навіть для тестувальника без досвіду. Створення тестів за допомогою цього інструменту відбувається елементарно: для початку ми просто додаємо умову, яка буде виконуватися перед кожним тестом за допомогою анотації @Before, потім створюємо тест, в якому буде перевірятися відповідність вхідних даних очікуваним. Тести створюються за допомогою анотації @Test. Ці тести допоможуть протестувати програму та перевірити, чи співпадають результати виконання з очікуваними результатами.

Щодо випадкових тестових даних, які подаються на вхід програмі, не можна відповісти однозначно. Більшість програм не призначені для роботи з випадковими або псевдовипадковими даними, тому вони будуть некоректно працювати з ними.

Головна перевага випадкового тестування полягає в тому, що оскільки довільне тестування може виконуватися автоматизованими системами, а не фахівцями-людьми, воно може мати переваги перед спрямованим тестуванням. Проте, спрямоване тестування може бути більш ефективним з точки зору обсягу необхідного тестування. Фахівці з тестування часто використовують термін «випадкове тестування» для позначення неефективного або навіть некомпетентного тестування, де спрямоване тестування розглядається як переважаючий метод.

Отже, підводячи підсумки, можна сказати, що даний вид тестування не є універсальним та не підходить для більшості програм. В основному, програми просто не прийматимуть на вхід випадкові або псевдовипадкові дані. Тим більш в нашому житті ми майже не стикаємося з генераторами повністю випадкових послідовностей. Даний вид тестування буде доречним в основному під час перевірки рівня безпеки та стійкості програми до різних помилок та непередбачуваних сценаріїв виконання. Тому даний вид тестування є затребуваним, але не в широкому колі всіх тестувальників, а лише на високих рівнях, наприклад при тестуванні екстремальних систем, які просто не повинні давати жодного збою, який міг би призвести до фатальних наслідків.

Перелік джерел посилання

1. Макгрегор Дж., Сайкс Д. «Тестування об'єктно-орієнтованого програмного забезпечення»
2. Стандарт IEEE 829 Software Test Documentation
3. ISO/IEC 9126 Software engineering
4. Липаев В.В. «Тестування програм»
5. Random Testing – [Електронний ресурс] - https://en.wikipedia.org/wiki/Random\_testing
6. Random Testing – [Електронний ресурс] - https://www.techopedia.com/definition/21595/random-testing

Додаток А

Код програми

Тест 1

Клас ButtonPush:

public class ButtonPush {

private double x;

private double y;

public double getX() {

return x;

}

public void setX(double x) {

this.x = x;

}

public double getY() {

return y;

}

public void setY(double y) {

this.y = y;

}

}

Клас TestButtonPush:

import org.junit.Before;

import java.io.IOException;

public class TestButtonPush {

ButtonPush bp = new ButtonPush();

@Before

public void setBp(){

Double a = (Math.random() \* 200);

Double b = (Math.random() \* 200);

bp.setX(a);

bp.setY(b);

System.out.println(bp.getX());

System.out.println(bp.getY());

}

@org.junit.Test

public void testPush() throws IOException {

boolean flag = true;

Double res = bp.getX();

Double res2 = bp.getY();

if ((res>50&&res<150)&&(res2>50&&res2<150)) {

flag = true;

System.out.println("Тест пройдено успішно");

} else {

flag = false;

System.out.println("Тест не пройдено");

}

org.junit.Assert.assertTrue(flag);

}

}

Тест 2

Клас Weather:

public class Weather {

private String weather;

public String getWeather() {

return weather;

}

public void setWeather(String weather) {

this.weather = weather;

}

}

Клас WeatherTest:

import org.junit.Before;

import java.io.IOException;

public class WeatherTest {

Weather w = new Weather();

@Before

public void setW(){

int a =(int) (Math.random() \* 5);

if(a == 1){

w.setWeather("Sunny");

}else if (a==2){

w.setWeather("Rainy");

}else if(a==3){

w.setWeather("Windy");

}else if(a==4){

w.setWeather("Snowy");

}else if(a==5){

w.setWeather("HOT");

}

System.out.println(w.getWeather());

}

@org.junit.Test

public void testWeather() throws IOException {

boolean flag = true;

String res = w.getWeather();

if (res=="Sunny") {

flag = true;

System.out.println("Тест пройдено успішно");

} else {

flag = false;

System.out.println("Тест не пройдено");

}

org.junit.Assert.assertTrue(flag);

}

}

Тест 3

Клас ZNO:

public class ZNO {

private int score;

public int getScore() {

return score;

}

public void setScore(int score) {

this.score = score;

}

}

Клас ZNOTest:

import org.junit.Before;

import java.io.IOException;

public class ZNOTest {

ZNO zno = new ZNO();

@Before

public void setZNO(){

int a = 95 + (int)(Math.random() \* 100);

zno.setScore(a);

System.out.println(zno.getScore());

}

@org.junit.Test

public void testRes() throws IOException {

boolean flag = true;

int res = zno.getScore();

if (res>100) {

flag = true;

System.out.println("Тест пройдено успішно");

} else {

flag = false;

System.out.println("Тест не пройдено");

}

org.junit.Assert.assertTrue(flag);

}

}

Додаток Б

Вміст pom.xml

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>CourseWork</groupId>

<artifactId>CourseWork</artifactId>

<version>1.0-SNAPSHOT</version>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.13-beta-2</version>

<scope>compile</scope>

</dependency>

</dependencies>

</project>