**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ**

**Національний авіаційний університет**

**Кафедра комп’ютеризованих систем управління**

ДОПУСТИТИ ДО ЗАХИСТУ

Завідувач кафедри

Литвиненко О.Є.

“\_\_\_\_\_”\_\_\_\_\_\_\_\_\_\_\_\_2021 р.

**ДИПЛОМНИЙ ПРОЕКТ  
(ПОЯСНЮВАЛЬНА ЗАПИСКА)**

**здобувача освітньоГО СТУПЕНЯ**

**“Бакалавр”**

**Тема:**  Модуль розпізнавання жестів рук для керування квадрокоптером

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Виконавець:**  Левчук В.В. а

**Керівник:**  к.т.н. доц. Глазок О.М.

**Консультанти з окремих розділів пояснювальної записки:**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Нормоконтролер:**  Тупота Є.В. а

**Київ 2021**

**Національний авіаційний університет**

Інститут комп’ютерних інформаційних технологій

Кафедра комп’ютеризованих систем управління

Освітньо-кваліфікаційний рівень магістр

Спеціальність 8.05010202 "Системне програмування"

ЗАТВЕРЖУЮ

Завідувач кафедри

Литвиненко О.Є.

« » 2014 р.

ЗАВДАННЯ

НАВИКОНАННЯДИПЛОМНОЇРОБОТИСТУДЕНТА

Кашкевич І-Ф. Ф.

(прізвище*,* ім'я*,* по батькові)

**1.**Т**ема проекту (роботи):**  “Методика використання платформи .*NET* при

створенні тригерів бази даних ”

затверджена наказом ректора від " 17 " січня 2014 року № 51/ст.

**2.**Т**ермін виконання проекту (роботи):** з 10.03.2014 до 22.06.2014

**3.** В**ихідні дані до проекту (роботи):** постановка задачі до виконання роботи*,*

мова програмування *C*#*,* середовище програмування *MS Visual Studio 2010,*

СУБД *MS* *SQL Server*

**4.** З**міст пояснювальної записки (перелік питань*,* що підлягають розробці):**

1) аналіз приципів створення і використання тригерів баз даних;

2) можливості використання тригерів баз даних;

3) розробка методики створення тригерів таблиць бази даних з використанням

платформи .*NET*.

**5.** П**ерелік обов’язкового графічного матеріалу:**

1) основне вікно програми;

2) вікно виконання сценаріїв;

3) структура таблиць бази даних;

4) схема алгоритму виконання тригерів в програмі;

5) порівняльна таблиця середнього часу обробки тригерів.

6. КАЛЕНДАРНИЙ ПЛАН

|  |  |  |  |
| --- | --- | --- | --- |
| №  п/п | Етапи виконання дипломної роботи | Термін виконання етапів | Примітка |
| 1 | Ознайомитись з постановкою задачі дипломного проектування | 17.03.14 | виконано |
| 2 | Вивчити спеціальну літературу і технічну документацію | 20.03.14 | виконано |
| 3 | Проаналізувати принципи створення тригерів при використанні платформи .*NET* | 29.03.14 | виконано |
| 4 | Написати розділ 1. | 10.04.14 | виконано |
| 5 | Проаналізувати можливості платформи .*NET* в роботі з тригерами | 22.04.14 | виконано |
| 6 | Написати розділ 2. | 29.04.14 | виконано |
| 7 | Провести розробку методики створення тригерів таблиць БД | 10.05.14 | виконано |
| 8 | Написати розділ 3. | 15.05.14 | виконано |
| 9 | Оформити пояснювальну записку | 05.06.14 | виконано |
| 10 | Підготувати графічний демонстраційний матеріал та доповідь | 16.06.14 | виконано |

7. Дата видачі завдання 04.02.2021 а

Керівник дипломної роботи Глазок О.М.

(підпис)

ЗавданняприйнявдовиконанняКашкевич І-Ф. Ф.

(підпис студента)

ПОЯСНЮВАЛЬНА ЗАПИСКА

Пояснювальна записка до дипломної роботи “Модуль розпізнавання жестів рук для керування квадрокоптером”: 95 с.*,* 14 рис.*,* 7 табл.*,* 27 літературних джерел*,* 4 додатки.

ШТУЧНИЙ ІНТЕЛЕКТ, ГЛИБОКЕ НАВЧАННЯ, ОДНОПЛАТНА ЕОМ, *Raspberry Pi, MediaPipe*, ЖЕСТ, КВАДРОКОПТЕР

**Об’єкт дипломного проектування** – розпізнавання жестів рук на основі нейромережевим алгоритмам.

**Предмет дипломного проектування** – модуль розпізнавання жестів рук для керування квадрокоптером.

**Мета дипломної роботи** – створення модуля на основі одноплатної ЕОМ для розпізнавання жестів рук з використанням алгоритмів штучного інтелекту з подальшою інтерпретацією результатів роботи модуля у команди для керування квадрокоптера.

**Метод проектування** – застосування засобів розпізнавання з поєднанням алгоритмів глибокого навчання та використання апаратних можливостей одноплатних ЕОМ.

**Галузь застосування** – дистанційне жестикуляційне керування динамічних систем.

**Значущість роботи та висновки** – модуль жестикуляційного керування забезпечує просте та інтуїтивне керування рухомими системами, такими як квадрокоптер.

**Прогнози та припущення щодо розвитку об’єкта дослідження** – створення робочого зразка модуля, інтеграція зразка у квадрокоптерта використання його для керування. Подальший розвиток системи передбачає додавання нового функціоналу у модуль, зокрема: розпізнавання та ідентифікація обличчя власника або кола персон, розширення спектру жестів, які підлягають розпізнаванню, додовання можливості розпізнавання звукових команд з подальшим виконання відповідних, до розпізнаних сигналів, завдань.

|  |  |
| --- | --- |
| ЗМІСТ  [ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ*,* СКОРОЧЕНЬ*,* ТЕРМІНІВ 6](#_bookmark0)  [ВСТУП 7](#_bookmark1)  [РОЗДІЛ 1 АНАЛІЗ ПІДХОДІВ ДО ТЕСТУВАННЯ ПРОГРАМНОГО](#_bookmark2) [ЗАБЕЗПЕЧЕННЯ 11](#_bookmark2)   * 1. [Зв'язок між тестування і забезпеченням якості 15](#_bookmark3)   2. [Автоматизація процесу тестування програмного](#_bookmark4)   [забезпечення 16](#_bookmark4)   * 1. [Обґрунтування генерації тестів 23](#_bookmark5)   2. [Висновки до розділу 25](#_bookmark6)   [РОЗДІЛ 2 ОБГРУНТУВАННЯ ПІДХОДУ ДО РОЗРОБКИ СИСТЕМИ 26](#_bookmark7)   * 1. [Формальна модель представлення вимог 26](#_bookmark8)   2. [Аналіз сучасних методик та засобів тестування програмних](#_bookmark9) [застосувань для мобільних пристроїв 31](#_bookmark9)   3. [Побудова тестового набору 39](#_bookmark10)   4. [Висновки до розділу 44](#_bookmark11)   [РОЗДІЛ 3 ОПИСАННЯ РОБОТИ СИСТЕМИ ВІДСЛІДКОВУВАННЯ](#_bookmark12) [ПОМИЛОК В ПРОГРАМНОМУ ЗАБЕЗПЕЧЕННІ](#_bookmark12)  [*WEB*-ДОДАТКІВ 45](#_bookmark12)   * 1. [Логіка роботи додатку 45](#_bookmark13)   2. [Структура програмного додатку 48](#_bookmark14)   3. [Інтерфейс додатку та параметри запуску 50](#_bookmark15)   4. [Висновки до розділу 55](#_bookmark16)   [ВИСНОВКИ 57](#_bookmark17)  [СПИСОК БІБЛІОГРАФІЧНИХ ПОСИЛАНЬ ВИКОРИСТАНИХ ДЖЕРЕЛ 60](#_bookmark18)  [ДОДАТОК А 62](#_bookmark19) | |
|  | *5* |

|  |  |
| --- | --- |
| ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ*,* СКОРОЧЕНЬ*,* ТЕРМІНІВ  *ЕОМ – електронна обчислювальна машина*  *ШІ – Штучний інтелект* | |
|  | *6* |

|  |  |
| --- | --- |
| ВСТУП  На сьогоднішній день, дедалі обширним та тісним стає вплив електронних систем на повсякденне життя людей. Ріст попиту людей на задоволення своїх потреб як у побуті, так у вирішенні низки проблем економічного та логістичного характеру безпрецедентно спонукає до розвитку науково-технічної галузі інтерактивних систем людино-комп’ютерної взаємодії. Виникає дедалі більший запит на системи, які забезпечують керування людиною різноманітних комплексів, і вимагають дедалі більш простого інтерфейсу для взаємодії з цими комплексами. Можна виділити кілька причин для спрощення систем інтерфейсу керування людино-машинних комплексів, а саме: легкість в освоєні персоналом простого та зручного інтерфейсу, зменшення витрат на навчання персоналу для керування комплексом, інтуїтивно зрозумілий інтерфейс зменшує ризик виникнення помилок під час експлуатації. Таким чином, простий та інтуїтивно зрозумілий інтерфейс надає значно кращі можливості для керування, що у свою чергу, має ліпший вплив на економічні витрати, зокрема під час освоєння та експлуатації подібних систем.  Однією з таких систем інтерфейсу керування є керування за допомогою жестикуляції рук людини, що у свою чергу складаються з інтерактивної системи розпізнавання жестів. Інтерфейси керування, які полягають у керуванні за допомогою жестикуляції рук починають лише впроваджуватися у життєві процеси людини, але дедалі більше досліджень здійснюють у цій області. Зокрема, як зазначалося раніше, дана система дозволяє значно спростити керування комплексом, звівши це до інтуїтивно зрозумілого рівня. Також дані системи надають можливість виключити фізичний контакт оператора з комплексом, що досить актуально, зважаючи на епідеміологічний стан у світі.  Керування жестикуляційним способом значно краще відображається на здатності людини виконувати кілька операцій одночасно – здатність людини здійснювати розподіл уваги і здійснювати кілька видів діяльності. | |
|  | *7* |

|  |  |
| --- | --- |
| Ключовим аспектом у виконанні одночасно кількох операцій є увага. Увага – це така форма психологічної діяльності людини, що виявляється у спроможності свідомості зосереджуватися на певних об’єктах, які мають те чи інше значення для людини у певний момент часу. Також, одна з умов виконання такої «багатозадачності», полягає у тому, що одна з таких операцій буде виражена у вигляді міміки кінцівок рук. Звісно, людина здатна виконувати різноманітні дії, які людина може виконувати одночасно, але найбільш успішно вона виконує одночасно дві дії, одна з яких – фізична робота, наприклад моторика рук.  Зокрема можна навести приклад подібних ситуацій, коли людина успішно та ефективно виконує кілька дій одночасно. Наприклад вона здатна в один і той же час мити посуд і вести розмову із співрозмовником на кухні, спілкуватися по телефону па в’язати світер, керувати машиною та спілкуватися з іншою людиною по голосовому зв’язку через гарнітуру. Людина не використовує великих розумових зусиль задля виконання привичних та інтуїтивних задач, наприклад, розглядаючи у даному випадку миття посуду. Вона без перешкод здатна вести діалог на різноманітні теми, що одночасно не несе ніякого негативного впливу на виконання другої задачі (миття посуду).  Окрім спілкування, людина здатна і на інші інтелектуальні задачі, одночасно з виконанням задач із залученням моторики рук. Завдяки довгому шляху еволюції людина набула навичок ефективно здійснювати розподіл уваги між виконанням розумової роботи та фізичної праці рук і переключення уваги з однієї діяльності на іншу стає непомітним для людини. Хоча варто зауважити, що на багатозадачність людини впливає безліч чинників, зокрема фізичний і психологічний стан людини і навколишнє середовище: індивідум не здатний якісно виконувати навіть одну задачу, у разі, якщо є певні чинники, котрі будуть негативно впливати на зосередженість (концентрацію) – інженер не здатний вести розрахунки для проектування каркасу будівлі, якщо на вулиці будуть здійснюватися будівельні роботи, які супроводжуються гучними та різкими звуками. | |
|  | *8* |

|  |  |
| --- | --- |
| Одним із ключових аспектів формування стійкої багатозадачності людини є стійкість уваги, що можна розтлумачити як здатність протягом тривалого часу утримуватися на якомусь об’єкті, дії. Розглядаючи дане визначення у контексті багатозадачності, можна сказати, що це здатність людини приділяти достатню кількість уваги протягом певного періоду часу для кожної роботи, яка виконується у даний момент, за який людина встигає виконати певний мінімальний обсяг робіт, при якому кожна з виконуваних дій людиною буде здійснюватися без проблем чи затримок, і не перешкоджатиме виконанню іншої дії, яку здійснює людина у цей час.  Одночасно, така успішна діяльність людини у плані багатозадачності втрачає ефективність, якщо дії, які виконує людина, будуть носити виключно характер розумової праці. Значно важкіше, а більшості випадках – неможливо, здійснювати одночасно кілька інтелектуальних задач – складно проводити відразу кілька обчислень, проводити постановку різних задач кільком групам людей і т.п.. В такому випадку відразу проявляється зниження ефективності виконання кожної з робіт, що може нести за собою вагомі наслідки, в залежності від того, наскільки критичною може бути помилка в тій чи іншій діяльності.  Ознайомившись з особливостями та проблематикою багатозадачності людини, можна зробити висновок, що системи керування жестами – ефективне рішення для керування різноманітними системами. Якщо розглядати можливі області застосування подібних інтерактивних систем для керування, перш за все впадає у вічі системи, що впроваджені у мобільних телефонах та мобільних додатках. Зокрема, у них впроваджена можливість захоплення та відстеження рук, розпізнавання та класифікація жестів рук, що надає зможу виконувати широкий спектр дій, як наприклад: здійснення знімку екрана, початок та закінчення відеозапису екрана телефона, розблокування пристрою, збільшення/зменшення гучності і т.п..  Також чудовим прикладом може слугувати інтегрована система керування жестами у бортових комп’ютерах сучасних автомобілів. Наприклад, застосування | |
|  | *9* |

|  |  |
| --- | --- |
| жестів передбачено у керування мультимедією автомобіля. Прості жести, такі як "вказати" або "змахнути" надають можливість викликати певні функції - наприклад, прийняти або відхилити дзвінок. Також передбачена регулювання гучність аудіосистеми завдяки обертовим рухом вказівного пальця. Здійснення планування маршруту в навігаційній системі також здійснюється за допомогою жестів рук, надана можливість за допомогою жестикуляції керувати навігацією в меню. Система управління жестами чудово доповнює канонічний інтерфейс керування мультимедією, такий як кнопковий.  Варто зауважити, що набуло поширення керування за допомогою жестикуляції і в системах «розумний дім». Набір як простих, так і більш складних, комбінованих жестів, відкриває перед користувачем великі можливості для зручного проживання у власній оселі. Таким чином, лише здійснивши кілька простих рухів руками, система надасть можливість керування освітлення, мульмедією будинку, побудовими приладами.  Одним з можливих рішень у застосуванні інтерактивних систем керування жестами, на думку автора, є керування літальними апаратами – мультикоптерами. Натхненний швидким та буремним розвитком коптеробудування та появою нових алгоритмів, методик та інструментів у галузях комп’ютерного зору та глибокого навчання (deep learning), автор розробив власний модуль розпізнавання жестів рук для керування квадрокоптером. Даний вид керування дозволить розширити можливості по управлінню літальними апаратами даного типу, а не лише застосовувати загальноприйняті пульти керування на радіоуправлінні.  Таким чином, інтерактивна система розпізнавання жестів – це складний але необхідний та зручний комплекс, який розкриває нову можливості для керування різноманітними комплексами. Системи такого типу інтуїтивно зрозумілі, завдяки тому, що керування здійснюється жестикуляцією рук оператора. Сам процес розпізнавання жестів рук ділиться на кілька етапів, протягом яких система отримує інформацію, аналізує її, обробляє і на основі неї здійснює висновок щодо жестикуляцій. Інтерактивні системи розпізнавання жестів є складним, але у той же час, надзвичайно вагомим знаряддям людино-машинного інтерфейсу. | |
|  | *10* |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| РОЗДІЛ 1  АНАЛІЗ ПІДХОДІВ ДО ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ  Програмні продукти демонструють стійку тенденцію до зростання і ускладнення. Це природно – продукти розвиваються*,* в них з'являються нові функції і можливості. Але при цьому зростає також складність самих продуктів*,* взаємозв'язків їх компонентів і підсистем*,* інтеграція з іншими додатками. Все це створює значну ймовірність виникнення дефектів в програмному забезпечені.  Тому зараз тестування є неодмінною умовою успішного функціонування як для гігантів індустрії таких як *Google, Microsoft, IBM* чи *Oracle,* так і для розробників програмного забезпечення "під замовлення".  Тестування програмного забезпечення складається з різних випробувань*,* через які повинен пройти програмний продукт і включає різні види діяльності*,* пов’язані з контролем його якості.  Тестування не є самостійною активністю. Воно має своє визначене місце у моделі життєвого циклу розробки продукту і саме ця модель має найбільший вплив на те*,* як процес тестування буде організовано. Від моделі залежить що і коли повинно бути перевірено*,* об’єм регресійного тестування*,* визначає які техніки тестування буде використано та багато іншого. Індустрія програмного забезпечення розвивається стрімкими темпами і сьогодні моделей життєвого циклу і їх варіацій існує досить багато. Класичними вважаються – каскадна*, V-* модель і ітеративна. Найпопулярнішими ж сьогодні є гнучкі моделі розробки.  Найраніша модель розробки ПЗ*,* запропонована 1970 році Уінстоном Ройсом. Вона передбачає послідовне виконання всіх етапів проекту в чітко фіксованому порядку (рис.1.1). | | | | | | | | | |
| Кафедра КСУ | | | | НАУ вввввв 000 ПЗ | | | | | |
| **Виконав** |  |  |  | Аналіз підходів до тестування програмного забезпечення | **Літера** | | | **Аркуш** | **Аркушів** |
| **Керівник** |  |  |  | Д |  |  | *11* | *61* |
| **Консульт*.*** |  |  |  | СП 4….. 123 | | | | |
| **Норм*.* контр*.*** |  |  |  |
| **Зав*.* Каф*.*** |  |  |  |

|  |  |
| --- | --- |
| Рис. 1.1. Каскадна модель або «водоспад»  Перехід на наступний етап означає повне завершення робіт на попередньому. Вимоги*,* визначені на стадії формування вимог*,* суворо документуються у вигляді технічного завдання і фіксуються на весь час розробки проекту. Кожна стадія завершується випуском повного комплекту документації*,* достатньої для того*,* щоб розробка могла бути продовжена іншою командою розробників.  Тестування у каскадній моделі розробки  Місце тестування у моделі «водоспад» є перед завершенням життєвого циклу проекту*,* тобто дефекти становляться виявленими близько до дати випуску продукту на ринок. Відомо*,* що вартість виправлення дефектів знайдених у кінці проекту набагато більша*,* ніж на ранніх етапах (рис.1.2). Найкритичнішою є ситуація*,* коли дефект знайдений у специфікації*,* що вертає розробку на початкову фазу. Інколи такі проекти стає вигіднішим закрити*,* чим починати розробку з початку.  *V*-модель була розроблена для вирішення деяких проблем*,* актуальних для традиційного «водоспаду»: виявлення дефектів на пізніх етапах життєвого циклу розробки продукту. Пізнє підключення тестування також призводить до збільшення затрат часу на нього. *V*-модель забезпечує механізм залучення тестування якомога раніше у життєвому циклі. | |
|  | *12* |

|  |  |
| --- | --- |
| Рис. 1.2. Ціна виправлення дефекту на різних етапах розробки ПЗ  *V*-модель демонструє як перевірка (верифікація і валідація) може бути інтегрована у кожний етап життєвого циклу (рис.1.3).  Рис. 1.3. *V*-модель тестування  Активності з тестування не обмежуються лише безпосереднім виконанням тестів. Є цілий ряд заходів*,* які повинні бути виконані до кінця фази написання коду. Ці заходи повинні проводитися паралельно з кодуванням*,* і спеціалісти з | |
|  | *13* |

|  |  |
| --- | --- |
| тестування повинні співпрацювати на цих етапах із розробниками і бізнес- аналітиками. Почавши дизайн тестів рано*,* працівники галузі тестування виявляють дефекти у специфікаціях та інших вхідних для них документах*,* тим самим попереджуючи перенесеннях цих помилок у код.  У *V*-моделі перевірочні випробування відбувається починаючи з ранніх етапів*,* наприклад*,* огляд потреб користувачів*,* і закінчується у кінці життєвого циклу*,* наприклад*,* під час приймальних випробувань.  Ітеративна модель  Ітеративний підхід означає розробку малого ядра*,* на яке пізніше нарощується інша функціональність. Замість тривалої послідовної розробки від початку до завершення продукту існує цикл певного числа самостійних ітерацій (рис.1.4). Результатом кожної ітерації є продукт*,* що містить частку функціональності кінцевого продукту.  Перевагами такого підходу є більш короткий час до виходу на ринок та можливість швидко внести зміни*,* якщо потреби замовника змінились.  Початок  Вимоги Дизайн  Приймальне Розробка і  тестування тестування  Завершення  Рис. 1.4. Ітеративна модель  Тестування у ітеративній моделі – послідовність ітерації зумовлює – тестування нового функціоналу*,* регресійне тестування існуючого (перевірка*,* що  внесені зміни не задали негативного впливу на інші області програми) та | |
|  | *14* |

|  |  |
| --- | --- |
| інтеграційне тестування між новою та існуючою частинами.  Регресійне тестування є важливим для кожної ітерації після першої. З цього випливає*,* що об’єм тестування для кожної наступної фази буде збільшуватись.  Гнучка розробка бере за основу ітеративний підхід*,* але зумовлює більш  «легкий» та більш орієнтований на людські ресурси механізм. Гнучкі процеси використовують зворотній зв'язок замість планування як основний механізм контролю. Зворотній зв'язок гарантується частими 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 зумовлює спочатку написання автоматичних модульних тестів*,* своєрідних прикладів*,* а потім написання коду безпосередньо.  Гнучка розробка завжди включає велику кількість ітерацій*,* кожна з яких потребує тестування. Тестування включає як повторний «прогін» автоматичних тестів*,* що були створені на стадії написанням коду (активність за яку відповідають зазвичай самі розробники)*,* так і незалежне тестування командою тестування або сторонньою організацію або самим замовником.  Таким чином*,* активності по тестуванню є різноманітними і проходять на різних стадіях життєвого циклу розробки ПЗ. Моделі*,* що є широко вживаними сьогодні зумовлюють залучення тестування якомога раніше і передбачають на кожну активність команди розробки відповідну активність команди тестування.  1.1. Зв'язок між тестування і забезпеченням якості  Тестування проводить аудит якості продукту*,* що знаходиться у розробці. Воно забезпечує зацікавлених людей своєчасним зворотнім зв’язком – надає у певному виражені*,* словесному або у вигляді певних метрик (кількість виявлених дефектів*,* їх критичність*,* кількість успішних/неуспішних тестів)*,* дані про стан продукту*,* на основі яких приймаються рішення.  Тестування – інформаційний сервіс. Тестування вимірює якість*,* однак не | |
|  | *15* |

|  |  |
| --- | --- |
| гарантує її.  Поняття тестування і забезпечення якості часто помилково тлумачать як еквівалентні. Однак ці активності переслідують різні цілі: ціль тестування полягає у знаходженні дефектів*,* цілі активностей по забезпеченню якості – у їх запобіганні.  Тобто тестування є процесом перевірки наперед 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*,* з іншого боку*,* набагато більше спрямоване на безперервне та послідовне поліпшення процесу*,* який дозволяє розробляти якісне програмне забезпечення*,* наприклад*,* підвищення кваліфікації персоналу*,* налагодження комунікації проектної команди тощо. Забезпечення якості повинне відбуватися на усіх етапах життєвого циклу розробки програмного забезпечення. За звичай це вимагає ретельного нотування і контролю багатьох артефактів*,* як наприклад*,* контроль версій продукту.  У традиційних моделях розробки підрозділ з забезпечення якості проводить перевірки на кожному етапі*,* з ретельним нотуванням*,* перевіркою та підписом. Тести та вихідний критерій основані на специфікації вимог*,* а випуск продукту – на результатах тестування.  У гнучких моделях розробки*,* в яких вимоги можуть бути змінені кожні декілька тижнів*,* коли замовники перевіряють чорновий варіант продукту*,* процес забезпечення якості повинен бути більш гнучким.  1.2. Автоматизація процесу тестування програмного забезпечення  Під автоматизацію процесу тестування в широкому сенсі розуміється використання інструментів на будь-якій фазі фундаментального процесу тестування. Це можуть бути інструменти для автоматичної генерації звітів за результати тестування*,* інструменти для створення тестових даних тощо. | |
|  | *16* |

|  |  |
| --- | --- |
| Інструменти тестування можуть поліпшити його ефективність шляхом автоматизації повторюваних завдань. Засоби тестування також можуть поліпшити надійність тестування шляхом*,* наприклад*,* автоматизації порівняння великих обсягів даних чи емуляції користувацької поведінки.  Сьогодні ринок програмного забезпечення*,* що асистує тестуванню досить різноманітний. Є безліч засобів тестування*,* які підтримують різні його аспекти. Існуючи інструменти можна класифікувати за декількома критеріями – цілі*,* вартість (комерційні*,* безкоштовні*,* з відкритим кодом)*,* за технологією*,* що покладено в основу тощо.  Найпоширенішою є класифікація за активностями*,* які інструмент підтримує. Деякі засоби мають лише одне призначення; інші можуть підтримувати більше*,* ніж одне*,* але класифікуються за тією функцією*,* з якою вони найбільш тісно пов'язані:   * інструменти*,* що асистують управлінню тестуванням і тестами; * інструменти*,* що асистують статичному тестуванню; * інструменти*,* що асистують створенню тестів; * інструменти*,* що асистують виконанню тестів та запису результатів; * інструменти*,* що асистують моніторинг продуктивності.   1.2.1. Інструменти*,* що асистують управлінню тестуванням і тестами Засоби управління застосовуються у всіх процесах тестування протягом  всього життєвого циклу розробки ПЗ.  Інструменти управління тестуванням. Інструменти цієї групи забезпечують:   * підтримку керування виконанням тестів і процесами тестування; * інтерфейс для засобів виконання тестів*,* засобів відстеження дефектів і засобів управління вимогами; * незалежний засіб керування версіями або інтерфейс для зовнішнього засобу управління конфігураціями; * засоби для трасуванню тестів*,* результатів тестування та звітів про інциденти із вхідною документацією*,* такою як специфікація вимог Запис | |
|  | *17* |

|  |  |
| --- | --- |
| результатів тестування та генерація звітів;   * кількісний аналіз (метрики)*,* пов'язані з тестами (наприклад*,* виконані тести і успішні тести) і тестовим об'єктах (наприклад*,* виявлені відмови системи)*,* для того*,* щоб дати інформацію про тестові об'єкти і для того*,* щоб контролювати і поліпшувати процес.   Засоби керування вимогами зберігають вимоги*,* перевіряють їх на узгодженість і наявність невизначених (пропущених) вимог*,* дозволяють розставляти пріоритети за вимогами і дозволяють окремим тестами трасуватися згідно до вимогам*,* функцій або опцій. Результати трасування можуть бути включені у звіти про тестування. Покриття вимог*,* функцій або опцій набором тестів також може бути відображено у звітах.  Засоби управління інцидентами  Засоби управління інцидентами зберігають і управляють звітами про відмови*,* тобто дефекти*,* збої або проблеми і відхилення*,* і забезпечують управління звітами про інциденти таким чином*,* що:   * допомагають виставляти пріоритети; * розподіляють необхідні дії між людьми (наприклад*,* тестування виправлень); * визначення статусів (наприклад*,* скасований*,* готовий бути протестований або відкладений до наступного випуску).   Ці засоби дозволяють постійно відстежувати звіти про інциденти*,* часто сприяють статистичному аналізу і надають звіти про інциденти. вони також відомі як засобу відстеження дефектів.  Засоби управління конфігураціями  Засоби управління конфігураціями не є*,* строго кажучи*,* засобами тестування*,* але зазвичай вони необхідні для того*,* щоб відслідковувати різні версії і збірки програмного забезпечення і тести.   * зберігають інформацію про версії та збірки ПЗ і проведене тестуванням; * забезпечують зіставлення між проведеним тестуванням і результатом роботи ПЗ та варіантами продукту.   Особливо корисні*,* коли розробка ведеться більш ніж на одній конфігурації | |
|  | *18* |

|  |  |
| --- | --- |
| апаратного та програмного забезпечення (наприклад*,* для різних версій операційних систем*,* різних бібліотек або компіляторів*,* різних браузерів або різних комп'ютерів)  1.2.2. Інструменти*,* що асистують статичному тестуванню  Засоби підтримки процесу експертизи можуть зберігати інформацію про процеси експертизи*,* зберігати і обмінюватися експертними коментарями*,* надавати інформацію про дефекти і обсяги робіт*,* керувати зв'язками з експертними правилами та / або списками перевірки*,* і забезпечувати трасуванню між документацією і вихідним кодом. Також вони можуть надавати можливість експертизи в реальному часі*,* яка корисна для географічно розподіленої команди.  Засоби статичного аналізу допомагають розробникам*,* спеціалістам з тестування і співробітникам забезпечення якості в пошуку дефектів до початку динамічного тестування. Основні цілі включають:  Дотримання стандартів програмування:   * аналіз структури і залежностей (наприклад*,* посилання на веб-сторінках); * допомога в розумінні коду; * засоби статичного аналізу можуть витягувати метрики з вихідного коду (наприклад*,* складність)*,* які можуть дати цінну інформацію*,* наприклад*,* для аналізу планування або ризиків.   Засоби моделювання здатні перевіряти достовірність моделей ПЗ. Наприклад*,* засіб контролю моделей баз даних може знайти дефекти і невідповідності в моделі даних; інші засоби моделювання можуть знайти дефекти в моделях станів або об'єктних моделях. Ці кошти найчастіше можуть допомогти в складанні деяких тестових сценаріях*,* заснованих на моделях. Основна перевага засобів статичного аналізу і засобів моделювання – вигода в вартості знаходження більшого числа дефектів на більш ранніх стадіях циклу розробки. Як результат*,* процес розробки може прискоритися і покращитися за рахунок меншого числа переробок. | |
|  | *19* |

|  |  |
| --- | --- |
| 1.2.3. Інструменти*,* що асистують створенню тестів  Засоби проектування тестів генерують вхідні дані або безпосередньо тести основуючись на вимогах*,* графічному інтерфейсові користувача*,* моделей проектування (станів*,* даних або об'єктів) або коду. Ці інструменти можуть також генерувати і очікувані результати (тобто використовувати роль тестового оракулу). Тести*,* генеровані з моделі станів чи об'єктів*,* корисні для перевірки реалізації моделі в ПЗ*,* але рідко достатні для перевірки всіх аспектів продукту або системи. Вони можуть скоротити дорогоцінний час і забезпечити підвищену глибину тестування за рахунок повноти тестів. Інші засоби цієї категорії можуть допомогти в підтримці генерації тестів*,* забезпечуючи структурні шаблони*,* іноді звані тестовими фреймами*,* які генерують тести чи тестові заглушки*,* таким чином збільшуючи швидкість проектування тестів.  Засоби підготовки тестових даних оперують з базами даних*,* файлами*,* або передачею даних для визначення тестових даних*,* які будуть використовуватися під час виконання тестів. Перевага таких інструментів у тому*,* що реальні дані передаються в тестове середовище анонімно для захисту даних.  Засоби виконання тестів дозволяють виконувати тести автоматично або напівавтоматично*,* використовуючи збережені вхідні дані і очікувані результати*,* з використанням мови скриптів.  Скриптова мова дозволяє керувати тестами з обмеженими витратами*,* наприклад*,* повторювати тест з різними даними або перевіряти різні частини системи однаковими кроками. Зазвичай такі інструменти включають функції динамічного порівнювання і надають можливість протоколювання кожного запуску тесту.  Засоби виконання тестів також можуть використовуватися механізм запису/відтворення тестів. Запис тестових вхідних даних під час дослідницького тестування може бути корисно для відтворення або документування тесту*,* наприклад*,* якщо відбулася відмова.  Тестові вузли можуть полегшувати тестування компонентів або частини  системи*,* імітуючи оточення*,* на якому об'єкт тестування буде працювати. Це може бути зроблено через те*,* що інші компоненти цього оточення ще не готові і | |
|  | *20* |

|  |  |
| --- | --- |
| замінені заглушками і / або драйверами*,* або просто для того*,* щоб забезпечити передбачуване і контрольоване оточення*,* в якому будь-які відмови можуть бути локалізовані в об'єкті тестування. Оболонка може бути створена там*,* де частина коду*,* об'єкт*,* метод або функція*,* модуль чи компонент можуть бути виконані*,* викликаючи об'єкт тестування або забезпечуючи зворотний зв'язок з цим об'єктом.  Тестові порівняльники встановлюють відмінності між файлами*,* базами даних або тестовими результатами. Засоби виконання тестів зазвичай включають динамічні порівняльники*,* але порівняння після виконання може бути виконано окремими засобами. Тестовий порівняльник може використовувати тестового оракула*,* особливо якщо він автоматизований.  Засоби покриття коду вимірюють відсоток конкретних типів структур коду*,* які були перевірені (наприклад*,* команди*,* гілки або рішення*,* і виклики модулів або функцій). Ці інструменти показують*,* наскільки глибоко вимірюваний тип структур був перевірений набором тестів.  Засоби безпеки перевіряють на комп'ютерні віруси та *DoS*-атаки. Інші інструменти безпеки навантажують систему*,* шукаючи певні уразливості в ній.  Засоби динамічного аналізу знаходять дефекти*,* які стають очевидними тільки*,* коли ПЗ виконується*,* такі як тимчасові затримки і витоки пам'яті. Зазвичай вони використовуються при тестуванні компонентів та їх інтеграції.  Засоби тестування продуктивності відстежують і повідомляють про те*,* як веде себе система при різних умовах використання. Вони імітують навантаження на додаток*,* базу даних або оточення системи*,* таке як мережа або сервер. Часто ці засоби називаються відповідно до того*,* що вони вимірюють*,* наприклад*,* для перевірки навантаження або стресових умов застосовуються засоби навантажувального і стресового тестування.  Засоби моніторингу не зовсім є засобами тестування*,* але вони надають інформацію*,* яка може бути використана з метою тестування і недоступна іншими способами.  Засоби моніторингу безперервно аналізують*,* перевіряють і повідомляють про використання конкретних системних ресурсів і видають попередження*,* якщо | |
|  | *21* |

|  |  |
| --- | --- |
| можливі проблеми. Вони зберігають інформацію про версію і збірку ПЗ та тестове середовище і забезпечують трасуванню.  Під автоматизацією тестування у вузькому сенсі мають на увазі залучення інструментів до перевірки програми під час її виконання – тобто під час динамічного тестування. Подібні інструменти емулюють дії користувача за допомогою спеціальних тестових фремворків.  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.  Автоматизація продуктів з графічним користувацьким інтерфейсом розвивалася протягом 4 поколінь інструментів та технік:  1.2.4. Інструменти запису і відтворення  Дані інструменти записують дії тестувальника під час ручного виконання тестів. Вони дозволяють виконувати тести без прямої участі людини протягом тривалого часу*,* значно збільшуючи продуктивність і усуваючи необхідність повторення одноманітних дій під час ручного тестування. У той же час*,* будь-яке незначна зміна інтерфейсу вимагає перезапису тестів. Тому це перше покоління інструментів не ефективне і не масштабується.  Сценарії (*Scripting*) – форма програмування на мовах*,* спеціально розроблених для автоматизації тестування ПЗ – пом'якшує багато проблем *capture* / *playback* інструментів. Але розробкою займаються програмісти високого рівня*,* які працюють окремо від тестувальників*,* безпосередньо запускають тести. До того ж скрипти найбільше підходять для тестування *GUI* і не можуть бути впровадженими*,* пакетними або взагалі яким-небудь чином об'єднані в систему. Нарешті*,* зміни в ПЗ вимагають складних змін у скриптах*,* і підтримка все зростаючою бібліотеки тестуючих скриптів стає врешті-решт непереборної завданням.  *Data*-*driven testing* – методологія*,* яка використовується в автоматизації | |
|  | *22* |

|  |  |
| --- | --- |
| тестування. Особливістю є те*,* що тестові скрипти виконуються та верифікуються на основі даних*,* які зберігаються в центральному сховищі даних або БД. Роль БД можуть виконувати *ODBC*-ресурси*, csv* або *xls* файли і т.д. *Data*-*driven testing* – це об'єднання декількох взаємодіючих тестових скриптів і їх джерел даних у фреймворк*,* який використовується в методології.  1.3. Обґрунтування генерації тестів  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 скриптів для автоматизації дії користувача коштує мінімум трьом циклам ручного тестування [1] і для короткострокових проектів не має економічного сенсу); а також через те*,* що 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.  Оскільки ручне виконання тестів не може бути замішено автоматичним і має своє місце у життєвому циклі розробки кожного продукту*,* для нього також необхідно залучати інструментальну підтримку з метою скорочення затрат часу спеціалістів відділу тестування і усунення «людського» фактору – можливості допуститися помилки.  Тому була висунута наступна задача для дослідження: віднаходження підходу до автоматизації процесу генерації тестів для ручного виконання.  Оскільки сьогодні найпоширенішим є клас програмного забезпечення з графічним інтерфейсом користувача*,* то саме цей клас і буде розглядатися як цільовий у даній роботі.  Підхід повинен відповідати наступним вимогам: | |
|  | *23* |

|  |  |
| --- | --- |
| Гнучкість*.* Підхід повинен бути досить універсальним*,* щоб бути поширеним на додатки з різних предметних галузей*,* виконаним з використанням різних архітектур і технологій.  Простота і легкість*.* Реалізація повинна бути досить простою*,* щоб не виникло необхідності тривалої і коштовної перепідготовки персоналу.  Тести будуються таким чином*,* щоб включати граничні*,* мінімальні та максимальні значення діапазонів з правильними і неправильними даними. Техніка може бути застосована як для вхідних так і вихідних значень.  За цим методом уся область вхідних (вихідних) даних розділяється на класи даних*,* з припущенням*,* що всі представники одного класу оброблюються додатком однаково. Цей метод використовується для досягнення двох цілей:   * зменшення тестових випадків у наборі; * для вибору правильних тестів*,* що покривають усі можливі сценарії.   Зазвичай техніки аналізу граничних значень і розбиття на еквівалентні класи використовуються у комбінації.  Діаграма станів та переходів використовується там*,* де аспекти системи можуть бути описані як скінчений автомат. Це означає*,* що система може перебувати в (скінченому) числі різних станів*,* і переходи з одного стану в інший визначаютьс правила «машини». На основі моделі скінченого автомату будуються тести.  Перевагами такого підходу є те*,* що в ньому перераховуються всі можливі комбінації станів – переходів*,* а не тільки ті*,* що є правильними: тестування неправильних шляхів використовується для додатків з високими ступенями ризику.  Створення подібної діаграми часто робить очевидними такі комбінації*,* які не були визначені*,* задокументовані*,* або розглянуті у неформальних вимогах. Це дуже вигідно для виявлення дефектів до початку безпосереднього створення коду.  Перші дві техніки широко застосовуються для перевірок різноманітного вводу у додатку*,* техніка діаграми станів і переходів дає змогу побудувати повний сценарій взаємодії користувача із цільовою системою. | |
|  | *24* |

|  |  |
| --- | --- |
| Тестові набори являють собою послідовність тестів*,* яка формується з урахуванням необхідного рівня тестового покриття. Тести в наборі не повинні дублювати перевірки*,* однак сформований набір повинен повністю задовольняти встановлену метрику якості робіт.  Таким чином загальна задача дослідження – автоматична генерація функціональних тестів для додатків з графічним інтерфейсом для ручного виконання може бути декомпозована на низку підзадач.   * 1. Висновки до розділу   У даному розділі було розглянуто загальну технологію використання програмних засобів для проведення автоматизованого тестування програмного забезпечення та визначено основні можливості і перспективи їх використання. 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;     - короткостроковий проект;     - через те*,* що 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. | |
|  | *25* |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| РОЗДІЛ 2  ОБГРУНТУВАННЯ ПІДХОДУ ДО РОЗРОБКИ СИСТЕМИ  2.1. Формальна модель представлення вимог  Формальне представлення функціональних вимог перетворює їх на  «машино читаємі»*,* тобто створює передумови повністю автоматизувати процес генерації тестів на їх основі.  Під формальною специфікаці*є*ю розуміють математичний опис програмної або апаратної системи*,* котра може бути реалізована на основі цього опису. Специфікується що повинна робити система*,* а не як вона повинна це робити.  Додатковим бонусом від формалізації вимог є по-перше*,* те*,* що протягом процесу їх перетворення виявляються недоліки – неточності або протиріччя у їх тексті*,* тобто проводиться їх статичне тестування. По-друге*,* вимоги у такому вигляді мають більшу тривалість життя*,* ніж у вигляді знань окремих індивідів*,* а також потребують менш зусиль для підтримки*,* ніж вимоги сформульовані на природній мові – що призводить до скорочення витрат коштів на підтримку.  Серед існуючих формальних моделей вимог*,* поведінки та оточення програмного забезпечення для аналізу його властивостей виділяють наступні групи логіко*-*алгебра*ї*чні*,* операційні та проміжні*.*  Логіко-алгебраїчні моделі. При моделюванні ПЗ модель такого типу описує деякий набір його властивостей*,* які можуть змінюватися з часом*,* але не дає точного уявлення про те*,* за рахунок чого змінюються ці властивості. Відмінність між логічним та алгебраїчним численням досить умовне*,* але*,* дещо спрощуючи*,* можна вважати*,* що логіка має справу з твердженнями в рамках якоїсь мови*,* а алгебра – з рівностями та нерівностями*,* які побудовані в деякій мові виразів. | | | | | | | | | |
| Кафедра КСУ | | | | НАУ вввввв 000 ПЗ | | | | | |
| **Виконав** |  |  |  | Обгрунтування підходу до розробки системи | **Літера** | | | **Аркуш** | **Аркушів** |
| **Керівник** |  |  |  | Д |  |  | *26* | *61* |
| **Консульт*.*** |  |  |  | СП 4……. 123 | | | | |
| **Норм*.* контр*.*** |  |  |  |
| **Зав*.* Каф*.*** |  |  |  |

|  |  |
| --- | --- |
| У першому випадку основним об'єктом уваги є твердження*,* хибні або істинні*,* а в другому – вирази або терми*,* пов'язані з якимось типом.  Приклади логічних числень такі:  Обчислення висловлювань. У ньому є атомарні висловлювання*,* які можливо залежать від об'єктних змінних*,* а також логічні зв'язки  («і»*,* сполучення)*,*  («або»*,* диз'юнкція)*,*  («не»*,* заперечення)*,* => («отже»*,* імплікація)*,* = (еквівалентність)*,* за допомогою яких можна з одних висловлювань будувати інші*,* більш складні.  Обчислення предикатів додає перелік висловлювань можливість використовувати квантори по об'єктним змінним для побудови нових тверджень. Квантори в цьому обчисленні бувають двох видів –  «загальності» та   «існування». У численні предикатів крім об'єктних змінних є функціональні і предикатні. Перші являють собою різноманітні функції*,* результат застосування функції до об'єкта – об'єкт. Другі невизначені твердження*,* результатом їх застосування до об'єкту має бути або «*true*»*,* або «*false*» . У не типізованому обчисленні всі об'єкти рівноправні*,* функції та предикати можуть приймати будь- які об'єкти в якості аргументів. У типізованих численнях кожен об'єкт має тип*,* а функціональні і предикатні змінні – сигнатуру*,* тобто список типів параметрів і тип результату для функцій. Відповідно*,* будувати формули можна лише дотримуючись відповідності типів параметрів типам виразів*,* які підставляють на місце цих параметрів.  Обчислення предикатів більш високих порядків*.* У цих обчисленнях можна використовувати квантори не тільки по об'єктним змінним*,* але і по функціональним або предикативним. Наприклад*,* визначення рівності об'єктів іноді формулюється так: *x*  *y*  *P*, *P*(*x*)  *P*( *y*), два об'єкти рівні*,* якщо будь-  яке твердження одночасно виконано або не виконано для них обох.  Обчислення з допомогою лямбда – оператор дозволяє будувати функції з виразів*,* наприклад*,* вираз *λx x*\**x* позначає функцію зведення в квадрат. У цьому прикладі також є зв’язаною змінною яка не має власного значення. У *λ –* численні з типами*,* так само як і в типізованому численні предикатів*,* об'єкти мають типи*,* а функції – сигнатури. *λ –* числення більш високих порядків | |
|  | *27* |

|  |  |
| --- | --- |
| дозволяють застосовувати *λ –* оператор не лише до об'єктів*,* а й до типів. При цьому виходять функції*,* що перетворюють типи в типи.  Приклади алгебраїчних моделей: реляційні алгебри*,* що лежать в основі реляційних систем управління базами даних.  Алгебри процесів. Це алгебраїчні обчислення*,* об'єктами яких є події та процеси*,* що створюють події або реагують на них. Зазвичай для процесів визначені операції послідовної (';') і паралельної ('||') композиції і операція вибору з двох альтернатив (альтернативна композиція*,* '+'). Послідовна композиція процесів моделює виконання спочатку першого з них*,* потім другого. Паралельна композиція моделює паралельне виконання обох процесів. Вибір з двох процесів моделює виконання або першого*,* або другого. У більшості таких числень процеси можуть взаємодіяти*,* обмінюючись подіями (один процес породжує подію*,* інший або інші його споживають). Найбільш широко відомі обчислення процесів *CSP* (*Communicating Sequential Processes*)*, CCS* (*Calculus of Communicating Systems*).  **Операційні моделі** характеризуються тим*,* що їх можна виконати*,* щоб простежити зміну властивостей змодельованого програмного забезпечення.  Всі види операційних моделей можна вважати розширенням і узагальненням кінцевих автоматів*,* тому їх варто розглянути в якості першого прикладу.  Кінцевий автомат являє собою систему з кінцевою множиною станів. В кінцевому автоматі визначено певний набір переходів між станами.  Вза*є*модіючі автомати представляють собою набір кінцевих автоматів*,* деякі з яких пов'язані каналами для передачі реакцій одного автомата як стимулів для іншого.  Ієрархічні автомати дозволяють визначати переходи з групи станів (але кінцевий стан такого переходу має бути лише один). Крім того*,* в ієрархічних автоматах можуть бути групи паралельних станів – кілька груп станів*,* що об'єднуються в паралельне сімейство.  Тимчасові автомати – зазвичай це розширені автомати*,* що містять додатковий набір змінних-таймерів*,* значення яких змінюються самі по собі з | |
|  | *28* |

|  |  |
| --- | --- |
| плином часу. Значення таймерів можна використовувати в умовах переходів*,* зміну змінних або значеннях параметрів реакцій. Крім цього*,* таймери можна запускати в діях*,* пов'язаних з переходами. Запущений таймер починає відлік часу з 0. Час може бути дискретним чи безперервним*,* що дозволяє моделювати поведінку різноманітних систем реального часу.  Гібридні автомати застосовувані для моделювання систем*,* що взаємодіють з безперервними фізичними процесами. У цих автоматах частина змінних зазвичай має значення*,* зміна яких описується системою диференціальних рівнянь  Широке використання операційних моделей пов’язано з їх наступними перевагами:  У моделі звичайно враховуються не всі властивості модельованого програмного забезпечення*,* а тільки важливі для розглянутої в даний момент завдання. Тому моделі виявляються значно простіше модельованих систем*,* їх набагато зручніше аналізувати.  Віртуальні машини використовуються на практиці мов програмування дуже складні і визначені нечітко*,* а віртуальні машини моделей значно більш прості і доступні для огляду. Це дозволяє провести вичерпний аналіз можливої поведінки моделі*,* виявити всі класи можливих при її роботі ситуацій.  Моделі проміжного типу мають риси як логіко-алгебраїчних*,* так і операційних. Основні їх види такі.  Логіки Хоара є специфічним видом логік*,* затвердження яких складаються з формул логіки деякого виду та програмних команд. У найпростішому вигляді це трійки {*Q*} *S* {*R*}*,* де *S* – частина програми на певній мові*,* а *Q* і *R* – формули обчислення висловлювань*,* що залежать від змінних*,* що входять до *S*. *Q* інтерпретується як умова*,* виконання перед початком виконання *S* (передумова)*,* а *R* – як умову*,* що має виконуватись після виконання *R* (післяумови). Якщо *R* дійсно завжди істинне після виконання *S* у стані*,* де істинно *Q,* така трійка теж вважається дійсною і має назву «трійка Хоара». У логіці Хоара для деякої мови програмування*,* семантика цієї мови задається у вигляді правил виведення*,* які дозволяють з тавтологій виводити крок за кроком істинні трійки Хоара. | |
|  | *29* |

|  |  |
| --- | --- |
| Узагальненням логік Хоара є динамічні чи програмні логіки. Вони є спеціальним типом модальних логік*,* в яких оператори модальності пов'язані з інструкціями програм. Зазвичай використовуються оператори [*S*] і <*S*>*,* де *S* – деяка програма. Твердження [*S*]*Q* означає*,* що завжди після виконання програми *S* формула *Q* істинна*,* а <*S*> *Q* – що після виконання *S, Q* може виявитися істинною. Трійка Хоара {*Q*} *S* {*R*} може бути представлена у динамічній логіці як *Q*=>[*S*]*R*.  Програмні контракти*,* є окремим випадком логіки Хоара*,* що звужує можливості використання логічних формул. Програмний контракт представляє собою опис поведінки набору програмних компонентів представлених у вигляді опису сигнатур операцій кожного з цих компонентів*,* структур їх станів*,* а також передумов і післяумов для кожної операції та наборів інваріантів для кожного компонента окремо. Інваріант компонента є предикатом*,* що залежить від елементів стану цього компонента*,* який повинен бути виконаний в станах*,* коли жодна з операцій компоненту не виконується. Інваріанти описують обмеження цілісності внутрішніх даних компонента. Передумову операції компонента являє собою предикат*,* що залежить від елементів стану цього компонента і параметрів цієї операції. При виклику операції з порушенням передумови її поведінка не визначена. Передумова є частиною контракту*,* яку зобов'язане дотримуватися оточення компонента*,* щоб забезпечити його коректну роботу. Післяумова операції являє собою предикат*,* що залежить від параметрів операції*,* її результату*,* елементів стану компонента до виклику операції і тих же елементів після закінчення її роботи. Контракти часто неможливо виконати безпосередньо*,* оскільки післяумова не визначає прямо коректні результати операцій і наступні стани*,* а лише оцінює передані ним дані.  Серед усіх формальних підходів*,* що можуть бути застосовані до представлення специфікацій широке практичне застосування у проектуванні і тестуванні засобів автоматики*,* зв’язку*,* обчислювальної техніки і програмного забезпечення знайшли скінчені автомати.  Вибір на користь скінченого автомату зумовлений тим*,* що побудова логіко  – алгебраїчних та проміжних моделей практично не вигідна в силу величезних | |
|  | *30* |

|  |  |
| --- | --- |
| витрат на розробку додаткових аксіом*,* тверджень та умов(передумов) та післяумов*,* що визначають заключні правила отримання правильного результату.  2.2. Аналіз сучасних методик та засобів тестування програмних застосувань для мобільних пристроїв  Існуючі на сьогоднішній день методи тестування ПЗ не дозволяють однозначно і повністю виявити всі дефекти і встановити коректність функціонування аналізованої програми*,* тому всі існуючі методи тестування діють у рамках формального процесу перевірки досліджуваного або розроблюваного ПЗ.  Такий процес формальної перевірки*,* чи верифікації*,* Може довести*,* що дефекти відсутні з точки зору використовуваного методу. (Тобто немає ніякої можливості точно встановити або гарантувати відсутність дефектів у програмному продукті з урахуванням людського фактора*,* присутнього на всіх етапах життєвого циклу ПЗ).  Існує безліч підходів до вирішення завдання тестування і верифікації ПЗ*,* але ефективне тестування складних програмних продуктів - це процес у вищій мірі творчий*,* не зводиться до слідування строгим і чітким процедурам або створенню таких [8].  З точки зору *ISO* 9126*,* якість програмного забезпечення можна визначити як сукупну характеристику досліджуваного ПЗ з урахуванням наступних складових:   * надійність; * супроводжуваність; * практичність; * ефективність; * мобільність; * функціональність.   Існує кілька ознак*,* за якими прийнято проводити класифікацію видів тестування. Зазвичай виділяють наступні: | |
|  | *31* |

|  |  |
| --- | --- |
| 1) 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*); * юзабиліті-тестування (*usability testing*); * тестування інтерфейсу користувача (*UI testing*); * тестування безпеки (*security testing*); * тестування локалізації (*localization testing*); * тестування сумісності (*compatibility testing*);   2) за наявністю доступу до системи:   * тестування Чорної Скриньки (*black box*); * тестування білої скриньки (*white box*) * тестування сірої скриньки (*grey box*);   3) за ступенем автоматизації:   * ручне тестування (*manual testing*); * автоматизоване тестування (*automated testing*); * напівавтоматизоване тестування (*semiautomated testing*);   4) за ступенем ізольованості компонентів:   * компонентне (модульне) тестування (*component*/*unit testing*); * інтеграціонне тестування (*integration testing*); * системне тестування (*system*/*end*-*to*-*end testing*);   5) за часом проведення тестування:   * альфа-тестування (*alpha testing*); * тестування при прийомі (*smoke testing*); * тестування нової функціональності (*new feature testing*); * регресійне тестування (*regression testing*); * тестування при здачі (*acceptance testing*); * бета-тестування (*beta testing*);   6) за ознакою позитивності сценаріїв: | |
|  | *32* |

|  |  |
| --- | --- |
| * позитивне тестування (*positive testing*); * негативне тестування (*negative testing*);   7) за ступенем підготовленості до тестування:   * тестування по документації (*formal testing*); * тестування *ad hoc* або інтуїтивне тестування (*ad hoc testing*).   Необхідність автоматизації тестування класифікується за наступними ознаками:  1) за об’єктом тестування:   * функціональному тестуванні (*functional testing*); * тестуванні продуктивності (*performance testing*); * навантажувальному тестуванні (*load testing*); * стрес-тестуванні (*stress testing*); * тестуванні стабільності (*stability* / *endurance* / *soak testing*);   2) за наявністю доступу до системи:  – тестування чорної скриньки (*black box*);  3) за ступенем ізольованості компонентів:  – системне тестування (*system*/*end*-*to*-*end testing*);  4) за часом проведення тестування:   * альфа-тестування (*alpha testing*) * тестування при прийомі (*smoke testing*) * регресійне тестування (*regression testing*) * тестування при здачі (*acceptance testing*) * Бета-тестування (*beta testing*)   Розглянемо більш детально види тестування в яких може застосовуватись автоматизація:  1) функціональне тестування – це тестування ПЗ в цілях перевірки можливості реалізації функціональних вимог*,* тобто здатності ПЗ в певних умовах вирішувати задачі*,* необхідні користувачам.  Функціональні вимоги визначають*,* що саме робить ПЗ*,* які задачі воно вирішує.  Функціональні вимоги включають в себе: | |
|  | *33* |

|  |  |
| --- | --- |
| * функціональну пригодність (англ. *suitability*); * точність (англ. *accuracy*); * здатність до взаємодії (англ. *interoperability*). * Відповідність стандартам і правилам (англ.*compliance*). * Захищеність (англ. *security*).   2) тестування продуктивності в інженерії програмного забезпечення - тестування яке проводиться з метою визначення*,* як швидко працює система або її частину під певним навантаженням. Також може служити для перевірки і підтвердження інших атрибутів якості системи*,* таких як масштабованість*,* надійність і споживання ресурсів.  Тестування продуктивності – це одна зі сфер діяльності розвивається в галузі інформатики інженерії продуктивності*,* яка прагне враховувати продуктивність на стадії моделювання та проектування системи*,* перед качаном Основному стадії кодування. У тестуванні продуктивності розрізняють наступні напрямки:   * стрес (*stress*); * навантажувальне (*load*); * тестування стабільності (*endurance or soak or stability*); * конфігураційне (*configuration*).   3) навантажувальне тестування (англ. *Load Testing*) – визначення або збір показників продуктивності і годині відгуку програмно-технічної системи або пристрою у відповідь на зовнішній запит з метою встановлення відповідності вимогам*,* що пред'являються до даної системи (пристрою). Для дослідження годині відгуку системи на високих або пікових навантаженнях проводиться стрес-тестування*,* при якому створювана на систему навантаження перевищує нормальні сценарії його використання. Не існує чіткої межі між навантажувальним та стрес-тестуванням*,* однак ці поняття не варто змішувати*,* так як ці види тестування відповідають на різні бізнес-питання і використовують різну методологію.  Термін тестування навантаження може бути використаний у різних значеннях в професійному середовищі тестування ПЗ. У загальному випадку він | |
|  | *34* |

|  |  |
| --- | --- |
| означає практику моделювання очікуваного використання додатка за допомогою емуляції роботи декількох користувачів одночасно. Таким чином*,* подібне тестування найбільше підходить для екстермінатуса мультикористувацьких систем*,* частіше - використовують клієнт-серверну архітектуру (наприклад*,* веб- серверів). Однак і інші типи систем ПЗ можуть бути протестовані подібним способом. Наприклад*,* текстовий або графічний редактор можна змусити прочитати дуже великий документ; а фінансовий пакет - згенерувати звіт на основі даних за декілька років. Найбільш адекватно спроектований навантажувальний тест дає більш точні результати.  Основна мета навантажувального тестування полягає в тому*,* щоб*,* створивши певну очікувану в системі навантаження (наприклад*,* за допомогою віртуальних користувачів) і*,* звичайно*,* використавши ідентичне програмне і апаратне забезпечення*,* спостерігати за показниками продуктивності системи. В ідеальному випадку в якості критеріїв успішності навантажувального тестування виступають вимоги до продуктивності системи*,* які формулюються і документуються на стадії розробки функціональних вимог до системи до початку програмування основних архітектурних рішень. Однак часто буває так*,* що такі вимоги не були чітко сформульовані або не були сформульовані зовсім. У цьому випадку перше навантажувальний тестування буде являтися пробним (*exploratory load testing*) і ґрунтуватися на розумних припущеннях про очікувану навантаженні і споживанні апаратної частини ресурсів.  Розробка методів побудови математичних моделей "чорної" скриньки є однією з важливих кібернетичних проблем. За умови наявності математичної моделі "чорної" скриньки з'являється можливість віднести його до якогось одного класу*,* всі системи якого ізоморфні по поведінці.  Для науки метод “чорної скриньки” має дуже велике значення. З його допомога в науці було зроблено дуже багато видатних відкриттів. Наприклад*,* вчений Гарвей Галі в *XVII* столітті передбачив будову серця. Він моделював роботу серця насосом*,* запозичивши ідеї із зовсім іншої області сучасних йому знань - гідравліки. Практична цінність методу “чорної скриньки” полягає по- перше*,* в можливості дослідження дуже складних динамічних систем*,* і*,* по-друге*,* | |
|  | *35* |

|  |  |
| --- | --- |
| в можливості заміни одного "ящика" іншим. Навколишня дійсність і біологія дають масу прикладів виявлення будови систем методом "чорної" скриньки.  Принципи тестування чорного ящика – у цьому методі програма розглядається як чорний ящик. Метою тестування ставиться з'ясування обставин*,* в яких поведінка програми не відповідає специфікації. Для виявлення всіх помилок у програмі необхідно виконати вичерпне тестування*,* тобто тестування на всіх можливих наборах даних. Для більшості програм таке неможливо*,* тому застосовують розумне тестування*,* при якому тестування програми обмежується невеликою підмножиною можливих наборів даних. При цьому необхідно вибирати найбільш відповідні*,* підмножини з найвищою імовірністю виявлення помилок.  Прийоми тестування чорного ящика:   * еквівалентне розбиття; * аналіз граничних значень; * аналіз причинно-наслідкових зв’язків; * допущення про помилку; * системне тестування.   2.2.1. Тестування через еквівалентне розбиття Основу методу складають два положення:   1. вихідні данні необхідно розбити на кінцеве число класів еквівалентності; 2. в одному класі еквівалентності містяться такі тести*,* що*,* якщо один тест з класу еквівалентності виявляє деяку помилку*,* то й будь який інший тест з цього класу еквівалентності має виявляти цю ж помилку.   Кожен тест має включати*,* по можливості*,* максимальну кількість класів еквівалентності*,* щоб мінімізувати загальне число тестів.  Розробка тестів цим методом здійснюється в два типа: виділення класів еквівалентності і побудова тесту.  Класи еквівалентності виділяються шляхом вибору кожної вхідної умови*,* які беруться з допомогою технічного завдання або специфікації і розбиваються на дві та більше групи. | |
|  | *36* |

|  |  |
| --- | --- |
| Виділення класів еквівалентності є евристичним способом*,* однак існує ряд правил:  Якщо вхідна умова описує область значень*,* наприклад "Ціле число приймає значення від 0 до 999"*,* то існує один правильний клас еквівалентності і два неправильних.  Якщо вхідна умова описує число значень*,* наприклад "Число рядків у вхідному файлі лежить в інтервалі (1.6)"*,* то також існує один правильний клас і два неправильних.  Якщо вхідна умова описує безліч вхідних значень*,* то визначається кількість правильних класів*,* рівна кількості елементів у множині вхідних значень. Якщо вхідна умова описує ситуацію "повинно бути"*,* наприклад "Перший символ має буті заголовним"*,* тоді один клас правильний і один неправильний.  Якщо є підстава вважати*,* що елементи всередині одного класу еквівалентності можуть програмою трактуватися по-різному*,* необхідно розбити даний клас на підкласи. На цьому кроці тестувальник на основі таблиці має скласти тести*,* що покривають собою всі правильні і неправильні класи еквівалентності. При цьому тестувальник має мінімізувати загальне число тестів.  Визначення тестів:  Кожному класу еквівалентності присвоюється унікальний номер.  Якщо ще залишилися не включені в тести правильні класи*,* то пишуться тести*,* які покривають максимально можливу кількість класів.  Якщо залишилися не включені в тести неправильні класи*,* то пишуть тести*,* які покривають тільки один клас [13].  2.2.2. Тестування через аналіз граничних значень  Граничні умови - це ситуації*,* що виникають на вищих і нижніх межах вхідних класів еквівалентності.  Аналіз граничних значень відрізняється від еквівалентного роздроблення наступним:  Вибір будь-якого елемента в класі еквівалентності в якості | |
|  | *37* |

|  |  |
| --- | --- |
| представницького здійснюється таким чином*,* щоб перевірити тестом кожний кордон цього класу.  При розробці тестів розглядаються не тільки вхідні значення (простір входів)*,* але і вихідні (простір виходів).  Метод вимагає певної міри творчості та спеціалізації в розглянутій задачі. Існує декілька правил:  Побудувати тести з неправильними вхідними даними для ситуації незначного виходу за межі області значень. Якщо вхідні значення повинні буті в інтервалі [-1.0. +1.0]*,* Перевіряємо - 1.0*,* 1.0*,* - 1.000001*,* 1.000001.  Обов'язково писати тести для мінімальної і максимальної межі діапазону.  Використовувати перші два правила для кожного з вхідних значень (використовувати пункт 2 для Всіх вихідних значень).  Якщо вхід і вихід програми представляє впорядкована множина*,* зосередити увагу на першому і останньому елементі списку.  Аналіз граничних значень*,* якщо він застосований правильно*,* дозволяє виявити велику кількість помилок. Однак визначення цих кордонів для кожної задачі може бути окремим важким завданням. Також метод не перевіряє комбінації вхідних значень.  2.2.3. Тестування через аналіз причинно-наслідкових зв'язків  У специфікації визначаються множини причин і наслідків. Під причиною розуміється окрема вхідна умова або клас еквівалентності. Слідство являє собою вихідну умову або перетворення системи. Тут кожній причині і слідству присвоюється номер.  На основі аналізу семантичного (смислового) змісту специфікації будується таблиця істинності*,* в якій послідовно перебираються всі можливі комбінації причин і визначаються слідства для кожної комбінації причин.  Таблиця забезпечується примітками*,* які задають обмеження і описують комбінації*,* які неможливі. Недоліком цього підходу є погане дослідження граничних умов. | |
|  | *38* |

|  |  |
| --- | --- |
| * + 1. Тестування через припущення про помилку   Тестувальник з великим досвідом вишукує помилки без всяких методів*,* але при цьому він підсвідомо використовує метод припущення про помилку. Даний метод у значній мірі ґрунтується на інтуїції. Основна ідея методу полягає в тому*,* щоб скласти список*,* який перераховує можливі помилки і ситуації*,* в яких ці помилки могли проявитися. Потім на основі списку складаються тести.   * + 1. Системне тестування   Системне тестування програмного забезпечення - це тестування програмного забезпечення (ПЗ)*,* що виконується на повній*,* інтегрованій системі*,* з метою перевірки відповідності системи вихідним вимогам. Системне тестування відноситься до методів тестування “чорної скриньки”*,* і*,* тим самим*,* не вимагає знань про внутрішню побудову системи.  Основним завданням системного тестування є перевірка як функціональних*,* так і не функціональних вимог до системи в цілому. При цьому виявляються дефекти*,* такі як невірне використання ресурсів системи*,* непередбачені комбінації даних користувацького рівня*,* несумісність з оточенням*,* непередбачені сценарії використання*,* відсутня або невірна функціональність*,* незручність використання і т.д. Для мінімізації ризиків*,* пов'язаних з особливостями поведінки системи в тій чи іншій середовищі*,* під час тестування рекомендується використовувати оточення максимально наближене до того*,* на яке буде встановлений продукт після видачі.  2.3. Побудова тестового набору  Побудова тестового набору передбачає віднаходження такої множини *R,* яка входить у *T* і забезпечує покриття всіх переходів модельного графу – формула  *co*(*R*) = *co*(*T*)*,*  де *T –* множина усіх тестів (шляхів графу); | |
|  | *39* |

|  |  |
| --- | --- |
| *R* – результуючий тестовий набір;  *co*(*R*) – покриття графу *G,* що досягається набором шляхів *R*;  *co*(*T*) *–* покриття графу *G,* що досягається набором шляхів *T,* такого що є розв’язком задачі оптимізації min*T*  min *ti*  *R* .  Тестовий набір будується з використанням жадібного алгоритму (рис. 2.1).  Жадібний алгоритм дозволяє отримати оптимальне рішення задачі шляхом здійснення ряду виборів.  У кожній точці прийняття рішення в алгоритмі робиться вибір*,* який в даний момент виглядає найкращим.  У записі алгоритму використовуються наступні умовні позначення:  | *S* | – потужність множини *S*;  *len*(*t*) – довжина тесту *t,* яка вимірюється у кількості тестових впливів;  {} – Порожня множина;  <> – Порожній упорядкований список.  Фіксуємо обмеження довжини тестів *M,* яке є параметром даного алгоритму. *C* – повний набір доступних транзакцій*, T* – повний набір раніше побудованих тестів.  Для кожного тесту *t* з *T* обчислюємо *co* (*t*)*,* як безліч тестових ситуацій з *C,*  що покриваються даним тестом. Якщо *co* (*t* )={}*,* то видаляємо *t* з *T*. *procedure* ПОБУДОВА\_ТЕСТОВОГО НАБОРУ  (*M*: максимальна довжина кроків тесту*,*  *P*: вага допустимого перевищення максимальної довжини на користь покриття );  *begin*  *t*1 *T* оголошуємо кращим  *for* кожного тесту *ti* з множини тестів *T*  *if len*(*t*1) ≤ *M* && *len*(*ti*) > *M,* то тест *t*1 кращий  *if len*(*t*1) ≤ *M* && *len*(*ti*) ≤ *M* && *co*(*t*1) > *co*(*ti*)*,* то тест *t*1 кращий  *if co*(*t*1) ≥ *co*(*ti*) && *len*(*t*1) < *len*(*ti*)*,* то тест *t*1 кращий  *if co*(*t*1) > *co*(*ti*) && *len*(*t*1) ≤ *len*(*ti*)*,* то тест *t*1 кращий | |
|  | *40* |

Початок

Отримання переліку вершин

Визначення

поточного індексу

Цикл опитування вершин

Присутні вихідні

ребра

ні

так

Вершина не

відвідувалась

ні

так

Відмітка про

відвідування

Відмітка про

вихідні ребра

ні

Остання

вершина

так

Кінець

Рис. 2.1. Схема алгоритму побудови всіх маршрутів на графі

*41*

|  |  |
| --- | --- |
| *if len*(*t*1)  *M* && *len*(*ti*)*,* а (*len*(*t*1) – *len*(*ti*)) *P* < (*co*(*t*1) – *co*(*ti*))*,* то тест *t*1 кращий. Додаємо знайдений кращий тест у набір *R*. Видаляємо із безлічі непокритих ситуації *C* всі елементи множини *co*(*t*кращий).  *end; end*;  Перше правило –гарантує*,* що в першу чергу всі можливі тестові ситуації будуть покриватися тестами з довжиною*,* що не перевищує максимальну.  *len*(*t*1) ≤ *M* && *len*(*ti*) > *M*.  Друге правило – формула вибирає тести з максимально можливим (в межах обмеження довжини) тестовим покриттям – у більшості випадків ця евристика дозволяє зменшити сумарну довжину тестів в наборі за рахунок зменшення їх кількості.  *len*(*t*1) ≤ *M* && *len*(*ti*) ≤ *M* && *co*(*t*1) > *co*(*ti*).  Трете і четверте правила – віддають перевагу тестам*,* що поліпшують покриття або довжину і не погіршує притому іншу з цих двох характеристик.  *if co*(*t*1) ≥ *co*(*ti*) && *len*(*t*1) < *len*(*ti*) *if co*(*t*1) > *co*(*ti*) && *len*(*t*1) ≤ *len*(*ti*)  Згідно з п*’*ятим правилом з двох тестів*,* що перевищують максимальну довжину*,* один краще іншого*,* якщо збільшення довжини компенсується (з урахуванням вагового параметра *P*) збільшенням покриття.  *len*(*t*1) > *M* && *len*(*ti*) > *M* && (*len*(*t*1) – *len*(*ti*)) *P* < (*co*(*t*1) – *co*(*ti*)).  Оскільки алгоритм евристичний (рис. 2.2)*,* і в ньому використовується перебір по неупорядкованій множині*,* в загальному випадку результати його роботи недетерміновані | |
|  | *42* |

Початок

Отримання всіх *M* і *Р*

Розрахунок

*C*, *T*, *R*

Для кожного тесту з

набору *Т*

L (кращ)<*M && L(*поточ*)<M*

ні

так

L (кращ)<*M &&*

*c(кращ)<c*(поточ)

ні

так

L (поточ)<*M &&*

*c(*кращ*)<c*(поточ)

ні

так

параметри кращого = параметри поточного

ні

Останній *М*

так

Кінець

Рис. 2.2. Схема алгоритму побудови тестового набору

*43*

|  |  |
| --- | --- |
| На різних запусках на одних і тих же даних можлива генерація різних тестових наборів*,* а проте в більшості випадків побудовані таким чином набори мають однакові метрики.  Допустимі значення: ціле позитивне число або + ∞. *P* задає вагу перевищення максимальної довжини тестів по відношенню до підвищення покриття. При великих значеннях параметра алгоритм в першу чергу вибирає з тестів*,* що перевищують максимальну довжину і додають хоча б якесь тестове покриття*,* самі короткі*,* що дозволяє зменшити максимальну довжину тестів*,* але може приводити до істотного збільшення сумарної складності тестового набору; при малих значеннях алгоритм в першу чергу вибирає тести*,* що покривають більшу кількість тестових ситуацій*,* що дозволяє мінімізувати сумарну складність тестового набору за рахунок додавання в нього дуже довгих тестів. Допустимі значення: невід’ємні числа (рис. 2.2).  У ході апробації алгоритму встановлено*,* що в більшості випадків оптимальне значення для параметра *M* трохи перевершує довжину найдовшого можливого в графі маршруту без циклів*,* що веде з початкового стану в кінцеве*,* а для параметра *P* лежить в діапазоні.  2.4. Висновки до розділу  Скінчений автомат є математичною абстракцією в теорії алгоритмів*,* що дозволяє описувати шляхи зміни стану об’єкта в залежності від його поточного стану і вхідних даних*,* за умови*,* що загальна можливу кількість станів скінчене. У процесі роботи скінченого автомату відбувається послідовна зміна скінченого числа його внутрішніх станів*,* причому стан автомата в певний момент часу однозначно визначається вхідним і вихідним сигналами. Такі автомати являють собою основу всієї сучасної обчислювальної техніки і всіляких дискретних систем автоматичного контролю і управління.  На даний час скінчені автомати – це один з основних інструментів побудови систем тестування. | |
|  | *44* |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| РОЗДІЛ 3  ОПИСАННЯ РОБОТИ СИСТЕМИ ВІДСЛІДКОВУВАННЯ ПОМИЛОК В ПРОГРАМНОМУ ЗАБЕЗПЕЧЕННІ *WEB*-ДОДАТКІВ  3.1. Логіка роботи додатку  Додаток повинен реалізувати наступну функціональність для користувача:   * для кожного з зареєстрованих користувачів вести облік проектів для тестування; * користувач повинен мати змогу будувати діаграму станів і переходів для продукту*,* що необхідно перевірити; * користувач запускає додаток*,* надаючи йому на вхід множину ребер графу переходів; * система повинна будувати усі шляхи графа (множину тестових випадків) та тестові набори*,* користуючись заданим критерієм покриття; * система повинна передбачати генерування випадкової послідовності взаємодій з додатком*,* що тестується; * користувач співставляє *id* станів їх описанню на природній мові*,* отримуючи таким чином тестові випадки для перевірки цільової системи.   Розроблюваний пакет має складатися з декількох модулів*,* що містять по кілька функціональних тестів. У підсумку пакет буде покривати весь базовий функціонал програмного забезпечення сайту.  На етапі моделювання системи проведемо аналіз основних видів її використання. На основі видів діяльності (рис. 3.1)*,* побудуємо послідовність подій (вхід до системи (рис. 3.2) та тестування *web*-додатків (рис. 3.3)). | | | | | | | | | |
| Кафедра КСУ | | | | НАУ ввввввв 000 ПЗ | | | | | |
| **Виконав** |  |  |  | Описання роботи системи відслідковування помилок в програмному забезпеченні *web*-додатків | **Літера** | | | **Аркуш** | **Аркушів** |
| **Керівник** |  |  |  | Д |  |  | *45* | *61* |
| **Консульт*.*** |  |  |  | СП 4….. 123 | | | | |
| **Норм*.* контр*.*** |  |  |  |
| **Зав*.* Каф*.*** |  |  |  |

|  |  |
| --- | --- |
| Введення імені і паролю  Перевірка наявності імені  Інакше  Реєстрація  Виведення списку поточних проектів  Генерація тестового набору  Запуск тесту на основі отриманого тестового  Підтвердження результатів тестування  Інакше  Формування звіту про тестування    Рис. 3.1. Діагарма основних видів діяльності при тестуванні програмного забезпечення в системі  Форма входу Модуль Інтерфейс Користувач в систему ідентифікації користувача   1. Введеня імені і паролю 2. Передача даних на 3. Повідомлення про помилку 4. Відкриття проектів та   історії дій користувача  Рис. 3.2. Діагарма послідовності дій для входу користувача в систему | |
|  | *46* |



|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | | | | | | |
| Користувач | | | | Меню користувача | | | Форма вибору Функція Модуль  проектів генерації тесту проведення |  |
|  | | | |  |  | |  | |
|  | | | 1. Вибір елементу ме | | ню "Проекти тестування" | | | |
|  | |  |  |  |  | 1. Виклик форми 2. Формування тестової послідовності 3. Виконання тестування 4. Вимога до   повторної генерації тестового набору  проходження тестування | | |
|  | |  |  |  |  |
|  | | |  | |  |
| 6. Звіт | | з |
|  | |  |  |  |  | | | |
|  | |  |  |
| Рис. 3.3. Діаграма послідовності дій для тестування програмного забезпечення  Усі модулі пакета є незалежними. Результат тестування*,* як і самі модулі знаходяться в єдиному файлі з розширенням *Trace*. Тести*,* що відповідають модулям пакета представлені у табл. 3.1.  Таблиця 3.1  Відповідність тестів модулям пакета | | | | | | | | |
|  | *Test*\_*Func* | | | | | Набор функцій*,* що використовуються в модулях пакета | |  |
| *Test\_App* | | | | | Запуск додатку*,* Меню Опцій*,* Меню Колекції*,* Головний  Екран | |
| *Test\_Collect* | | | | | Повний тест функціоналу меню Колекції | |
| *Test\_Diff* | | | | | Тест розбиття зображення по Важкості (Використання  кнопок) | |
| *Test\_Help* | | | | | Повний тест функціоналу Екрану Допомоги | |
| *Test\_Main* | | | | | Повний тест функціоналу Головного Екрану | |
| *Test\_Picture*\_*Contr* | | | | | Тестування переключення зображень кнопками  Наступна*,* Попередня | |
| *Test\_Susp* | | | | | Додатковий тест для визначення поведінки сайту в  момент згортання | |
| *Test\_Orient* | | | | | Додатковий тест для визначення поведінки сайту в момент зміни положення пристрою (для мобільних  браузрів) | |
|  |  | | | | | | |  |
|  | | | | | | | | *47* |



|  |  |
| --- | --- |
| * 1. Структура програмного додатку      1. Класи програмного додатку   Клас *graph\_test\_coverage* являє собою абстракцію покриття модельного графу для формування дерева варіантів . Він містить контейнер з ребрами та відповідним кожному ребру булевою змінною*,* що містить інформацію про те*,* чи покрите ребро чи ні. Об’єкт класу *graph\_coverage* використовується під час побудови тестового набору.  *#include < graph\_test\_coverage.h>*  Реалізовані наступні методи класу:  *graph\_test\_coverage <Graph>(const Graph & graph)*  Конструктор класу*,* створює об’єкт – абстракцію покриття графу і ініціалізує значення покриття кожного ребра графу *graph* як *FALSE* (непокрите).  *aditional\_coverage\_for\_path(std::list<size\_t> path) const*  Функція повертає додатне число типу *size*\_*t,* якщо шлях *path*покриває хоча б одне досі непокрите ребро*,* і 0 – у протилежному випадку.  *fully\_covered() const*  Функція повертає булеве значення. *TRUE,* якщо кожне ребро модельного графу покрито шляхами(тестами)*,* що увійшли до результуючого набору*,* і *FALSE*  – у протилежному випадку.  *add\_coverage\_for\_path(const std::list<size\_t>& path)*  Функція помічає всі вершини*,* що зустрічаються у шляху *path* як покриті.  Об’єкт класу *test\_edge*\_*id* містить унікальне описання ребра графу*,* яким слугують його початого і цільова вершини.  *#include " test\_edge.h"*  *test\_edge\_id(size\_t vertex\_source, size\_t vertex\_target)*  Конструктор класу*,* ініціалізує *m\_vertex\_source* та *m\_vertex\_target*  значеннями переданих параметрів.  Клас *Graph* є синонімом класу *boost::adjacency\_list* і використовується для представлення модельного графу. | |
|  | *48* |

|  |  |
| --- | --- |
| *#include <boost/graph/adjacency\_list.hpp>*  Наступні методи із тих*,* що пропонує клас *boost::adjacency\_list* було використано під час реалізації додатку:  *out\_edges(vertex\_descriptor u, const adjacency\_list& g)*  Повертає пару літераторів*,* що ітерує діапазон ребер*,* що виходять вершини  *U* графу *G*.  *target(edge\_descriptor e, const adjacency\_list& g)*  Функція повертає значення типу *vertex\_descriptor,* що є значенням кінцевої вершини ребра *e* графу *g.*  *num\_vertices(const adjacency\_list& g)*  Функція повертає кількість вершин графу *g. vertex(vertices\_size\_type n, const adjacency\_list& g)* Функція повертає *n-*у вершину графу *g*.  3.2.2. Методи  Основні функціъ*,* що використовують при розв’язанні задачі:  *read\_values\_in\_set<T>(std::istream & input, size\_t values\_count, std::set<T>\* result)*  Функція*,* що читає із вхідного потоку *input, values\_count* кількість ребер*,* що представлені парами вершин і зберігає їх у контейнер *result.*  *output\_test(const std::list<size\_t>& path\_accumulator, std::ostream & output)*  Функція*,* що виводить у вихідний потік *output* шлях на графі(тест)*,* що міститься у *path\_accumulator.*  *save\_to\_collection(const std::list<size\_t>& path, std::list<std::list<size\_t>>& pathes\_collection)*  Функція поміщає шлях *path* у *pathes\_collection* – фінальний набір тестів.  Функція*,* що реалізує алгоритм побудови всіх маршрутів на графі із заданим обмеженням (відвідування однієї вершини не більше двох разів). Результат поміщається у *pathes\_collection.* Функція рекурсивна.  Параметрами виступають модельний граф *graph,* поточна вершина  *current\_vertex,* множина кінцевих вершин графу *final\_vertices,* контейнер для | |
|  | *49* |

|  |  |
| --- | --- |
| шляху*,* що будується *path\_accumulator, visits\_count\_vector –* вектор*,* в якому міститься інформація про кількість відвідування поточної вершини*, pathes\_collection –* тестовий набір*,* контейнер*,* в який буде поміщено *path\_accumulator,* коли буде досягнуто одну з вершин множини *final\_vertices.*  *read\_case(Graph & states\_graph, std::set<size\_t>& final\_states)*  Функція*,* що відповідає за отримання усіх вхідних даних для генерації тестового набору*,* а саме – значення кількості вершин модельного графу*,* значення кількості кінцевих вершин*,* множина кінцевих вершин і множина ребер графу.  *generate\_all\_tests(const Graph & states\_graph, const std::set<size\_t>& final\_states, std::list<std::list<size\_t>>& tests\_collection)*  Функція*,* яка через виклик функції *trace\_graph(),* будує усі шляхи на графі.  *choose\_best\_tests(const size\_t M, const double P, const Graph& states\_graph, const size\_t states\_count, std::list<std::list<size\_t> >& tests\_collection, std::list<std::list<size\_t> >& best\_tests)*  Функція*,* що реалізує алгоритм побудови тестового набору. Вхідні параметри: *M –* максимальна довжина переходів у тесті*, P* – вага допустимого перевищення довжини тесту на користь покриття*, states\_graph –* модельний граф*, states\_count –* кількість станів модельного графу *states\_graph, tests\_collection –* повний набір тестів*,* що побудовано у результаті роботи функції *generate\_all\_tests(),best\_tests –* контейнер*,* що містить результуючу тестову послідовність.   * 1. Інтерфейс додатку та параметри запуску      1. Модуль тестування сценаріїв   Розробка тестів*,* заснованих на використанні сценаріїв*,* здійснюється за такою методикою:  – визначається модель використання*,* що включає операційне оточення продукту і "акторів". Актором може бути користувач*,* інший продукт*,* апаратна | |
|  | *50* |

|  |  |
| --- | --- |
| частина тощо*,* тобто все*,* з чим продукт обмінюється інформацією. Поділ на оточення і акторів умовно і служить для опису оптимальних способів використання продукту;   * розробляються сценарії використання продукту. Опис сценарію в залежності від продукту і обраного підходу може бути строго певним*,* параметризованим або дозволяти певний рівень невизначеності; * розробляється набір тестів*,* що покривають задані сценарії. З урахуванням ступеня невизначеності*,* закладеної в сценарії*,* кожен тест може покривати один сценарій*,* кілька сценаріїв*,* або*,* навпаки*,* частина сценарію.   Використання сценаріїв не вимагає наявності повної формальної специфікації вимог*,* але зате може зажадати більше часу на розробку і аналіз.  Для кожного проекту можна обрати окремий сценарій тестування (рис.  3.4)*,* що дозволяє уніфікувати систему.  Рис. 3.4. Вікно з переліком активних проектів  Ще одна особливість тестування сценаріїв полягає в тому*,* що цей метод направляє тестування на перевірку конкретних режимів використання продукту*,* що дозволяє знаходити дефекти*,* які метод тестування за вимогами може пропустити.  3.3.2. Модуль генераціъ тестів  Тест формалізований у наступному вигляді (рис. 3.5). Тут кожна стрілка з позначкою *DoTransaction, DoAddressTenure* або *DoDataTenure* представлет собою | |
|  | *51* |

|  |  |
| --- | --- |
| виклик відповідної функції продукту з передачею параметрів. Стрілка *checkTr* відповідає перевірці проходження по шині транзакції з відповідними параметрами. Кожна зі стрілок діаграми генератором тестів перетворюється в здійсненний код*,* при цьому стрільцям*,* які представляють собою виклики функцій може відповідати досить простий і невелику ділянку коду*,* що викликає відповідну функцію і перевіряючий її вихідне значення на наявність помилок.  Рис. 3.5. Формальний запис сценарного тесту  Слід зазначити*,* що стрілки*,* відповідні перевірки транзакцій*,* можуть після генерації перетворитися в досить складний код*,* який буде виконувати очікування появи транзакції на шині протягом заданого при генерації часу - тайм-ауту*,* перевіряти фази транзакції і звіряти обчислені значення параметрів із заданими еталонними значеннями.  В результаті в розглянутому прикладі виграш від застосування генераційні підходу досягається в основному за рахунок використання наочного візуального представлення тестів*,* що може бути нівельовано витратами на створення генераційні сценарію на *MSC*.  Можлива куди більш ефективна формалізація сценарію для генерації тестів (рис. 3.6)*,* яка представляє інший спосіб формалізації тесту*,* що виконує ті ж самі перевірки.  На рис. 3.6 перевірки транзакцій згруповані з породжують їх викликами в | |
|  | *52* |

|  |  |
| --- | --- |
| окремі фрагменти*,* а паралелізм*,* який використовується при виконанні фрагментів*,* заданий через *Par* - формальну конструкцію*,* яка застосовується для зображення паралелізму.  Рис. 3.6. Формальний запис сценарного тесту з використанням паралелізму При генерації тестів по діаграмі рис. 3.6 тестовий генератор перебирає всі  можливі і не повторюються варіанти виклику тестованих функцій*,* зберігаючи при цьому коректність порядку перевірок*,* що в даному прикладі дає три згенерованих тесту. Нескладно бачити*,* що витрати на створення діаграми.  Для успішного розвитку технологій тестування на основі моделей потрібно одночасно підтримувати їх еволюцію в трьох напрямках: нарощувати функціональність технологій*,* нарощувати спряженість цих технологій з сучасними процесами розробки і нарощувати зручність їх використання. Особливо важливо збільшувати зручність використання технологій для вирішення найбільш часто зустрічаються завдань.  3.3.3. Описання інтерфейсу системи  Наша Система дозволяє:  – зберігати інформацію про помилку в наступному виді: хто повідомив про проблему*,* дата і час*,* коли була виявлена проблема; серйозність проблеми; опис | |
|  | *53* |

|  |  |
| --- | --- |
| неправильної поведінки програми; стан помилки (рис. 3.7).  Рис. 3.7. Вікно з описанням помилки в роботі програмного забезпечення   * дозволяє менеджеру проектів прийняти помилку для подальшого усунення та відіслати программісту для її усунення; * надає можливість відсортувати баг за серйозністью*, id,* тегами (рис. 3.8)*,* а також провести пошук конкретної помилки (рис. 3.9); * за всіма тестами можна сформувати звітність та передивитись статистику (рис. 3.10).   Рис. 3.8. Вікно з списком знайдених помилок в роботі програмного забезпечення | |
|  | *54* |

|  |  |
| --- | --- |
| Рис. 3.6. Вікно знайдених помилок з інструментарієм пошуку  Рис. 3.10. Вікно історії проведення тестів  3.4. Висновки до розділу  Побудовано *web*-додаток*,* що реалізує підхід до генерації тестів і проведені експерименти з його використанням.  Заявлений підхід налічує наступні переваги:   * гнучкість; * фіксація максимальної кількості кроків тесту; | |
|  | *55* |

* недетермінованість;
* мінімізація розміру тестового набору;
* гарантованість досягнення встановленого рівні покриття;
* статичне тестування вимог.

*56*

|  |  |
| --- | --- |
| ВИСНОВКИ  Під час написання даної роботи було розроблено алгоритм для генерації тестового набору для ручного виконання і побудовано додаток*,* що реалізує наведений підхід. Алгоритм демонструє застосування теоретичних відомостей про скінчені автомати і графи для побудови технології*,* що може асистувати відділу тестування у найкритичнішій його активності – виборі скінченого числа перевірок серед їх безкінечної множини*,* які з найбільшою ймовірністю викриють помилки у програмному забезпечені.  Робота проводилась поетапно*,* у її процесі було детально розглянуто наступні проблеми і вирішено наступні задачі:   1. Розглянуто процес тестування і його складові. Виявлені проблема наявності помилок у існуючих системах*,* що є складними і часто асистують людству у критичних завданнях; необхідність тестування з метою віднаходження та попередження помилок під час використання; та конфлікт безкінечних можливостей для перевірок і скінчених ресурсів – людських та часу; і як результат*,* необхідність залучення інструментальної підтримки процесу тестування. 2. Залучення автоматизації до виконання і генерації тестів. Недоліки і переваги обох підходів та обґрунтування вибору автоматизації створення тестів як задачі дослідження. 3. Існуючи способи формального представлення вимог до програмного продукту – логіко-алгебраїчні та операційні моделі. Обґрунтування вибору на користь скінчено – автоматної моделі для описання еталону поведінки програмного забезпечення. 4. Розглянуто варіанти представлення скінченого автомату і обґрунтовано вибір на користь його візуального представлення діаграмою станів і переходів*,* що дає можливість залучення до розв’язання глобальної задачі дослідження такого потужного інструменту як теорія графів. 5. Розробка і обґрунтування алгоритмів побудови тестів і тестового | |
|  | *57* |

|  |  |
| --- | --- |
| набору з використанням рекурсивного і жадібного алгоритмів.  6. Побудовано додаток*,* що реалізує підхід до генерації тестів і проведені експерименти з його використання.  Заявлений підхід налічує наступні переваги:   1. Гнучкість. Підхід може бути використаний для побудови тестового набору для продуктів з різних предметних областей*,* побудованих з використанням різних архітектур і технологій. 2. Фіксація максимальної кількості кроків тесту. У тестовий набір буде залучено тести перевищуючі максимальну встановлену довжину*,* тільки якщо існують переходи*,* недосяжні з початкової вершини за максимальну кількість кроків. Таким чином забезпечується зниження ймовірності допущення помилки під час виконання тестів*,* що є актуальною проблемою під час виконання довгих сценаріїв. 3. Недетермінованість. Евристичний алгоритм відбору тестів до тестового набору може обирати різні тести із встановленою метрикою тестового покриття*,* таким чином мінімізуючи ризик настання «парадоксу пестициду». 4. Мінімізація розміру тестового набору. Тестовий набір включає кращі тести для досягнення встановленого покриття*,* замість усіх можливих тестів. 5. Гарантованість досягнення встановленого рівні покриття. Для складних систем*,* що представлені графом із сотнею вершин і більше*,* гарантувати досягнення покриття усіх переходів створюючи тестовий набір у ручний спосіб складно і існує ймовірність помилки*,* на відміну від автоматичного способу. 6. Статичне тестування вимог. Під час перетворення представлення вимог на природній мові у скінчено-автоматну модель відбувається їх статичне тестування: виявляються їх недоліки*,* неточності і протиріччя.   Серед недоліків можна виділити наступні:   1. Обмеження на модель. Тестовий набір може бути побудований з використанням даного підходу лише для програмного забезпечення*,* еталонна модель якого може бути представлена у вигляді графу станів і переходів скінченого автомату. 2. Неповна автоматизація. Перетворення моделі графу – автомату у список | |
|  | *58* |

|  |  |
| --- | --- |
| ребер і співставлення унікальних номерів ребер їх семантичному значенню не автоматизовано на даному етапі.  Додаток може бути успішно застосованим вже на поточному етапі у тестуванні програмних продуктів. Перспективним етапом дослідження є пошук такої модифікації алгоритму*,* за яким тести обираються до набору*,* який гарантував би покриття лише заданих переходів. Такий напрямок дослідження дасть змогу автоматично будувати набори для регресійного тестування*,* коли перевірці підлягатимуть лише переходи*,* які могли зазнати негативного впливу в результаті внесення змін до програмного продукту. Описаний підхід може бути вбудований у середовище підтримки неперервної інтеграцій*,* тоді відділ тестування буде кожного разу після внесення змін отримувати не лише нову версію продукту*,* а й набір перевірок*,* що треба провести. | |
|  | *59* |

|  |  |
| --- | --- |
| СПИСОК БІБЛІОГРАФІЧНИХ ПОСИЛАНЬ ВИКОРИСТАНИХ ДЖЕРЕЛ   1. *IP VPNs for Service Providers*: *The Foundation for Profitable* [*http*://*www*.*cisco*.*com*/.](http://www.cisco.com/) 2. *Beazley D*. *Python essential reference* // *BHV*. – 2015. – 734 *c*. 3. *Jones C*.*A*.*, Drake F*.*L*. *Python* & *XML* // *BHV*. – 2014. – 807 *c*. 4. ДСТУ 2851-94 Програмні засоби ЕОМ. Документування результатів випробувань. 5. НД ТЗІ 1.1-003-99. Термінологія у області захисту інформації в комп'ютерних системах від несанкціонованого доступу. // Департамент спеціальних телекомунікаційних систем і захисту інформації Служби безпеки України. – Київ*,* 1999. 6. Гамма Э.*,* Холм Р.*,* Джонсон Р.*,* Влиссидес Дж. Приемы объектно- ориентированного проектирования. Паттерны проектирования. – СПб: Питер*,* 2006. – 366 с. 7. Доусон М. Программируем на *Python*. – СПб.: Питер*,* 2014. – 416 с. 8. Лутц М. Изучаем *Python,* 4-е издание. – Пер. с англ. – СПб.: Символ- Плюс*,* 2011. – 1280 с. 9. Лутц М. Программирование на *Python,* том *I,* 4-е издание. – Пер. с англ.    * СПб.: Символ-Плюс*,* 2011. – 992 с. 10. Лутц М. Программирование на *Python,* том *II,* 4-е издание. – Пер. с англ.     * СПб.: Символ-Плюс*,* 2011. – 992 с. 11. Прохоренок Н.А. *Python* 3 и *PyQt*. Разработка приложений. – СПб.: БХВ- Петербург*,* 2012. – 704 с. 12. Прохоренок Н.А. Самое необходимое. — СПб.: БХВ-Петербург*,* 2011. — 416 с. 13. Фаулер М.*,* Скотт К. *UML*. Основы. – Пер. с англ. – СПб.: Символ- Плюс*,* 2002. | |
|  | *60* |

|  |  |
| --- | --- |
| 1. Фаулер*,* Мартин – Архитектура корпоративных программных приложений.: Пер. с англ. – М.: Издательский дом "Вильяме"*,* 2006. – 544 с. 2. Хахаев И.А. Практикум по алгоритмизации и программированию на   *Python*. – М.: Альт Линукс*,* 2010. — 126 с.   1. Шаньгин В. Ф. Защита компьютерной информации*,* М: ДМК. 2008 – 544с. 2. Бойченко С.В.*,* Іванченко О.В. Положення про дипломні роботи (проекти) випускників Національного авіаційного університету. – К.: НАУ*,* 2017. – 63 с. | |
|  | *61* |

|  |  |
| --- | --- |
| Додаток А  Програмний код розробленого пакета  *Collections*\_*Test*. *js*  #*import* "*Join*\_*It*\_*Functions*. *js*"  *var target* = *UIATarget*. *localTarget* (); *UIALogger*. *logStart* ("*Join It Temp Script*"); *target*. *delay* (1);  *target*. *frontMostApp* (). *mainWindow* (). *elements* () [0]. *tap* (); // *Cancel video target*. *delay* (2);  *target*. *frontMostApp* (). *mainWindow* (). *buttons* () ["*Collections*"]. *tap* ();  *target*. *delay* (1);  *target*. *frontMostApp* (). *mainWindow* (). *buttons* () ["*Add Pictures*"]. *tap* ();  *target*. *delay* (1);  *target*. *frontMostApp* (). *mainWindow* (). *popover* (). *tableViews* () [0]. *cells* () [*Random* (8)]. *tap* ();  *target*. *delay* (5);  *for* (*i*=0; *i*<3; *i*++)  {  *target*. *frontMostApp* (). *mainWindow* (). *popover* (). *scrollViews* () [0]. *elements*  () [*i*]. *tap* ();  *target*. *delay* (2);}  *target*. *delay* (5);  *target*. *frontMostApp* (). *mainWindow* (). *popover* (). *navigationBar* (). *buttons* () ["*Back*"]. *tap* ();  *target*. *delay* (1);  *target*. *frontMostApp* (). *mainWindow* (). *popover* (). *tableViews* () [0]. *cells* () ["*My Photos*"]. *tap* ();  *target*. *delay* (1); | |
|  | *62* |

|  |  |
| --- | --- |
| *target*. *frontMostApp* (). *mainWindow* (). *popover* (). *tableViews* () ["*Empty list*"]. *cells* () [0]. *tap* ();  *target*. *delay* (1);  *target*. *frontMostApp* (). *mainWindow* (). *popover* (). *tableViews* () ["*Empty list*"]. *cells* () [20]. *images* () [0]. *tap* ();  *for* (*j*=0; *j*<10; *j*++)  {  *target*. *frontMostApp* (). *mainWindow* (). *tableViews* () ["*Empty list*"]. *cells* () [*j*].  *tap* ();  *target*. *delay* (1);  }  *target*. *frontMostApp* (). *mainWindow* (). *tableViews* () ["*Empty list*"]. *cells* () ["*Join It* - *Original Collection*"]. *tap* ();  *target*. *delay* (1);  *target*. *frontMostApp* (). *mainWindow* (). *scrollViews* () [0]. *elements* () [0]. *tap*  ();  *target*. *delay* (1);  *target*. *frontMostApp* (). *mainWindow* (). *tableViews* () ["*Empty list*"]. *cells* () ["*All Pictures*"]. *tap* ();  *target*. *delay* (1);  *target*. *frontMostApp* (). *mainWindow* (). *buttons* () ["*Edit*"]. *tap* ();  *target*. *delay* (1);  *for* (*k*=0; *k*<4; *k*++)  {  *target*. *frontMostApp* (). *mainWindow* (). *scrollViews* () [0]. *elements* () [*Random* (16)]. *tap* ();  *target*. *delay* (1);  }  *target*. *frontMostApp* (). *mainWindow* (). *buttons* () ["*Delete*"]. *tap* ();  *target*. *delay* (1); | |
|  | *63* |

|  |  |
| --- | --- |
| *target*. *frontMostApp* (). *mainWindow* (). *popover* (). *buttons* () ["*Delete Selected Pictures*"]. *tap* ();  *target*. *delay* (1);  *target*. *frontMostApp* (). *mainWindow* (). *buttons* () ["*Back*"]. *tap* ();  *target*. *delay* (1);  *UIALogger*. *logPass* ("*Done*");  *Smoke*\_*Test*. *js*  #*import* "*Join*\_*It*\_*Functions*. *js*"  *var target* = *UIATarget*. *localTarget* (); *UIALogger*. *logStart* ("*Join It Temp Script*"); *target*. *delay* (1);  *target*. *frontMostApp* (). *mainWindow* (). *elements* () [0]. *tap* (); // *Cancel video target*. *delay* (1);  *First*\_*Launch*\_*Check* (); *Choose*\_*Random*\_*Picture* (); *Change*\_*Difficulty* ("*Easy*"); *target*. *delay* (1); *Change*\_*Difficulty* ("*Medium*"); *target*. *delay* (1); *Change*\_*Difficulty* ("*Hard*"); *target*. *delay* (1); *Change*\_*Difficulty* ("*Insane*"); *target*. *delay* (1);  *Change*\_*Difficulty*\_*Drag* (*Random* (1));  *target*. *delay* (1); *Change*\_*Difficulty* ("*Easy*"); *target*. *delay* (1); *Open*\_*Options* (); *Tap*\_*Sticking* (); *Tap*\_*Rotation* (); *Tap*\_*Timer* (); | |
|  | *64* |

|  |  |
| --- | --- |
| *Tap*\_*Sound* (); *Tap*\_*Iphone* (); *Tap*\_*Ipad* (); *Tap*\_*Piece*\_*Style* (1);  *Tap*\_*Piece*\_*Style* (2);  *Tap*\_*Piece*\_*Style* (3);  *Tap*\_*Piece*\_*Style* (4); *Tap*\_*Piece*\_*Style* (); *Choose*\_*Background* (*Random* (8)); *Close*\_*Options* ();  *Start*\_*Game* (5);  *Game* ();  *UIALogger*. *logPass* ("*Done*");  *target*. *frontMostApp* (). *mainWindow* (). *images* () [3]. *doubleTap* ();  *Suspend*\_*Test*. *js*  #*import* "*Join*\_*It*\_*Functions*. *js*"  *var target* = *UIATarget*. *localTarget* (); *UIALogger*. *logStart* ("*Join It Temp Script*"); *target*. *delay* (1);  *target*. *frontMostApp* (). *mainWindow* (). *elements* () [0]. *tap* (); // *Cancel video target*. *delay* (2);  // *First*\_*Launch*\_*Check* ();  *Deactivate* (3);  *target*. *delay* (1);  *target*. *frontMostApp* (). *mainWindow* (). *buttons* () ["*Collections*"]. *tap* ();  *target*. *delay* (1);  *Deactivate* (3);  *target*. *delay* (1);  *target*. *frontMostApp* (). *mainWindow* (). *scrollViews* () [0]. *elements* () [0]. *tap*  ();  *target*. *delay* (1); | |
|  | *65* |

|  |  |
| --- | --- |
| *Deactivate* (3);  *target*. *delay* (1); *Open*\_*Options* (); *target*. *delay* (1);  *Deactivate* (3);  *target*. *delay* (1); *Close*\_*Options* (); *target*. *delay* (1);  *Deactivate* (3);  *target*. *delay* (1);  *Start*\_*Game* (5);  *target*. *delay* (1);  *Deactivate* (3);  *target*. *delay* (1);  *target*. *frontMostApp* (). *mainWindow* (). *staticTexts* () ["*Resume*"]. *tap* ();  *target*. *delay* (1); *Exit*\_*Game* (); *target*. *delay* (1);  *Deactivate* (3);  *target*. *delay* (1);  *UIALogger*. *logPass* ("*Done*");  *Main*\_*Screen*\_*Test*. *js*  #*import* "*Join*\_*It*\_*Functions*. *js*"  *var target* = *UIATarget*. *localTarget* (); *UIALogger*. *logStart* ("*Join It Temp Script*"); *target*. *delay* (1);  *target*. *frontMostApp* (). *mainWindow* (). *elements* () [0]. *tap* (); // *Cancel video target*. *delay* (2);  *target*. *frontMostApp* (). *mainWindow* (). *buttons* () ["*HelpToolbar*"]. *tap* ();  *target*. *delay* (1);  *target*. *frontMostApp* (). *mainWindow* (). *buttons* () ["*BackToolbar*"]. *tap* (); | |
|  | *66* |

|  |  |
| --- | --- |
| *target*. *delay* (1); *Change*\_*Difficulty* ("*Easy*"); *target*. *delay* (2); *Change*\_*Difficulty* ("*Medium*"); *target*. *delay* (2); *Change*\_*Difficulty* ("*Hard*"); *target*. *delay* (2); *Change*\_*Difficulty* ("*Insane*"); *target*. *delay* (2);  *target*. *frontMostApp* (). *mainWindow* (). *elements* () [7]. *tap* (); // *Enter Hi Score target*. *delay* (1);  *target*. *frontMostApp* (). *mainWindow* (). *elements* () [8]. *tap* (); // *Exit Hi Score target*. *delay* (1);  *Change*\_*Difficulty*\_*Drag* (0);  *target*. *delay* (1);  *Change*\_*Difficulty*\_*Drag* (0.5);  *target*. *delay* (1);  *if* (*target*. *frontMostApp* (). *mainWindow* (). *buttons* () [2]. *isVisible* () ==*true*)  {  *target*. *frontMostApp* (). *mainWindow* (). *buttons* () [2]. *tap* ();  *target*. *delay* (1);  }  *Open*\_*Options* ();  *Close*\_*Options* ();  *target*. *frontMostApp* (). *mainWindow* (). *images* () [2]. *buttons* () ["*Info*"]. *tap* ();  *target*. *delay* (4);  *target*. *frontMostApp* (). *mainWindow* (). *images* () [2]. *buttons* () ["*saveButton*"].  *tap* ();  *target*. *frontMostApp* (). *mainWindow* (). *images* () [2]. *buttons* () ["*zoomButton*"]. *tap* ();  *target*. *delay* (1); | |
|  | *67* |

|  |  |
| --- | --- |
| *target*. *frontMostApp* (). *mainWindow* (). *images* () [2]. *tap* ();  *target*. *delay* (1);  *target*. *frontMostApp* (). *mainWindow* (). *buttons* () ["*Next*"]. *tap* ();  *target*. *delay* (2);  *target*. *frontMostApp* (). *mainWindow* (). *buttons* () ["*Previous*"]. *tap* ();  *target*. *delay* (2);  *target*. *flickFromTo* ({*x*: 360*, y*: 200}*,* {*x*: 160*, y*: 200}); // *Flick Picture Change*  (*NEXT*)  *target*. *delay* (2);  *target*. *flickFromTo* ({*x*: 160*, y*: 200}*,* {*x*: 260*, y*: 200}); // *Flick Picture Change*  (*PREVIOUS*)  *target*. *delay* (2);  *target*. *frontMostApp* (). *mainWindow* (). *buttons* () ["*Collections*"]. *tap* ();  *target*. *delay* (1);  *target*. *frontMostApp* (). *mainWindow* (). *buttons* () ["*Back*"]. *tap* ();  *target*. *delay* (1);  *target*. *frontMostApp* (). *mainWindow* (). *images* () [2]. *buttons* () ["*Go*"]. *tap* ();  *UIALogger*. *logPass* ("*Done*");  *Difficulty*\_*Test*. *js*  #*import* "*Join*\_*It*\_*Functions*. *js*"  *var target* = *UIATarget*. *localTarget* (); *UIALogger*. *logStart* ("*Join It Temp Script*"); *target*. *delay* (1);  *target*. *frontMostApp* (). *mainWindow* (). *elements* () [0]. *tap* (); // *Cancel video target*. *delay* (1);  *Change*\_*Difficulty* ("*Easy*");  *Start*\_*Game* (5);  *target*. *captureScreenWithName* ("*Easy Game*"); *Exit*\_*Game* ();  *Change*\_*Difficulty* ("*Medium*");  *Start*\_*Game* (10); | |
|  | *68* |

|  |  |
| --- | --- |
| *target*. *captureScreenWithName* ("*Medium Game*"); *Exit*\_*Game* ();  *Change*\_*Difficulty* ("*Hard*");  *Start*\_*Game* (15);  *target*. *captureScreenWithName* ("*Hard Game*"); *Exit*\_*Game* ();  *Change*\_*Difficulty* ("*Insane*");  *Start*\_*Game* (20);  *target*. *captureScreenWithName* ("*Insane Game*"); *Exit*\_*Game* ();  *UIALogger*. *logPass* ("*Done*");  *Help*\_*Test*. *js*  #*import* "*Join*\_*It*\_*Functions*. *js*"  *var target* = *UIATarget*. *localTarget* (); *UIALogger*. *logStart* ("*Join It Temp Script*"); *target*. *delay* (1);  *target*. *frontMostApp* (). *mainWindow* (). *elements* () [0]. *tap* (); // *Cancel video target*. *delay* (2);  *target*. *frontMostApp* (). *mainWindow* (). *buttons* () ["*HelpToolbar*"]. *tap* ();  *target*. *delay* (1);  *target*. *captureScreenWithName* ("*Help*\_*Screenshot*\_*iPad* 1");  *target*. *delay* (1);  *for* (*i*=2; *i*<5; *i*++)  {  *target*. *frontMostApp* (). *mainWindow* (). *scrollViews* () [0]. *scrollViews* () [0].  *webViews* () [0]. *scrollDown* ();  *target*. *delay* (1);  *target*. *captureScreenWithName* ("*Help*\_*Screenshot*\_*iPad* " +*i*);  *target*. *delay* (1);  }  *target*. *delay* (2); | |
|  | *69* |

|  |  |
| --- | --- |
| *target*. *frontMostApp* (). *mainWindow* (). *scrollViews* () [0]. *scrollViews* () [0].  *webViews* () [0]. *staticTexts* () ["*iPhone Controls*"]. *tap* ();  *target*. *delay* (1);  *target*. *captureScreenWithName* ("*Help*\_*Screenshot*\_*iPhone* 1");  *target*. *delay* (1);  *for* (*i*=2; *i*<5; *i*++)  {  *target*. *frontMostApp* (). *mainWindow* (). *scrollViews* () [0]. *scrollViews* () [0].  *webViews* () [0]. *scrollDown* ();  *target*. *delay* (1);  *target*. *captureScreenWithName* ("*Help*\_*Screenshot*\_*iPhone* " +*i*);  *target*. *delay* (1);  }  *target*. *delay* (2);  *target*. *frontMostApp* (). *mainWindow* (). *scrollViews* () [0]. *scrollViews* () [0].  *webViews* () [0]. *links* () ["\*nJoin It website*"]. *staticTexts* () ["*Join It website*"]. *tap* ();  *target*. *delay* (1);  *target*. *frontMostApp* (). *mainWindow* (). *buttons* () ["*BackToolbar*"]. *tap* ();  *UIALogger*. *logPass* ("*Done*"); *Picture*\_*Change*\_*Test*. *js* #*import* "*Join*\_*It*\_*Functions*. *js*"  *var target* = *UIATarget*. *localTarget* (); *UIALogger*. *logStart* ("*Join It Temp Script*"); *target*. *delay* (1);  *target*. *frontMostApp* (). *mainWindow* (). *elements* () [0]. *tap* (); // *Cancel video target*. *delay* (1);  *target*. *frontMostApp* (). *mainWindow* (). *buttons* () ["*Collections*"]. *tap* ();  *target*. *delay* (1);  *target*. *frontMostApp* (). *mainWindow* (). *scrollViews* () [0]. *elements* () [0]. *tap*  ();  *target*. *delay* (2); | |
|  | *70* |

|  |  |
| --- | --- |
| *for* (*i*=1; *i*<16; *i*++)  {  *target*. *flickFromTo* ({*x*: 360*, y*: 200}*,* {*x*: 160*, y*: 200}); // *Flick Picture Change*  (*NEXT*)  *target*. *delay* (2);  }  *for* (*i*=1; *i*<16; *i*++)  {  *target*. *flickFromTo* ({*x*: 160*, y*: 200}*,* {*x*: 260*, y*: 200}); // *Flick Picture Change*  (*PREVIOUS*)  *target*. *delay* (2);  }  *for* (*j*=1; *j*<16; *j*++)  {  *target*. *frontMostApp* (). *mainWindow* (). *buttons* () ["*Next*"]. *tap* (); // *Button Picture Change* (*NEXT*)  *target*. *delay* (2);  }  *for* (*j*=1; *j*<16; *j*++)  {  *target*. *frontMostApp* (). *mainWindow* (). *buttons* () ["*Previous*"]. *tap* (); // *Button Picture Change* (*PREVIOUS*)  *target*. *delay* (2);  }  *UIALogger*. *logPass* ("*Done*");  *Piece*\_*Style*\_*Test*. *js*  #*import* "*Join*\_*It*\_*Functions*. *js*"  *var target* = *UIATarget*. *localTarget* (); *UIALogger*. *logStart* ("*Join It Temp Script*"); *target*. *delay* (1);  *target*. *frontMostApp* (). *mainWindow* (). *elements* () [0]. *tap* (); // *Cancel video* | |
|  | *71* |

|  |  |
| --- | --- |
| *target*. *delay* (1);  // *First*\_*Launch*\_*Check* ();  *for* (*j*=1; *j*<5; *j*++)  {  *Open*\_*Options* (); *Tap*\_*Piece*\_*Style* (*j*); *target*. *delay* (1); *Close*\_*Options* (); *Start*\_*Game* (5);  *target*. *captureScreenWithName* ("*Piece*\_*Syle* " + *j*); *Exit*\_*Game* ();  }  *UIALogger*. *logPass* ("*Done*");  *High*\_*Score*\_*Test*. *js*  #*import* "*Join*\_*It*\_*Functions*. *js*"  *var target* = *UIATarget*. *localTarget* (); *UIALogger*. *logStart* ("*Join It Temp Script*"); *target*. *delay* (1);  *target*. *frontMostApp* (). *mainWindow* (). *elements* () [0]. *tap* (); // *Cancel video target*. *delay* (2);  *target*. *frontMostApp* (). *mainWindow* (). *elements* () [7]. *tap* (); // *High SCore Change*\_*Difficulty* ("*Easy*");  *target*. *delay* (2);  *target*. *captureScreenWithName* ("*High Score for Easy*"); *Change*\_*Difficulty* ("*Medium*");  *target*. *delay* (2);  *target*. *captureScreenWithName* ("*High Score for Medium*"); *Change*\_*Difficulty* ("*Hard*");  *target*. *delay* (2);  *target*. *captureScreenWithName* ("*High Score for Hard*"); *Change*\_*Difficulty* ("*Insane*"); | |
|  | *72* |

|  |  |
| --- | --- |
| *target*. *delay* (2);  *target*. *captureScreenWithName* ("*High Score for Insane*");  *target*. *frontMostApp* (). *mainWindow* (). *buttons* () ["*HelpToolbar*"]. *tap* (); *target*. *frontMostApp* (). *mainWindow* (). *buttons* () ["*BackToolbar*"]. *tap* (); *Open*\_*Options* ();  *Close*\_*Options* ();  *target*. *frontMostApp* (). *mainWindow* (). *elements* () [8]. *tap* (); // *Main Menu UIALogger*. *logPass* ("*Done*");  *Join*\_*It*\_*Functions*. *js*  *var target* = *UIATarget*. *localTarget* ();  *function Deactivate* (*t*)  {  *UIATarget*. *localTarget* (). *deactivateAppForDuration* (*t*);  }  *function Start*\_*Game* (*t*)  {  *target*. *frontMostApp* (). *mainWindow* (). *images* () [3]. *buttons* () ["*Go*"]. *tap* ();  *target*. *delay* (*t*);  *target*. *tap* ({*x*: 350.00*, y*: 350.00});  }  *function Game* ()  {  *Open*\_*Options* (); *Set*\_*Reference*\_*Picture*\_*Visibility* (*Random* (1)); *Tap*\_*Sticking* ();  *Tap*\_*Picture*\_*Frame* (); *Set*\_*Table*\_*Scale* (*Random* (1)); *Tap*\_*Zoom*\_*Lock* (); *Tap*\_*Rotation* ();  *Tap*\_*Timer* ();  *Tap*\_*Sound* (); | |
|  | *73* |

|  |  |
| --- | --- |
| *Tap*\_*Sharing* (); *Choose*\_*Background* (*Random* (8)); *Close*\_*Options* ();  *Exit*\_*Game* ();  }  *function Exit*\_*Game* ()  {  *target*. *frontMostApp* (). *mainWindow* (). *buttons* () ["*Pause*"]. *tap* ();  *target*. *delay* (1);  *target*. *frontMostApp* (). *mainWindow* (). *staticTexts* () ["*Main Menu*"]. *tap* ();  }  *function First*\_*Launch*\_*Check* ()  {  *if* (*target*. *frontMostApp* (). *mainWindow* (). *staticTexts* () ["*What*'*s New*"]. *isVisible* () ==*true*)  {  *target*. *delay* (20);  *target*. *frontMostApp* (). *mainWindow* (). *buttons* () ["*Continue*"]. *tap* ();  *target*. *delay* (1);  *target*. *frontMostApp* (). *mainWindow* (). *buttons* () ["*iPadButtonSwitchPressed*"]. *tap* ();  *target*. *delay* (1);  *target*. *frontMostApp* (). *mainWindow* (). *buttons* () ["*OK*"]. *tap* ();  *target*. *delay* (1);  }  }  *function Open*\_*Options* ()  {  *if* (*target*. *frontMostApp* (). *mainWindow* (). *buttons* () ["*HelpToolbar*"]. *isVisible*  () ==*true*)  { | |
|  | *74* |

|  |  |
| --- | --- |
| *target*. *frontMostApp* (). *mainWindow* (). *buttons* () ["*SettingsSmall*"]. *tap* (); //  *The Options*  }  *else*  {  *target*. *frontMostApp* (). *mainWindow* (). *buttons* () ["*SettingsControl*"]. *tap* (); //  *The InGame Options*  }  *target*. *delay* (1);  }  *function Close*\_*Options* ()  {  *target*. *tap* ({*x*: 651.00*, y*: 63.00});  }  *function Random* (*p*)  {  *if* (*p*<=1)  {  *r* = *Math*. *random* ();  }  *else*  {  *r* = *parseInt* (*Math*. *random* () \* *p*) +1;  }  *UIALogger*. *logMessage* ("*Random Number is* " +*r*)  *return r*;  }  *function Set*\_*Reference*\_*Picture*\_*Visibility* (*r*)  {  *target*. *frontMostApp* (). *mainWindow* (). *scrollViews* () [0]. *sliders* () [0].  *dragToValue* (*r*); | |
|  | *75* |

|  |  |
| --- | --- |
| }  *function Tap*\_*Sticking* ()  {  *target*. *frontMostApp* (). *mainWindow* (). *scrollViews* () [0]. *elements* () [6]. *tap*  (); // *STICKING y*={95: 123}  *target*. *delay* (1);  }  *function Tap*\_*Picture*\_*Frame* ()  {  *target*. *frontMostApp* (). *mainWindow* (). *scrollViews* () [0]. *elements* () [10]. *tap*  ();  }  *function Set*\_*Table*\_*Scale* (*r*)  {  *target*. *frontMostApp* (). *mainWindow* (). *scrollViews* () [0]. *sliders* () [1].  *dragToValue* (*r*);  }  *function Tap*\_*Zoom*\_*Lock* ()  {  *target*. *frontMostApp* (). *mainWindow* (). *scrollViews* () [0]. *elements* () [10]. *tap*  ();  }  *function Tap*\_*Rotation* ()  {  *target*. *frontMostApp* (). *mainWindow* (). *scrollViews* () [0]. *elements* () [22]. *tap*  (); // *ROTATION y*={430: 460}  *target*. *delay* (1);  }  *function Tap*\_*Timer* ()  { | |
|  | *76* |

|  |  |
| --- | --- |
| *target*. *frontMostApp* (). *mainWindow* (). *scrollViews* () [0]. *elements* () [25]. *tap*  (); // *TIMER y*={479: 509}  *target*. *delay* (1);  }  *function Tap*\_*Sound* ()  {  *target*. *frontMostApp* (). *mainWindow* (). *scrollViews* () [0]. *elements* () [28]. *tap*  (); // *SOUND y*={526: 556}  *target*. *delay* (1);  }  *function Tap*\_*Sharing* ()  {  *target*. *frontMostApp* (). *mainWindow* (). *scrollViews* () [0]. *elements* () [30]. *tap*  (); // *SHARING y*={573: 603}  *target*. *delay* (1);  }  *function Tap*\_*Iphone* ()  {  *target*. *frontMostApp* (). *mainWindow* (). *scrollViews* () [0]. *elements* () [34]. *tap*  (); // *IPHONE*  *target*. *delay* (1);  }  *function Tap*\_*Ipad* ()  {  *target*. *frontMostApp* (). *mainWindow* (). *scrollViews* () [0]. *elements* () [35]. *tap*  (); // *IPAD*  *target*. *delay* (1);  }  *function Tap*\_*Piece*\_*Style* (*Piece*\_*Num*)  {  *switch* (*Piece*\_*Num*) | |
|  | *77* |

|  |  |
| --- | --- |
| {  *case* 1:  *target*. *frontMostApp* (). *mainWindow* (). *scrollViews* () [0]. *scrollViews* () [0].  *elements* () [0]. *tap* (); // *PIECE STYLE* 1  *target*. *delay* (1);  *break*; *case* 2:  *target*. *frontMostApp* (). *mainWindow* (). *scrollViews* () [0]. *scrollViews* () [0].  *elements* () [1]. *tap* (); // *PIECE STYLE* 2  *target*. *delay* (1);  *break*; *case* 3:  *target*. *frontMostApp* (). *mainWindow* (). *scrollViews* () [0]. *scrollViews* () [0].  *elements* () [2]. *tap* (); // *PIECE STYLE* 3  *target*. *delay* (1);  *break*;  *case* 4: *target*. *frontMostApp* (). *mainWindow* (). *scrollViews* () [0]. *scrollViews*  () [0]. *elements* () [3]. *tap* (); // *PIECE STYLE* 4  *target*. *delay* (1);  *break*; *default*:  *target*. *frontMostApp* (). *mainWindow* (). *scrollViews* () [0]. *scrollViews* () [0].  *elements* () [2]. *tap* (); // *PIECE STYLE* 3  *target*. *delay* (1);  }  }  *function Choose*\_*Background* (*r*)  {  *target*. *frontMostApp* (). *mainWindow* (). *scrollViews* () [0]. *scrollViews* () [0].  *elements* () [3]. *scrollToVisible* ();  *target*. *delay* (1); | |
|  | *78* |

|  |  |
| --- | --- |
| *target*. *frontMostApp* (). *mainWindow* (). *scrollViews* () [0]. *buttons* () [8]. *tap* ();  *target*. *delay* (1);  *switch* (*r*)  {  *case* 1:  *target*. *frontMostApp* (). *mainWindow* (). *scrollViews* () [0]. *buttons* () [1]. *tap* ();  // *PIECE STYLE* 1  *target*. *delay* (1);  *break*; *case* 2:  *target*. *frontMostApp* (). *mainWindow* (). *scrollViews* () [0]. *buttons* () [2]. *tap* ();  // *PIECE STYLE* 2  *target*. *delay* (1);  *break*; *case* 3:  *target*. *frontMostApp* (). *mainWindow* (). *scrollViews* () [0]. *buttons* () [3]. *tap* ();  // *PIECE STYLE* 3  *target*. *delay* (1);  *break*; *case* 4:  *target*. *frontMostApp* (). *mainWindow* (). *scrollViews* () [0]. *buttons* () [4]. *tap* ();  // *PIECE STYLE* 4  *target*. *delay* (1);  *break*; *case* 5:  *target*. *frontMostApp* (). *mainWindow* (). *scrollViews* () [0]. *buttons* () [5]. *tap* ();  // *PIECE STYLE* 1  *target*. *delay* (1);  *break*; *case* 6: | |
|  | *79* |

|  |  |
| --- | --- |
| *target*. *frontMostApp* (). *mainWindow* (). *scrollViews* () [0]. *buttons* () [6]. *tap* ();  // *PIECE STYLE* 2  *target*. *delay* (1);  *break*; *case* 7:  *target*. *frontMostApp* (). *mainWindow* (). *scrollViews* () [0]. *buttons* () [7]. *tap* ();  // *PIECE STYLE* 3  *target*. *delay* (1);  *break*; *case* 8:  *target*. *frontMostApp* (). *mainWindow* (). *scrollViews* () [0]. *buttons* () [8]. *tap* ();  // *PIECE STYLE* 4  *target*. *delay* (1);  *break*; *default*:  *target*. *frontMostApp* (). *mainWindow* (). *scrollViews* () [0]. *buttons* () [1]. *tap* ();  // *PIECE STYLE* 3  *target*. *delay* (1);  }  *target*. *delay* (1);  }  *function Change*\_*Difficulty* (*Difficulty*)  {  *switch* (*Difficulty*)  {  *case* "*Easy*":  *target*. *frontMostApp* (). *mainWindow* (). *elements* () [3]. *tap* (); // *EASY target*. *delay* (1);  *break*;  *case* "*Medium*":  *target*. *frontMostApp* (). *mainWindow* (). *elements* () [4]. *tap* (); // *MEDIUM* | |
|  | *80* |

|  |  |
| --- | --- |
| *target*. *delay* (1);  *break*;  *case* "*Hard*":  *target*. *frontMostApp* (). *mainWindow* (). *elements* () [5]. *tap* (); // *HARD target*. *delay* (1);  *break*;  *case* "*Insane*":  *target*. *frontMostApp* (). *mainWindow* (). *elements* () [6]. *tap* (); // *INSANE target*. *delay* (1);  *break*; *default*:  *target*. *frontMostApp* (). *mainWindow* (). *elements* () [4]. *tap* (); // *MEDIUM target*. *delay* (1);  }  }  *function Change*\_*Difficulty*\_*Drag* (*Difficulty*\_*Drag*)  {  *target*. *frontMostApp* (). *mainWindow* (). *sliders* () [0]. *dragToValue*  (*Difficulty*\_*Drag*);  *target*. *delay* (2);  }  *function Choose*\_*Random*\_*Picture* (*Random*\_*Image*)  {  *target*. *frontMostApp* (). *mainWindow* (). *buttons* () ["*Collections*"]. *tap* ();  *r*=*Random* (6);  *if* (*r*==1)  {  *r*++;  }  *target*. *frontMostApp* (). *mainWindow* (). *tableViews* () ["*Empty list*"]. *cells* () [*r*].  *tap* (); | |
|  | *81* |

|  |  |
| --- | --- |
| *target*. *delay* (2);  *r*=*Random* (15);  *target*. *frontMostApp* (). *mainWindow* (). *scrollViews* () [0]. *elements* () [*r*]. *tap*  ();  *target*. *delay* (1);  *r*++;  *if* (*target*. *frontMostApp* (). *mainWindow* (). *buttons* () ["*Edit*"]. *isVisible* ()  ==*true*)  {  *target*. *delay* (30);  *target*. *frontMostApp* (). *mainWindow* (). *scrollViews* () [0]. *elements* () [*r*]. *tap*  ();  *target*. *delay* (1);  }  } | |
|  | *82* |