МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ КІБЕРБЕЗПЕКИ, КОМП'ЮТЕРНОЇ ТА ПРОГРАМНОЇ ІНЖЕНЕРІЇ

Кафедра Комп'ютерних інформаційних технологій

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач кафедри
Аліна САВЧЕНКО
«_____» _____ 2021р.

ДИПЛОМНА РОБОТА

(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ВИПУСКНИКА ОСВІТНЬОГО СТУПЕНЯ

"МАГІСТРА"

ЗА ОСВІТНЬО-ПРОФЕСІЙНОЮ ПРОГРАМОЮ "ІНФОРМАЦІЙНІ УПРАВЛЯЮЧІ СИСТЕМИ ТА ТЕХНОЛОГІЇ"

Тема: "Методи та засоби автоматизованого тестування web-сервісів на мові Java (Scala)"

Виконавець: *Гомель Олександр Олегович*

Керівник: *к.т.н., доцент, Райчев Ігор Едуардович*

Нормоконтролер: *Ігор РАЙЧЕВ*

Київ – 2021

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет *Кібербезпеки, комп'ютерної та програмної інженерії*

Кафедра *Комп'ютерних інформаційних* технологій

Галузь знань, спеціальність, освітньо-професійна програма: 12 "Інформаційні технології", 122 "Комп'ютерні науки", "Інформаційні управляючі системи та технології"

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____Аліна САВЧЕНКО
«____»_____2021р.

ЗАВДАННЯ

на виконання дипломної роботи студента

Гомеля Олександра Олеговича
(прізвище, ім'я, по батькові)

1. Тема роботи: «Методи та засоби автоматизованого тестування web-сервісів на мові Java (Scala)» затверджена наказом ректора 2228/ст. від 12.10.21р.

2. Термін виконання роботи: 12.10.2021 – 31.12.2021

3. Вихідні данні до роботи: процеси тестування ПЗ в ІТ-компаніях; тестування навантаження; завдання на створення автоматизованої множини тестових наборів даних для тестування веб-сервісів різних типів.

4. Зміст пояснювальної записки: вступ, місце тестування в процесі розробки програмного забезпечення, методології тестування, види тестування та класифікація методів тестування ПС, техніки створення тестів, тестування навантаження та автоматизоане функціональне тестування.

5. Перелік обов'язкового графічного матеріалу: життєвий цикл продукту згідно технології створення програмних продуктів RUP; ітерації життєвого циклу програмного продукту; методи черного, білого та сірого ящика; схеми процесу тестування; рівні тестування; життєвий цикл бага.

6. Календарний план-графік

№ п/п	Завдання	Термін виконання	Підпис керівника
1.	Отримання завдання на дипломну роботу, створення плану дипломної роботи та побудова плану-графіку виконання робіт.	12.10.2021 – 15.10.2021	
2.	Огляд та аналіз наукової літератури по темі дипломної роботи та написання Розділу 1.	16.10.2021 – 26.10.2021	
3.	Написання Розділу 2 дипломної роботи.	27.10.2021 – 07.11.2021	
4.	Написання Розділу 3 і Розділу 4 дипломної роботи. Завершення створення пояснювальної записки дипломної роботи.	08.11.2021 – 09.12.2021	
5.	Оформлення та друк пояснювальної записки.	10.12.2021 – 13.12.2021	
6.	Створення презентації, доповіді та підготовка до захисту дипломної роботи.	14.12.2021 – 17.12.2021	
7.	Підготовка матеріалів дипломної роботи для передачі секретарю ДЕК (папка, конверт, диск із файлом диплому, рецензія, відгук).	18.12.2021 – 21.12.2021	

7. Дата видачі завдання: «12» жовтня 2021 р.

Керівник дипломної роботи _____ Ігор РАЙЧЕВ
(підпис керівника) (П.І.Б.)

Завдання прийняв до виконання _____ Олександр ГОМЕЛЬ
(підпис випускника) (П.І.Б.)

РЕФЕРАТ

Пояснювальна записка до дипломної роботи «Методи та засоби автоматизованого тестування web-сервісів на мові Java (Scala)» складається зі вступу, чотирьох розділів, висновку, списку використаних джерел та двох додатків і містить 126 сторінок, 22 таблиці та 51 рисунок. Список використаних джерел складається з 15 найменувань.

Ключові слова: ТЕСТУВАННЯ, АВТОМАТИЗОВАНЕ ТЕСТУВАННЯ, WEB-СЕРВІС, ТЕСТУВАННЯ НАВАНТАЖЕННЯ, ТЕСТ-КЕЙС, ТЕСТОВІ НАБОРИ ДАНИХ, ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ, ПРОГРАМНА СИСТЕМА.

Актуальність. Нині етап тестування є обов'язковою частиною процесу виробництва ПЗ, воно спрямоване на виявлення та усунення якомога більшої кількості помилок. Наслідком такої діяльності є підвищення якості властивостей ПЗ. Однак, незалежно від кількості методик тестування, немає однієї єдиної техніки, за допомогою якої можна було б виявити, що ПЗ не містить помилок.

Метою дипломної роботи є дослідження процесу тестування, видів дефектів ПЗ та їх відстеження, способів створення і застосування тест-кейсів. Як результат, на основі отриманих знань необхідно розробити проект авто-тестів для веб-сервісу. Додатковою метою є проведення навантажувального тестування для веб-сервісу.

Для досягнення поставленої мети необхідно вирішити такі **завдання**:

- виявити місце, яке займає тестування в процесі розробки ПС;
- детально ознайомитися з процесом тестування в IT-компаніях;
- розглянути найвідоміші види дефектів і причини їх виникнення, а також дослідити системи пошуку та відслідковування помилок;
- ознайомитися з техніками створення тестів та їх застосуванням;
- виконати огляд ПЗ для навантажувального тестування;
- обрати платформу для навантажувального тестування веб-сервісу;
- оіолодіти процесом автоматизації функціонального тестування;
- розробити проект автоматизації для початківців-тестувальників;

- розробити проект авто-тестів для веб-сервісу.

Об'єктом дослідження є процес тестування ПЗ в ІТ-компаніях.

Предметом дослідження є техніки створення тест-кейсів і можливість автоматизації створення множини тест-кейсів (тестових наборів даних).

Методи дослідження включають у себе:

- методи тестування програмного забезпечення;
- методи визначення якості продукту;
- методи верифікації програмного забезпечення;
- методи автоматизації створення тест-кейсів.

Теоретичною основою дипломної роботи стали вітчизняні та зарубіжні дослідження щодо забезпечення якості програмного забезпечення та публікації на сайтах, присвячені питанням тестування програмних систем.

Теоретична і практична значимість роботи полягає в тому, що на основі отриманих знань:

1) можна в короткий термін ознайомитися з необхідною теорією з тестування ПЗ, що може допомогти успішному проходженню технічної співбесіди для влаштування на роботу тестувальником;

2) по зібраному матеріалу можна сформувати методичний посібник для студентів і включити його в програму навчання в якості додаткового курсу до дисципліни "Тестування ПЗ інформаційних систем";

3) розроблено проект автоматизованого створення тестів для тестування веб-сервісу, який успішно введено в експлуатацію.

На захист виносяться наступні положення:

1) процес тестування ПЗ в ІТ-компаніях;

2) тестування навантаження;

3) проект автотестів для веб-сервісу.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

ПЗ	Програмне забезпечення
CMMI	Capability Maturity Model Integration – комплексна модель продуктивності і зрілості – набір моделей (методологій) вдосконалення процесів в організаціях різних розмірів і видів діяльності. CMMI містить набір рекомендацій у вигляді практик, реалізація яких, дозволяє реалізувати цілі, необхідні для повної реалізації певних областей діяльності
IEEE	Institute of Electrical and Electronics Engineers – Інститут інженерів з електротехніки та електроніки – міжнародна некомерційна асоціація фахівців в області техніки, світовий лідер в області розробки стандартів з радіоелектроніки та електротехніки
RUP	Rational Unified Process – раціональний уніфікований процес розробки програмного забезпечення, технологія програмування створена компанією Rational Software
Баг	Жаргонне слово у тестуванні, позначає помилку в програмі або системі, яка видає несподіваний або неправильний результат
ISO 9660	Стандарт, випущений Міжнародною організацією зі стандартизації, що описує файлову систему для дисків CD-ROM
I/O	Введення/вихід – взаємодія між оброблювачем інформації (наприклад, комп'ютер) і зовнішнім світом, який може представляти як людина, так і будь-яка інша система обробки інформації. Введення – сигнал або дані, отримані системою, а вихід – сигнал або дані, надіслані нею (або з неї)
MDAC	Microsoft Data Access Components – сукупність технологій компанії Microsoft, що дозволяють отримати уніфікований спосіб доступу до даних з різних реляційних і не реляційних джерел. Термін MDAC є загальним позначенням для всіх розроблених компанією Microsoft

	технологій, пов'язаних з базами даних
ООП	Об'єктно-орієнтоване програмування
FTP	File Transfer Protocol – стандартний протокол, призначений для передачі файлів по TCP/IP-мережі
JDBC	Java DataBase Connectivity – переносних незалежний промисловий стандарт взаємодії Java-додатків з різними СУБД
СУБД	Система управління базами даних
ОПФ	Організаційно-правова форма
ОВД	Основний вид діяльності
ФА	Фінансовий аналіз
GUI	Різновид призначеного для користувача інтерфейсу, в якому елементи інтерфейсу (меню, кнопки, значки, списки тощо), виконані у вигляді графічних зображень
ІС	Інформаційна система
ПС	Програмна система
ІТ	Інформаційні технології
ТНД	Тестовий набір даних

ЗМІСТ

ВСТУП

Основний сплеск інтересу до процесу тестування припав на 90-і роки минулого сторіччя і почався у США. Бурхливий розвиток систем автоматизованої розробки ПЗ (CASE-засобів) та мережевих технологій призвів до зростання ринку виробництва ПЗ і до перегляду питань забезпечення якості та надійності розроблюваних програмних систем (ПС), чого можна було досягти тільки впровадженням у життєвий цикл розробки ПЗ поглибленого тестування.

Різко посилилася конкуренція між виробниками ПЗ, користувачі зажадали високої якості створюваних продуктів, оскільки тепер у споживача був вибір: багато фірм пропонували свої продукти і послуги за досить прийнятними цінами, а тому можна було звернутися до тих, хто розробить ПС не тільки швидко і дешево, але й якісно. Ситуація ускладнилась тим фактом, що нині комп'ютеризацією охоплені практично всі сфери людського життя. І питання про якість ПЗ починає набувати особливу важливість: сьогодні це вже не тільки комфорт під час роботи в тій чи іншій системі, нині програмні системи управляють обладнанням в лікарнях, диспетчерськими системами в аеропортах, атомними реакторами тощо.

Усвідомивши той факт, що забезпечення високої якості розроблюваного ПЗ — це реальний шлях «обійти» конкурентів, багато компаній у всьому світі вкладають все більше коштів на забезпечення якості своїх програмних продуктів, створюючи власні групи і відділи, які займаються тестуванням, або передаючи тестування своїх продуктів стороннім організаціям.

Найбільші компанії, що піклуються про свою репутацію, та бажаючі пройти сертифікацію і досягти високого рівня CMMI (Capability Maturity Model Integration), створюють свої власні системи управління якістю (Quality Management System), спрямовані на постійне вдосконалення виробничих процесів і постійне підвищення якості програмних продуктів, що розробляються.

Сьогодні тестування стало обов'язковою частиною процесу виробництва ПЗ. Воно спрямоване на виявлення та усунення якомога більшого числа помилок.

Наслідком такої діяльності є підвищення якості ПЗ по всіх його характеристиках.

Існуючі на сьогоднішній день методи тестування програмного забезпечення не дозволяють однозначно і повністю усунути всі дефекти та помилки і встановити коректність функціонування програмного продукту. Тому, всі існуючі методи тестування діють в рамках формального процесу перевірки досліджуваного або розроблюваного програмного продукту.

Такий процес формальної перевірки або верифікації може довести, що дефекти відсутні, з точки зору використовуваного методу. З цього випливає, що немає ніякої можливості точно встановити або гарантувати відсутність дефектів в програмному продукті з урахуванням людського фактора, присутнього на всіх етапах життєвого циклу виготовлення програмного забезпечення.

Існує багато підходів до вирішення завдання тестування і верифікації програмного забезпечення, але ефективне тестування складних програмних продуктів — це процес надзвичайно творчий, який не зводиться до проходження суворих і чітких процедур тестових випробувань або створення таких.

Кінцевою метою будь-якого процесу тестування є забезпечення такого ємного сукупного поняття як *якість програмного родукту*, з урахуванням всіх, або найбільш критичних для даного конкретного випадку складових властивостей.

Тестування програмного забезпечення — це спроба визначити, чи виконує програма те, що від неї очікують. Як правило, ніяке тестування не може дати абсолютної гарантії працездатності програми в майбутньому. Завдання тестування ПЗ — знизити вартість розробки шляхом раннього виявлення дефектів.

РОЗДІЛ 1. ОГЛЯД ПРОЦЕСУ ТЕСТУВАННЯ

1.1. Визначення тестування

У відповідності з IEEE Std 829: тестування — це процес аналізу ПЗ, спрямований на виявлення відмінностей між його реально існуючими і необхідними властивостями (пошук дефектів), та на оцінку властивостей ПЗ.

Відповідно до ГОСТ/ISO МЕК 12207 в життєвому циклі ПЗ визначені допоміжні процеси верифікації, атестації, спільного аналізу та аудиту. Процес верифікації є процесом визначення того, що програмні продукти функціонують в повній відповідності з вимогами або умовами, реалізованими в попередніх етапах. Даний процес може включати аналіз, перевірку та випробування (тестування). Процес атестації є процесом визначення повноти відповідності встановлених вимог, створеної ПС або програмного продукту їх функціональному призначенню. Процес спільного аналізу є процесом оцінки станів і, при необхідності, результатів робіт над програмним продуктом згідно з проектом. Процес аудиту є процесом визначення відповідності вимогам, планам та умовам договору. У сумі ці процеси й складають те, що зазвичай називають тестуванням.

Тестування грунтується на тестових процедурах з конкретними вхідними даними, початковими умовами і очікуваним результатом, що розроблені для певної мети, такої, як перевірка окремої програми або верифікація відповідності певної вимоги. Тестові процедури можуть перевіряти різні аспекти функціонування програми — від правильної роботи окремої функції до адекватного виконання бізнес-вимог.

При виконанні проекту необхідно враховувати, відповідно до яких стандартів та вимог буде проводитися тестування програмного продукту. Також які інструментальні засоби будуть використовуватися для пошуку і для документування знайдених дефектів. Якщо пам'ятати про тестування з самого початку виконання проекту, тестування продукту, що розробляється, не завдасть неприємних несподіванок. А значить і якість продукту, швидше за все, буде досить високою.

1.2. Життєвий цикл програмних продуктів та тестування

Останнім часом частіше використовуються ітеративні процеси розробки ПЗ, зокрема, технологія RUP — Rational Unified Process. На рис.1.1 можна побачити життєвий цикл продукту згідно RUP. При використанні такого підходу тестування перестає бути другорядним процесом, який запускається тільки після того, як програмісти написали весь необхідний код. Робота над тестами починається з самого початкового етапу виявлення вимог впритул до закінчення виготовлення майбутнього продукту і тісно інтегрується з поточними завданнями. А це висуває нові вимоги до тестерів. Їх роль не зводиться просто до виявлення помилок якомога повніше і якомога раніше. Вони повинні брати участь у загальному процесі виявлення та усунення найбільш істотних ризиків проекту. Для цього для кожної ітерації визначається мета тестування і методи її досягнення. А в кінці кожної ітерації визначається, наскільки ця мета досягнута, чи потрібні додаткові випробування, і чи не потрібно змінити принципи та інструменти проведення тестів. У свою чергу, кожен виявлений дефект повинен пройти через свій власний життєвий цикл.

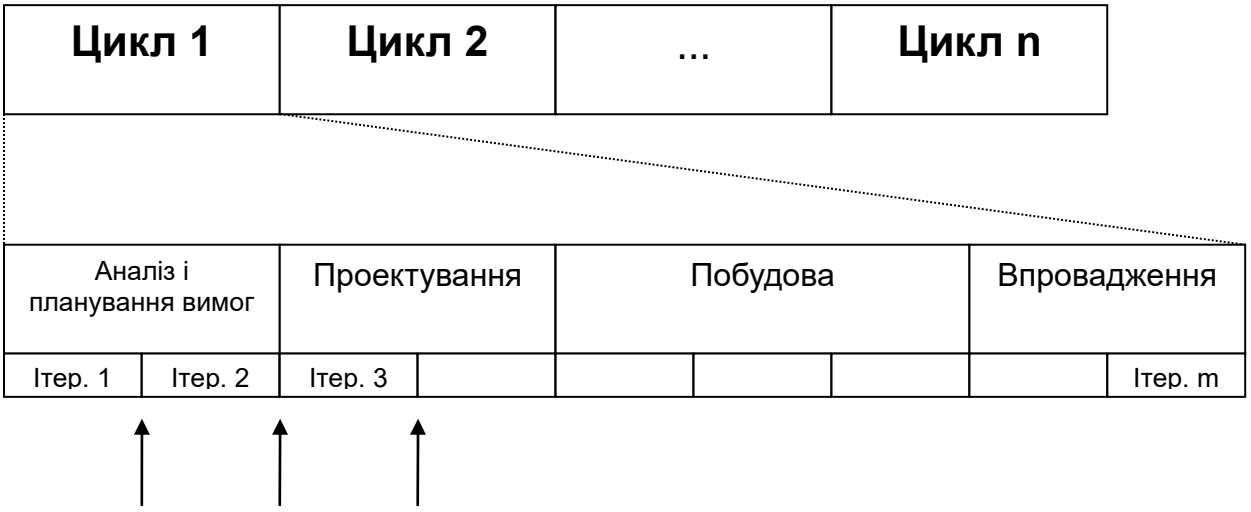


Рис.1.1. Життєвий цикл продукту згідно RUP

Тестування зазвичай проводиться циклами, кожен з яких має конкретний список завдань і цілей. Цикл тестування може збігатися з ітерацією або відповідати її певній частині. Як правило, цикл тестування проводиться для

конкретної збірки системи.

Життєвий цикл програмного продукту, який зображений на рис.1.2, складається з серії щодо коротких ітерацій. Ітерація — це є завершений цикл розробки, що приводить до випуску кінцевого продукту або деякої його скороченої версії, яка розширюється від ітерації до ітерації, щоб, врешті-решт, стати закінченою системою.

Кожна ітерація включає, як правило, завдання планування робіт, аналізу, проектування, реалізації, тестування та оцінки досягнутих результатів. Однак задачі в ітерації групуються у фази. У першій фазі - Початок - основна увага приділяється завданням аналізу. У ітераціях другої фази - Розробка - основна увага приділяється проектування і випробування ключових проектних рішень. У третій фазі - Побудова - найбільш велика частка задач розробки і тестування. А в останній фазі - Передача - вирішуються в найбільшій мірі завдання тестування і передачі системи Замовнику.

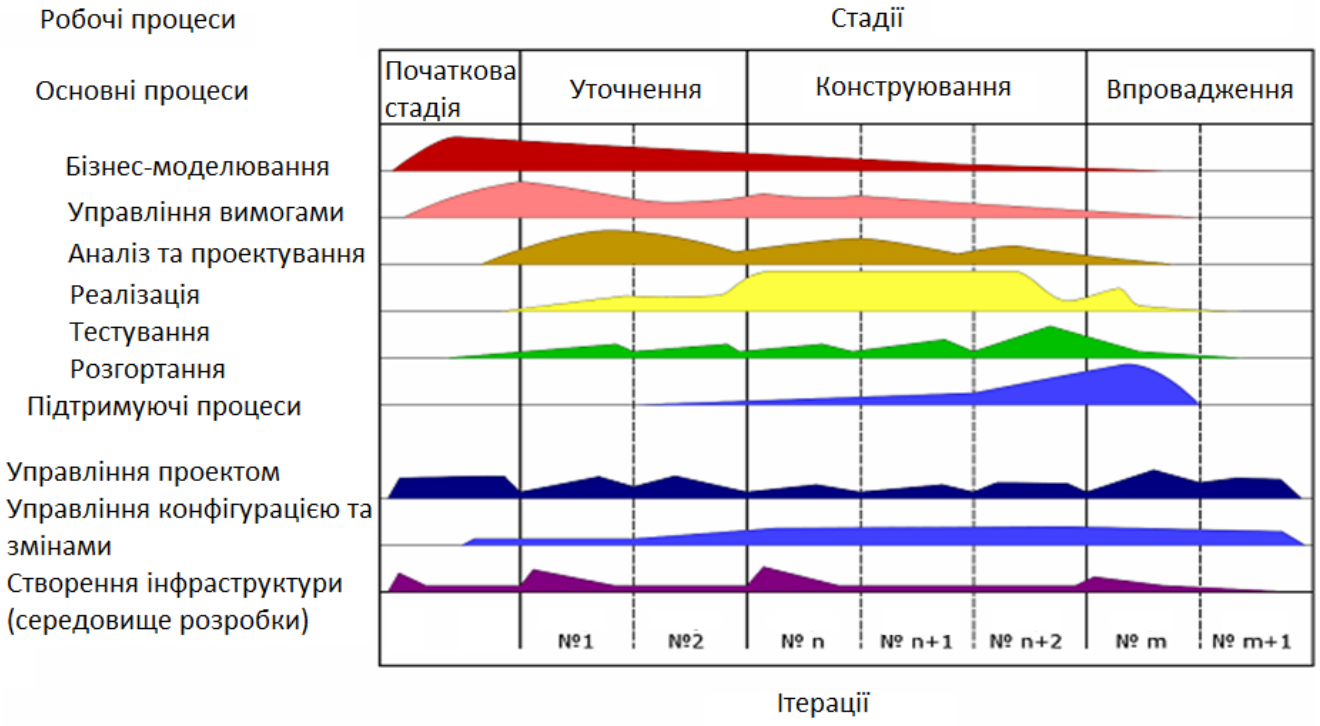


Рис.1.2. Ітерації життєвого циклу програмного продукту

Кожна фаза має свої специфічні цілі в життєвому циклі продукту і вважається виконаною, коли ці цілі досягнуті. Все ітерації, крім, ітерацій

Початкової стадії, завершуються створенням функціонуючої версії розроблюваної системи.

1.3. Місце тестування в процесі розробки ПЗ

Структура фірми-розробника програмного забезпечення відображає етапи життєвого циклу програмного засобу. Той чи інший підрозділ забезпечує виконання робіт на одному або декількох етапах життєвого циклу програмного забезпечення.

Аналітичний відділ. До завдань аналітичного відділу входять:

- визначення концепцій і функціонального спрямування розвитку програмного продукту;
- проведення передпроектного обстеження;
- визначення функціональних можливостей системи;
- визначення (спільно з розробниками) технічних вимог до системи;
- опис бізнес-процесів предметної області в термінах, зрозумілих розробникам (Постановки завдань і специфікації на розробку);
- написання постановок завдань і специфікацій на доопрацювання програмного засобу при зміні законодавства, вимог клієнтів, розширенні функціональних можливостей продукту;
- контроль процесу реалізації нових можливостей в програмних продуктах компанії.

Відділ документації. Часто даний відділ не виділяється в відокремлену структуру, він може входити, наприклад, до складу аналітичного відділу.

До завдань відділу входять написання технічної документації для кінцевого користувача, відстеження змін в програмному засобі і актуалізація в документації.

Відділ розробки. Це ключовий відділ для фірми. Якщо без інших відділів часто можна обійтися, то без відділу розробки не можна. У його завдання входять:

- визначення (спільно з аналітиками) технічних вимог до системи;

- реалізація базових функцій програмного засобу;
- розширення переліку функцій програмного засобу (реалізація доробок);
- виправлення знайдених помилок;
- адаптація програмного продукту для функціонування в інших умовах (перехід на нову СУБД, нову мову програмування та ін.);
- оптимізація програмного продукту (збільшення швидкодії, надійності та ін.).

Відділ технічної підтримки (гаряча лінія). Здійснює консультації користувачів з питань, пов'язаних з установкою і експлуатацією програмного засобу по різних каналах зв'язку (телефон, пошта, електронна пошта).

Відділ тестування. До завдань відділу входять:

- комплексний контроль якості;
- підготовка тестової документації (плани тестування та ін.);
- виявлення та локалізація помилок у функціонуванні програмних продуктів;
- фіксування і відстеження помилок у функціонуванні програмних засобів;
- перевірка відповідності документації програмного продукту стандартам і реально реалізованим функціям;
- участь в розробці та впровадженні системи якості;
- автоматизація тестування;
- оцінка продуктивності розроблюваних програмних засобів на різних програмно-апаратних платформах і їх специфічних конфігураціях.

У деяких компаніях на відділ тестування покладаються збірка і випуск програмного забезпечення (в деяких компаніях цим займається відділ розробки).

Всі відділи компанії взаємодіють між собою, дані взаємодії впорядковані між собою і представляють виробничі технологічні процеси. Технологічні про-процеси, як правило, регламентовані внутрішніми документами або

внутрішньокорпоративними стандартами; в сукупності являють собою технологічний цикл виробництва про- грамного засобу.

Типових технологічних ланцюжків всередині компанії - розробника програмного забезпечення велика кількість. Як приклад розглянемо схему взаємодії відділу тестування програмного забезпечення з іншими відділами при виявленні помилки під час функціонування програмного забезпечення у користувача.

1.4. Методологія тестування

У термінології професіоналів тестування (програмного і деякого апаратного забезпечення), визначення «тестування білого ящика» і «тестування чорного ящика» відносяться до того факту, чи має розробник тестів доступ до вихідного коду тестованого програмного забезпечення, або ж тестування виконується через інтерфейс або прикладний програмний інтерфейс, наданий тестованим модулем.

Під час тестування методом білого ящика (white-box testing, також говорять — прозорого ящика, воно ж структурне тестування), розробник тесту має доступ до вихідного коду програм і може писати код, який пов'язаний з бібліотеками тестованого програмного забезпечення. На рис.1.3 можна побачити схематичне зображення даного методу.

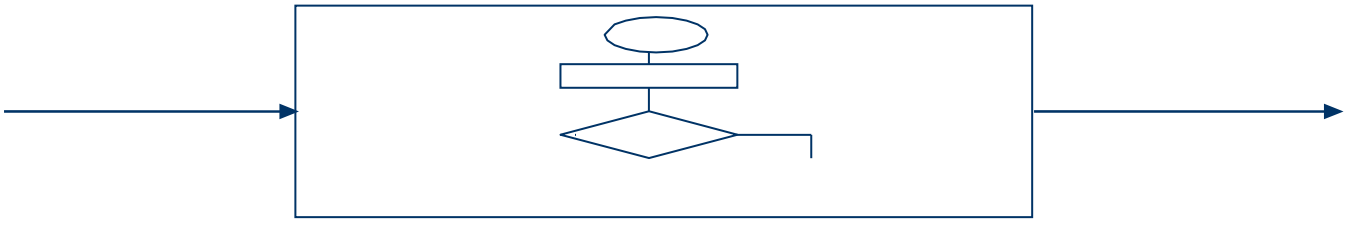


Рис.1.3. Метод білого ящика

Тести створюються на основі знань про структуру самої ПС і про те, як вона працює. Критерії повноти засновані на відсотку елементів коду, які відпрацювали в ході виконання тестів. Для оцінки ступеня відповідності вимогам можуть залучатися додаткові знання про зв'язок вимог з певними

обмеженнями на значення внутрішніх даних системи (наприклад, на значення параметрів викликів, результатів і локальних змінних).

В разі тестуванні методом чорного ящика, схему якого можна побачити на рис.1.4, тестувальник має доступ до програмного забезпечення тільки через ті ж самі інтерфейси, що і замовник або користувач, або через зовнішні інтерфейси, що дозволяють з іншого комп'ютера або іншому процесу підключитися до системи для виконання тестування.

Рис.1.4. Метод чорного ящика

Тестування чорного ящика націлене на перевірку вимог. Тести для нього і критерії повноти тестування створюються на основі вимог і обмежень, чітко зафіксованих в специфікаціях, стандартах та внутрішніх нормативних документах. Часто таке тестування називається тестуванням на відповідність (conformance testing). Окремим випадком його є функціональне тестування — тести для нього, а також використовувані критерії повноти проведеного тестування визначають на основі вимог до функціональності.

Крім методів тестування «білого ящика» і «чорного ящика» розрізняють тестування методом «сірого ящика», схема якого зображена на рис.1.5.

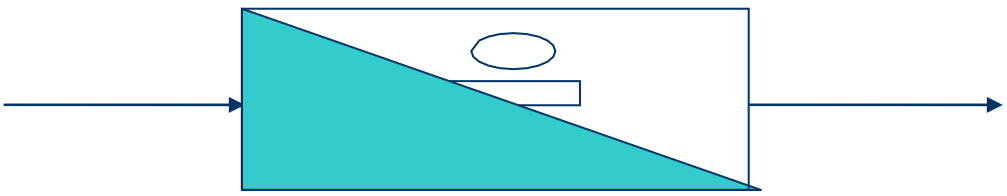


Рис.1.5. Метод сірого ящика

В даному випадку у людини, яка розробляє тест-кейси, є деяка інформація про внутрішню структуру програми або про деталі її реалізації. Даний метод застосовується останнім часом частіше попередніх.

1.5. Рівні тестування

Існує класифікація видів тестування, яка заснована на тому рівні деталізації робіт проекту, на який вона спрямована. На рис.1.6 зображені рівні тестування. Ці ж різновиди тестування можна пов'язати з фазою життєвого циклу, на якій вони виконуються.

Модульне тестування (Unit-testing) — рівень тестування, на якому тестується мінімально можливий для тестування компонент, наприклад, окремий клас або функція. На цьому рівні застосовуються методи «білого ящика». У сучасних проектах модульне тестування («юніт-тестінг») здійснюється розробниками.

Модульне тестування призначене для перевірки правильності окремих модулів, незалежно від їх оточення. При цьому перевіряється, що якщо модуль отримує на вхід дані, що задовольняють певним критеріям коректності, то й результати його коректні. Для опису критеріїв коректності вхідних і вихідних даних часто використовують програмні контракти-передумови, що описують для кожної операції, на яких вхідних даних вона призначена працювати; постумови, що описують для кожної операції, як повинні співвідноситися вхідні дані з повертаються нею результатами, а також інваріанти, що визначають критерії цілісності внутрішніх даних модуля.

Основний недолік модульного тестування полягає в тому, що проводити його можна, тільки коли перевіряємий елемент програми вже розроблено. Знизити вплив цього обмеження можна, готуючи тести (а це найбільш трудомістка частина тестування) на основі вимог заздалегідь, коли вихідного коду ще немає.

Модульне тестування є важливою складовою частиною налагоджувального тестування, виконуваного розробниками для налагодження написаного ними коду.

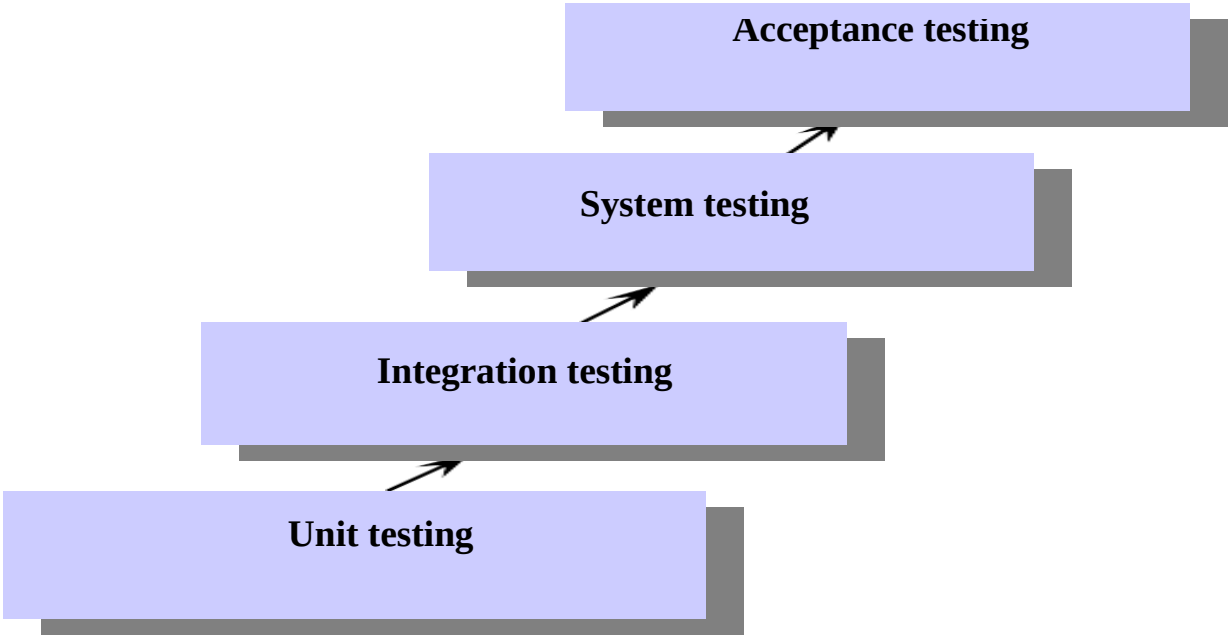


Рис.1.6. Рівні тестування

Інтеграційне тестування (Integration testing) — це рівень тестування, на якому окремі програмні модулі об'єднуються і тестуються в групі. Зазвичай інтеграційне тестування проводиться після модульного тестування (юніт-тести для модулів повинні бути виконані і знайдені помилки виправлені).

Інтеграційне тестування в якості вхідних даних використовує модулі, над якими було проведено модульне тестування, групує їх в більш великі множини, виконує тести, передбачені в плані тестування для цих множин, і представляє їх в якості вихідних даних, а також і вхідних для подальшого системного тестування.

При цьому перевіряється, що в ході спільної роботи модулі обмінюються даними і викликами операцій, не порушуючи взаємних обмежень на таку взаємодію, наприклад, передумов виклику операцій. Інтеграційне тестування виконується розробниками під час налагодження, але на більш пізньому етапі розробки.

Системне тестування (System testing) — це тестування ПЗ, яке виконується на повній, інтегрованій системі, з метою перевірки відповідності системи вихідним вимогам. Системне тестування відноситься до методів тестування «чорного ящика», і, тим самим, не вимагає знань про внутрішній устрій програмної системи.

Системне тестування виконується через зовнішні інтерфейси програмного забезпечення і тісно пов'язане з тестуванням призначеного для користувача інтерфейсу (або через призначений для користувача інтерфейс), що проводяться за допомогою імітації дій користувачів над елементами цього інтерфейсу. Окремими випадками цього виду тестування є тестування графічного інтерфейсу користувача (Graphical User Interface, GUI) та призначеного для користувача інтерфейсу Web-додатків (WebUI). Системне тестування виконується інженерами з тестування.

Приймальне тестування (Acceptance testing) — це тестування готового продукту кінцевими користувачами в реальному середовищі, в якому буде функціонувати тестована ПС. Приймальні тести розробляються користувачами, зазвичай, у вигляді сценаріїв. Для того, щоб знайти більше помилок рекомендується планувати не тільки системне тестування та приймальне,але модульне та інтеграційне.

Статичне тестування (Static testing) — тестування, в ході якого тестована програма (код) не виконується (не запускається). Аналіз програми відбувається на основі вихідного коду, який прораховується вручну, або аналізується спеціальними інструментами.

Приклади статичного тестування:

- огляди (Reviews);
- інспекції (Inspections);
- наскрізні перегляди (Walkthroughs);
- аудити (Audits).

Динамічне тестування (Dynamic testing) - тестування, в ході якого тестована програма (код) виконується (запускається).

Альфа-тестування - тестування в процесі розробки.

Бета-тестування - тестування виконується користувачами (end-users).

Перед тим, як виконується випуск програмного забезпечення, як мінімум, воно повинно пройти стадії альфа (внутрішнє пробне використання) і бета (пробне використання з залученням відібраних зовнішніх користувачів) версій.

Звіти про помилки, що надходять від користувачів цих версій продукту, обробляються відповідно до визначених процедур, що включають підтверджуючі тести (будь-якого рівня), що проводяться фахівцями групи розробки. Даний вид тестування не може бути заздалегідь спланований.

Регресійне тестування (Regression testing) — це тестування функціональності, яка була вже протестована, після внесення у систему будь-яких змін.

Після внесення змін до чергової версії програми, регресійні тести підтверджують, що зроблені зміни не вплинули на працездатність решти функціональності програми, додатку або ПС. Регресійне тестування може виконуватися як вручну, так і з залученням засобів автоматизації тестування.

Визначення успішності регресійних тестів позначено в стандарті IEEE 610-90 "Standard Glossary of Software Engineering Terminology", де сказано: "повторне вибіркове тестування системи або компонент з метою перевірки зроблених модифікацій не повинно призводити до непередбачуваних ефектів". На практиці це означає, що якщо система успішно проходила тести до внесення модифікацій, вона повинна їх проходити і після внесення цих модифікацій. Основна проблема регресійного тестування полягає у визначенні компромісу між наявними ресурсами і необхідністю проведення таких тестів в процесі внесення кожної зміни. Певною мірою, завдання полягає в тому, щоб визначити критерії "масштабів" змін, з досягненням яких необхідно проводити регресійні тести.

«Смок-тест» (Smoke Testing, «димове тестування») в тестуванні означає мінімальний набір тестів на явні помилки. Димової тест зазвичай виконується самим програмістом; якщо програма не проходить цей тест, то програму не має сенсу віддавати на подальше більш глибоке тестування.

1.6. Види тестування

Функціональне тестування (Functional testing) — мета даного тестування полягає в тому, щоб переконатися в належному функціонуванні

об'єкта тестування:

- кожна функціональна вимога транслюється в тест-кейси (тестові випадки), використовуючи техніки «чорного ящика», для того, щоб перевірити, що система функціонує в точності, як і описано в специфікаціях (функціональних вимогах до системи);
- перевіряється, чи всі функціональні вимоги дійсно запрограмовані/реалізовані.

Тестування продуктивності (Perfomance testing) — тестування, яке проводиться з метою визначення, як швидко працює система або її частина під певним навантаженням. Також може служити для перевірки і підтвердження інших атрибутів якості системи, таких як масштабованість, надійність і споживання ресурсів. Існує особливий підвид таких тестів, коли робиться спроба досягнення кількісних меж, обумовлених характеристиками самої системи та операційного середовища, де вона функціонує. Необхідно:

- продемонструвати, що система задовольняє критеріям продуктивності при заданих умовах;
- виміряти, яка частина системи є причиною «поганої» продуктивності системи;
- виміряти час реакції ПС на дію користувача, час відгуку на запит, і т.д.

Стресове тестування (Stress testing) зазвичай використовується для розуміння меж пропускної здатності додатків. Цей тип тестування проводиться для визначення надійності системи під час екстремальних або диспропорційних навантажень і відповідає на питання про достатню продуктивність системи в разі, якщо поточне навантаження сильно перевищує очікуваний максимум.

Тестування навантаження (Load testing) — це найпростіша форма тестування продуктивності. Тестування навантаження зазвичай проводиться для того, щоб оцінити поведінку програми під час заданого очікуваного навантаження. Цим навантаженням може бути, наприклад, очікувана кількість

одночасно працюючих користувачів додатка, які виконують задане число транзакцій за інтервал часу.

Такий тип тестування зазвичай дозволяє отримати час відгуку всіх найважливіших бізнес-транзакцій. У разі спостереження за базою даних, сервером додатків, мережею і т.д. Цей тип тестування може також ідентифікувати деякі вузькі місця реалізації системи.

Тестування стабільності (Stability testing) проводиться з метою, щоб переконатися в тому, що додаток витримує очікуване навантаження протягом тривалого часу. При проведенні цього виду тестування здійснюється спостереження за споживанням додатком пам'яті, щоб виявити потенційні витоки. Таке тестування виявляє деградацію продуктивності, що виражається в зниженні швидкості обробки інформації і/або збільшенні часу відповіді програм після тривалої їх роботи в порівнянні з початком тесту.

Тестування зручності використання (Usability testing) — дослідження, яке виконується з метою визначення, чи зручний деякий штучний об'єкт (такий як веб-сторінка, призначений для користувача інтерфейс або пристрій) для його передбачуваного застосування. Таким чином, перевірка ергономічності вимірює ергономічність об'єкта або системи. Перевірка ергономічності зосереджена на певному об'єкті або невеликому наборі об'єктів, в той час як дослідження взаємодії людина-комп'ютер в цілому формулює універсальні принципи.

Метод оцінки зручності продукту у використанні, заснований на залученні користувачів в якості тестувальників, випробувачів і підсумовуванні отриманих від них висновків.

В разі випробування багатьох продуктів користувачеві пропонують в «лабораторних» умовах вирішити основні завдання, для виконання яких цей продукт розроблявся, і просять висловлювати під час виконання цих тестів свої зауваження.

Процес тестування фіксується в протоколі і/або на відеотехніці з метою подальшого більш детального аналізу. Якщо юзабіліті-тестування виявляє

будь-які труднощі (наприклад складності в розумінні інструкцій, виконання дій або інтерпретації відповідей системи), то розробники повинні допрацювати продукт і повторити тестування.

Спостереження за тим, як люди взаємодіють з програмним продуктом, нерідко дозволяє знайти для нього більш оптимальні рішення. Якщо при тестуванні використовується модератор, то його завдання — тримати респондента сфокусованим на завданнях (але при цьому не "допомагати" йому вирішувати ці завдання). Основну складність після проведення процедури юзабіліті-тестування нерідко представляють великі обсяги і безладність отриманих даних. Тому для подальшого аналізу важливо зафіксувати:

1) мову модератора і респондента;

2) вираз обличчя респондента (знімається на відеокамеру);

3) зображення екрану комп'ютера, з яким працює респондент;

4) різні події, що відбуваються на комп'ютері, пов'язані з діями користувача:

- переміщення курсору і натиснення на кнопки миші;

- використання клавіатури;

- переходи між екранами (браузера або іншої програми).

Всі ці потоки даних повинні бути синхронізовані з тайм-кодами, щоб під час аналізу їх можна було б співвідносити між собою.

Поряд з модератором в тестуванні нерідко беруть участь спостерігачі. У міру виявлення проблем вони роблять свої замітки про хід тестування так, щоб після можна було синхронізувати їх з основним записом. В результаті кожен значущий фрагмент запису тесту виявляється прокоментований в нотатках спостерігача. В ідеалі провідник (тобто модератор) представляє розробника, спостерігачі — замовника (наприклад, видавця), а випробувачі — кінцевого користувача (наприклад покупця).

Існує ще один підхід до usability-тестування: для вирішення завдання, запропонованого користувачеві, розробляється "ідеальний" сценарій вирішення цього завдання. Як правило, це сценарій, на який орієнтувався розробник. При виконанні завдання користувачами реєструються їх відхилення від задуманого сценарію для подальшого аналізу. Після декількох ітерацій доопрацювання програми і подальшого тестування можна отримати задовільний інтерфейс з точки зору користувача.

Тестування інтерфейсу користувача (UI testing) — тестування графічного інтерфейсу користувача для того, щоб переконатися, що він відповідає прийнятим стандартам та їх вимогам і рекомендаціям. Перевіряється, як додаток обробляє введення з клавіатури і «мишки» і як відображаються елементи графічного інтерфейсу (текст, кнопки, меню, списки та інші елементи).

Тестування безпеки (security testing) - оцінка уразливості програмного забезпечення до різних атак. Комп'ютерні системи дуже часто є мішенню незаконного проникнення. Під проникненням розуміється широкий діапазон дій: спроби хакерів проникнути в систему з спортивного інтересу, помста розгніваних службовців, злом шахраями для незаконної наживи. Тестування безпеки перевіряє фактичну реакцію захисних механізмів, вбудованих в систему, на проникнення. В ході тестування безпеки випробувач грає роль зломщика. Йому дозволено все:

- спроби дізнатися пароль за допомогою зовнішніх засобів;
- атака системи за допомогою спеціальних утиліт, які аналізують захисту;
- придушення, приголомшення системи (в надії, що вона відмовиться обслуговувати інших клієнтів);
- цілеспрямоване введення помилок в надії проникнути в систему в ході відновлення;
- перегляд несекретних даних в надії знайти ключ для входу в систему.

Тестування локалізації (Localization testing) — це багатогранна річ, що

має на увазі перевірку множини аспектів, пов'язаних з адаптацією продукту для користувачів з інших країн. Наприклад, тестування локалізації для користувачів з Японії може полягати в перевірці того, чи не видасть система помилку, якщо цей користувач на сайті знайомств введе розповідь про себе символами Kanji, а не англійським шрифтом.

Тестування сумісності (Compatibility testing) — це вид нефункціонального тестування, основною метою якого є перевірка коректної роботи продукту в певному оточенні (середовищі). Оточення може включати в себе наступні елементи:

- апаратна платформа та мережеві пристрої;
- периферія (принтери, DVD-приводи, веб-камери, flash-пам’ять та ін.);
- операційна система (Unix, Windows, MacOS, ...);
- бази даних (Oracle, MS SQL, MySQL, ...);
- системне програмне забезпечення (веб-сервер, файєрвол, антивірус, ...);
- браузери (Internet Explorer, Firefox, Opera, Chrome, Safari).

1.7. Процес тестування. Етапи та завдання тестування

Тестування — це перевірка відповідності ПЗ вимогам, здійснювана за допомогою спостереження за його роботою в спеціальних, штучно побудованих ситуаціях. Такого роду ситуації називають тестовими випадками або просто тестами.

Тестування — це скінченна процедура. Однак, набір ситуацій, в яких буде перевірятися тестоване програмне забезпечення, не завжди скінченний. Але цей процес необхідно звести до того, щоб усі необхідні процедури тестування можна було провести в рамках проекту розробки ПЗ, не надто збільшуючи його бюджет і терміни. Це означає, що, як правило, під час проведення тестування завжди перевіряється лише невелика частка з множини всіх можливих тестових ситуацій.

Тестування може використовуватися для досить упевненого винесення оцінок про якість програмного забезпечення. Для цього необхідно обрати

критерії повноти тестування, що описують важливість різних ситуацій для оцінки якості, а також еквівалентності і залежності між ними. Наприклад, критерій може стверджувати, що все одно в якій із ситуацій, А або В, слід перевіряти правильність роботи програмного забезпечення, або, якщо програма правильно працює в ситуації А, то, швидше за все, в ситуації В все теж буде правильно. Часто критерії повноти тестування задаються за допомогою визначення еквівалентності ситуацій, що може дати кінцевий набір класів ситуацій. В цьому випадку вважають, що все одно, яку з ситуацій одного класу використовувати в якості тесту. Такий критерій називають критерієм тестового покриття, а відсоток класів еквівалентності всіляких ситуацій, що трапилися під час тестування — досягнутим тестовим покриттям.

Таким чином, основні завдання тестування: побудувати такий набір ситуацій, який був би досить представницьким і дозволяв би завершити тестування з достатнім ступенем впевненості в правильності ПЗ взагалі, щоб переконатися, що в конкретній ситуації програмне забезпечення працює правильно, відповідно до вимог.

Організація тестування ПЗ регулюється такими стандартами в сфері ІТ:

- IEEE 829-1998 Standard for Software Test Documentation. Описує види документів, які є для підготовки тестів;
- IEEE 1008-1987 (R1993, R2002) Standard for Software Unit Testing. Описує організацію модульного тестування;
- ISO/IEC 12119-1994 (аналог AS/NZS 4366:1996 і ГОСТ Р-2000, також IEEE 1465-1998) Information Technology. Software packages - Quality requirements and testing. Описує вимоги до процедур тестування ПС.

На рис.1.7 зображена схема процесу тестування. Розробка тестів відбувається на основі вимог і критеріїв повноти тестування. Розроблені тести формуються в Тейс-кейс (набір тестів) і виконуються на ПЗ, яке потрібно протестувати. Після прогону всіх тестів аналізуються результати, в результаті чого виявляються помилки в ПС.

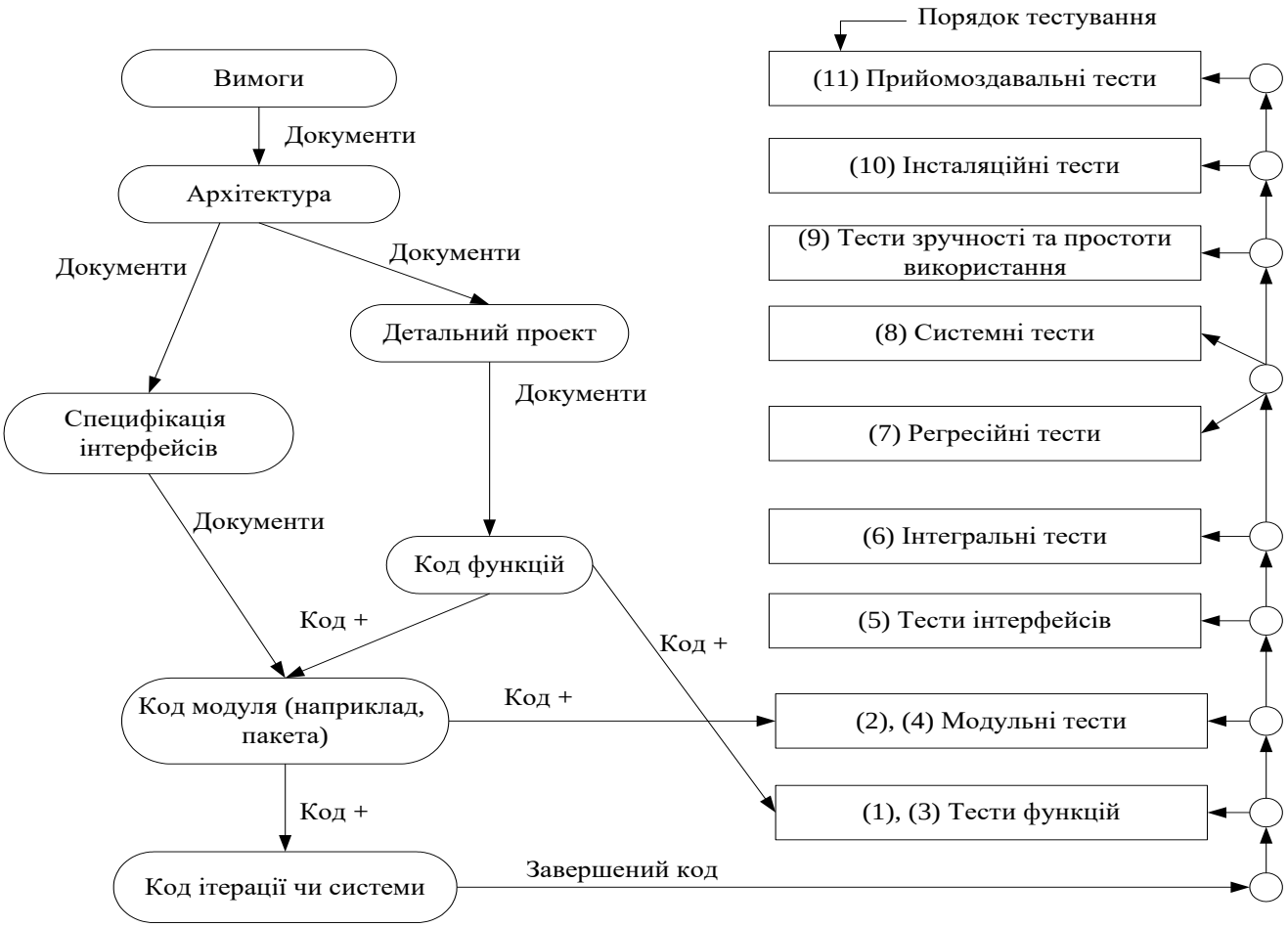


Рис.1.7. Огляд процесу тестування. Етапи та артефакти тестування

Тестувати можна на дотримання будь-яких вимог, відповідність яким перевіряється під час роботи ПЗ. З поміж характеристик якості згідно ISO/IEC 9126 цієї властивості не мають тільки атрибути зручності супроводу. Тому виділяють види тестування, пов'язані з перевіркою певних характеристик і атрибутів якості — тестування функціональності, надійності, зручності використання, переносимості і продуктивності, а також тестування захищеності, функціональної придатності та ін. Крім того, особливо виділяють навантажувальне або стресове тестування, що перевіряє працездатність ПС і показники її продуктивності в умовах підвищених навантажень — при великій кількості користувачів, інтенсивному обміні даними з іншими системами, великому обсязі переданих або використовуваних даних тощо.

1.7.1. Принципи організаційного тестування

Тест є хорошим тільки в тому випадку, в якому існує висока

ймовірність виявити помилку. Ця аксіома є фундаментальним принципом тестування. Оскільки неможливо довести, що програма не має помилок і, отже, всі такі спроби безплідні, процес тестування має являти собою спробу виявити в програмі раніше не знайдені помилки.

Одна з найскладніших проблем під час тестування — вирішити, коли потрібно його зупинити. Повне тестування (тобто випробування на всіх вхідних значеннях) неможливе. Таким чином, в процесі тестування відбувається зіткнення з економічною проблемою: як обрати скінченне число ТНД, яке дає максимальну віддачу (ймовірність виявлення помилок) для заданого об'єму витрат. Відомо дуже багато випадків, коли створені ТНД мали вкрай малу ймовірність виявлення нових помилок, в той час як досить очевидні хороші тести залишалися непоміченими.

Слід уникати тестування програми її автором. Жоден програміст не повинен намагатися тестувати свою власну програму. Це відноситься до всіх форм тестування: як до тестування ПС, так і до тестування зовнішніх функцій, інтерфейсів і, навіть, окремих модулів. Тестування повинне бути надзвичайно руйнівним процесом, але є глибокі психологічні причини, за якими програміст не може ставитися до своєї власної програмою як руйнівник. Додатковий тиск (наприклад, жорсткий графік) впливає на окремого програміста, або весь колектив розробників проекту часто заважає програмісту, або всьому колективу виконати адекватне тестування. Якщо модуль містить дефекти внаслідок якихось помилок перекладу, досить висока ймовірність того, що програміст допустить ту ж помилку перекладу (наприклад, неправильно інтерпретує специфікації) і при підготовці тестів. Всі помилки в його розумінні інших модулів і їх сполученні також позначаться на його тестових випадках.

Звідси не випливає, що програміст не може тестувати свою програму. Багато програмістів з цим цілком успішно справляються. Тут лише робиться висновок про те, що тестування є більш ефективним, якщо воно виконується будь-ким іншим. Ці міркування не мають відношення до налагодження, тобто до виправлення вже відомих помилок. Така робота ефективніше виконується

самим автором програми.

Необхідна частина будь-якого тесту — опис очікуваних вихідних даних або результатів. Одна з найпоширеніших помилок тут полягає в тому, що результати кожного тесту не прогнозуються до його виконання. Очікувані результати потрібно визначати заздалегідь, щоб не виникла ситуація, коли око бачить те, що хоче побачити. Щоб зовсім виключити таку можливість, краще користуватися інструментами тестування, здатними автоматично звіряти очікувані і фактичні результати.

Іноді, наприклад, при тестуванні математичного програмного забезпечення, допускаються виключення. Математичні програми володіють такою властивістю, що вихідні дані є тільки наближенням правильного результату. Це відбувається через використання обчислювальних процесів замість нескінченних математичних процесів, через помилки округлення, пов'язаних з кінцевою точністю машинної арифметики і неточністю уявлення чисел, а також помилок через скінченну точність представлення вхідних даних і констант. Тому в багатьох випадках виявляється важливим не абсолютна точність, а кореляція помилок. Наприклад, коли математична програма повертає масив чисел, може виявитися важливим, щоб обчислене рішення було точним рішенням для набору вхідних даних, що апроксимує реальні вихідні дані. Тому під час тестування математичного ПЗ прогнозування точних вихідних даних є нетрівіальною задачею.

Тести для неправильних і непередбачуваних вхідних даних слід розробляти так само ретельно, як для правильних і передбачуваних. Під час тестування ПЗ є природна тенденція концентрувати увагу на правильних і передбачуваних вхідних умовах, а неправильним і непередбачуваним вхідним даним приділяється мало уваги. Наприклад, в разі тестування завдання про трикутники, лише небагато хто зможе привести в якості тесту довжини сторін 1, 2 і 5, щоб переконатися в тому, що трикутник ***не буде*** помилково інтерпретований як нерівносторонній. Багато помилок, які потім несподівано виявляються в працюючих програмах, проявляються внаслідок не

передбачених дій користувачів. Тести, що представляють несподівані або неправильні вхідні дані, часто краще виявляють помилки, ніж правильні тести.

Необхідно перевіряти не тільки, чи виконує програма те, для чого вона призначена, але чи не робить вона те, що не повинна робити. Це логічно випливає з попереднього принципу. Необхідно перевіряти ПЗ на небажані побічні ефекти. Наприклад, програма розрахунку зарплати, яка виробляє правильні платіжні чеки, виявиться невірною, якщо вона зробить зайві чеки для працюючих або двічі запише перший запис в перелік особового складу підприємства.

Слід детально вивчати результати кожного тесту. Найвитонченіші тести нічого не варті, якщо їх результати удостоюються лише побіжного погляду. Тестування програми означає більше, ніж виконання достатньої кількості тестів; воно також передбачає вивчення результатів кожного тесту.

Не слід викидати тести, навіть якщо програма вже не потрібна. Ця проблема найчастіше виникає в разі використанні інтерактивних систем налагодження. Після внесення змін або виправлення помилок необхідно повторювати тестування, тоді доводиться заново винаходити тести. Як правило, цього намагаються уникати, оскільки повторне створення тестів потребує значної роботи. В результаті повторне тестування буває менш ретельним, ніж початкове, тому якщо модифікація стосувалася функціональної частини програми, і при цьому була допущена помилка, то вона часто може залишитися непоміченою.

Не можна планувати тестування в припущенні, що помилки не будуть виявлені. Таку помилку зазвичай допускають керівники проекту, що використовують невірне визначення тестування як процесу демонстрації відсутності помилок у ПЗ, тобто коректного функціонування програм.

По мірі того, як число помилок, виявлених в деякому компоненті програмного забезпечення, збільшується, зростає відносна ймовірність існування в цьому компоненті ще невиявлених помилок. Помилки утворюють кластери, тобто містяться групами. З ростом числа помилок, виявлених в

деякому компоненті програми (наприклад, у модулі, підсистемі, функції користувача), збільшується також ймовірність існування в цьому ж компоненті ще не виявлених помилок. Якщо під час тестування двох модулів в них виявлені одна і вісім помилок відповідно, крива показує, що для модуля з вісьмома помилками ймовірність того, що в ньому є ще помилки, вище.

Тестування, як майже будь-яка інша діяльність, має починатися з постановки цілей. Цикл тестування подібний повному циклу розробки ПЗ. Тести повинні бути спроектовані, реалізовані, перевірені і, нарешті, виконані. Тому завдання тестування повинні бути сформульовані на кожному його етапі, наприклад, для кожного конкретного типу тестування повинні бути визначені орієнтири (число пройдених шляхів, перевірених умовних переходів, частка помилок, які повинні бути виявлені на цьому етапі).

Тестування — процес творчий. Для тестування великої програми або ПС потрібно мати більш творчий потенціал, ніж для її проектування.

1.7.2. Планування тестування

Питання, що визначають процес планування

Процес тестування знаходиться в прямій залежності від процесу розробки ПЗ, але сильно відрізняється від нього, оскільки переслідує інші цілі. Розробка орієнтована на побудову ПП, тоді як тестування відповідає на питання, чи відповідає програмний продукт, що розроблюється, вимогам, де зафіксовано початковий задум розробки (тобто те, що замовив замовник).

Разом обидва процеси охоплюють види діяльності, необхідні для отримання якісного ПП. Помилки можуть бути привнесені на кожній стадії розробки. Тому кожному етапу розробки повинен відповідати етап тестування. Відносини між цими процесами такі, що якщо щось розробляється, то воно піддається тестуванню, а результати тестування використовуються для визначення, чи відповідає це набору пропонованих вимог. Процес тестування повертає виявлені ним помилки в процес розробки. Процес розробки передає процесу тестування нові і виправлені проектні версії. Вочевидь планування тестування теж залежить від обраної моделі розробки.

Однак незалежно від моделі розробки під час планування тестування необхідно відповісти на п'ять питань, які визначають цей процес:

- хто буде тестувати і на яких етапах? Розробники продукту, незалежна група тестувальників або спільно;

- які компоненти треба тестувати? Чи будуть піддані тестуванню всі компоненти програмного продукту або тільки ті компоненти, які загрожують найбільшими втратами для всього проекту;

- коли треба тестувати? Чи буде це безперервний процес, або вид діяльності, що здійснюється в спеціальних контрольних точках, або вид діяльності, що здійснюється на завершальній стадії розробки;

- як треба тестувати? Чи буде тестування зосереджено тільки на перевірці того, що даний ПП повинен виконувати, або також на тому, як це реалізовано;

- в якому обсязі тестувати? Як визначити, чи в достатньому обсязі виконано тестування, або як розподілити обмежені ресурси під тестування.

Всі відповіді на поставлені питання і багато іншого фіксується в документі ***Тест план.***

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

Тест план містить:

1) перелік тестових ресурсів;
2) перелік функцій і підсистем, що підлягають тестуванню;
3) тестову стратегію:
   - аналіз функцій і підсистем з метою визначення слабких місць, що вимагають вичерпного тестування, тобто ділянок функціональності, де поява дефектів є найбільш ймовірною;
   - визначення стратегії вибору ТНД для тестування. Оскільки в реальних додатках множина вхідних даних ПП практично нескінченна, вибір скінченної підмножини для проведення

тестування є складним завданням. Для її вирішення можуть бути застосовані методи покриття класів вхідних і вихідних даних, аналіз граничних значень, покриття випадків використання і т.п. Обрана стратегія повинна бути обгрунтована і задокументована;

- визначення потреби в автоматизації процесу тестування. При цьому рішення про використання існуючої, або про створення нової автоматизованої системи тестування повинно бути обгрунтованим, а також продемонстрована оцінка витрат на створення нової системи або на впровадження вже існуючої.

4) графік (розклад) тестових циклів;

5) вказівку конкретних параметрів апаратури, операційних середовищ та програмного оточення;

6) визначення тестових метрик, які необхідно збирати і аналізувати, таких як: покриття набору вимог, покриття коду, кількість і рівень серйозності дефектів, обсяг тестового коду тощо.

Тест план повинен як мінімум відповідати на такі питання:

1) **що треба тестувати?** Опис об'єкта тестування: системи, додатка, обладнання;
   що будете тестувати? Список функцій і опис тестованої системи і її компонент окремо;

2) **як будете тестувати?** стратегія тестування, а саме: методології, види тестування і їх застосування по відношенню до тестованого об'єкту, пріоритети, автоматизація та ін .;

3) **коли будете тестувати?** Послідовність проведення робіт: фази, цикли тестування, процедура тестування — підготовка (Test Preparation), тестування (Testing), аналіз результатів (Test Result Analisys) в розрізі запланованих фаз розробки;

4) **де будете тестувати?**

- оточення тестованої системи — опис;
- необхідне для тестування обладнання та програмні засоби.

5) **хто буде тестувати?**
   - ролі і обов'язки;
   - імена і прізвища.

6) **критерії початку тестування**:
   - готовність оточення;
   - готовність тест кейсів;
   - закінченість розробки необхідного функціоналу;
   - виконання юніт-тестів;
   - білд побудований і задовольняє певним критеріям.

7) **критерії закінчення тестування:**
   - результати тестування задовольняють певним критеріям;
   - вимоги до кількості відкритих багів виконані (визначаються заздалегідь).

8) **критерії передачі системи для приймального тестування:**
   - приймальні тести — де зберігаються і коли виконуються;
   - процедура приймання.

9) **які ризики існують і як ми ними будемо керувати?** Ризики та їх дозвіл.

10) **метрики і звіти:**
    - метрики продукту — хто збирає, з якою періодичністю;
    - звіти - хто створює, кому відправляє, і т.п.

11) **розклад білдів.**

Відповівши в тест-плані на ці запитання, можна вважати, що на руках є хороша чернетка документа з планування тестування. Далі, щоб документ набув більш серйозного вигляду, слід доповнити його наступними пунктами:

- оточення тестованої системи (опис програмно-апаратних засобів);

- необхідне для тестування обладнання та програмні засоби (тестовий стенд, програмні платформи для автоматизованого тестування);
- ризики та шляхи їх вирішення.

Види тест-планів

Найчастіше доводиться стикатися з такими видами тест-планів:

1) **майстер тест план (Master Plan або Master Test Plan);**
2) **тест план (Test Plan);**
3) **план приймальних випробувань (Product Acceptance Plan)** — документ, що описує набір дій, пов'язаних з приймальним тестуванням: стратегія, дата проведення, відповідальні і т.д .;
4) **план автоматизації (Test Automation Plan)** — документ опису дій з автоматизації тестування: стратегія, правила, відповідальні і т.д.

Явна відмінність Master Test Plan від просто Test Plan в тому, що він є більш статичним в силу того, що містить у собі високорівневу інформацію, яка не схильна до частої зміни в процесі тестування і перегляду вимог. Сам же детальний тест план, який містить більш конкретну інформацію по стратегії, видам тестування, розкладом виконання робіт, є "живим" документом, який постійно зазнає змін, що відображають реальний стан речей на проекті.

У повсякденному житті на проекті може бути один Майстер Тест План і кілька детальних тест планів, що описують окремі модулі однієї програми.

1.8 Тестовий випадок (Test case). Види. Структура.

Тестовий випадок (Test Case) – це:

- мінімальний елемент тестування (всього одна верифікація, тобто перевірка);
- сукупність кроків, конкретних умов і параметрів, необхідних для перевірки реалізації тестованої функції або її частин;
- опис певних дій і умов, які необхідні для того, щоб виявити той чи інший баг (дефект).

Під тест-кейсом розуміється структура виду: Action> Expected Result>

Test Result. У таблиці 1.1 показаний приклад test case.

Таблиця 1.1. Приклад тестового випадку

Action	**Expecred Result**	**Test Result (passed/failed/blocked)**
Open page "Login"	Login page is opened	Passed

Види тестових випадків

Тест-кейси поділяються за типом очікуваного результату на позитивні і негативні:

- позитивний тест-кейс використовує тільки коректні дані і показує, що додаток правильно виконав функцію, що викликається;
- негативний тест-кейс оперує як коректними, так і некоректними даними (мінімум один некоректний параметр) і ставить за мету перевірку виняткових ситуацій (спрацьовування валідаторів), а також перевіряє, що викликана додатком функція не виконується коректно при спрацьовуванні валідатора.

Структура Тестових Випадків (Test Case Structure)

Кожен тест кейс повинен складатися з трьох частин. У таблиці 1.2 показані ці частини.

Таблиця 1.2. Структура Test case

Pre conditions	Список дій, які призводять систему до стану придатного для проведення основної перевірки. Або список умов, виконання яких говорить про те, що система знаходиться в придатному для проведення основного тесту стану.
Test case description	Список дій, які переводять систему з одного стану в інший, для отримання результату, на підставі якого можна зробити висновок про задоволенні реалізації, поставленим вимогам
Post conditions	Список дій, які переводять систему в первинний стан (стан до проведення тесту - initial state)

Примітка: Post Conditions не є обов'язковою частиною. Ця частина *актуальна* при автоматизованому тестуванні, коли за один прогін можна наповнити базу даних сотнею або навіть тисячею некоректних документів.

Приклад тест-кейса

do A1, verify B1; do A2, verify B2; do A3, verify B3

У наведеному прикладі кінцева перевірка B3. Це означає, що саме вона є ключовою. Значить, A1 і A2 – це дії, що призводять систему в тестопригодний стан. Тоді B1 і B2 – умови того, що система знаходиться в стані придатному для тестування. Таким чином, в таблиці 1.3 показано умова тест-кейса.

Таблиця 1.3. Умова тест кейса

Action	**Expected Result**	**Test Result (passed/failed/blocke**
Preconditions		
do A1	verify B1	
do A2	verify B2	
Test Case Description		
do A3	verify B3	
Postconditions		

PostConditions в даному прикладі не були описані, але за логікою речей треба виконати кроки, які б повернули систему в первинний стан (наприклад, видалили створені записи, або скасували б зміни зроблені в документі).

Потрібно відповісти на питання: "Чому дане розбиття зручно використовувати?" Відповідь в табл.1.4: кінцева перевірка одна, тобто в разі якщо тест провалений (test failed) буде відразу ясно, через що. Оскільки якщо провальними виявляться перевірки B1 і/або B2, то тест кейс буде заблокований (test blocked), через те, що функцію неможливо привести в тестопригодний стан (стан придатний для проведення тестування), але це не означає, що тестована опція спрацює.

Таблиця 1.4. Один з варіантів результату

Action	**Expected Result**	**Test Result (passed/failed/blocked)**
Preconditions		
do A1	verify B1	passed
do A2	verify B2	*failed*
Test Case Description		
do A3	verify B3	**blocked**
Postconditions		

Деталізація опису тест кейсів

Рівень деталізації тест-кейсів повинен бути таким, щоб забезпечувати розумне співвідношення часу проходження до тестового покриття. Тобто, поки покриття тестами певного функціоналу не змінюється, можна зменшувати деталізацію тест-кейсів. У таблиці 1.5 можна побачити приклад деталізації тест кейса:

Таблиця 1.5. Приклад тест кейса 1

Перевірка відображення сторінки		
Дія	**Очікуваний результат**	**Результат тесту**
Відкрити сторінку "Вхід в систему"	- вікно "Вхід в систему" відкрито; - назва вікна - Вхід в систему; - логотип компанії відображається в правому верхньому куті; - на формі 2 поля - Ім'я та Пароль; - кнопка Вхід доступна; - посилання "забув пароль" - доступна.	...

Приклад тест-кейса 2:

Назва: Перевірка відображення сторінки. **Дія**: Відкрити сторінку "Вхід в систему". **Перевірка:** Перевірте, що відображається сторінка відповідає сторінці на зображенні 1 (і додаємо зображення сторінки "Вхід в систему").

У прикладі 1 і 2 покриття буде однаковим, але ось час, який буде потрібно для проходження, буде різним. Другий приклад буде наочнішим.

Атрибути тест-кейса

У таблиці 1.6 представлені часто використовувані атрибути тест-кейса.

Таблиця 1.6. Атрибути тест-кейса

Атрибут тест кейса	Опис
Test Case ID	Унікальне значення в межах не тільки документа, але і всієї фірми <
Test Case Priority	Пріоритет. Вимірюється від 1 до n 1 - найвищий n - найнижчий (для не дуже великих проектів раціонально вибирати n = 4) < IDEA Опис ідеї, що перевіряється тест кейсом SETUP and ADDITIONAL INFO Вхідні параметри <
IDEA	Опис ідеї, що перевіряється тест кейсом
SETUP and ADDITIONAL INFO	Вхідні параметри <
Revision History	Історія редагування. Приклад формату: Created on <date> by <name> Modified on <date> by <name> Change - які зміни і навіщо вони
Execution Part	Здійсненне частина тест кейса. Приклад формату: Action - список команд EXPECTED RESULT - очікуваний результат TEST RESULT - (passed, failed, blocked)

Приклад тест кейса: нехай для тестування можливості оплати послуги на сайті необхідно виконати наступний алгоритм:

1) відкрити vk.com; 2) ввести в поле "Ім'я користувача" значення "user";

3) ввести в поле "Пароль" значення "password"; 4) натиснути кнопку "Увійти";

5) ввести в поле "Шукати людей" значення "Петро Петров";

6) натиснути кнопку "Пошук"; 7) натиснути на сторінку з Петро Петров;

8) в сторінці натиснути на посилання "Відправити подарунок";

9) в сторінці натиснути на будь-яке посилання з подарунком;

10) у сторінці, натиснути на кнопку "Подарувати";

11) у сторінці, натиснути на посилання "Банківські картки";

12) ввести в поле "Номер карти" значення карти VISA "1111-1111-1111-1111";

13) ввести в поле "Дійсне" значення "07/21"; 14) ввести в поле "CVC" значення "111";

15) натиснути кнопку "Оплатити"; 16) записати номер замовлення;

17)запросити базу даних "select res from payment where id = <номер замовлення>".

Очікуваний результат: "10".

У таблиці 1.7 можна побачити як для даного прикладу буде виглядати тест-кейс. Перевага такої структури тест-кейса на відміну від початкового списку полягає в тому, що є можливість протестувати за тим же сценарієм інші дані (наприклад: user2, password2, Олександр Гомель, 2222-2222-2222-2222).

Для виконання одного і того ж сценарію на N наборах тестових даних створюється N тестів.

Цьому правилу необхідно слідувати. Таким чином, щоб зробити подарунок Івану Іванову, потрібно скопіювати вміст тест-кейса VV12345, наприклад, в тест-кейс VV12346 і переписати тільки вхідні параметри.

Такий вид тест кейса називається керований даними (data-driven).

Проблеми сценарію в прикладі:

- пункти 1-4 можуть змінюватися у зв'язку з міграцією сайту на новий домен, зміною інтерфейсу і т.д .;

- пошук збігу в пунктах 5-7 "Петро Петров" може привести в нікуди в разі видаленні сторінки;

- пункти 8-15 можуть бути легко змінені за рахунок нового дизайну сайту.

Отже, при будь-якій такій зміні доведеться переписувати весь тест-кейс, щоб заново протестувати первинне завдання.

Таблиця 1.7. Тест кейс для прикладу

Test Case ID/Priority	VV12345	1
IDEA: Оплата карткою VISA "подарунка одному" **SETUP and ADDITIONAL INFO**: Аккаунт: user / password Ім'я одного: Петро Петров Номер карти: 1111-1111-1111-1111 Термін дії: 07/15 CVC: 111 Запит SQL: select res from payment where id = <номер замовлення> <		
Revision History		
Created on 23/01/2014 by Иван Иванов	Новий тест кейс	
Execution Part		
ACTION	**EXPECTED RESULT**	**TEST RESULT**

1) відкрити vk.com; 2) ввести в поле "Ім'я користувача" значення "user"; 3) ввести в поле "Пароль" значення "password"; 4) натиснути кнопку "Увійти"; 5) ввести в поле "Шукати людей" значення "Петро Петров"; 6) натиснути кнопку "Пошук"; 7) натиснути на сторінку з Петро Петров; 8) в сторінці натиснути на посилання "Відправити подарунок"; 9) в сторінці натиснути на будь-яке посилання з подарунком; 10) у сторінці, натиснути на кнопку "Подарувати"; 11) у сторінці, натиснути на посилання "Банківські картки"; 12) ввести в поле "Номер карти" значення карти VISA "1111- 1111-1111-1111"; 13) ввести в поле "Дійсне" значення "07/21"; 14) ввести в поле "CVC" значення "111"; 15) натиснути кнопку "Оплатити"; 16) записати номер замовлення; 17) запросити базу даних "select res from payment where id = <номер замовлення> ".) <Номер замовлення> ".	**10**	

Якщо ж розбити задачу на декілька тест-кейсів, наприклад, за пункти 1-4 відповідатиме тест-кейс "Вхід в систему", за пункти 5-7 "Пошук друга", 8-15 "Оплата подарунка" (на внутрішньому рівні кожного з них можливий більш детальний поділ), то можна переписати тест-кейс так, як вказано в таблиці 1.8.

Можливий варіант, коли все, що потрібно — це виконання команди номер 5, за умови що інші команди виконані заздалегідь. У таблиці 1.9 вказано спрощений варіант тест-кейса.

Таблиця 1.8. Розбиття тест-кейса

Test Case ID/Priority	VV12347	1
IDEA: Оплата карткою VISA “подарунку другу” **SETUP and ADDITIONAL INFO:** Аккаунт: user/password Имя друга: Петр Петров Номер карти: 1111-1111-1111-1111 Срок действия: 07/15 CVC: 111 Запрос SQL: select res from payment where id = <номер заказа> <		
Revision History		
Created on 23/01/2014 **by** Марина Гончарова	Новий тест кейс	
Modified on 23/01/2014 **by** Иванов Евгений	Спрощення шагів	
Execution Part		
ACTION	**EXPECTED RESULT**	**TEST RESULT**
1) увійти в систему; 2) пошук друга; 3) оплата подарунка; 4) записати номер замовлення; 5) запросити базу даних "select res from payment where id = <номер замовлення>".	**10**	

Таблиця 1.9. Спрощення тест-кейса

Test Case ID/Priority	VV12348	1
IDEA: Підтвердження оплати **SETUP and ADDITIONAL INFO:** Номер замовлення: параметр		
Revision History		
Created on 23/01/2014 **by** Марина Гончарова	Новий тест кейс	
Modified on 23/01/2014 **by** Иванов Евгений	Спрошення шагів	
Modified on 24/01/2014 **by** Сергей Галкин	Зміна структури	
Execution Part		
ACTION	**EXPECTED RESULT**	**TEST RESULT**
Запросити базу даних "select res from payment where id = <номер замовлення>".	**10**	

Невідомий параметр <Номер замовлення> після завершення виконання тесту отримає своє значення, яке буде задокументовано.

РОЗДІЛ 2. ДЕФЕКТИ. ТЕХНІКИ СТВОРЕННЯ ТЕСТІВ ДЛЯ ЧОРНОГО ЯЩИКА

Помилками в програмному забезпеченні є всі можливі невідповідності між характеристиками його якості, які демонструються, та сформульованими (чи такими, що маються на увазі) вимогами, а також очікуваннями користувачів. В англомовній літературі використовуються кілька термінів, що часто однаково перекладають як "помилка" на нашу мову:

- **defect** — саме загальне порушення будь-яких вимог або очікувань, не обов'язково виявляється зовні (до дефектів відносяться порушення стандартів кодування, програмування, недостатня гнучкість системи та ін.);
- **failure** — спостережуване порушення вимог, що виявляється при якомусь реальному сценарії роботи ПЗ. Це можна назвати проявом помилки, що іноді призводить до збоїв у роботі ПС;
- **fault** — помилка в коді програми, що викликає порушення вимог при роботі (failures), та місце, яке треба виправити. Хоча це поняття використовується досить часто, воно, взагалі кажучи, не цілком чітке, оскільки для усунення порушення можна виправити програму в декількох місцях. Що саме треба виправляти, залежить від додаткових умов, виконання яких хочемо при цьому забезпечити, хоча в деяких ситуаціях накладення додаткових обмежень не усуває неоднозначність;
- **error** — використовується в двох сенсах. Перший сенс (або зміст) — це помилка в ментальній моделі програміста, помилка в його міркуваннях про програму, яка змушує його робити помилки в коді (**faults**). Це, власне, помилка, яку зробила людина в своєму розумінні властивостей програми.

Другий зміст — це некоректні значення даних (вихідних або внутрішніх), які виникають при помилках в роботі програми.

Ці поняття деяким чином пов'язані з основними джерелами помилок. Оскільки при розробці програм необхідно спочатку зрозуміти задачу, потім придумати її рішення і закодувати його в вигляді програми, то, відповідно, основних джерел помилок три:

- **неправильне розуміння завдань**. Розробники ПЗ не завжди розуміють, що саме потрібно зробити. Іншим джерелом нерозуміння слугує відсутність його у самих користувачів і замовників, досить часто вони можуть просити зробити трохи не те, що їм дійсно потрібно. Для запобігання неправильного розуміння завдань програмної системи служить аналіз предметної області;
- **неправильне рішення задач**. Найчастіше, навіть правильно зрозумівши, що саме потрібно зробити, розробники вибирають неправильний підхід до того, як це робити. Обране рішення може забезпечити лише деякі з необхідних властивостей, вони можуть добре підходити для даної задачі в теорії, але погано працювати на практиці, в конкретних обставинах, в яких має працювати ПЗ. Допомогти у виборі правильного рішення може зіставлення альтернативних рішень і ретельний їх аналіз на предмет відповідності всім вимогам, підтримання постійного зв'язку з користувачами та замовниками, надання їм необхідної інформації по обраних рішеннях, демонстрація прототипів, аналіз придатності обраних рішень для роботи в тому контексті, в якому вони будуть використовуватися;
- **неправильне перенесення рішень у код**. Маючи правильне рішення, правильно зрозуміле завдання, люди, проте, здатні зробити досить багато помилок при втіленні цих рішень. Коректним поданням рішень у коді можуть перешкодити як звичайні помилки, так і забудькуватість програміста або його небажання відмовитися від

звичних прийомів, які не дають можливості акуратно записати прийняте рішення. З помилками такого роду можна впоратися за допомогою інспектування коду, взаємного контролю, при якому розробники уважно читають код один одного, випереджаючої розробки модульних тестів і тестування.

У програмуванні **баг** (bug - жук) зазвичай позначає помилку в програмі або системі, яка видає несподіваний або неправильний результат. Більшість багів виникають через помилки розробників у вихідному коді, або проекті ПС. Також деякі баги виникають через некоректну роботу компілятора, який виробляє некоректний код. Програму, яка містить велику кількість багів і/або баги, що серйозно обмежують її функціональність, називають нестабільною, або на жаргонній мові «глючною», «забагованою» (unstable, buggy).

Термін «баг» зазвичай вживається стосовно помилок, які проявляють себе на стадії роботи, на відміну від помилок проектування або синтаксичних помилок. Звіт, що містить інформацію про баг, також називають звітом про помилку або звітом про проблему (bug report). Звіт про критичну проблему (crash), що викликає аварійне завершення програми, називають креш-репортом (crash report). «Баги» локалізуються і усуваються в процесі тестування і налагодження програми.

2.1. Система відстеження помилок

Система відстеження помилок (bug tracking system) — прикладна програма, розроблена з метою допомогти розробникам ПЗ (програмістам, тестувальникам і ін.) враховувати і контролювати помилки (баги), знайдені в програмах, побажання користувачів, а також стежити за процесом усунення цих помилок і виконання або невиконання побажань.

Головний компонент такої системи — база даних, що містить відомості про виявлені дефекти. Ці відомості фіксуються в звіті про помилку.

Звіт про помилки (Баг репорт) — це документ, что опісує сітуацію або послідовність Дій прізвела до некоректної роботи об'єкта тестування, Із зазначену причин и очікуваного результату.

Структура баг-репорта

Системи відслідковування помилок, пропонують різні поля для заповнення і структури опису помилок. У табл.2.1 надано шаблон баг- репорта.

Таблиця 2.1. Шаблон баг-репорта

Шапка	
Короткий опис (Summary)	Короткий опис проблеми, явно вказує на причину і тип помилкової ситуації.
Проект (Project)	Назва тестованого проекту
Компонент додатка (Component)	Назва частини або функції тестованого продукту
Номер версії (Version)	Версія, де була знайдена помилка
Серйозність (Severity)	Найбільш поширена п'ятирівнева система градації серйозності дефекту: – s1 Блокуючий (Blocker); – s2 Критичний (Critical); – s3 Значний (Major); – s4 Незначний (Minor); – s5 Тривіальний (Trivial).
Пріоритет (Priority)	Пріоритет дефекту: – p1 Високий (High); – p2 Середній (Medium); – p3 Низький (Low). (докладніше в розділі **Серйозність** і пріоритет помилки)
Статус (Status)	Статус бага. Залежить від процедури і життєвого циклу бага (bug workflow and life cycle)
Автор (Author)	Творець баг-репорта
Призначено на (Assigned To)	Ім'я співробітника, призначеного на вирішеня проблеми
Оточення	
ОС / Сервіс Пак та ін. / Браузера + версія / ...	Інформація про оточення, де був знайдений баг: операційна система, сервіс пак, для WEB-тестування - ім'я і версія браузера і т.д.
Опис	
Кроки відтворення (Steps to Reproduce)	Кроки, за якими можна легко відтворити ситуацію, яка призвела до помилки.
Фактичний Результат (Result)	Результат, отриманий після проходження кроків до відтворення
Очікуваний результат (Expected Result)	Очікуваний правильний результат
Доповнення	
Прикріплений файл (Attachment)	Файл з логами (журнал), скріншот або будь-який інший документ, який може допомогти прояснити причину помилки або вказати на спосіб вирішення проблеми.

Серйозність та пріорітет помилки

Різні системи відслідковування помилок пропонують різні шляхи опису серйозності і пріоритету баг-репорта, незмінним залишається лише зміст, вкладений в ці поля.

- **серйозність (Severity)** – атрибут, що характеризує вплив помилки на працездатність додатку;
- **пріоритет (Priority)** –атрибут, який вказує на черговість виконання завдання або усунення помилки. Це інструмент менеджера з планування робіт. Чим вище пріоритет, тим швидше потрібно виправити дефект.

Градація серйозності дефекту (severity):

- **s1 Блокучий (Blocker).** Блокуюча помилка, що приводить додаток у неробочий стан, в результаті якого подальша робота з тестованої системою або її ключовими функціями стає неможливою. Рішення проблеми необхідно для подальшого функціонування системи;
- **s2 Критична (Critical).** Критична помилка, неправильно працює ключова бізнес-логіка, діра в системі безпеки, проблема, яка призвела до тимчасового падіння сервера або приводить в неробочий стан деяку частину системи, без можливості вирішення проблеми, використовуючи інші вхідні точки. Рішення проблеми необхідно для подальшої роботи з ключовими функціями тестованої системи;
- **s3 Значна (Major).** Значна помилка, частина основної бізнес-логіки не функціонує належним чином. Помилка не критична або є можливість для роботи з тестованої функцією, використовуючи інші вхідні точки;
- **s4 Незначна (Minor).** Незначна помилка, що не порушує бізнес-логіку тестованої частини програми, очевидна проблема призначеного для користувача інтерфейсу.
- **s5 Тривіальна (Trivial).** Тривіальна помилка, яка не стосується

бізнес-логіки додатка, погано відтворена проблема, малопомітна через призначений для користувача інтерфейс, проблема сторонніх бібліотек або сервісів, проблема, що не надає ніякого впливу на загальну якість продукту.

Градація пріоритету помилки (Priority):

- **p1 Високий (High).** Помилка повинна бути виправлена якомога швидше, тому що її наявність є критичною для проекту;
- **p2 Середній (Medium).** Помилка повинна бути виправлена, її наявність не є критичною, але вимагає обов'язкового рішення;
- **p3 Низький (Low).** Помилка повинна бути виправлена, її наявність не є критичною, і не потребує термінового вирішення.

Порядок виправлення помилок за їх пріоритетам: High> Medium> Low.

2.2. Написання баг-репорта

Баг-репорт — це технічний документ, і в зв'язку з цим, мова опису проблеми повинна бути технічною. Необхідно використовувати правильну термінологію під час використання назв елементів призначеного для користувача інтерфейсу (editbox, lightbox, combobox, link, text area, button, menu, popup menu, title bar, system tray і т.д.), дій користувача (click link, press the button , select menu item і т.д.) і отриманих результатів (window is opened, error message is displayed, system crashed і т.д.).

Вимоги до обов'язкових полів баг-репорта

Обов'язковими полями баг-репорта є: короткий опис (Bug Summary), серйозність (Severity), кроки до відтворення (Steps to reproduce), результат (Actual Result), очікуваний результат (Expected Result). Нижче наведені вимоги і приклади щодо заповнення цих полів.

Короткий опис. В одній пропозиції потрібно вмістити сенс всього баг-репорта, а саме: коротко і ясно, використовуючи правильну термінологію сказати що і де не працює.

Наприклад:

- додаток зависає, при спробі збереження текстового файлу розміром

більше 50Мб;

- дані на формі "Профайл" не зберігаються після натискання кнопки "Зберегти".

Серйозність. У двох словах слід відзначити, що якщо проблема знайдена в ключовий функціональності програми і після її виникнення додаток стає повністю недоступним, а подальша робота з ним неможлива, то вона є блокуючою. Зазвичай блокуючі проблеми знаходяться під час первинної перевірки нової версії ПП (Build Verification Test, Smoke Test), тому що їх наявність не дозволяє повноцінно проводити тестування. Якщо ж тестування може бути продовжено, то серйозність даного дефекту критична. Щодо значних, незначних і тривіальних помилок питання залежить від ситуації.

Кроки до відтворення/Результат/Очікуваний результат. Дуже важливо чітко описати всі кроки, з вказанням всіх даних, що вводяться (імені користувача, даних для заповнення форми) і проміжних результатів.

Наприклад:

1) увійдіть в системи: Користувач Тестер1, пароль xxxXXX:
   - ви ввійшли в систему.
2) клікніть лінк Профайл: - сторінка Профайл відкрилася.
3) введіть Нове ім'я користувача: Тестер2;
4) натисніть кнопку Зберегти.

Результат. На екрані з'явилася помилка. Нове ім'я користувача не було збережено. **Очікуваний результат.** Сторінка профайл перевантажилась. Нове значення імені користувача збережено.

Основні помилки під час написання баг-репортів

Недостатність наданих даних. Не завжди одна і таж проблема проявляється при всіх впроваджених значеннях і для будь-якого користувача в системі, тому настійно рекомендується вносити всі необхідні дані в баг-репорт.

Визначення серйозності. Дуже часто відбувається або завищення, або заниження серйозності дефекту, що може привести до неправильної черговості

вирішення проблем.

Мова опису. Часто для опису проблеми використовується неправильна термінологія або складні мовні звороти, які можуть ввести в оману людину, відповідальну за вирішення проблеми.

Відсутність очікуваного результату. У випадках, якщо ви не вказали, яка має бути необхідна поведінка системи, ви витрачаєте час розробника на пошук цієї інформації, тим самим уповільнюючи виправлення дефекту. Ви повинні вказати пункт у вимогах, написаний тест-кейс або ж вашу особисту думку, якщо ця ситуація не була документована.

Заповнення полів баг-репорта

У таблиці 2.2 представлені основні поля баг-репорта і роль працівника, відповідального за заповнення даного поля. Жирним курсивом виділені обов'язкові для заповнення поля.

Таблиця 2.2. Заповнення полів баг репорта

Поле	**Відповідальний за заповнення поля**
Короткий опис (Summary)	Автор баг-репорта (зазвичай це Тестувальник)
Проект (Project)	Автор баг-репорта (зазвичай це Тестувальник)
Компонент додатка (Component)	Автор баг-репорта (зазвичай це Тестувальник)
Номер версії (Version)	Автор баг-репорта (зазвичай це Тестувальник)
Серйозність (Severity)	Автор баг-репорта (зазвичай це Тестувальник), роте даний атрибут може бути змінений вищестоящим менеджером
Пріоритет (Priority)	Менеджер проекту або менеджер відповідальний за розробку компонента, на який написаний баг-репорта
Статус (Status)	автор баг-репорта (зазвичай це Тестувальник), але багато систем баг-трекінгу виставляють статус за замовчуванням
Автор (Author)	Встановлюється за замовчуванням, якщо немає, то вказується ім'я автора
Призначено на (Assigned To)	Менеджер проекту або менеджер відповідальний за розробку компонента, на який написаний баг репорта
ОС / Сервис Пак и т.д. / Браузера + версія / ...	Автор баг-репорта (зазвичай це Тестувальник)
Кроки відтворення (Steps to Reproduce)	Автор баг-репорта (зазвичай це Тестувальник)
Фактичний Результат (Result)	Автор баг-репорта (зазвичай це Тестувальник)
Очікуваний результат (Expected Result)	Автор баг-репорта (зазвичай це Тестувальник)
Прикріплений файл (Attachment)	Автор баг-репорта (зазвичай це Тестувальник), а також будь-який член командної групи, який вважає, що прикріплені дані допоможуть у виправленні бага

Життєвий цикл бага

Рис.2.1 ілюструє життєвий цикл дефекту, прийнятий у багатьох значних компаніях. Баг може знаходиться в одному з представлених на рисунку станів.

Виявлений (submitted). Тестувальник знаходить баг і представляє його на розгляд в систему стеження за вадами. З цього моменту баг починає своє офіційне життя і про його існування знають необхідні люди.

Призначений (assigned). Тестувальник або провідний розробник розглядає баг і призначає цей напрямок комусь з команди розробників.

Виправлений (fixed). Розробник, якому було призначено виправлення дефекту, виправляє його і повідомляє про те, що завдання виконано.

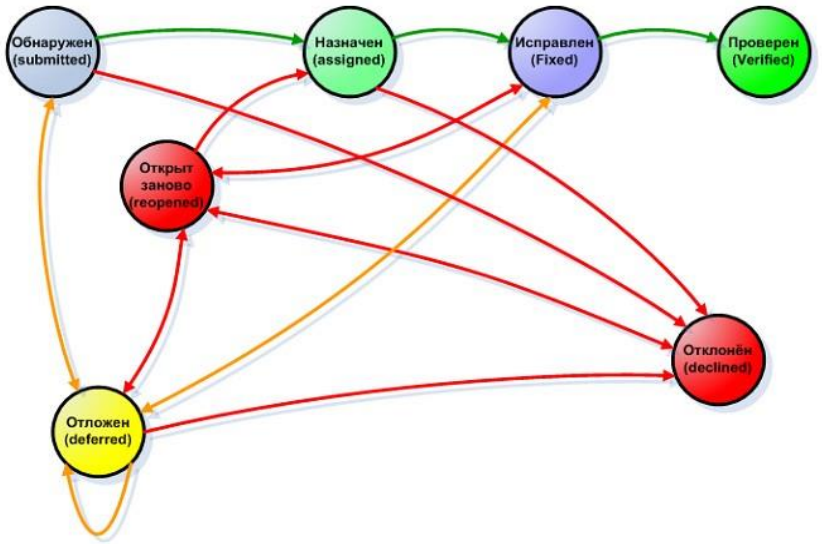


Рис. 2.1. Життєвий цикл бага

Перевірений (verified). Тестувальник, який виявив помилку перевіряє на новому білді (в якому виправлення цієї помилки заявлено), чи виправлений баг насправді. І тільки в тому випадку, якщо помилка не проявиться на новому білді, тестувальник змінює статус бага на Verified.

Відкритий заново (reopened). Якщо баг проявляється на новому білді, тестувальник знову відкриває цей дефект. Баг набуває статусу Reopened.

Відхилений (declined). Баг може бути відхилено. По-перше, тому що для замовника якісь помилки перестають бути актуальними. По-друге, це може трапитися з вини тестувальника через погане знання продукту, вимог (дефекту насправді немає).

Відкладений (deferred). Якщо виправлення конкретного бага зараз не дуже важливо, або замовник поки думає, чи ми чекаємо якусь інформацію, від якої залежить виправлення бага, тоді баг набуває статусу Deferred.

Закриті (closed) баги. Закритим вважається баг в станах Перевірений (verified) і Відхилений (declined).

Відкриті (open) баги. Відкритими є баги у станах Виявлений (submitted), Призначений (assigned), Відкритий заново (reopened). Іноді до відкритих відносять і баги в станах Виправлений (fixed) і Відкладений (deferred).

2.3. Методи тестування та створення тестових наборів даних

Метою тестування ставиться з'ясування обставин, в яких поведінка програми не відповідає специфікаціям. Для виявлення всіх помилок в програмі необхідно виконати вичерпне тестування, тобто тестування на всіх можливих ТНД. Для більшості програм таке неможливо, тому застосовують розумне тестування, при якому для тестування програми обмежуються невеликою підмножиною всіх можливих тестових наборів даних. При цьому необхідно вибирати найбільш підходящі підмножини, тобто підмножини з найвищою імовірністю виявлення помилок.

Властивості правильно обраного тесту:

- зменшує більш, ніж на одне число (іноді на порядок) кількість інших тестів, які повинні бути розроблені для розумного тестування;
- покриває значну частину інших можливих тестів, що в деякій мірі свідчить про наявність чи відсутність помилки до і після тестування на обмеженій множині ТНД.

Техніки чорного ящика:

- еквівалентне розбиття;
- аналіз граничних значень;
- аналіз причинно-наслідкових зв'язків;

- попарне тестування;
- припущення про помилку.

2.4. Еквівалентне розбиття

Основу методу еквівалентного розбиття складають два положення:

- вихідні дані необхідно розбити на скінченне число класів еквівалентності. В одному класі еквівалентності містяться такі тести, що, якщо один тест з деякого класу еквівалентності виявляє помилку визначеного типу, то і будь-який інший тест з цього класу еквівалентності повинен виявляти ту саму помилку;
- кожен тест повинен включати, по можливості, якомога більше класів еквівалентності, щоб мінімізувати загальне число тестів.

Розробка тестів цим методом здійснюється в два етапи: виділення класів еквівалентності і побудова тестів.

Еквівалентний клас — це непуста множина значень введення, до яких ПЗ застосовує однакову логіку.

Класи еквівалентності виділяються шляхом вибору кожної вхідної умови, що беруться з технічного завдання або специфікацій, і розбиваються на дві або декілька груп. Виділення класів еквівалентності є евристичним способом, однак є 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 (<0, >999);
- 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 множину вхідних значень, то кількість елементів у правильному класі дорівнює кількості елементів у множині вхідних значень. Якщо вхідна умова описує ситуацію «має бути», наприклад «Перший символ повинен бути з великої

літери», тоді один клас правильний і один неправильний;

- якщо є підстави вважати, що елементи всередині одного класу еквівалентності можуть програмою трактуватися по-різному, необхідно розбити даний клас на підкласи. На цьому кроці тестер на основі таблиці повинен скласти тести, що покривають собою все правильні і всі неправильні класи еквівалентності. При цьому тестувальник повинен мінімізувати загальне число тестів.

Кілька тестів є еквівалентними, якщо:

- вони спрямовані на пошук однієї і тієї ж помилки;
- якщо один з тестів виявляє помилку, інші її теж, скоріше за все, виявлять;
- якщо один з тестів не виявляє помилку, інші її теж, скоріше за все, не виявлять;
- тести використовують один і той же набір вхідних даних (ТНД);
- для виконання тестів доводиться здійснювати одні й ті ж операції;
- тести генерують однакові вихідні дані або приводять додаток в один і той же стан;
- всі тести призводять до спрацьовування одного й того ж блоку обробки помилок ("error handling block");
- жоден з тестів не призводить до спрацьовування блоку обробки помилок ("error handling block").

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 цих класів;
- якщо залишилися не включені в тести неправильні класи, то пишуть тести, кожний з яких покриває лише один такий клас.

2.5. Аналіз граничних значень

Граничні умови — це ситуації, що виникають на вищих і нижніх межах (границях) вхідних класів еквівалентності.

Аналіз граничних значень відрізняється від еквівалентного розбиття наступним:

- вибір будь-якого елементу в класі еквівалентності як представницького здійснюється таким чином, щоб перевірити тестом кожний кордон цього класу;
- під час розробки тестів розглядаються не тільки вхідні значення (простір входів), але й вихідні (простір виходів).

Метод аналізу граничних значень вимагає певною мірою творчості і високого рівня спеціалізації в задачі, що розглядається.

Існує кілька правил:

1) Необхідно побудувати тести з неправильними вхідними даними для ситуації незначного виходу за межі області значень. Якщо вхідні значення повинні бути в інтервалі [-1.0 .. +1.0], потрібно перевірити -1.0, 1.0, -1.000001, 1.000001;

2) обов'язково написати тести для мінімальної і максимальної межі діапазону значень кожного з класів еквівалентності;

3) використовувати перші два правила для кожного з вхідних значень (також використовувати пункт 2 для всіх вихідних значень);

4) якщо вхід і вихід програми представляє впорядкована множина даних, слід зосередити увагу на першому й останньому елементах списку цих даних.

Аналіз граничних значень, якщо він застосований правильно, дозволяє виявити велику кількість помилок. Однак визначення таких границь для кожного завдання може бути окремою важкою задачею. Також цей метод не перевіряє можливі комбінації вхідних значень.

2.6. Аналіз причинно-наслідкових зв'язків

Аналіз причинно-наслідкових зв'язків дозволяє системно обирати високорезультативні тести. Метод використовує алгебру логіки та оперує поняттями «причина» і «наслідок». Причиною в даному випадку називають окрему вхідну умову або клас еквівалентності. Слідством є вихідна умова або перетворення системи.

Ідея методу полягає у співвіднесенні всіх наслідків і причин, тобто в уточненні причинно-наслідкових зв'язків. Даний метод дає корисний побічний ефект, дозволяючи виявляти неповноту і неоднозначність формалізованих вихідних специфікацій.

Побудова тестів здійснюється в декілька етапів. Спочатку, оскільки таблиці причинно-наслідкових зв'язків при застосуванні методу до великих специфікацій стають громіздкими, специфікації розбивають на «робочі» ділянки, намагаючись по можливості виділяти в окремі таблиці незалежні групи причинно-наслідкових зв'язків. Потім для специфікації визначають множину причин і наслідків. Далі на основі аналізу семантичного (смислового) змісту специфікації будують таблицю істинності, в якій кожній можливій комбінації причин ставиться у відповідність наслідок. При цьому доцільно істину позначати як «True» (1), брехню як «False» (0), а для позначення байдужих станів умов застосовувати позначення «X», яке передбачає довільне значення умови (1 або 0).

Таблицю супроводжують примітками, які задають обмеження і описують комбінації причин і/або наслідків, які є неможливими через синтаксичні або зовнішні обмеження. При необхідності аналогічно будується таблиця істинності для класів еквівалентності. І, нарешті, кожен рядок таблиці перетворюють у тест. При цьому рекомендується по можливості поєднувати тести з незалежних таблиць.

Даний метод дозволяє будувати високорезультативні тести і виявляти неповноту і неоднозначність вихідних умов.

2.7. Попарне тестування

Pairwise testing — це техніка формування ТНД, де кожне тестоване значення кожного з перевіряємих параметрів хоча б один раз поєднується з кожним тестованим значенням усіх інших параметрів, що перевіряються.

Для демонстрації даної техніки візьмемо в якості прикладу абстрактну систему з великим числом параметрів, що впливають на її роботу, яку потрібно протестувати. Досвідчений тестувальник знає, що для перевірки всіх комбінації не вистачить часу. Наприклад, для перевірки всіх сполучень 10 параметрів з 10 значеннями кожен, потрібно 10 000 000 000 тестів, в той час як метод перебору пар дозволяє реалізувати порівнянне за якістю тестування (з огляду на кількість і критичність знайдених в результаті багів) використовуючи всього 177 тестів.

Метод попарного тестування заснований на досить простій, але від того не менш ефективним ідеї, що переважна більшість помилок виявляється тестом, що перевіряє один параметр, або поєднання двох. Помилки, причиною яких стали комбінації трьох і більше параметрів, як правило значно менш критичні, ніж пари параметрів або одного. Якщо цього вимагає ситуація, то можна доповнити тестове покриття тест-кейсами на бажані комбінації параметрів.

Перебрати всі пари не складно, складність полягає в тому, щоб забезпечити при цьому мінімум тестів, комбінуючи перевірки декількох пар в одному тесті. Математичні методи можуть забезпечити такий необхідний мінімум тестів. Одним з таких методів є ортогональні матриці.

Для розгляду того як відбувається оптимізація, візьмемо для прикладу табл.2.3 параметрів і значень.

Таблиця 2.3. Параметри та їх значення

Параметр 1	Параметр 2	Параметр 3
Значення 1.1	Значення 2.1	Значення 3.1
Значення 1.2	Значення 2.2	Значення 3.2

Переберемо значення першого параметра з другим (рядки №1-4), першого з третім (рядки №5-8) і другого з третім (рядки №9-12). Результат показаний в таблиці 2.4. Видаливши повторювані набори параметрів (виділені сірим),

отримаємо наступний результат, показаний в таблиці 2.5.

Таблиця 2.4. Результат перебору значень

#	Параметр 1	Параметр 2	Параметр 3
1	Значення 1.1	Значення 2.1	Значення 3.1
2	Значення 1.1	Значення 2.2	Значення 3.1
3	Значення 1.2	Значення 2.1	Значення 3.1
4	Значення 1.2	Значення 2.2	Значення 3.1
5	Значення 1.1	Значення 2.1	Значення 3.1
6	Значення 1.1	Значення 2.1	Значення 3.2
7	Значення 1.2	Значення 2.1	Значення 3.1
8	Значення 1.2	Значення 2.1	Значення 3.2
9	Значення 1.1	Значення 2.1	Значення 3.1
10	Значення 1.1	Значення 2.1	Значення 3.1
11	Значення 1.1	Значення 2.2	Значення 3.1
12	Значення 1.1	Значення 2.2	Значення 3.2

Таблиця 2.5. Скорочений результат перебору

#	Параметр 1	Параметр 2	Параметр 3
1	Значення 1.1	Значення 2.1	Значення 3.1
2	Значення 1.1	Значення 2.2	Значення 3.1
3	Значення 1.2	Значення 2.1	Значення 3.1
4	Значення 1.2	Значення 2.2	Значення 3.1
5	Значення 1.1	Значення 2.1	Значення 3.2
6	Значення 1.2	Значення 2.1	Значення 3.2
7	Значення 1.1	Значення 2.2	Значення 3.2

Зеленим виділені унікальні пари в таблиці. Значення виділені білим не є необхідними для перебору всіх пар в таблиці, тому можуть бути замінені на будь-яке інше значення. Замінивши їх можна оптимізувати тести, додавши перевірку пар з 5, 6 і 7 рядків у другий і третій рядки. У таблиці 2.6 показаний результат оптимізації.

Таблиця 2.6. Результат оптимізації

#	Параметр 1	Параметр 2	Параметр 3
1	Значение 1.1	Значение 2.1	Значение 3.1
2	Значение 1.1	Значение 2.2	Значение 3.2
3	Значение 1.2	Значение 2.1	Значение 3.2
4	Значение 1.2	Значение 2.2	Значение 3.1

Оптимізація навіть такого малого набору параметрів не так проста. При цьому складність завдання зростає пропорційно зростанню числа параметрів.

2.8. Припущення про помилку

Програміст з великим досвідом вишукує помилки без всяких методів, але при цьому він підсвідомо використовує метод припущення про помилку. Даний метод в значній мірі заснований на інтуїції. Основна ідея методу полягає в тому, щоб скласти список, який перераховує можливі помилки і ситуації, в яких ці помилки можуть проявитися. Потім на основі списку складаються тести.

2.9. Застосування еквівалентного розбиття і граничної умови

Застосування даних методів розглянемо на прикладі додатка, який на вхід приймає рядок, але за змістом рядок має інтерпретуватися як число. На рис.2.2 показана програма, яка є онлайн-калькулятором для перекладу одиниць часу.

Рис.2.2. Онлайн-калькулятор часу

Всі можливі рядки можна розділити на два великі класи: "числа" і "не числа". Але даними класами можна задати інші, довші, але при цьому більш точні назви:

- множини рядків, які програма інтерпретує як числа;
- множини рядків, які програма не може інтерпретувати як числа.

У цьому формулюванні стає ясно, що програма на вхід отримує рядок, який перетворюється у число. Якщо це перетворення пройшло успішно, то отримане число використовується в обчисленнях. Але якщо перетворити рядок в число не вдалося, то отримуємо інформацію про це або у вигляді повідомлення про проблему, або у вигляді безглуздого результату обчислень.

Можна визначити, як саме перетворювач часу поводиться в тій і в іншій ситуації, для цього достатньо подати на вхід якесь значення, яке тлумачитиметься як число, наприклад, "5". На рис.2.3 зображено введення вхідного значення.

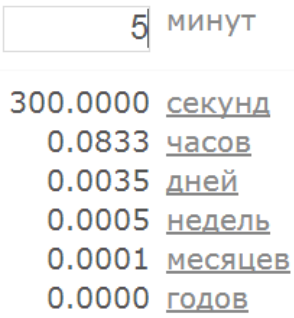


Рис.2.3. Вхідне значення 5 хвилин

Введемо також значення, яке точно не є числом, наприклад, "test". На рис.2.4 показано введення вхідного нечислового значення.

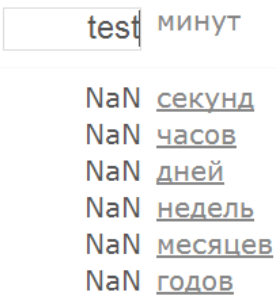


Рис.2.4. Вхідне значення не число

Спостерігається випадок, коли невалідність вхідного значення призводить до безглуздого результату (NaN означає "Not a Number").

Потрібно визначити, які рядки програма буде інтерпретувати як числа, тобто виділити в великих класах підкласи меншого розміру, але зате описані конструктивно. Рядок, що представляє собою послідовність цифр, інтерпретується як ціле число. Потрібно зрозуміти, якої довжини послідовність цифр інтерпретується як число. Щоб це зробити, необхідно застосувати техніку розбиття на підобласті. Мінімальна довжина послідовності — нуль. Максимальна довжина — "максимально можлива".

В даному додатку не вказано жодних обмежень на розмір поля введення. Є кілька ситуацій:

- можливо браузер накладає яке-небудь обмеження, однак тестеру про це нічого не відомо, якщо воно є, то в різних браузерах воно різне;
- якщо введені дані передаються на сервер у вигляді GET-запиту, можливо, є обмеження на довжину запиту — відповідно до стандарту RFC 2068 вони повинні підтримувати не менше 255 байтів, але реально здатні обробляти запити більшої довжини, і це залежить від браузера і від веб-сервера.

Конвертер, який використовується в якості прикладу, реалізований на мові JavaScript, на сервер ніяких даних не відправляється, всі обчислення виконуються браузером. Google Chrome успішно впорався з рядком, що складається з 10 000 000 дев'яток, рядок з 100 000 000 дев'яток він обробити вже не може — з'являється помилка з пропозицією перезавантажити сторінку. Отже, між цими значеннями і знаходиться максимальна довжина. Тому потрібно уточнити підклас: “Рядок, що представляє собою послідовність цифр, інтерпретується як ціле число, якщо довжина рядка не перевищує деяке Максимальне Значення”.

Тестові випадки на істотно більш коротких послідовностях (з 1000 дев'яток) під час обчислень призвели до переповнення, проте Infinity — це не NaN, тобто згідно описаному вище вважається, що така послідовність (а також і більш довгі послідовності цифр) може вважатися числом.

Послідовність нульової довжини — це порожній рядок. Додаток інтерпретує порожній рядок як число нуль. На рис.2.5 зображена дана інтерпретація.

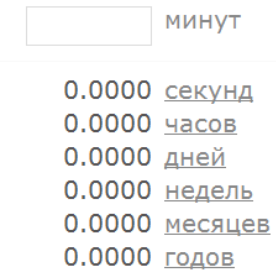


Рис.2.5. Послідовність нульової довжини

Перший підклас, послідовність цифр, створений. Другий підклас — послідовність переставлень цифр. "Не цифрами" можуть бути пробіли на початку і в кінці, а також провідні нулі. Вони обрізаються, а решта рядка інтерпретується як число. Тому:

Рядок, що інтерпретується як число, також є числом, якщо додати на початку кілька нулів, при цьому провідні нулі ігноруються. Рядок, що інтерпретується як число, також є числом, якщо додати на початку або в кінці кілька пробілів, при цьому всі пробіли ігноруються.

По-перше, важливо не забувати про максимальне значення довжини, якщо додати занадто багато нулів або пробілів, рядок перестане бути числом. По-друге, додавати спочатку нулі, а потім пробіли можна, а навпаки — ні.

Рядок, що інтерпретується як число, також інтерпретується як число, якщо додати на початку знак мінус або плюс. Негативні значення і нецілі числа можуть ставати новими підкласами.

Тому рядок (String), що складається з двох нерозривних ланцюжків цифр, розділених десятковою крапкою, інтерпретується як число.

Для нецілих чисел можна застосувати техніку розбиття на підобласті до кількості значущих цифр, або до кількості знаків після коми, в залежності від того, як інтерпретується поняття точності в конкретному додатку. Але при цьому слід зазначити, що для чисел з плаваючою точкою техніка розбиття на підобласті працює погано тому JavaScript реалізує стандарт IEEE-754.58.

Наступний підклас, який може представляти собою не послідовність цифр, але інтерпретуватися як число, може бути рядок, що складається з числа, за яким іде символ 'e', за яким слідує ціле число.

Рядок, що складається з символів '0x', за якими слідує послідовність шістнадцятирічних цифр, інтерпретується як шістнадцяткове ціле число.

Таким чином, отримані наступні підкласи, за якими можна починати тестувати додаток:

1) рядок, що представляє собою послідовність цифр, інтерпретується як ціле число, якщо довжина рядка не перевищує деяке Максимальне Значення;

2) рядок, що інтерпретується як число, також інтерпретується як число, якщо додати на початку кілька нулів, при цьому провідні нулі ігноруються;

3) рядок, що інтерпретується як число, також інтерпретується як число, якщо додати на початку або в кінці кілька пробілів, при цьому пробіли ігноруються;

4) рядок, що інтерпретується як число, також інтерпретується як число, якщо додати на початку знак мінус або плюс;

5) рядок, що складається з двох нерозривних ланцюжків цифр, розділених десятковою крапкою, інтерпретується як число;

6) рядок, що складається з числа, за яким слідує символ 'e', за яким слідує ціле число, інтерпретується як число;

7) рядок, що складається з символів '0x', за якими слідує нерозривна послідовність шістнадцятирічних цифр, інтерпретується як шістнадцяткове ціле число.

2.10. Попарне тестування

Метод ефективний на пізніх етапах розробки, або доповнений основними функціональними тестами. Якщо проводиться конфігураційне тестування, то перш ніж використовувати попарне тестування слід переконатися, що основний сценарій функціонує на всіх операційних системах з параметрами за замовчуванням (побудувати тест верифікації). Це значно полегшить

локалізацію майбутніх багів, адже при парному тестуванні в одному тесті фігурує множина параметрів зі значеннями не за замовчуванням, кожен з яких може стати причиною збою і його локалізація в цьому випадку має певні складнощі. Тому метод слід використовувати лише на стабільному функціоналі, коли поточні тести вже втрачають свою ефективність.

Щоб скористатися методом необхідно виконати кілька простих кроків:

Визначитися з функціональністю, яку потрібно перевірити

Для цього необхідно розділити функціональність на частини: компоненти, функції, сценарії. Функціональність невеликої програми, наприклад, по запису дисків, спрощено можна представити у вигляді всього двох сценаріїв: запис диска, стирання диска. Розглянемо на прикладі "запис диска" і перейдемо до наступного кроку.

Дослідити обраний сценарій і виявити його параметри та їх значення

На даному етапі необхідно виявити параметри сценарію, який можуть вплинути на його роботу. Як параметри можуть виступати як налаштування самої програми, так і зовнішні чинники.

Спрощено, параметри та їх значення для запису диска можна представити у вигляді таблиці 2.7.

Таблиця 2.7. Параметри та їх значення під час запису диска

Тип носія	Швидкість запису	Файлова система	Мультисесія	Об’єм даних
CD	2	ISO	Ні	100 Mb
DVD	4	UDF	Почати	700 Mb
	8	UDF/ISO	Продовжити	4,7 Gb
	16			
	24			
	36			
	52			

Звернемо увагу, що параметр «Швидкість запису» має значення, неприпустимі для DVD. Для цього потрібно розділити таблицю на дві: табл.2.8 і табл.2.9. Варто враховувати, що на практиці параметрів в цьому сценарії набагато більше, і нестиковок може бути значно більше.

Таблиця 2.8. Параметри та їх значення для CD

Швидкість запису	Файлова система	Мультисесія	Об’єм даних
2	ISO	Ні	100 Mb
4	UDF	Почати	700 Mb
8	UDF/ISO	Продовжити	
16			
24			
36			
52			

Таблиця 2.9. Параметри зі значеннями для DVD

Швидкість запису	Файлова система	Мультисесія	Обєм данних
2	ISO	Ні	100 Mb
4	UDF	Почати	4,7 Gb
8	UDF/ISO	Продовжити	
16			
24			

Під час вибору параметрів і значень слід пам'ятати, що негативні тести не варто включати в таблицю, бо в одному тесті може перевірятися декілька пар, а в разі негативного тесту буде виконана перевірка лише одного параметра, в результаті деякі пари можуть залишитися неперевіреними. З цієї причини в даному прикладі відсутні значення обсягу даних рівні нулю і ті, що перевищують обсяг диска. Якщо їх додати, то в результаті використання методу може вийти ТНД, в якому на нульовому обсязі даних буде перевірятися наприклад пара Файлова система ISO і Початок Мультисесії. В результаті, успішно переконавшись в коректній обробці спроби запису порожнього диска, ми упустимо перевірку пари: ISO — почати Мультисесію.

Якщо буде зроблена спроба Інкрементального доповнення тестового покриття — кількість тестів збільшитися незрівнянно. Забігаючи наперед, для даного прикладу із записом DVD методом перебору пар вийде 17 тестів. Вирішивши доповнити вихідну таблицю всього одним значенням, наприклад швидкістю запису DVD 32x, загальна кількість тестів збільшиться на 8,

оскільки прагнення зберегти цілісність методу призведе до перебору цього значення з усіма значеннями інших параметрів. Доцільно доповнити тестове покриття одним-двома тестами на перевірку нового значення складеними вручну, або ще раз створити таблицю, як це описано в наступному кроці.

Слід застосувати алгоритм, що встановить оптимальне число тестів з повним перебором пар. Складати тести за методом парного тестування без використання технічних засобів вкрай складно, тому щоб спростити собі життя, слід скористатися програмними рішеннями.

В якості прикладу візьмемо Allpairs. Allpairs — це програма, що підбирає унікальні пари для вхідного набору даних. Працює з командного рядка. В якості вхідних даних для програми використовується .txt файл з таблицею параметрів, стовпці якої розділені табуляцією. Для створення такого файлу найзручніше використовувати MS Excel в якому є можливість зберігати "текстові файли з роздільниками табуляції (* .txt)". Маючи вихідний файл, необхідно запустити консоль і набрати там рядок, яка зображена на рис.2.6.

Рис.2.6. Запуск Allpairs

де: - C: \ pairs \ allpairs.exe — повний шлях до програми allpairs.exe;

- test.txt — шлях до вихідного файлу з таблицею параметрів;
- re.txt — шлях і ім'я файлу, який буде створений в результаті роботи програми.

Результуючий файл буде містити в собі готовий перелік перевірок. В результаті, взявши таблицю з параметрами для DVD, виходить перелік тестів, вказаний в таблиці 2.10.

Символ «~» означає, що замість зазначеного значення може бути використаний будь-який, оскільки воно не становить пари в даному тесті.

Таблиця 2.10. Рузультат Allpairs для DVD

#	Швидкість запису	Файлова система	Мультисесія	Об'єм даних
1	2	ISO	Ні	100 Mb
2	2	UDF	Почати	4,7 Gb
3	4	ISO	Почати	100 Mb
4	4	UDF	Ні	4,7 Gb
5	8	ISO	Продовжити	4,7 Gb
6	8	UDF	Продовжити	100 Mb
7	16	UDF/ISO	Ні	100 Mb
8	16	UDF/ISO	Почати	4,7 Gb
9	24	UDF/ISO	Продовжити	100 Mb
10	24	ISO	Ні	4,7 Gb
11	2	UDF/ISO	Продовжити	~4,7 Gb
12	4	UDF/ISO	Продовжити	~100 Mb
13	8	UDF/ISO	Почати	~100 Mb
14	16	UDF	Продовжити	~100 Mb
15	24	UDF	Почати	~100 Mb
16	8	~ISO	Ні	~4,7 Gb
17	16	ISO	~ Почати	~4,7 Gb

Якщо перебирати всі пари послідовно: всі значення швидкості запису з усіма файловими системами, плюс всі значення швидкості запису з усіма значеннями мультисесії, і так далі до повного перебору всіх пар значень, то вийде 61 тест. Це наочно демонструє можливості оптимізації. Отримавши таку таблицю тестувальник має практично готові тести: він знає сценарій, а також значення параметрів, які задаються при його проходженні. Тому вже в поточному вигляді тести придатні для виконання. У табл.2.11 міститься результат для параметрів таблиці для CD.

Таблиця 2.11. Результат Allpairs для CD

#	Швидкість запису	Файлова система	Мультисесія	Об’єм даних
1	2	ISO	Нема	100 Mb
2	2	UDF	Почати	700 Mb
3	4	ISO	Почати	100 Mb
4	4	UDF	Ні	700 Mb
5	8	ISO	Продовжити	700 Mb
6	8	UDF	Продовжити	100 Mb
7	16	UDF/ISO	Ні	100 Mb
8	16	UDF/ISO	Почати	700 Mb
9	24	UDF/ISO	Продовжити	100 Mb
10	24	ISO	Ні	700 Mb
11	36	UDF	Почати	100 Mb
12	36	UDF/ISO	Продовжити	700 Mb
13	52	ISO	Почати	100 Mb
14	52	UDF	Ні	700 Mb
15	2	UDF/ISO	Продовжити	~100 Mb
16	4	UDF/ISO	Продовжити	~700 Mb
17	8	UDF/ISO	Ні	~100 Mb
18	16	ISO	Продовжити	~700 Mb
19	24	UDF	Почати	~100 Mb
20	36	ISO	Ні	~100 Mb
21	52	UDF/ISO	Продовжити	~100 Mb
22	8	~UDF	Почати	~700 Mb
23	16	UDF	~ Продовжити	~100 Mb

Разом: 23 тести проти 87, якщо виконувати повний перебір параметрів.

Попарне тестування за допомогою PICT

PICT — це програма, яка дозволяє генерувати компактний набір значень тестових параметрів, що представляє собою всі тестові сценарії для всебічного комбінаторного покриття параметрів, які вводяться. Приміром, у користувача є наступні параметри для тестування, що вказані в таблиці 2.12.

Таблиця 2.12. Набір даних для тестування

Type	Size	Format method	File system	Cluster size	Compression
Primary	10	Quick	FAT	512	On
Logical	100	Slow	FAT32	102	Off
Single	500		NTFS	2048	
Span	1000			4096	
Stripe	5000			8192	
Mirror	10000			16384	
RAID-5	40000			32768	
				65536	

Існує більше 4700 комбінацій цих значень. Буде дуже складно провести тестування за розумний час. Дослідження показують, що тестування всіх пар можливих значень забезпечує дуже хороше покриття і кількість тест-кейсів залишається в межах розумного. Наприклад, {Первинна, FAT} це одна пара і {10, повільна} інша; один тест-кейс може покривати багато пар.

Запускається PICT з командного рядка, як показано на рис.2.7.

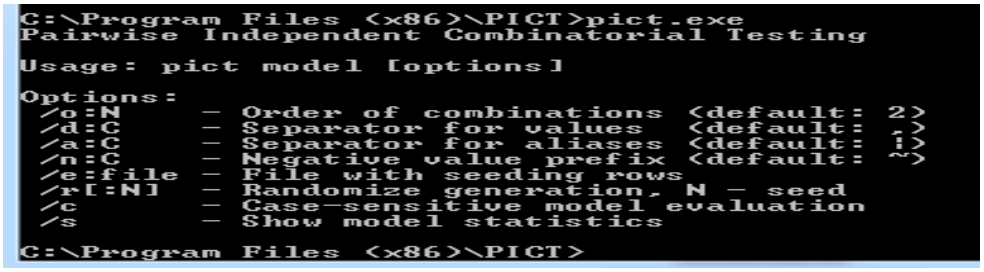


Рис. 2.7. Запуск PICT

На вхід програма приймає простий текстовий файл з параметрами, званий Моделлю, а на вихід видає згенеровані тестові сценарії. Розглянемо роботу програми на прикладі. Маємо такі параметри і їх значення (таблиця 2.13).

Таблиця 2.13. Параметри і їх значення для PICT

Стать	Вік	Наявність дітей
Чоловік	До 25	Так
Жінка	Від 25 до 60	Ні
	Більше 60	

Якщо перебирати всі можливі значення, то кількість сценаріїв буде 12. Складемо файл, де вказуються параметри та їх значення. Вміст текстового файлу model.txt буде виглядати наступним чином:

SEX: Чоловік, Жінка

AGE: до 25 років, 25-60, старше 60 років

Children: так, ні

де: стать, вік, діти - це назви параметрів; Чоловік, Жінка, до 25 років і т.п. - це значення параметрів, які вказуються через кому.

Щоб PICT отримала результат, в командному рядку потрібно написати "Pict" і шлях до моделі. Використовуючи модель і вийде 6 тестових сценаріїв, (замість 12). Вони показані на рис.2.8.

```
C:\Program Files (x86)\PICT>pict model.txt
SEX     Age     Children
Female  25-60   No
Male    Under 25        Yes
Female  Older than 60   Yes
Female  Under 25        No
Male    25-60   Yes
Male    Older than 60   No

C:\Program Files (x86)\PICT>
```

Рис. 2.8. Набір тест-кейсів

Можна використовувати збереження тест-кейсів в Excel (рис. 2.9).

```
C:\Program Files (x86)\PICT>pict model.txt > f:\example.xls
```

Рис. 2.9. Збереження результатів в Excel-файл

В результаті буде створений файл з таким вмістом (рис.2.10).

	A	B	C
1	SEX	Age	Children
2	Female	25-60	No
3	Male	Under 25	Yes
4	Female	Older than	Yes
5	Female	Under 25	No
6	Male	25-60	Yes
7	Male	Older than	No

Рис. 2.10. Результат сформований у файлі Excel

Якщо розглянути рішення першого прикладу в цьому розділі, то PICT отримав 60 тестових сценаріїв, проти 4700 за умови повного перебору, що зображено на рис.2.11.

	A	B	C	D	E	F	G
1	#	Type	Size	Format m	File syste	Cluster s	Compression
2	1	Mirror	10	quick	FAT	32768	off
3	2	RAID-5	10	slow	FAT32	512	on
4	3	Stripe	500	quick	NTFS	512	off
5	4	Span	1000	slow	NTFS	1024	on
6	5	Primary	100	quick	FAT32	16384	off
7	6	Single	1000	slow	FAT	8192	off
8	7	Primary	5000	slow	FAT	2048	on
9	8	RAID-5	40000	quick	NTFS	8192	on
10	9	Logical	10	slow	NTFS	65536	on
11	10	Span	100	quick	FAT	65536	off
12	11	Mirror	10000	slow	FAT32	65536	on
13	12	Logical	1000	quick	FAT32	512	off
14	13	Logical	40000	slow	FAT	4096	off
15	14	Single	1000	quick	NTFS	4096	on
16	15	Stripe	500	slow	FAT32	32768	on
17	16	Mirror	100	quick	NTFS	2048	off
18	17	Span	10	slow	FAT32	4096	off
19	18	Single	40000	quick	FAT32	65536	off
20	19	RAID-5	5000	quick	FAT	65536	off
21	20	Stripe	1000	slow	FAT32	2048	on
22	21	Primary	10000	quick	NTFS	8192	off
23	22	Span	10000	slow	FAT	16384	on
24	23	Primary	1000	slow	FAT32	65536	on
25	24	Single	5000	quick	FAT32	1024	off
26	25	RAID-5	100	slow	FAT	1024	on
27	26	Single	500	slow	NTFS	2048	off
28	27	Mirror	500	quick	FAT	1024	on
29	28	Stripe	100	quick	FAT	4096	on
30	29	Primary	40000	quick	FAT32	1024	off
31	30	Single	10	quick	NTFS	16384	on
32	31	Logical	5000	slow	NTFS	32768	off
33	32	Stripe	10	slow	FAT	1024	off
34	33	Primary	500	slow	NTFS	4096	off
35	34	Mirror	1000	quick	FAT	16384	on
36	35	Stripe	40000	quick	FAT	16384	off
37	36	Mirror	10	slow	FAT32	8192	on
38	37	Span	40000	quick	NTFS	32768	off
39	38	Logical	10000	slow	NTFS	1024	off
40	39	Span	5000	quick	FAT	512	on
41	40	Logical	100	slow	FAT32	8192	on
42	41	RAID-5	500	quick	NTFS	16384	on
43	42	Stripe	5000	slow	NTFS	8192	off
44	43	Mirror	5000	slow	NTFS	4096	off
45	44	Span	500	quick	FAT	65536	off
46	45	Span	10000	slow	NTFS	2048	on
47	46	Stripe	10000	quick	FAT32	65536	off
48	47	Primary	10	quick	FAT	2048	off
49	48	RAID-5	10000	slow	NTFS	4096	on
50	49	Primary	10000	quick	NTFS	32768	on
51	50	RAID-5	1000	quick	FAT32	32768	on
52	51	Primary	10000	quick	FAT	512	off
53	52	Mirror	40000	slow	FAT32	512	on
54	53	Single	100	slow	NTFS	512	off
55	54	Logical	500	quick	FAT32	16384	off
56	55	Single	100	slow	NTFS	32768	on
57	56	Mirror	5000	quick	FAT32	16384	off
58	57	Span	500	slow	FAT	8192	on
59	58	RAID-5	40000	slow	FAT	2048	off
60	59	Logical	10	quick	FAT	2048	off
61	60	Single	10000	slow	FAT32	65536	on

Рис.2.11. Результат рівнозначний повному перебору

На рис.2.12 зображений результат, отриманий PICT для DVD.

	A	B	C	D	E
1	#	Speed	FS	ulstisessid	Capacity
2	1	2	ISO	Begin	100Mb
3	2	2	UDF/ISO	No	4.7Gb
4	3	2	UDF	Continue	4.7Gb
5	4	4	UDF/ISO	Begin	4.7Gb
6	5	4	UDF	Continue	4.7Gb
7	6	4	ISO	No	100Mb
8	7	8	ISO	No	4.7Gb
9	8	8	UDF/ISO	Begin	100Mb
10	9	8	UDF	Continue	4.7Gb
11	10	16	ISO	Continue	100Mb
12	11	16	UDF	Begin	4.7Gb
13	12	16	UDF/ISO	No	4.7Gb
14	13	24	UDF	No	100Mb
15	14	24	ISO	Begin	4.7Gb
16	15	24	UDF/ISO	Continue	100Mb

Рис.2.12. Результат для DVD в PICT

Разом 15 сценаріїв проти 17 у Allpairs. Це означає, що в даному прикладі PICT здатний згенерувати меншу кількість сценаріїв, ніж Allpairs, надаючи однакове покриття ТНД.

Додаткові Можливості PICT:

- можна вказувати порядок угруповання значень. За замовчуванням використовується порядок 2 і створюються комбінації пар значень (що і складає попарне тестування). Але можна вказати 3 і тоді будуть використовуватися триплети. Максимальний порядок для простої моделі дорівнює кількості параметрів, що створюють набір варіантів;
- можна групувати параметри в підмоделі і вказувати їм окремий порядок для комбінацій, у випадку, коли комбінації певних параметрів повинні бути протестовані ретельніше або повинні бути об'єднані окремо;
- можна створювати умови і обмеження. Наприклад, можна вказати, що один із параметрів буде приймати певне значення тільки тоді, коли кілька інших параметрів приймуть потрібні значення. Це дозволяє відсікти створення непотрібних перевірок;
- можна позначати невалідність значення для параметрів, для створення комбінацій та негативних тест-кейсів;
- використовуючи вагові коефіцієнти, можна вказати програмі віддавати переваги певним значенням при генерації комбінацій;
- можна використовувати опцію мінімізації (запускати програму кілька разів з цією опцією), щоб отримати мінімальну кількість тест кейсів.

РОЗДІЛ 3. АВТОМАТИЗАЦІЯ. НАВАНТАЖУВАЛЬНЕ ТЕСТУВАННЯ

3.1. Термінологія навантажувального тестування

Щоб обговорювати підходи до тестування навантаження і проблеми, які вирішуються з його допомогою, потрібно почати з основ, тобто термінології.

1) **віртуальний користувач** (Virtual User) — програмний процес, що циклічно виконує модельовані операції. *Приклад: звичайний користувач, який хоче скористатися системою;*
2) **ітерація (ітерація)** — це один повтор виконуваної в циклі операції. *Приклад ітерації: Вхід в систему -> проведення аналізу -> отримання результату аналізу - > Вихід із системи;*
3) **інтенсивність виконання операції (операція Інтенсивність)** — частота виконання операції за одиницю часу, в тестовому скрипті задається інтервалом часу між ітераціями. *Приклад: 3 користувача, які входять в систему через хвилину, після того, як зайшов попередній користувач;*
4) **навантаження (завантаження)** — сукупне виконання операцій на загальному ресурсі. *Приклад: загальна кількість виконаних операцій в системі трьома користувачами за певний проміжок часу;*
5) **продуктивність (Performance)** — кількість виконуваних операцій за період часу. Якщо система не може виконати певну кількість операцій за заданий час, вона починає гальмувати, що означає брак продуктивності;
6) **масштабованість додатку (Application Scalability)** — пропорційне зростання продуктивності при збільшенні навантаження;
7) **профіль навантаження (Performance Profile)** — це набір операцій з заданими інтенсивностями, отриманий на основі збору статистичних даних, або певним шляхом аналізу вимог до тестованої системи. Також він називається сценарієм (скриптом);

8) **навантажувальної точкою** називається розрахована (або задана Замовником) кількість віртуальних користувачів в групах, що виконують операції з певними інтенсивностями.

Розглянемо, як ці сутності пов'язані між собою. Вимірявши інтенсивність через інтервал часу між ітераціями, можна побачити, що зростання інтенсивності виконуваних операцій — це скорочення інтервалів часу. Зростання навантаження пропорційне зростанню інтенсивності. Також при збільшенні інтенсивності зростає продуктивність. При цьому збільшується ступінь використання (завантаженості) ресурсів. З якогось моменту зростання продуктивності припиняється (а навантаження може продовжувати рости), відбувається насичення і потім деградація системи. Можна помітити, що під час тестування зміна інтенсивності операцій може відбуватися згідно закону Пуассона, або бути рівномірною протягом усього тесту.

3.2. Мета навантажувального тестування

Основною метою навантажувального тестування є:

- оцінка продуктивності і працездатності програми на етапі розробки і передачі в експлуатацію;
- оцінка її продуктивності і працездатності на етапі випуску нових релізів та патчів;
- оптимізація продуктивності додатка, включаючи настройки серверів та оптимізацію коду;
- підбір відповідної для цього додатка апаратної (програмної платформи) і конфігурації сервера.

В межах однієї мети можуть використовуватися різні види тестування навантаження, наприклад, для першої, другої та третьої мети потрібно робити як тестування продуктивності так і тестування стабільності.

Але при плануванні навантажувального тестування логічніше все ж відштовхуватися від технічних цілей (а не комерційних, перерахованих вище), які досягаються в результаті тестування та класифікувати тести по ним таким чином:

- якщо цікавить дослідження продуктивності додатка, а саме час відгуку для операцій на різних навантаженнях в досить широких діапазонах, включаючи стресові навантаження, то це тестування продуктивності;
- якщо метою є розуміння наскільки додаток стійкий в режимі тривалого використання (виключення витоків пам'яті, некоректних конфігураційних налаштувань і т.д.), то проводиться довгий навантажувальний тест — це тестування стабільності. При цьому аналіз часу відгуку може мати місце, але не бути першим пріоритетом, головне щоб система "не впала";
- стрес-тестування має на меті перевірити чи повертається ПС після позамежного навантаження (і як скоро) до нормального режиму, також цілями стресового тестування можуть бути перевірки поведінки ПС у випадках, коли один з серверів в пулі перестає працювати, аварійно змінилася апаратна конфігурації сервера бази даних і т.д. Потрібно відзначити, що при стресовому тестуванні перевіряється не продуктивність ПС, а її здатність до регенерації після наднавантаження.

3.3. Етапи проведення навантажувального тестування

Розглянемо етапи проведення навантажувального тестування:

1) аналіз вимог і збір інформації про тестовану систему;
2) конфігурація тестового стенду для тестування навантаження;
3) розробка моделі навантаження;
4) вибір інструментальної платформи для навантажувального тестування;
5) створення і налагодження тестових скриптів;
6) проведення тестування;
7) аналіз результатів;
8) підготовка, відправка та публікація звіту по проведеним видам тестування навантаження.

3.3.1. Аналіз вимоги і збір інформації про тестовану систему

Під час аналізу вимог необхідно визначити основні критерії успішності проведених тестів. Для цього виділимо наступні характеристики:

-**час відгуку** (час, необхідний для відкриття сторінки або отримання очікуваного результату);

-**інтенсивність** (число запитів в секунду);

-**використовувані ресурси** (завантаження процесора, кількість використовуваної пам'яті, дисковий і мережевий I/ і т.д.);

-**максимальна кількість користувачів** (визначає число користувачів, здатних працювати з системою в умовах заданої конфігурації).

Задані у вимогах характеристики і є базовими навантажувальними точками програми. Всі одержані результати будуть порівнюватися з ними для прийняття рішення про завершення тестування або подальшого профілювання продуктивності.

Зауваження:

- основною проблемою при аналізі вимог є їх відсутність. Оскільки не завжди бізнес-аналітики, чи люди відповідальні за написання вимог з продуктивності, представляють, як система повинна працювати під навантаженням, які саме вимоги повинні бути надані, дуже часто цифри беруться просто «зі стелі», а тому доводиться не тільки виділяти наявні вимоги, але й проводити глибокий аналіз на предмет їх коректності;
- характеристика "Максимальна кількість користувачів" насправді є малоінформативною, в разі якщо для тестування необхідно буде працювати з декількома групами користувачів. Тому вкрай важливо буде знати більш-менш точну кількість користувачів в кожній групі.

3.3.2. Аналіз вимог в залежності від типу проекту

При аналізі вимог необхідно врахувати: чи розробляється нове ПЗ (проект запуску), або ж проект спрямований на профілювання навантаження для вже існуючого додатку, що вже експлуатується (профілювання проекту).

Залежно від типу проекту розробляються вимоги по продуктивності.

Для проектів запуску:

- аналіз загальноприйнятих критеріїв продуктивності;
- аналіз продуктивності конкуруючих додатків;

- аналіз експертної думки розробників, системних і мережевих адміністраторів, адміністраторів баз даних і інженерів з тестування;
- аналіз очікувань цільових користувачів (груп користувачів).

Для профілювання проектів:

- аналіз загальноприйнятих критеріїв продуктивності;
- аналіз продуктивності конкуруючих додатків;
- аналіз продуктивності експлуатованої версії додатка, з метою визначення функцій, що вимагають профілювання, процесів, операцій і т.д;
- аналіз експертної думки розробників, системних адміністраторів і адміністраторів баз даних та інженерів тестування навантаження;
- аналіз думки цільових користувачів (груп користувачів).

І вже після отримання даних з усіх джерел можна отримати більш, менш точні вимоги щодо продуктивності для тестової програми.

3.3.3. Конфігурація тестового стенда для навантажувального тестування

На результати тестування навантаження можуть впливати різні чинники, такі як конфігурація тестового стенда, завантаженість мережі, заповненість бази даних і багато інших. Причому вплив їх на продуктивність програми може бути значним і мати нелінійну залежність, тому висловити її формулою буде практично неможливо.

Отже, чим менше будуть різнитися параметри тестової і реальної інфраструктури, тим менше буде похибка в отриманих результатах.

Потрібно відзначити ті частини конфігурації, які вимагають особливої уваги:

Hardware: процесор (тип, частота, кількість ядер і т.д); оперативна пам'ять (тип, обсяг, таймінг, ефективна частота); жорсткі диски та флеш-диски (тип, швидкість).

Software: операційна система; драйвера.

Network: топологія мережі; пропускна здатність; протокол передачі даних.

Application: архітектура; база даних (структура даних); програмне забезпечення, необхідне для роботи програми (наприклад, для Java додатків-JVM).

У самому ідеальному випадку тестовий стенд дублює конфігурацію

реального сервера, на якому працює або ж буде працювати додаток. Однак, ідеальних випадків практично не буває (то пам'яті мало, то процесора такої частоти немає в наявності, то операційна система не тієї версії, то вартість деякого серверного ПО не вкладається в бюджет). Перерахуємо основні причини, через які не завжди виходить продублювати конфігурацію системи на тестовому стенді:

1) складність дублювання дорогого серверного заліза для тестових потреб;

2) обмеження на використання ліцензій необхідного ПЗ;

3) закритість архітектури додатку з боку замовника з міркувань безпеки;

4) труднощі відтворення або транспортування бази даних програми;

5) складність відтворення необхідної архітектури мережі.

Доцільність ж відтворення інфраструктури необхідно оцінити з урахуванням виділених ресурсів, часу і зусиль, бо не завжди результат виправдовує засоби.

3.3.4. Розробка моделі навантаження

Визначившись з видами навантажувального тестування, цілями і термінологією, перейдемо до основного завдання тестування — розробці моделі навантаження.

Для цього необхідно визначити наступне:

- список тестованих операцій;
- інтенсивність виконання операцій;
- залежність зміни інтенсивності виконання операцій від часу.

У список тестованих завдань повинні увійти операції, критичні з точки зору бізнесу, а також з технічної точки зору:

- критичними **з точки зору бізнесу** є операції, швидкість виконання яких, реально впливає на продуктивність бізнес-процесу. *Наприклад, збільшення тривалості обслуговування клієнтів в банку, неможливість виконання необхідної кількості операцій протягом дня і так далі;*
- критичними з **технічної точки зору** є ресурсомісткі операції, що вимагають велику кількість пам'яті, серйозно задіють процесор, що створюють значний мережевий трафік. *Як правило, це операції виконуються*

одночасно великою кількістю бізнес-користувачів або для створення складних звітів, в які входять так звані "важкі" запити до бази даних.

Потрібно підкреслити, що під ступенем критичності операції мається на увазі її вплив на бізнес-процес і працездатність системи. *Наприклад, якщо створення звіту повністю завантажує сервер бази даних в нічний час, це не буде носити високий пріоритет для оптимізації, а в робочі години буде мати максимальний пріоритет.*

Модель тестування продуктивності

Поступове збільшення навантаження, додаючи нових користувачів з деяким інтервалом часу, дозволяє визначити:

- вимірювання часу виконання обраних операцій при певних интенсивностях виконання цих операцій;
- кількість користувачів, здатних одночасно працювати з додатком;
- межі прийнятної продуктивності при збільшенні навантаження;
- продуктивність під час різних навантажень.

Модель стресового тестування

Збільшуючи інтенсивність операцій вище пікових (максимально дозволених) значень, або збільшуючи кількість користувачів до тих пір, доки навантаження не стане вище максимально припустимих значень, перевіряємо, що система працездатна в умовах стресу. Далі, опустивши навантаження до середніх значень, перевіряємо здатність системи до регенерації, тобто, що система повернулася до нормального стану (основні характеристики навантажень не перевищують базові).

Модель об'ємного тестування

Можна використовувати ту ж модель, що і для тестування продуктивності, однак метою буде перевірка роботи системи з прогнозом на майбутнє зростання обсягу даних. Отже, найважливішою передумовою тесту буде збільшення обсягів бази даних програми до необхідних розмірів. Таким чином можна перевірити і оцінити продуктивність, прогнозуючи зростання системи на рік, два або три вперед.

Модель тестування стабільності або надійності

Використовуючи базовий навантажувальний профіль, запускаємо тест тривалістю від декількох годин до декількох днів, з метою виявлення витоків пам'яті,

перезапуску серверів та інших аспектів, що впливають на навантаження.

3.4. Огляд програм навантажувального тестування веб-сервісів

Здаючи веб-сервер в повсякденну експлуатацію, потрібно бути впевненим, що він витримає плановане навантаження. Тільки створивши умови, наближені до бойових, можна оцінити, чи достатня потужність ПС, чи правильно встановлені додатки, які беруть участь у створенні веб-контенту, і інші чинники, що впливають на роботу веб-сервера. У цій ситуації на допомогу прийдуть спеціальні інструменти, які допоможуть дати якісну й кількісну оцінку роботи як веб-вузла в цілому, так і окремих його компонентів.

OpenSTA (рис.3.1) — це більше ніж додаток для тестів, це відкрита архітектура, спроектована навколо відкритих стандартів. Проект створений групою компаній CYRANO, яка підтримувала комерційну версію ПП, але нині OpenSTA поширюється як додаток з відкритим кодом під ліцензією GNU GPL. Для роботи вимагає компоненти Microsoft Data Access (MDAC), які можна завантажити з сайту корпорації.

Поточний інструментарій дозволяє провести навантажувальне випробування HTTP/HTTPS сервісів, хоча його архітектура здатна на більше. OpenSTA дозволяє створювати тестові сценарії на спеціалізованій мові SCL (Script Control Language). Для спрощення їх створення є Tools – Canonicalize URL, де в полі запускається веб-браузер. Користувач просто ходить по сайтах, збираючи посилання, які будуть збережені в скрипт. Всі параметри запиту піддаються редагуванню, можлива підстановка змінних. Структура тесту і заголовки будуть виводитися у вкладках в панелі зліва. Тести зручно об'єднувати в ТНД. Налаштування проксі задаються в самому скрипті, тому можна вказати кілька серверів. Реалізована можливість організації розподіленого тестування, що підвищує реалістичність, або коли з одного комп'ютера не виходить навантажити потужний сервер. Кожна з машин такої системи може виконувати свою групу завдань, а господар сховища здійснює збір і зберігання результатів. Після установки на кожній тестованій системі запускається сервер імен, робота якого обов'язкова. Підтримується аутентифікація користувачів на веб-ресурсі і встановлення з'єднань по протоколу SSL. Параметри роботи навантажувальної

системи можна контролювати за допомогою SNMP і засобів Windows. Результати тестування, що включають час відгуків, кількість переданих байт в секунду, коди відповіді для кожного запиту і кількість помилок виводяться у вигляді таблиць і графіків. Використання великого числа фільтрів дозволяє відібрати необхідні результати. Результат можна експортувати в CSV-файл. Можливості щодо виведення звітів дещо обмежені, але по посиланнях на сайті можна знайти скрипти і плагіни, що спрощують, в тому числі, аналіз отриманої інформації.

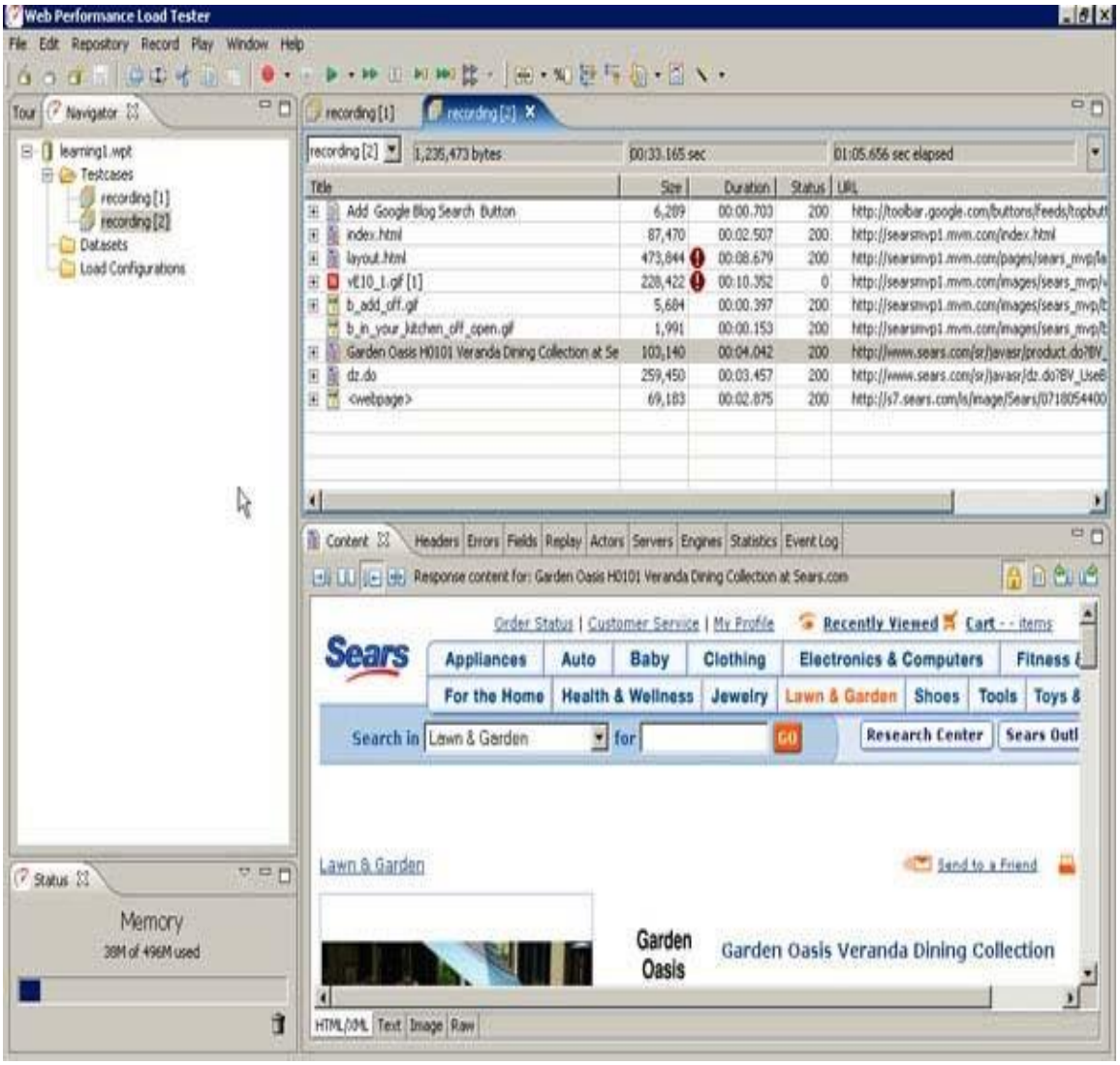


Рис.3.1. OpenSTA

Apache Jmeter (рис.3.2) — це єдиний інструмент для навантажувального тестування, який з одного боку безкоштовний і з відкритим вихідним кодом, а з іншого боку досить розвинений і має можливість створення тестового навантаження одночасно з декількох комп'ютерів.

Тести для JMeter створюються візуально і мають деревоподібну структуру в вікні редагування тесту. Запуск тестів здійснюємо як з вікна програми, так і з командного рядка, що корисно, якщо ТНД запускати за розкладом, наприклад, вночі.

Для чого він найкраще підходить

Інструмент позиціонує себе як універсальний, що дозволяє працювати: з HTTP запитами, з FTP, з JDBC-запитами, SOAP, Web Services, TCP, LDAP, JMS, пошта тестерів. Але він найкраще підходить для навантажувального тестування веб-додатків.

Якщо для користувачів розробники створюють багато-користувацький веб-додаток, то перед його випуском у них однозначно виникнуть питання:

- чи стабільно працює додаток під великим навантаженням;
- яке максимально можливе число користувачів витримає додаток на певній конфігурації;
- наскільки швидше став працювати додаток після поліпшення архітектури його коду.

На всі ці питання з відносно невеликими витратами часу відповість JMeter.

З чого складається тест

При створенні тесту JMeter пропонує кілька типів компонент:

1) *пробники* — основні елементи, які безпосередньо спілкуються з тестованим додатком, наприклад HTTP-*пробовідбірник* для звернення до веб-додатку;
2) *логічні контролери* — елементи, що дозволяють групувати інші елементи в цикли, групи паралельного запуску, і т.д.;
3) *затверджувачі* — елементи, що виконують контроль. З їх допомогою ви можете перевірити текст, який ви очікуєте на веб-сторінці або, наприклад, вказати, що ви очікуєте відповідь від сервера не більше ніж 2 секунди. Якщо який-небудь *Затверджувач* не буде задоволений, тест буде мати негативний результат.

Як виглядає звіт

Що буде міститися у звіті, користувач налаштовує сам. Зручно те, що до звіту можна включати таблиці та графіки.

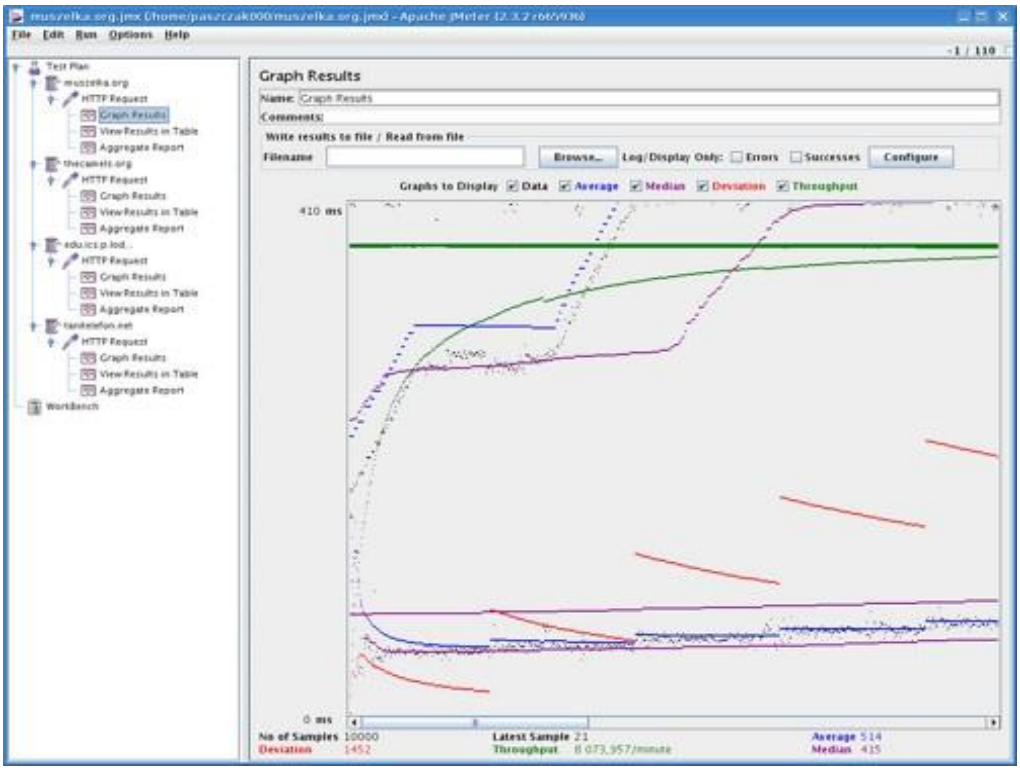


Рис.3.2. Apache Jmeter

Чи можна розширювати JMeter

Jmeter є зручним інструментом для запуску та моніторингу тестів і для перегляду звітів. Якщо користувачеві потрібні особливі види тестів, які він може написати на Java, то все що потрібно, це написати код самого тесту. Далі його слід оформити як *пробовідбірник*, після чого JMeter допоможе зробити всю іншу роботу по організації, конфігурації і виконанню тестів (також і з декількох комп'ютерів).

WAPT (Тестування Web Application, рис.3.3) — дозволяє випробувати стійкість веб-сайту та інших додатків, що використовують веб-інтерфейс, до реальних навантажень. Розроблний компанією SoftLogica LLC. Це одна з найпростіших у використанні програм огляду. Для проведення простого тесту навіть не потрібно заглядати в документацію, інтерфейс простий, але не локалізований. Для перевірки WAPT може створювати множину віртуальних користувачів, кожен з індивідуальними параметрами. Підтримується декілька видів аутентифікації. Сценарій дозволяє змінювати затримки між запитами і динамічно генерувати деякі випробувальні параметри, максимально імітуючи поведінку реальних користувачів. У запит можуть бути підставлені різні варіанти HTTP-заголовка, в налаштуваннях

можна вказати кодування сторінок. Параметри User-Agent, X-Forwarded-For, IP вказуються в настройках сценарію. Значення параметрів запиту можуть бути розраховані кількома способами, в тому числі, визначені відповіддю сервера на попередній запит, використовуючи змінні і функції. Підтримується робота по захищеному протоколу HTTPS (і всі типи проксі-серверів). Створені сценарії, які зберігаються у файлі XML-формату, можна використовувати повторно. Крім продуктивності і стандартних налаштувань, в переліку присутні кілька інших тестів, що дозволяють визначити максимальну кількість користувачів і тестувати сервер під навантаженням протягом довгого періоду.

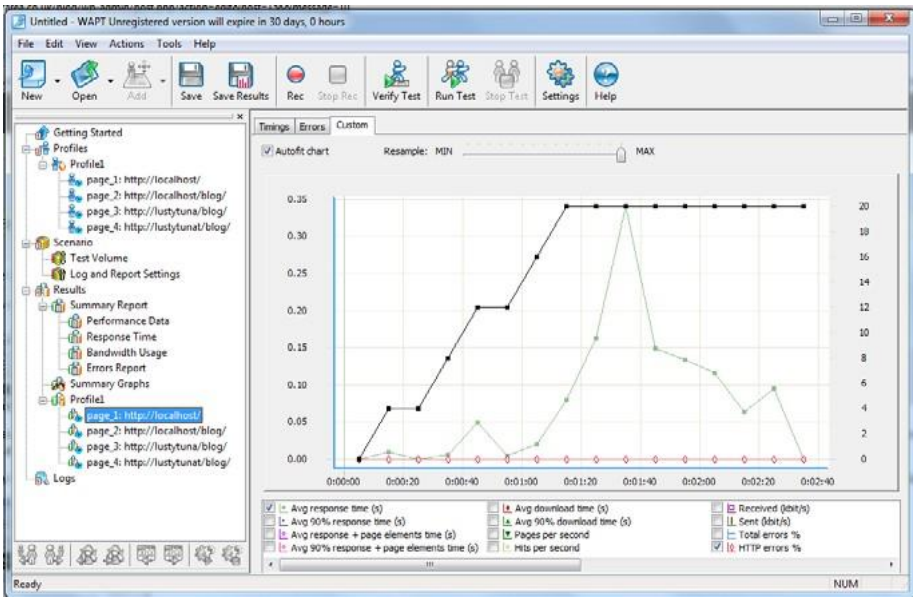


Рис.3.3. WAPT

NeoLoad (рис.3.4) — система, що дозволяє провести тестування навантаження веб-додатків. Написана на Java. У звіті можна отримати детальну інформацію по кожному завантаженому файлі. NeoLoad вельми зручний для оцінки роботи окремих компонентів (AJAX, PHP, ASP, CGI, Flash, аплетів і ін.). Можлива установка часу затримки між запитами (ThinkTime) глобально і індивідуально для кожної сторінки. Тестування проводиться як з використанням досить зручної графічної оболонки, так і за допомогою командного рядка (використовуючи заздалегідь підготовлений XML-файл). Підтримує роботу з протоколом HTTPS, з HTTP і HTTPS проксі, основні аутентифікації та cookies, автоматично визначаючи дані під час запису сценарію, а потім програє їх під час тесту. Для роботи з різними профілями та для реєстрації користувачів можуть бути використані змінні. При проведенні тесту можна задіяти

додаткові монітори (SNMP, WebLogic, WebSphere, RSTAT і Windows, Linux, Solaris), що дозволяють контролювати і параметри системи, на якій працює веб-сервер.

За допомогою NeoLoad можна виконувати і розподілені тести. Один з комп'ютерів є контролером, на інші встановлюються генератори навантаження (loadGenerator). Контролер розподіляє навантаження між loadGenerator і збирає отриману статистику. Дуже зручно реалізована робота з віртуальними користувачами. Користувачі з індивідуальними налаштуваннями об'єднуються у групи населення (маємо створити як мінімум одну Популяцію), в популяціях можна задати загальну поведінку (наприклад, 40% користувачів популяції відвідують динамічні ресурси, 20% читають новини). Віртуальні користувачі можуть мати індивідуальну IP-адресу, смугу пропускання і свій сценарій тесту.

Використовуючи утиліти навантажувального тестування, можна отримати інформацію про роботу веб-сервісу, вжити необхідних заходів щодо усунення виявлених недоліків і гарантувати необхідну продуктивність.

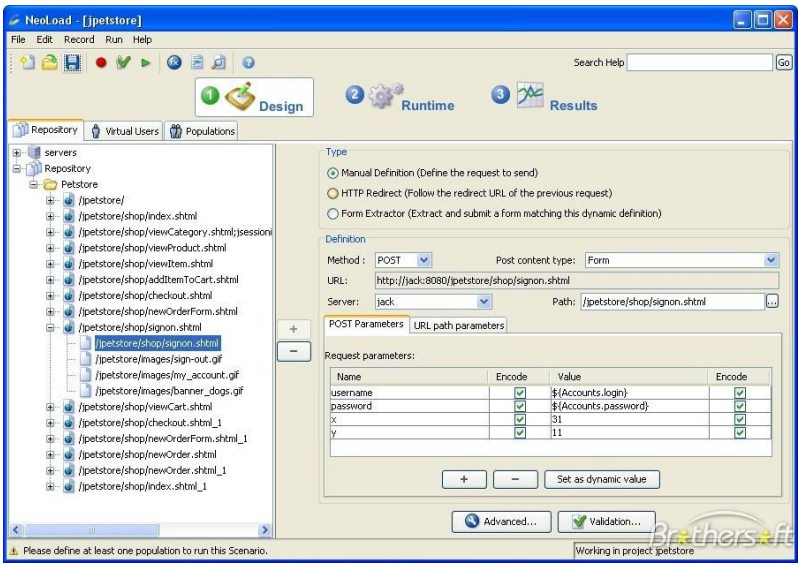


Рис.3.4 NeoLoad

Продукти Microsoft. Корпорація Microsoft пропонує два продукти, що дозволяють протестувати веб-сервер під навантаженням. Це Microsoft Application Stress Tool і Web-засіб аналізу можливостей. Перший поширюється як окремий продукт і має графічний інтерфейс. Другий входить до складу комплекту інструментів Internet Information Services Resource Kit Tools і працює з командного рядка. MAST більш наочний, в створенні тесту допоможе простий майстер створення тестів,

можливе регулювання навантаження за різними URL. Сценарій тестування може бути створений вручну, або записаний за допомогою веб-браузера та за необхідності відредагований. У WAST рівень навантаження (рівень стресу) регулюється шляхом задання кількості ниток, що здійснюють запити до сервера, а число віртуальних користувачів розраховується як добуток кількості ниток на число гнізд, відкритих у кожній з ниток. Після закінчення тесту отримуємо простий звіт в текстовій формі, в якому подано інформацію про кількість оброблених запитів в одиницю часу, середній час затримки, швидкість передачі даних на сервер і з сервера, кількість помилок і т.д. Звіт можна експортувати в CSV-файл. Ніяких можливостей по статистичній обробці не передбачено, тобто з його допомогою можна тільки оцінити роботу за певних умов.

Зважаючи на найкращу функціональність, з відомих програмних платформ автоматизованого тестування, що були розглянуті, обрано інструментальну платформу **Jmeter**, в тому числі для виконання навантажувального тестування ПЗ.

3.5. Навантажувальне тестування за допомогою Jmeter

Робота з програмою Jmeter була розглянута на простому прикладі створення скрипта для 5 користувачів, які хочуть: увійти в систему; провести повний фінансовий аналіз та вийти з системи.

На перший погляд може здатися, що сценарій занадто короткий, проте по-перше, створений скрипт не повинен містити багато дій. Якщо мета полягає в тому, що потрібно розглянути більше сценаріїв користувача, то по-перше, скрипти можна комбінувати між собою в ланцюжок для виявлення слабких місць в системі. А по-друге, навіть такий короткий скрипт допоможе розібратися в Jmeter і використовувати отримані знання для самостійного оволодіння програмою.

Одна з головних особливостей Jmeter — автоматичний запис скрипта. Суть її в тому, що коли створюється скрипт для сервісу, який тестується методом чорного ящика, тестувальник не знає всіх тонкощів і особливостей його роботи. Для полегшення роботи існує вбудований елемент HTTP проксі-сервер. Емулюючи роботу проксі сервера, він буде записувати всі отримані/надіслані запити.

3.5.1. Підготовчі дії

Перед використанням проксі потрібно зрозуміти, куди записувати отримані кроки. Їх можна записати прямо в "котушку ниток" (Thread), на "верстак" (Workbanch), або в елемент "контролер запису". Краще використовувати останній елемент з двох причин: по-перше, так краще відстежується і формується структура тесту, а по-друге, це дозволить при необхідності відключити (Ctrl+T) весь лог разом. Всі дії по додаванню і редагуванню відбуваються після натискання правої кнопки миші в контекстному меню. Створені елементи можна просто "перетягувати". Але, для використання елемента "Контролер запису" потрібно додати хоча б одну "котушку". Для цього потрібно натиснути правою кнопкою на План тестування> Add> Теми> Група розроблення. Створену котушку можна перейменувати для зручності і розуміння. Щоб це зробити, необхідно натиснути на створений елемент, праворуч буде поле Ім'я, замість групи потоків (Threads) можна написати що завгодно. Котушка була названа "dwarf" — це ім'я сервера, де знаходиться сервіс, з яким будемо проводити роботу. Після додавання котушки додається контролер запису. Для цього потрібно натиснути правою кнопкою на створений Thread Group> Add> логічний контролер> Контролер запису. Створений елемент був названий як "вхід в систему". На рис.3.5 зображено рузультат.

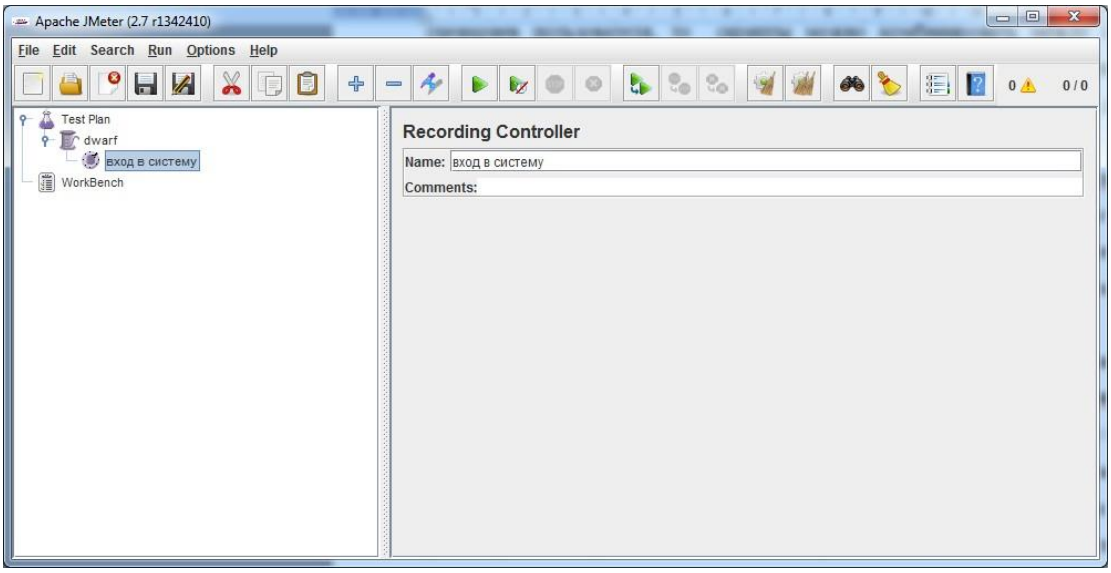


Рис.3.5. Thread Group та Recording Controller

3.5.2. Запис скрипта за допомогою HTTP Proxy Server

Коли підготовчі дії були завершені, можна додати на «верстак» проксі-елемент:

Стільниця> Додати> Non-Test Elements> HTTP проксі-сервер. У ньому багато налаштувань, але важливий поки тільки порт. Якщо port 8080 зайнятий, можна встановити 18000, або інший. Так само можна побачити, що за замовчуванням в полі цільовий контролер варто використовувати Контролер запису. На рис.3.6 було явно зазначено, куди потрібно записувати скрипт, тобто dwarf > вход в систему.

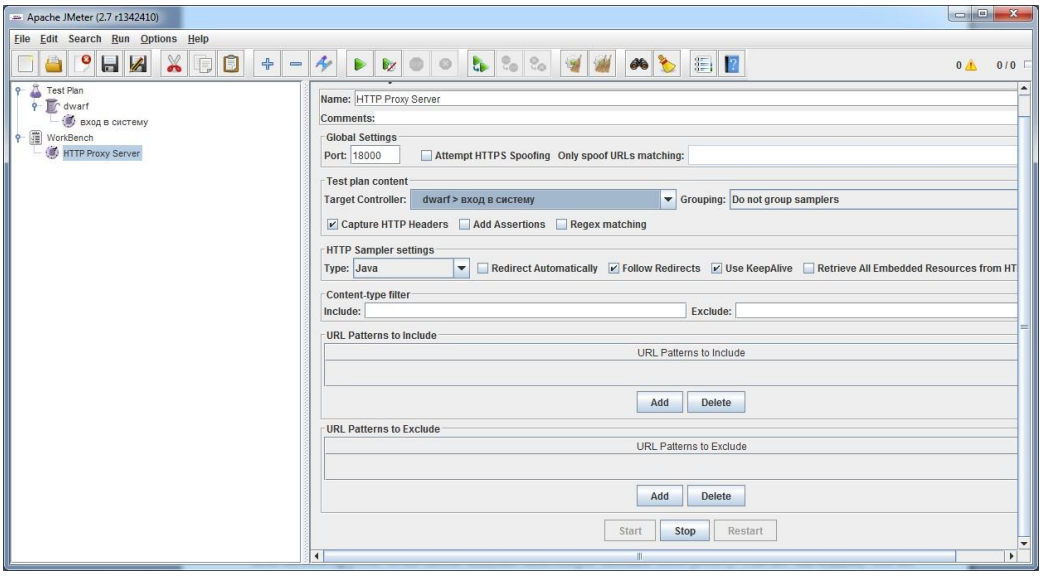


Рис.3.6. Налаштування HTTP Proxy Server

Після того, як порт був виставлений, необхідно перейти до налаштувань браузера. В даному прикладі показано: Сервіс - Властивості браузера-Підключення - кнопка Налаштування мережі. Відзначається «Використовувати проксі-сервер ...» На рис.3.7 в полі Адреса вказується localhost, а порт як у Jmeter'a: 18000.

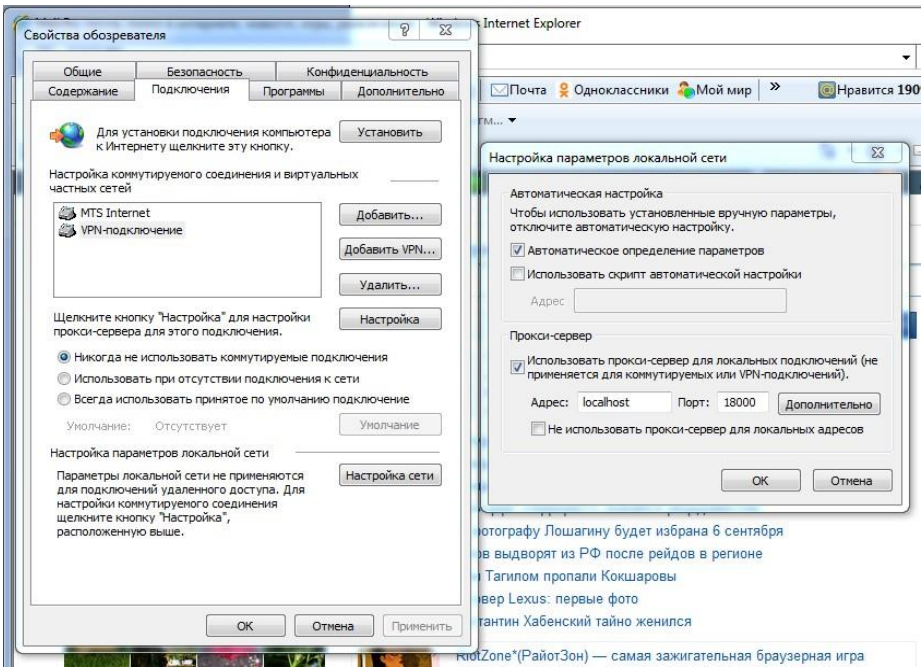


Рис.3.7. Налаштування браузера

Тепер все готово для запису. Потрібно повернутися в Jmeter на проксі і внизу натиснути кнопку Start. Після натискання всі дії користувача в браузері почнуть записуватися у створений Recording Controller. Необхідно увійти в систему. Записувати дії бажано за кілька кроків, бо скрипт може виявитися надмірно великим і його буде складно редагувати. Тому перша дія обрана як авторизація на сервісі. Після натискання Start і проходження процедури авторизації на сервері і процедури авторизації на сайті, слід повернутися в Jmeter і натиснути Stop. У записаному Recording Controller є багато дочірніх елементів, які зображені на рис.3.8. Це те, що було послано на сервер, а сервер відповів. Необхідно вивчити скрипт і прибрати зайве.

3.5.3. Налагодження скрипта

Налагодження скрипта — це видалення різних .jpg, .png і посилань на сторонні ресурси. Все це можна вичищати. В цілому, також можна вичистити *.js . Головне, знайти запит, який передає у своєму тілі облікові дані вашого користувача. А також знайти запит, який веде на сторінку, на якій користувач логінився (реєструвався).

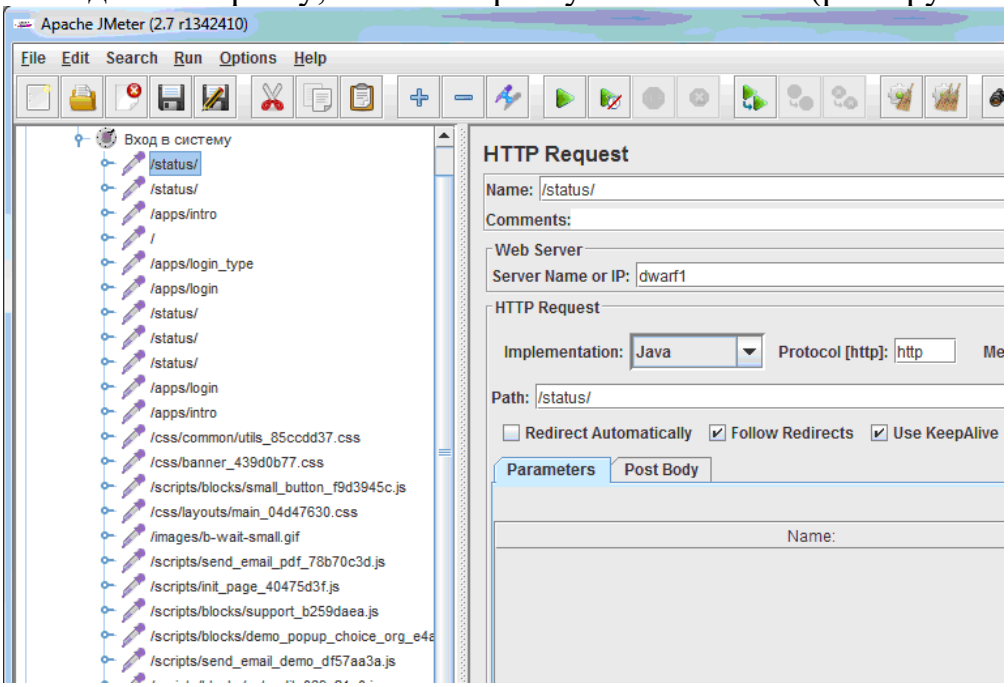


Рис.3.8. Створений скрипт входу в систему

Таким чином, разом ці запити моделюють зв'язку в діях користувача «зайшов на сторінку - залогінився». Після того, як сміття було прибрано, а потрібні запити були залишені, бажано дати їм читабельні імена. Щоб перевірити, чи все було видалено, скрипт треба запустити. В даному прикладі, якщо запустити скрипт, то він

виконається з помилками, оскільки для тестування потрібно авторизуватися на сервері. Для цього необхідно додати в верхівку дерева HTTP Autorization Manager (Thread Group> Add> Config Elements> HTTP Autorization Manager). Він інтуїтивно зрозумілий. У ньому необхідно вказати адресу ресурсу, на якому відбувається тестування навантаження, обов'язково з http: //, ім'я користувача і пароль.

Якщо Менеджер Авторизації потрібен для авторизації на сервері, то абсолютно точно при авторизації на сайті використовуються *cookies*. Необхідно додати Cookie Manager після Менеджера Авторизации (TestPlan> Add> Config Elements> HTTP Cookie Manager). Цей елемент потрібно додавати завжди, якщо мова йде про те, що користувачеві потрібно залогінитися (рис.3.9). Перш ніж запустити і перевірити скрипт додається елемент View Results Tree (dwarf> Add> Listener> View Results Tree), в якому можна буде спостерігати виконання. Цей елемент зручно розташовувати завжди знизу.

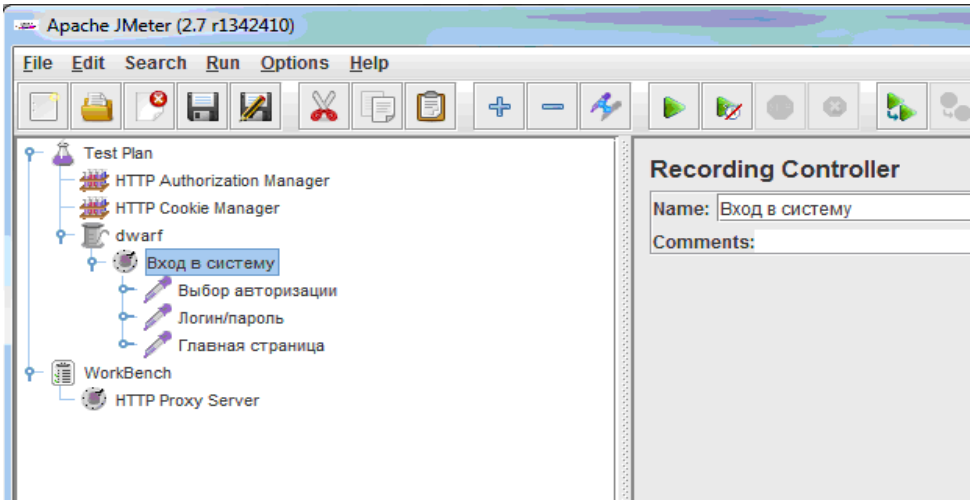


Рис. 3.9. Встановлення менеджера авторизації і cookies

Для запуску використовується поєднання клавіш ctrl + r. На рис.3.10 зображений результат виконання скрипта.

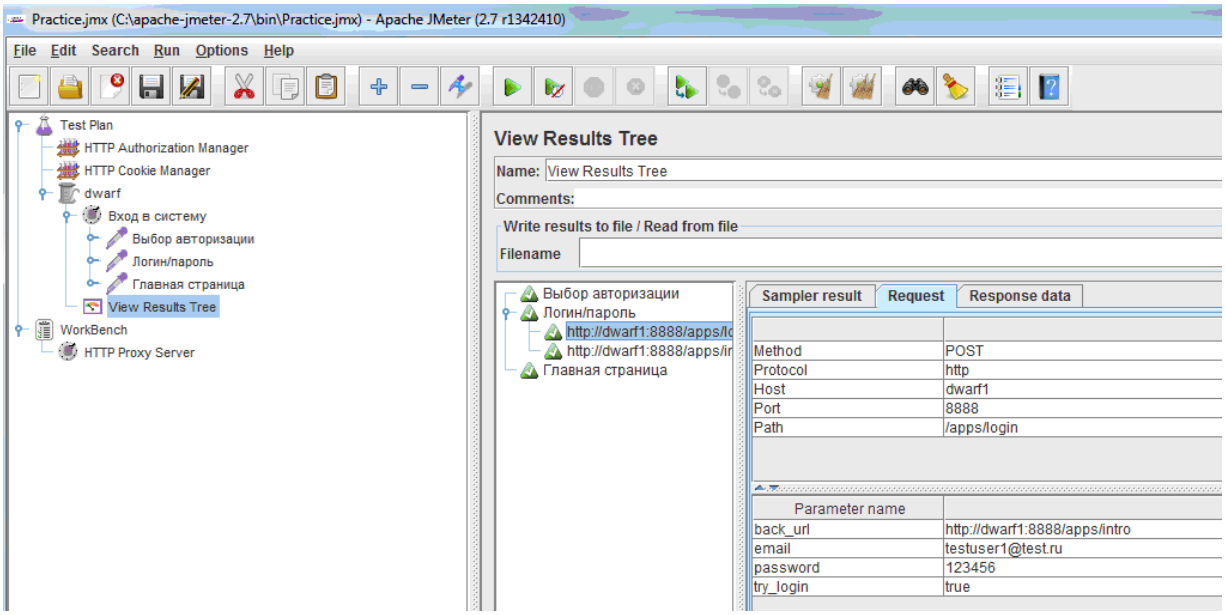


Рис.3.10. Результат виконання скрипта

Для того, щоб переконатися, що користувач був успішно авторизований, можна зайти в дерево результатів і подивитися на відповідь. Для цього потрібно натиснути на вкладку Response Data. У відповіді сервера є інформація, яка свідчить про успішну авторизацію користувача. В даному прикладі це ім'я користувача, яке було вказано при реєстрації e-mail, тому що після авторизації воно висвічується на сторінці. На рис. 3.11 можна побачити цього користувача у Response Data.

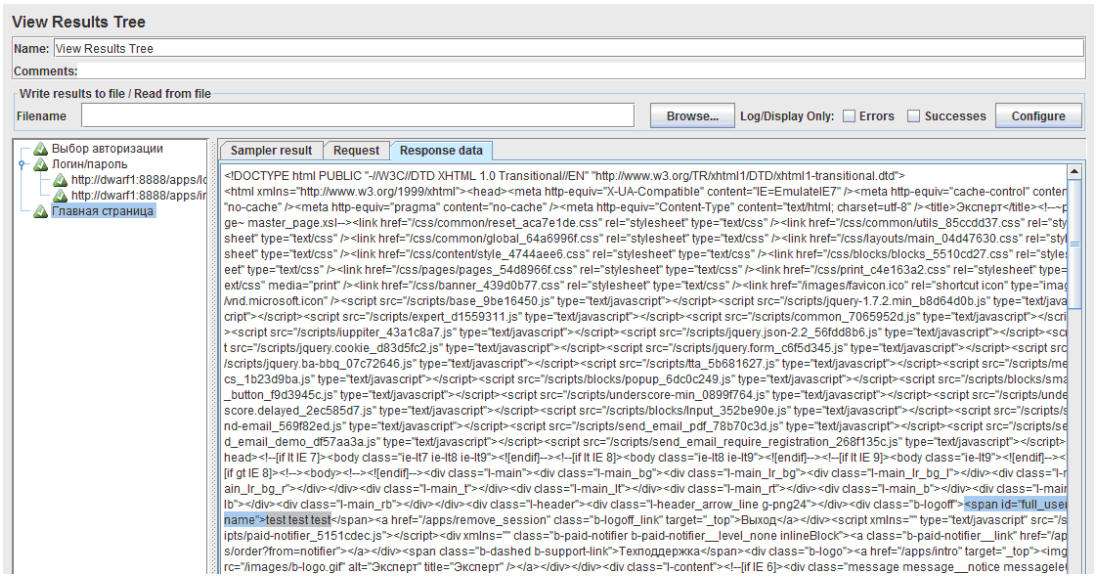


Рис.3.11. Response Data

Тепер вирішимо завдання залучення п’яти користувачів. Для цього необхідно виконати невеликий приклад по параметризації запитів.

3.5.4. Параметризація

Для параметризації запитів додається в самий верх елемент User Define Variables, тобто Thread> Config Element> User Define Variables. Внизу є кнопка Add. Після її натискання додаються дві змінні, тому що змінюватися будуть тільки два параметри: логін і пароль. Їм задаються імена: user і pswd. А в якості значень вказуються конкретні значення поточного користувача (див. рис.3.12).

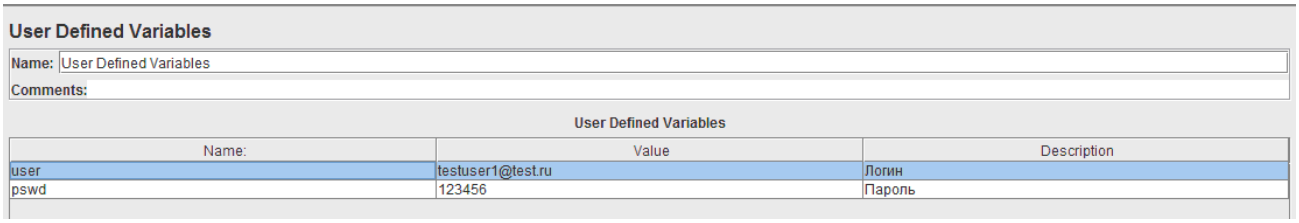


Рис.3.12. User Defined Variables

Щоб їх використовувати потрібно перейти в запит Логін/пароль, і замість конкретних значень записати імена змінних в форматі $ {ім'я змінної}. В даному прикладі вказуються в поля Value для логіна - $ {user}, для пароля - $ {pswd}. На рис.3.13 показано додавання параметрів.

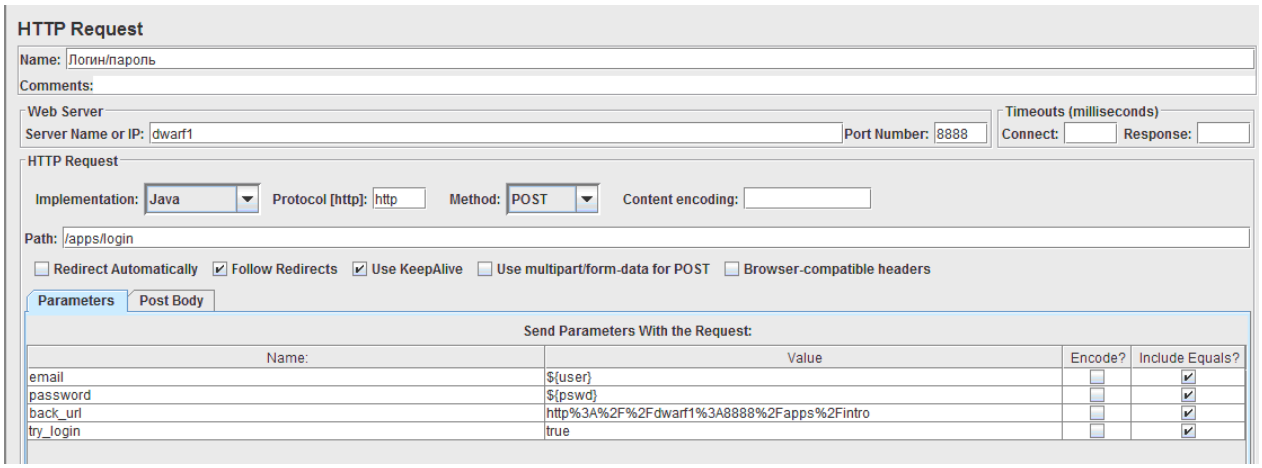


Рис.3.13. Параметризація: логін і пароль

Для налагодження передачі параметрів додається ще один елемент - Debug Sampler: Thread> Add> Samplers> Debug Sampler. В налаштуваннях елемента залишається тільки Jmeter variables, інші параметри приймають значення false (дивись рис.3.14).

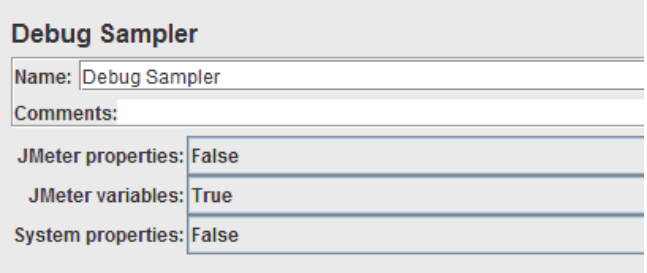


Рис.3.14. Налаштування Debug Sampler

Після запуску у View Results Tree повинні з'явиться результати. З них можна побачити, що в якості змінних передалися саме ті значення, зображення яких були задані, і запити в цілому повернули те, що потрібно (рис.3.15).

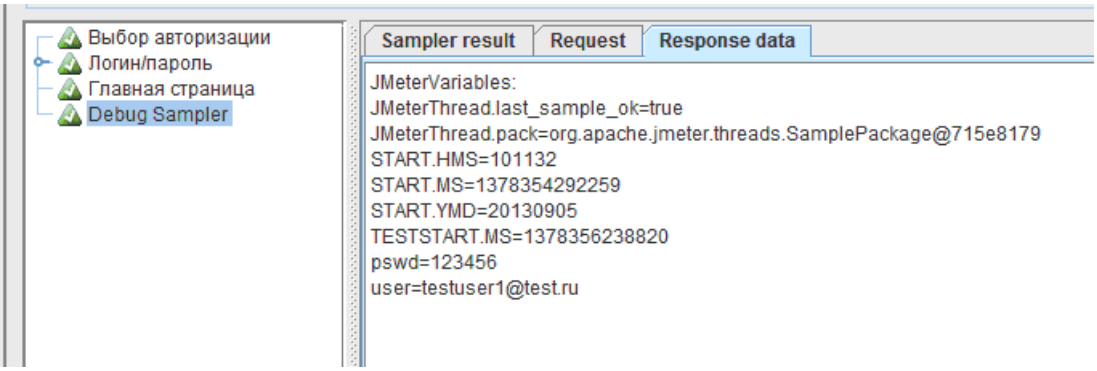


Рис.3.15. Перегляд параметрів

Тепер можна передати дані 5 користувачів. Для цього використовується CSV Data Set Config.

3.5.5. CSV Data Set Config

Створюється **.csv**-файл у блокноті. Для користувачів файл повинен являти собою набір з 5 рядків виду «user; pswd », як показано на рис.3.16.

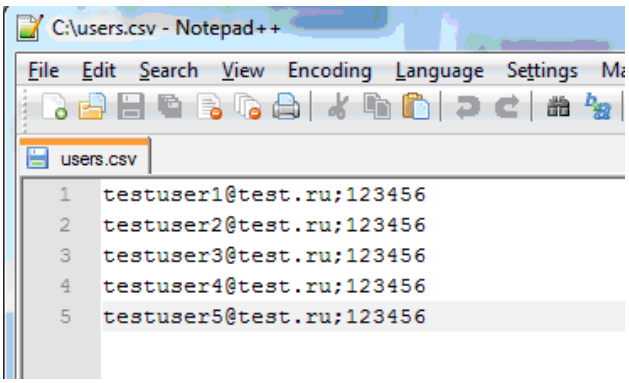


Рис.3.16. CSV-файл з користувачами і паролями

Далі в дерево додається елемент CSV Data Set Config (Add> Config Element> CSV Data Set Config). Краще його додати не в “котушку”, а в тест-план, нижче користувальницьких змінних. Як filename для файлу потрібно вказати повністю шлях до файлу і його ім'я. Нижче вказати кодування (на ваш вибір). Якщо всі дані англійською, то поле можна залишити порожнім. Нижче необхідно написати змінні, які вважаються у файлі є. Як приклад, це будуть user і pswd. Як роздільник потрібно вказати крапку з комою, так як в файлі використовується вона. Потрібно поставити в false повтор файлу після досягнення кінця, а також зупинку котушки. Налаштування CSV Data Set зображено на рис.3.17.

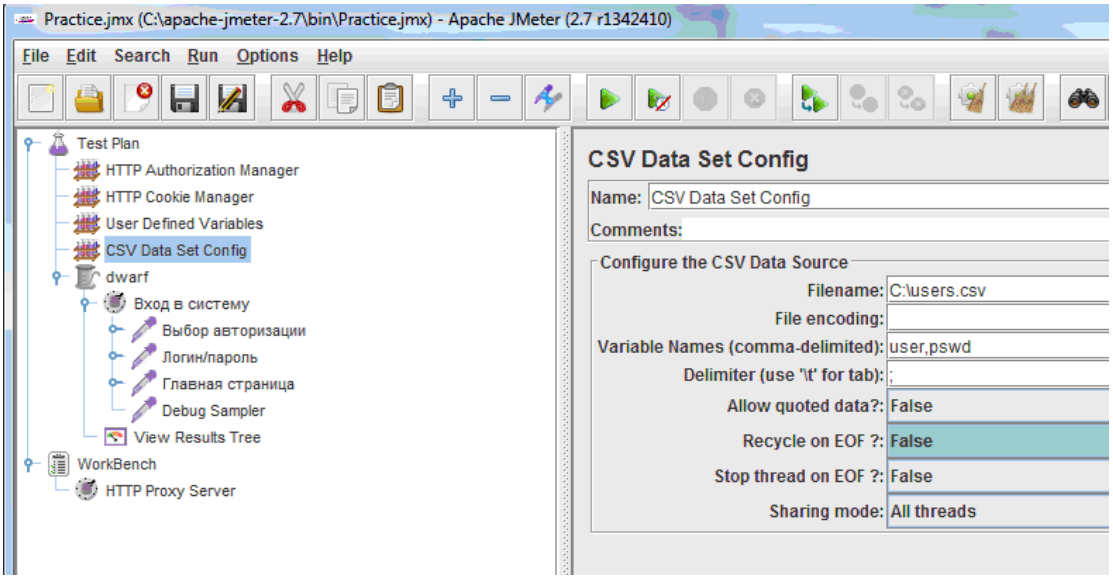


Рис.3.17. Налаштування CSV Data Set Config

Далі необхідно видалити подібні змінні з User Defined Variables. Після цього в самій котушці кількість потоків (Number of Threrads) встановлюється на 5. Тепер можна запустити скрипт і спостерігати в Debug Sampler і View Results Tree результати (рис.3.18). Потрібно не забути перевірити, чи потрібні дані повернув сервер та чи є в них імена створених користувачів.

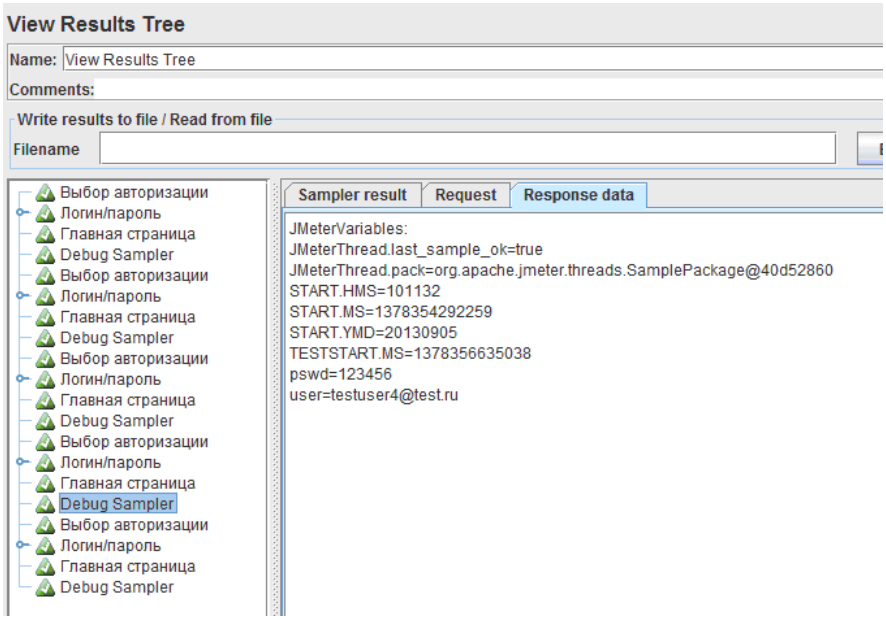


Рис.3.18. Результат

3.5.6. Створення фінансового аналізу

Після створення скрипта на вхід в систему, потрібно виконати те ж саме, тільки для створення фінансового аналізу (ФА). Для цього потрібно знову включити проксі-сервер і поставити галочку в браузері "використовувати проксі-сервер". У Jmeter створюється другий Recording Controller, який був названий "Створення ФА". Потрібно не забути поміняти Target Controller в проксі на dwarf> Створення ФА, як показано на рис.3.19.

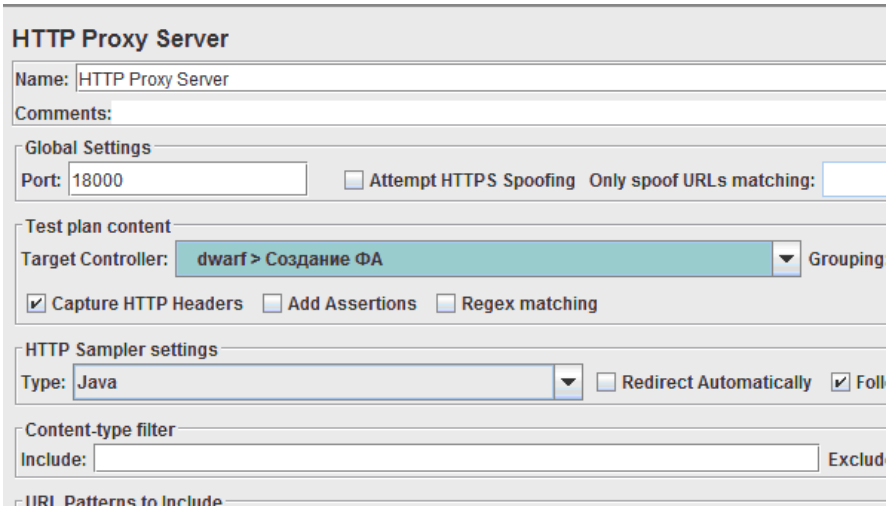


Рис.3.19. Зміна мети

Після запису скрипта (створення ФА) виходить наступний результат, зображений на рис.3.20.

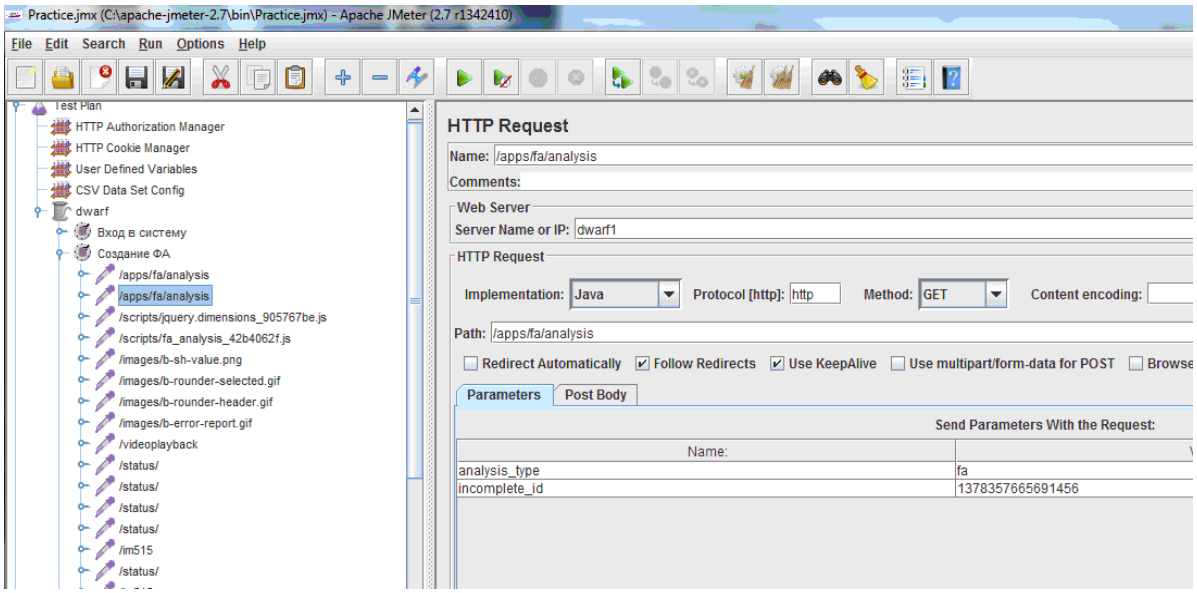


Рис.3.20. Скрипт створення ФА

Даний скрипт потрібно відчистити від сміття, залишити тільки найнеобхідніше і поміняти для зручності імена запитів.

Під час створення скрипта ФА потрібно знати наступне:

- при створенні ФА присвоюється унікальний і неповторний id;
- результат ФА також має унікальний прихований id.

Отже, необхідно параметризуванти ці 2 id.

Для параметризації прихованого id використовується CSV Data Set Config ще раз, як показано на рис.3.21.

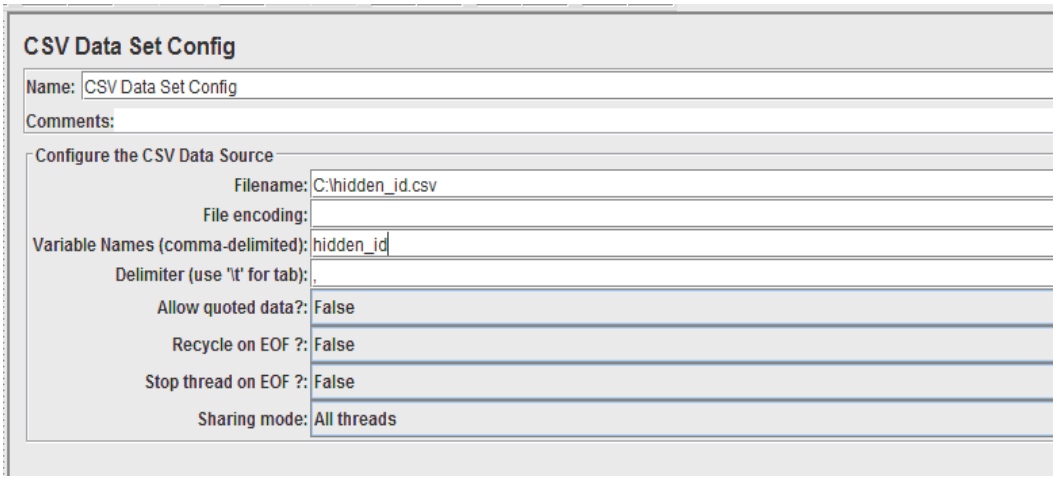


Рис.3.21. Параметризація прихованого id

Для параметризації унікального id, при створенні ФА, потрібно його спочатку дістати. Для цього використовується елемент ***Regular Experssion Extractor (dwarf> Add> Post Processors> Regular Experssion Extractor)***. Унікальний id знаходиться в адресному рядку і має приблизно такий вигляд: fa / analysis? Analysis_type = fa & incomplete_id = 1378393412734465, де incomplete_id і є ідентифікатор.

У Reference Name вказується ім'я параметра в Regular Expression записується регулярний вираз, яке буде діставати потрібний id при створенні нового ФА. $ 1 $ - означає порядок розстановки регулярного виразу, в нашому випадку воно одне. Якби було їх 2, то запис повинна бути $ 1 $$ 2 $. На рис.3.22 зображені налаштування Regular Expression Extractor.

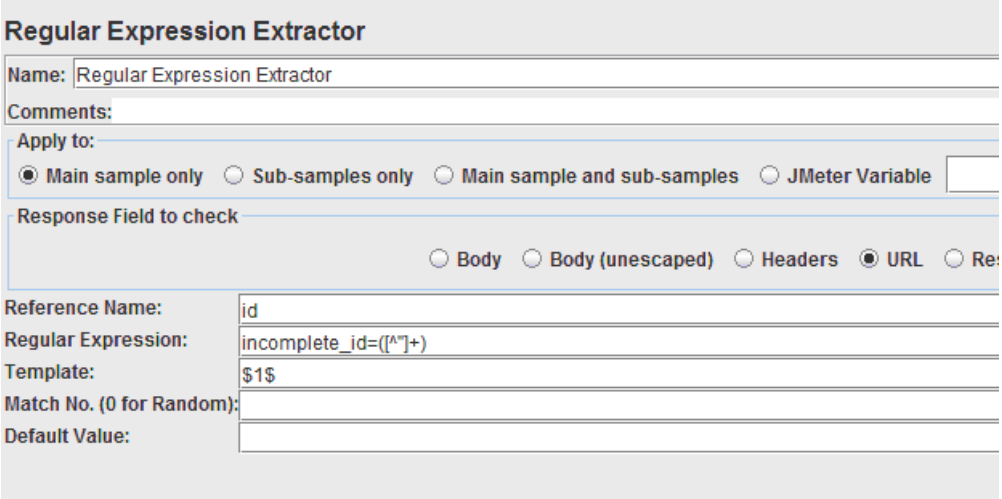


Рис.3.22. Налаштування Regular Expression Extractor

Тепер потрібно замінити встановлений id в скрипті на параметризований. Замість фіксованого ідентифікатора встановлюється створений параметр (рис.3.23).

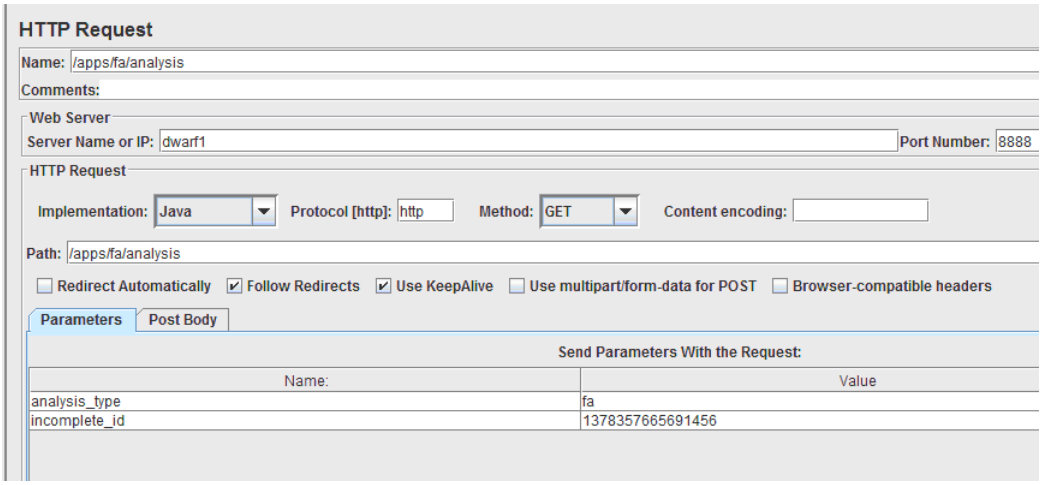


Рис.3.23. Заміна на параметр

Всі ці маніпуляції з id зроблені для того, щоб кожен користувач створював свій власний новий ФА, а не знову перестворював раніше вже створений аналіз.

Поставлена наступна задача: потрібно з'ясувати в якому місці, під час створення ФА, серверу необхідно найбільше часу для відповіді, при високому навантаженні.

Для цього була зімітована ситуація, коли в системі знаходиться одночасно 50

користувачів і вони створюють фінансовий аналіз. Для цього в Thread Group було проставлено кількість користувачів 50, а час, протягом якого ці 50 користувачів з'являються/виявляються в системі, 10 секунд (рис.3.24). Тобто за 10 секунд в системі буде працювати ~ 50 користувачів. Можливо така ситуація насправді малоймовірна, але існує завдання з'ясувати потенційно слабке місце в системі.

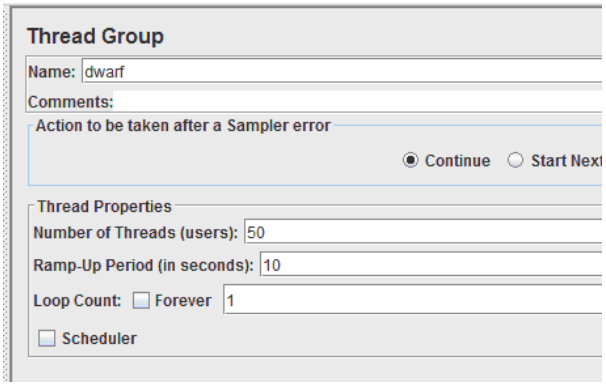
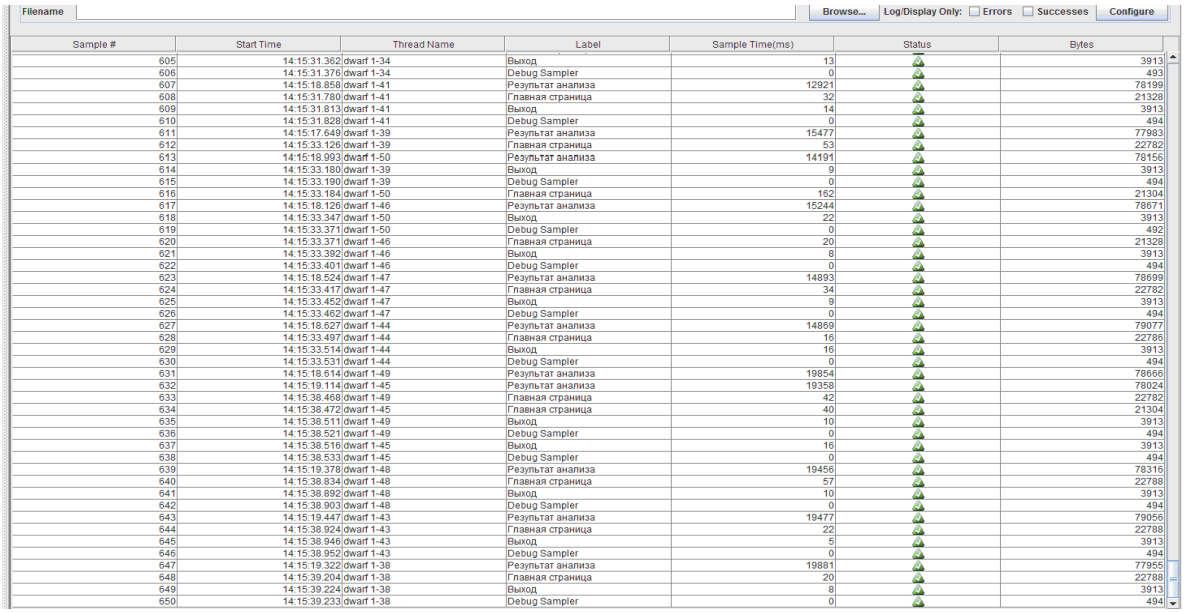


Рис.3.24. Установка 50 користувачів

Для того, щоб точно визначити слабку ланку був використаний елемент ***View Results in Table (dwarf> Add> Listener> View Results in Table)***. На рис.3.25 зображений результат тестування.



Filename | Browse... | Log/Display Only: Errors Successes | Configure

Sample #	Start Time	Thread Name	Label	Sample Time(ms)	Status	Bytes
605	14:15:31.362	dwarf 1-34	Выход	13		3913
606	14:15:31.376	dwarf 1-34	Debug Sampler	0		493
607	14:15:18.858	dwarf 1-41	Результат анализа	12921		78199
608	14:15:31.780	dwarf 1-41	Главная страница	32		21328
609	14:15:31.813	dwarf 1-41	Выход	14		3913
610	14:15:31.828	dwarf 1-41	Debug Sampler	0		494
611	14:15:17.649	dwarf 1-39	Результат анализа	15477		77983
612	14:15:33.126	dwarf 1-39	Главная страница	53		22782
613	14:15:18.993	dwarf 1-50	Результат анализа	14191		78156
614	14:15:33.180	dwarf 1-39	Выход	9		3913
615	14:15:33.190	dwarf 1-39	Debug Sampler	0		494
616	14:15:33.184	dwarf 1-50	Главная страница	182		21304
617	14:15:18.126	dwarf 1-46	Результат анализа	15244		78671
618	14:15:33.347	dwarf 1-50	Выход	22		3913
619	14:15:33.371	dwarf 1-50	Debug Sampler	0		492
620	14:15:33.371	dwarf 1-46	Главная страница	20		21328
621	14:15:33.392	dwarf 1-46	Выход	8		3913
622	14:15:33.401	dwarf 1-46	Debug Sampler	0		494
623	14:15:18.524	dwarf 1-47	Результат анализа	14893		78699
624	14:15:33.417	dwarf 1-47	Главная страница	34		22782
625	14:15:33.452	dwarf 1-47	Выход	9		3913
626	14:15:33.462	dwarf 1-47	Debug Sampler	0		494
627	14:15:18.627	dwarf 1-44	Результат анализа	14869		79077
628	14:15:33.497	dwarf 1-44	Главная страница	16		22786
629	14:15:33.514	dwarf 1-44	Выход	16		3913
630	14:15:33.531	dwarf 1-44	Debug Sampler	0		494
631	14:15:18.614	dwarf 1-49	Результат анализа	19854		78666
632	14:15:19.114	dwarf 1-45	Результат анализа	19358		78024
633	14:15:38.468	dwarf 1-49	Главная страница	42		22782
634	14:15:38.472	dwarf 1-45	Главная страница	40		21304
635	14:15:38.511	dwarf 1-49	Выход	10		3913
636	14:15:38.521	dwarf 1-49	Debug Sampler	0		494
637	14:15:38.516	dwarf 1-45	Выход	16		3913
638	14:15:38.533	dwarf 1-45	Debug Sampler	0		494
639	14:15:19.378	dwarf 1-48	Результат анализа	19456		78316
640	14:15:38.834	dwarf 1-48	Главная страница	57		22788
641	14:15:38.892	dwarf 1-48	Выход	10		3913
642	14:15:38.903	dwarf 1-48	Debug Sampler	0		494
643	14:15:19.447	dwarf 1-43	Результат анализа	19477		79056
644	14:15:38.924	dwarf 1-43	Главная страница	22		22788
645	14:15:38.946	dwarf 1-43	Выход	5		3913
646	14:15:38.952	dwarf 1-43	Debug Sampler	0		494
647	14:15:19.322	dwarf 1-38	Результат анализа	19881		77955
648	14:15:39.204	dwarf 1-38	Главная страница	20		22788
649	14:15:39.224	dwarf 1-38	Выход	8		3913
650	14:15:39.233	dwarf 1-38	Debug Sampler	0		494

Рис.3.25. Результат автоматизованого тестування роботи підсистеми Фінансового Аналізу

З таблиці можна побачити, що найслабша ланка в системі — це формування результатів аналізу, бо час, який потрібно серверу для цієї дії, щоразу зростає, і в підсумку може досягти критичної позначки 30 сек. В результаті цього користувач буде довго чекати свій аналіз, а у веб-додатку це критичний параметр. Аби цього не відбувалося, потрібно підвищити ефективність формування результату, щоб ця дія займала менше часу, а навантаження на сервер було знижене.

РОЗДІЛ 4. АВТОМАТИЗОВАНЕ ФУНКЦІОНАЛЬНЕ ТЕСТУВАННЯ

Автоматизоване тестування ПЗ (Automation Testing) — це процес верифікації програмного забезпечення, при якому основні функції та кроки тесту, такі як запуск, ініціалізація, виконання, аналіз і видача результату, виконуються автоматично за допомогою інструментів для автоматизованого тестування.

4.1. Переваги та недоліки

З автоматизацією тестування, як і з багатьма іншими вузьконаправленими IT-дисциплінами, пов'язано багато хибних уявлень. Для того, щоб уникнути неефективного застосування автоматизації, слід обходити її недоліки і максимально використовувати переваги.

Переваги автоматизованого тестування:

- повторюваність — всі написані тести завжди будуть виконуватися однаковим чином, тобто виключений «людський фактор». Тестувальник не пропустить тест з необережності і нічого не наплутає в результатах;
- швидке виконання — автоматизованому скрипту не потрібно звірятися з інструкціями і документаціями, це сильно економить час виконання;
- менші витрати на підтримку — коли автоматичні скрипти вже написані, на їх підтримку і аналіз результатів потрібно, як правило, менший час, ніж на проведення того ж обсягу тестування вручну;
- звіти автоматично розсилаються, також зберігаються звіти про результати тестування;
- виконання без втручання — під час виконання тестів інженер-тестувальник може займатися іншими корисними справами, або тести можуть виконуватися в неробочий час (цей метод краще, тому що навантаження на локальні мережі вночі знижена).

Недоліки автоматизації тестування:

- повторюваність — всі написані тести багато разів будуть виконуватися одноманітно. Це одночасно є недоліком, бо тестер, виконуючи тест вручну, може звернути увагу на деякі деталі і, провівши кілька додаткових операцій, знайти дефект, а скрипт цього зробити не зможе;
- витрати на підтримку — незважаючи на те, що у випадку автоматизованих тестів вони менше, ніж витрати на ручне тестування того ж функціоналу, вони все ж є. Чим частіше змінюється додаток, тим вони вищі;
- великі витрати на розробку — розробка автоматизованих тестів це складний процес, так як фактично йде розробка програми, яка тестує іншу програму. У складних автоматизованих тестах також є фреймворки, утиліти, бібліотеки та інше. Природно, все це потрібно тестувати і налагоджувати, а це вимагає часу;
- вартість інструменту для автоматизації — в разі якщо використовується ліцензійне ПЗ, його вартість може бути досить висока. Вільно розповсюджувані інструменти, як правило, відрізняються більш скромним функціоналом і меншою зручністю роботи;
- пропуск дрібних помилок — автоматичний скрипт може пропускати дрібні помилки, на перевірку яких він не запрограмований. Це можуть бути неточності в позиціонуванні вікон, помилки в написах, які не перевіряються, помилки контролю і форм, з якими не здійснюється взаємодія під час виконання скрипта.

Щоб прийняти рішення про доцільність автоматизації тестування додатка, слід відповісти на питання «чи переважують в нашому випадку переваги?» хоча б для деякої функціональності нашого застосування. Якщо не можна знайти таких частин, а недоліки в цьому випадку неприйнятні, від автоматизації варто утриматися.

При прийнятті рішення варто пам'ятати, що альтернатива — це тільки ручне тестування, у якого є множина своїх недоліків.

4.2. Застосування автоматизації

Автоматизацію потрібно застосовувати в наступних випадках:

1) важкодоступні місця в ПС (бекенд процеси, логування файлів, запис в БД);

2) часто використовувана функціональність, ризики від помилок в якій досить високі. Автоматизувавши перевірку критичної функціональності, можна гарантувати швидке знаходження помилок та швидке їх усунення;
3) рутинні операції, такі як перебори даних (форми з великою кількістю введених полів). Можна автоматизувати заповнення полів різними даними і їх перевірку після збереження;
4) валідаційні повідомлення (автоматизувати заповнення полів некоректними даними і перевірку на появу тієї чи іншої валідації);
5) довгі end-to-end сценарії;
6) перевірка даних, що вимагають точних математичних розрахунків;
7) перевірка правильності пошуку даних.

А також, багато іншого, в залежності від вимог до тестованої системи і можливостей обраного інструменту для тестування.

Для більш ефективного використання автоматизованого тестування краще розробити окремі тест-кейси, що перевіряють:

- базові операції створення / читання / зміни / видалення сутностей (так звані CRUD операції - Create / Read / Update / Delete). Приклад: створення, видалення, перегляд і зміна даних про користувача;
- типові сценарії використання додатка, або окремі дії. Приклад: користувач заходить на поштовий сайт, гортає листи, переглядає нові, пише і відправляє листи, виходить з сайту. Це end-to-end сценарій, який перевіряє сукупність дій. Ми пропонуємо вам використовувати саме такі сценарії, бо вони дозволяють повернути систему в стан, максимально близький до вихідного, а значить мінімально впливає на інші тести;
- інтерфейси, робота з файлами та інші моменти, незручні для тестування вручну. Приклад: система створює певний .xml файл, структуру якого необхідно перевірити.

Це і є функціональність, від автоматизації тестування якої, можна отримати найбільшу віддачу.

4.3. Як автоматизувати тестування

В даному розділі розглянуті аспекти, що впливають на вибір інструменту автоматизації тестування.

По-перше, потрібно звернути увагу наскільки добре інструмент для автоматизації розпізнає елементи управління в вашому додатку. У разі коли елементи не розпізнаються варто пошукати плагін, або відповідний модуль. Якщо такого немає, від інструменту краще відмовитися. Чим більше елементів може розпізнати інструмент — тим більше часу можна заощадити на написанні і підтримці скриптів.

По-друге, потрібно звернути увагу на те скільки часу потрібно на підтримку скриптів, написаних за допомогою інструменту. Для цього необхідний простий скрипт, який обирає пункт меню, а потім уявіть що змінився пункт меню, який необхідно вибрати. Якщо для відновлення працездатності сценарію доведеться перезаписати скрипт цілком, то інструмент не оптимальний (реальні сценарії набагато складніше). Найкращий той інструмент, який дозволяє винести назву кнопки в змінну на початку скрипта і швидко замінити її значення. В ідеалі: описати меню, як клас.

По-третє, наскільки зручний інструмент для написання нових скриптів. Скільки потрібно на це часу, наскільки можна структурувати код (підтримка ООП), наскільки код програмного додатка читабельний, наскільки зручне середовище розробки для проведення рефакторінга (переробки коду) і т.п.

Найкраще, для прийняття правильного рішення про автоматизацію, відповісти на питання: «Навіщо? Що? Як?». Саме в такому порядку. Це допоможе уникнути даремно витраченого часу і фінансів та отримати надійність, швидкість і якість.

4.4. Рівні автоматизації тестування

Умовно, тестований додаток можна розбити на 3 рівня: **unit tests layer; functional tests layer (Non-UI); GUI tests layer.**

Для забезпечення кращої якості продукту, рекомендується автоматизувати всі 3 рівня. Розглянемо більш детально стратегію автоматизації тестування на основі трирівневої моделі:

Рівень модульного тестування (Unit Test Layer)

Під автоматизованими тестами на цьому рівні розуміються Компонентні або Модульні тести написані розробниками. Тестувальникам ніхто не забороняє писати такі тести, які будуть перевіряти код, якщо їх кваліфікація дозволяє це. Наявність подібних тестів на ранніх стадіях проекту, а також постійне їх поповнення новими тестами, перевіряючими «баг-фікси», вберігає проект від багатьох серйозних проблем.

Рівень функціонального тестування (Functional Test Layer (non-UI))

Чи не всю бізнес-логіку можна протестувати через GUI-шар. Це може бути особливістю реалізації, яка ховає бізнес-логіку від користувачів. Саме з цієї причини за домовленістю з розробниками, для команди тестування може бути реалізований доступ безпосередньо до функціонального шару, що дає можливість тестувати безпосередньо бізнес-логіку додатка, минаючи призначений для користувача інтерфейс.

Рівень тестування через призначений для користувача інтерфейс (GUI Test Layer)

На даному рівні є можливість тестувати не тільки інтерфейс користувача, але також і функціональність, виконуючи операції, що викликають бізнес-логіку програми. З нашої точки зору, такого роду наскрізні тести дають більший ефект, ніж просто тестування функціонального шару, так як ми тестуємо функціональність, емулюючи дії кінцевого користувача, за допомогою графічного інтерфейсу.

4.5. Архітектура тестів

Для зручності накладення автоматизованих тестів, на вже наявні тест-кейси, структура тест-скриптів повинна бути аналогічна структурі тестового випадку — *Precondition, Steps & Post Condition.*

Отримуємо правило, що кожен тест-скрипт повинен мати: **precondition; steps (Test); post Condition.**

Перелічимо основні функції скрипта:

1) precondition:

- ініціалізація додатка (наприклад, відкриття головної сторінки, вхід під тестовим користувачем, перехід в необхідну частину програми та підведення системи до стану придатного для тестування);
- ініціалізація тестових даних.

2) steps:

- безпосереднє проведення тесту;
- занесення даних про результат тесту, з обов'язковим збереженням причин провалу (failed) і кроків, за якими проходив тест.

3) post condition:

- видалення, створених в процесі виконання скрипта, непотрібних тестових даних;
- коректне завершення роботи програми.

Рекомендується, також, створити загальну бібліотеку по обробці помилок і виняткових ситуацій, наприклад: *PreConditionException; TestCaseException; PostConditionException.*

У підсумку, скориставшись вищеописаними рекомендаціями, будемо реалізовувати загальну архітектуру тест-скриптів і сценаріїв.

4.6. Створення проекту з автоматизованого тестування

Розгляд даного проекту переслідує наступну мету: стати керівництвом для тестера-автоматизатора, допомогти йому швидко створити перші авто-тести і продовжити в подальшому автоматизувати процес тестування в своїй роботі.

Встановлення програм та налаштування проекту

Для роботи будуть потрібні наступні інструментальні програмні засоби:

- Git — лідер сучасних систем управління версіями файлів (спільна робота);
- Visual Studio 2012 (або вище) — для написання та автоматизованного виконання ТНД, використовуючи інфраструктуру Nunit через Test Explorer і Run All. Результати тестів підсумовуються;
- Nunit — відкрите середовище для модульного тестування застосунків та додатків, яке перенесене з мови Java (бібліотека JUnit).
- Resharper — для збільшення продуктивності роботи та автоматизації рефакторингу в середовищі Microsoft Visual Studio, а також для зручності запуску тестів і виконання дебага (налагодження);
- Selenium IDE — плагін для Firefox (для зручності розпізнавання локаторів елементів на сторінках під час тестування).

Щоб закачати проект зі сховищ, необхідно в папці, де слід його розмістити, запустити git bash і виконати команду: *git clone git: //github.com/4gott3n/AT.git master* .

Далі потрібно відкрити файл AT.sln в Visual Studio. Відкриється Solution, що містить проект AT (фреймворк), а також приклад тестового проекту, в якому можна побачити реалізацію сторінок, тестів тощо. Цей проект є зручним прикладом для створення свого власного проекту.

Наступний крок — це створення нового проекту, який буде зберігати всі тести, сторінки і все нобхідне.

Що потрібно зробити:

1) Додати в Solution новий проект (Class library). Цю назву пізніше потрібно прописати в App.config;

2) Підключити посилання.

У підсумку все повинно бути, як на рис.4.1.

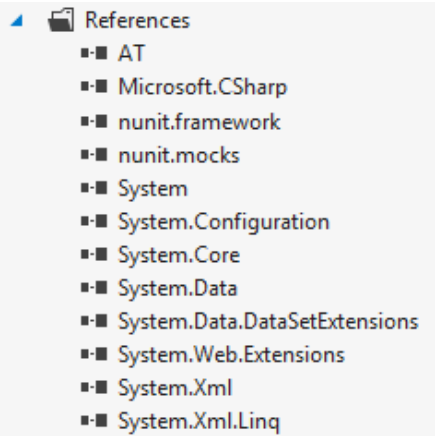


Рис.4.1. Новий проект

3) Створити файл конфігурації App.config;

У цьому файлі міститимуться всі основні параметри проекту.

Детальний зміст файлу App.config:

```
<add key="project_name" value="SampleTestProject"/> <!-- назва проекту -->

<!-- блок налаштувань для розсилки звіту про запуск тестів -->

<add key="smtp_server" value="адреса smtp сервера" />
<add key="smtp_port" value="порт" />
<add key="smtp_login" value="логін" />
<add key="smtp_password" value="пароль" />
<add key="mail_from" value="ваш email" />
<add key="mail_to" value="список розсилки через кому" />

<!-- end notif config -->
<!-- блок налаштувань браузера -->
```

```
<add key="ImplicitlyWait" value="час очікування елемента (в секундах)" />
<add key="WaitForAjax" value=" час очікування виконання ajax (в секундах)" />
<add key="browser" value="поточний браузер (firefox, chrome чи iexplorer)" />
<add key="browser_check_url" value="http://ya.ru" />

<!-- сторінка для перевірки роботи браузера -->
<!-- browser config end --> <!-- налаштування сторінок(webpages) -->

<add key="Yandex" value="http://yandex.ru/" /> <!-- відповідність назви папки всередині WebPages головній сторінці системи, що тестується (в даному випадку всередині папки WebPages повинна бути папка Yandex) -->

<add key="dash_prefix" value="   " />
<!-- позначення тире в назві namespace (три нижніх слеша)-->

<add key="extension_prefix" value="   " /> <-- префікс перед розширенням файлів сторінок (два нижніх слеша), наприклад сторінка index-1.html повинна мати назву класу index 1 html -->
<add key="folder_index_prefix" value="_fld" />
<!-- відкрити папку без імені файлу сторінки, якщо нам потрібно відкрити папку, не вказуючи назви файлу, наприклад сторінка https://yandex.ru/test/, у цьому випадку в папці Yandex має бути папка test, а в ній клас _fld -->

<!-- pages config end-->
<!-- блок налаштувань бази даних -->

<add key="database_name_prefix" value="db_" /> <!-- префікс рядків конфігу, що містять рядки ініціалізації БД -->
<add key="database_selected_rows_limit" value="50" /> <!-- обмеження на вибірку (кількість рядків) -->
<add key="db_test" value="172.18.XX.XX;1521;SID;LOGIN;PASSWORD;oracle" /> <!-- приклад рядка ініціалізації БД -->

<!-- формат: host;port;sid;login;password;type (type може бути або Oracle, або Mssql -->

<!-- data base config end -->

<add key="test_step_prefix" value="step_" /> <!-- префікс у назвах способів для кроків тест-кейсу, наприклад  step_01 -->
<add key="test_case_prefix" value="test_" /> <!-- префікс у назвах класів тестів -->

<add key="datetime_string_format" value="yyyy-MMM-dd -> hh:mm:ss" /> <!-- формат дати для логування -->
 <add key="log_file_name" value="system.log" /> <--! назва файлу для логів -->
```

4) Створити папку для сторінок (WebPages);

У цій папці будуть знаходитися файли з класами сторінок.

Детальніше щодо WebPages.

Кілька правил:

- папка зі сторінками повинна називатися WebPages;

- вона повинна знаходитися в корені проекту;

- всередині неї повинні міститися кореневі папки ваших тестованих систем

(наприклад якщо тестувати Yahoo, то в папці WebPages повинна бути папка yahoo);

- сторінки в папках повинні знаходитися в тій же ієрархії, як вони знаходяться в тестованої системі.

Приклад:

Сторінка: test.ru/step1/service/index.html

Ієрархія папок: WebPages -> Test -> step1 -> service -> index.html.cs

(Важливо: імена папок і класів мають бути з урахуванням регістру).

- назви файлів класів сторінок повинні відповідати реальним іменам сторінок.

Назви класів сторінок:

Приклад:

Сторінка: index-1.html, Клас: index 1 html

Тут:

- дефіс в назві сторінки замінюється трьома *слеш* (можна змінити, параметр dash_prefix в App.config);
- точка файлу замінюється двома *слеш* (можна змінити, параметр extension_prefix в App.config).

Назви класів, коли немає назви файлу, а є тільки папка:

Приклад:

Сторінка: test.ru/step1/service/

Клас: _fld (можна змінити, параметр folder_index_prefix в App.config).

5) створити клас Pages.cs;

```
public static class Pages { }
```

Клас буде містити об'єкти класів сторінок (WebPages).

Детальніше щодо складу Pages.cs

Цей клас містить об'єкти класів сторінок, через які здійснюється доступ до елементів і т.д.

Кілька правил:

- в даному класі ієрархія підкласів повинна збігатися з ієрархією в папці

WebPages;

- клас і всі об'єкти всередині нього повинні бути типу public static .

Приклад класу:

```
public static class Pages
{
        public static class Test
        {
                public static index html Index = new index
                html(); public static class step1
                {
                        public static class service
                         {
                                public static _fld Main = new _
                                fld();
                         }
                         public static add  php Add = new add  php();
                }
        }
}
```

При такому записі сторінки будуть доступні в тестах приблизно так:

```
Pages.Test.Index.Open();
Pages.Test.step1.service.Main.Open
();
```

6) Створити папку для тестів (Tests);

Тут немає особливих правил, досить створити в корені проекту папку Tests, всередині неї створити папки з тестами по тестованим системам.

Детальніше. Tests

На рисунку 4.2 зображено як це зроблено в проекті:

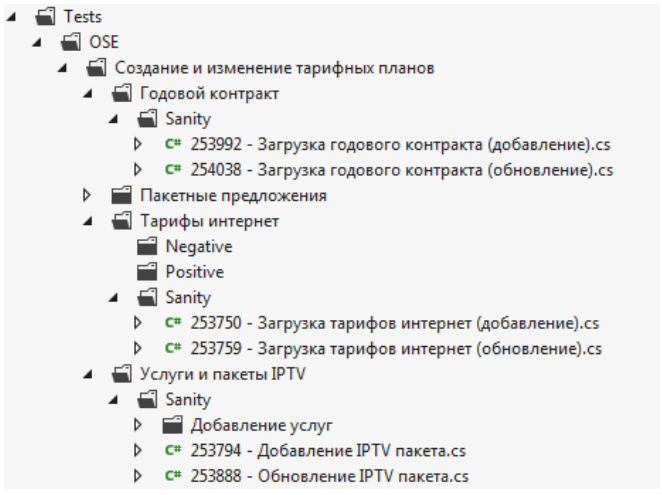


Рис.4.2. Ієрархія Tests

Рекомендується створити два «службових» тести, які завжди будуть запускатися

першим і останнім. У першому тесті можна виконувати різні дії з налаштування середовища, в останньому, наприклад, виконувати очистку стенду (платформи) тестування і запускати нотифікацію. Nunit запускає тести всередині категорії в алфавітному порядку (по повній назві класів).

Для запуску нотифікації необхідно виконати код:

```
AT.Service.Notifier.SendNotif();
```

Приклад створення сторінки (WebPages)

Всі створювані сторінки успадковуються від базового класу PageBase, який містить необхідні методи, однакові для всіх сторінок: «відкрити», «відкрити з параметром», «Отримати Url».

Приклад класу сторінки:

```
public class index  php : PageBase
  {
     public void OpenVpdnTab()
     {
        new WebElement().ByXPath("//a[contains(@href, '#internet')]").Click();
     }
     public string VpdnAction
     {
        get                    {                    return                    new
WebElementSelect().ByXPath("//select[@name='action']").GetSelectedValue(); }
        set                                 {                                    new
WebElementSelect().ByXPath("//select[@name='action']").SelectByValue(value); }
     }
     public string VpdnLid
     {
        set { new WebElement().ByXPath("//input[@name='lid']").SendKeys(value);
        }
     }
     public string VpdnTechlist
     {
        set { new
        WebElement().ByXPath("//input[@name='file']").SendKeys(value); }
     }
     public string VpdnStartDate
     {
        set                                 {                                    new
WebElement().ByXPath("//input[@name='start_date']").SendKeys(value); }
     }
     public void VpdnSubmit()
     {
        new
        WebElement().ByXPath("//input[@value='Применить']").Click();
     }
  }
```

Правило:

Всі елементи, які присутні на сторінці, повинні ініціалізуватися в момент звернення до них.

Приклади роботи зі сторінками:

`Pages.Test.Index.Open();` — відкрити

`Pages.Test.Index.Open("?id=1");` — відкрити з параметром

`var url = Pages.Test.Index.Url;` — отримання адреси сторінки

`Pages.Test.Index.VpdnSubmit();` — запуск функції, прописаної в класі сторінки вище

Приклад створення тест-кейса (Test)

Як вже зазначалося вище, всі тест-кейси повинні бути розміщені в папці Tests. Всі тест-кейси повинні бути public і успадковуватися від базового класу TestBase. Перед назвою класу з тест-кейсом повинен бути вказаний атрибут [TestFixture]. Перед кожним кроком тест-кейса повинен бути вказаний атрибут [Test].

Детальніше про атрибути тестів можна почитати в документації NUnit.

Приклад класу з тест-кейсом

```
namespace TestProject.Tests.OSE
{
   [TestFixture]
   [Category("OSE"), Category("OSE_Internet")] /*  категорії,  nunit  дозволяє
запускати тести вибірково за категоріями */
   public class test_253750 : TestBase
   {
      [Test]
      public void step_01()
      {
         Pages.OSE.Inaclogin.Open();
         Pages.OSE.Inaclogin.Login = "user";
         Pages.OSE.Inaclogin.Password =
         "password";
         Pages.OSE.Inaclogin.Submit();

         Assertion( "Помилка авторизації", () =>
     Assert.AreEqual(Pages.OSE.Inaclogin.IsAuthSuccess, true));
      }
    }
}
```

`Assertion` – це перевірка виконання умов.

Формат запису:

```
Assertion (текст помилки, () => Assert.будь-який_accert_из_nunit() );
```

Чому не використовується просто Assert? Всі Assert'и відловлюються

спеціальним класом, в якому виконуються дії по реєстрації помилок, логуванню і т.д.

Приклади роботи з БД

Для роботи з БД використовується клас AT.DataBase.Executor, що містить методи:

- виконання запитів на вибірку (select);
- виконання запитів типу insert, update, delete (unselect);
- виконання збережених процедур.

Приклади:

Select:

```
var query = select col1, col2 from table_name";
var list = Executor.ExecuteSelect(query, Environment.Ім’я_БД);
```

Unselect:

```
var query = "DELETE FROM table_name";
Executor.ExecuteUnSelect(query,
Environment.Имя_БД);
```

Виконання процедури:

```
Executor.ProcedureParamList.Ad ProcedureParam("varchar    "ім’я_параметр
d(new                                                    ",          ");"
Executor.ProcedureParamList.Ad  ProcedureParam("varchar                  "/*
значення */
var res = Executor.ExecuteProcedure("ім’я_процедури",
Environment.Ім’я_БД);
```

Ім' я_БД має відповідати значенням sid з рядка ініціалізації БД в App.config.

```
<add  key="db_test" value="172.18.XX.XX;1521;SID;LOGIN;PASSWORD;oracle" />  <!
      --
приклад рядка ініціалізації БД -->
```

Збір результатів тестування і приклад звіту:

Як уже згадувалося вище, для запуску нотифікації і отримання звіту на пошту необхідно після запуску останнього тесту виконати код AT.Service.Notifier.SendNotif (). Логіка NUnit така, що він запускає тести в алфавітному порядку, тому, щоб потрібний тест запустився останнім, його потрібно назвати відповідним чином. Налаштування оповіщення вказуються в файлі App.config. На рисунку 4.3 зображено приклад звіту.

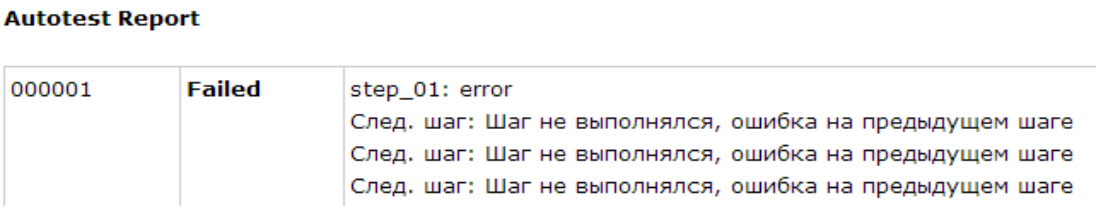


Рис.4.3. Приклад звіту

Виконавши всі пункти цієї інструкції тестер-автоматизатор зможе в короткі терміни створити свій проект і отримати необхідний мінімум на початку цього шляху.

4.7. Проект авто-тестів для веб-сервісу

Тепер розглянемо завершений проект з автоматизованого тестування веб-сервісу. В якості прикладу візьмемо веб-сервіс для оцінки фінансового стану підприємства та оцінки ймовірності його податкової перевірки. Мета створення проекту: 1) прискорити регресійне тестування і випуск оновлень; 2) прискорити час проходження всіх тестів.

Спочатку автоматизовані тести були написані на мові C#, але з огляду на їх повільне проходження (через велику їх кількість, специфіку самого веб-сервісу і те, що працювати з ними можна було тільки під Windows), було прийнято рішення переписати всі тести на мові функціонального програмування Scala, яка потребує Java-машину (в нашому випадку – середовище розробки Eclipse). Технічний драйвер, який пов'язує тести і бразуер – Selenium Web Driver, який не є інструментом автоматизації тестування, а лише засобом управління браузером.

Як приклад для написання тестів, розглянута сторінка, зображена на рис.4.4. Виконано написання для неї інфраструктури та тест-кейсів і ТНД. Налаштування майданчиків для тестування і конфігураційних файлів, а також сам запуск тестів будуть опущені, бо в якості платформи для автоматизованого тестування можна використати **Jmeter** або **Microsoft Visual Studio**, які сконфігуровано і розглянуто в п.3.5 та п.4.6.

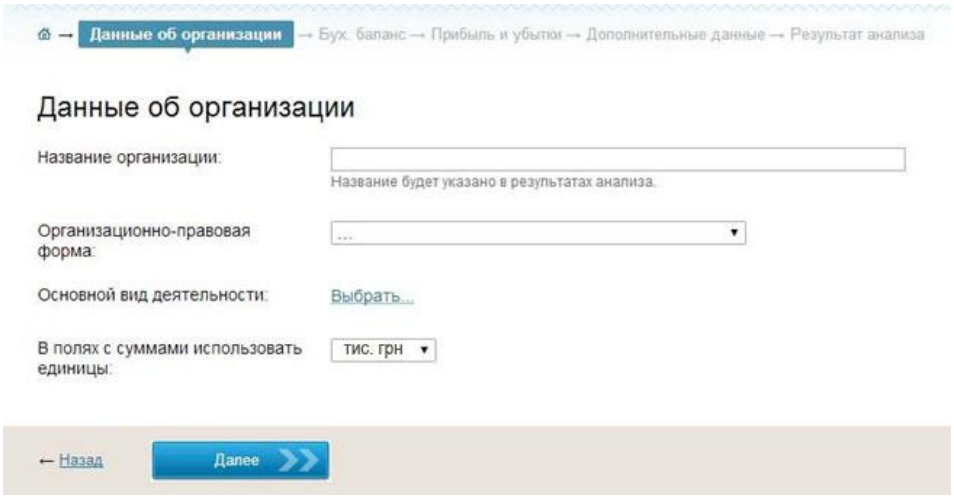


Рис.4.4. Дані про організацію

4.7.1.Опис інфраструктури сторінки

Перед тим як почати писати тест-кейси для сторінки, її потрібно описати, тобто створити інфраструктуру. Інфраструктура являє собою опис тестованих елементів розглянутої сторінки. В даному випадку це поля: назви організації, ОПФ, ОВД і грошової одиниці; кнопки "назад" і "далі"; а також різні валідаційні повідомлення.

Щоб описати інфраструктуру елементів сторінки, потрібно знати як звернутися до цих елементів. В цьому допоможе консоль розробника в браузері, зображена на рис.4.5.

Створення нового елемента є одночасно перевіркою того, що цей елемент є на сторінці і він видимий. Якщо елементи з'являються після певних дій, або в повному обсязі елементи можуть одночасно бути присутніми на сторінці, потрібно використовувати **lazy val** за допомогою модифікатора. В такому випадку, перше звернення до елементу буде одночасно і перевіркою його існування на сторінці.

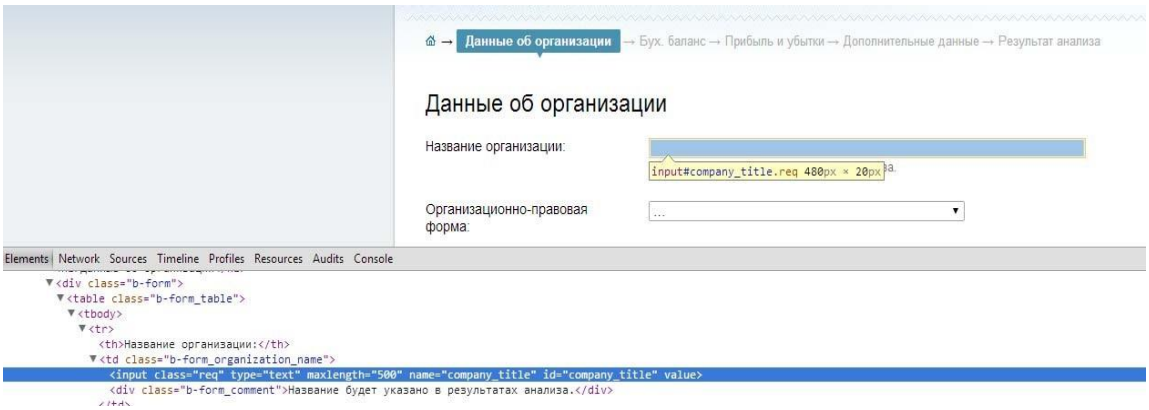


Рис.4.5. Консоль розробника

Дізнавшись як звертатися до елементу, можна використати це у коді:

```
val companyTitle = new ExpertInput(ByJQ("input#company_title"))
```

Оскільки у поле "Назва організації" відбувається введення даних, то використовується елемент ExpertInput і запит JQ, щоб можна було визначити дане поле на сторінці.

За тим же принципом описується елемент ОПФ (Організаційно-правові форми). Але ОПФ є списком, який зображений на рис.4.6, а тому використовується вже не ExpertInput, а ExpertSelect. При цьому потрібно описати всі елементи цього списку, для цього використовується об'єкт OrgTypes:

```
object OrgTypes extends StringEnumeration {
  val empty = StringValue("…")
  val OOO = StringValue("Общество с ограниченной ответственностью")
  val OAO = StringValue("Открытое акционерное
  общество") val ZAO = StringValue("Закрытое
  акционерное общество") val UP =
  StringValue("Унитарное предприятие")
  val ODO = StringValue("Общество с дополнительной ответственностью")
  val PT  = StringValue("Полное товарищество")
  val TV  = StringValue("Товарищество на вере")
  val PK  = StringValue("Производственный кооператив")
  val KKH = StringValue("Крестьянское (фермерское) хозяйство")
  val NKO = StringValue("Некоммерческая организация")
  val IP  = StringValue("Индивидуальный предприниматель")
  val OBPUL = StringValue("Организация без прав юридического лица")
}
```

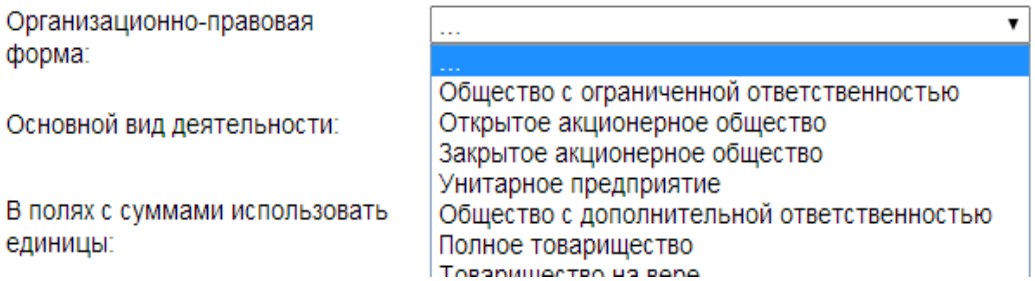


Рис.4.6. Список ОПФ

Грошові одиниці теж є списком, але тому, що він складається всього з двох елементів, прийнято рішення описати вибір між елементами як дві функції.

```
  def chooseThousandsUnit : FaCompanyInfoPage[Intro,
    Result] = {
    unitOfMeasure.select(UnitsOfMeasure.thousands)
    new FaCompanyInfoPage(url)
  }

  def chooseMillionsUnit : FaCompanyInfoPage[Intro,
    Result] = {
    unitOfMeasure.select(UnitsOfMeasure.millions)
    new FaCompanyInfoPage(url)
  }
```

На сторінці також є й приховані елементи, показані на на рис.4.7, які з'являються на ній тільки за певних умов. В нашому випадку — це валідаційні повідомлення. Їх теж необхідно описати, тому що всі тест-кейси будуть націлені на перевірку валідації, тобто на правильну обробку і результати, які відповідають вимогам до ПС.

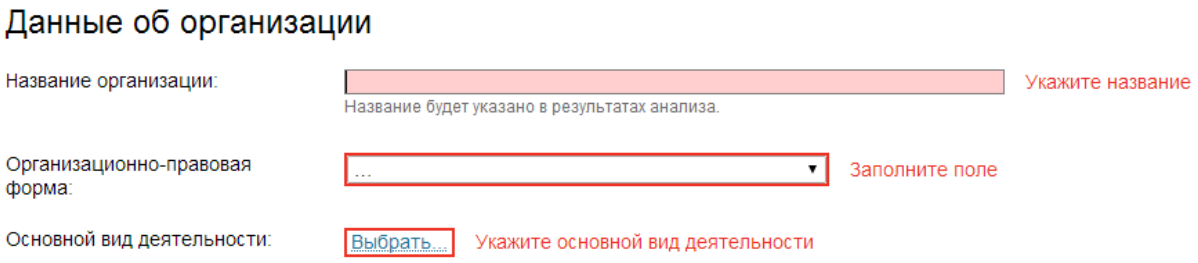


Рис.4.7. Приховані елементи

Код для тексту валідації:

```
val companyNameValidation = "Вкажіть назву"
val orgTypeValidation = "Заповніть поле"
val onvdCodeValidation = "Вкажіть основний вид діяльності"
lazy val moneyUnitValidation = "Перемикання одиниць виміру неможливо - значення
повинні бути цілими числами"
```

Функції перевірки валідації:

```
def checkCompanyNameValidation() =
checkText(ByJQ("label[for='company_title']"), companyNameValidation)

def checkOrgTypeValidation() = checkText(ByJQ("label[for='org_type']"),
orgTypeValidation)

def checkOnvdCodeValidation() =
checkText(ByJQ("label[for='onvd_code']"), onvdCodeValidation)

def checkMoneyValidation() {
  moneyUnitElement.text should be(moneyUnitValidation) }
```

Повний опис сторінки можна побачити у Додатку А.

4.7.2.Написання тест-кейсів

Розглянемо простий сценарій: після того як користувач потрапляє на сторінку "Дані про організацію" він натискає кнопку "Далі", не заповнивши жодного поля. Очікуваний результат: спрацює валідація на всі поля. Після цього він заповнить одне поле — тоді валідація з усіх полів зникає.

Оскільки для кожного тесту потрібно знаходитися на сторінці "Дані про організацію", то необхідно повторюваний код ввести у функцію. В даному випадку — це кроки вибору фінансового аналізу, періоду і можливість заповнення полів вручну.

Код у цьому випадку виглядає наступним чином:

```
  protected def setup = {
    IntroPage().new_fa.proceed().chooseManual.selectPeriod(Years.year2019,
    PeriodTypes.year)
```

Після цього йде опис кроків у коді:

```
"Click next with empty fields than fill onvd code - all fields" should "be valid" in
{
   val companyInfoPage = setup
   companyInfoPage.nextPageInactive.click()
   companyInfoPage.checkCompanyNameValidati
   on()
   companyInfoPage.checkOnvdCodeValidation(
   )
   companyInfoPage.checkOrgTypeValidation()

   companyInfoPage.onvdCodeSelectorLink.proceed().checkOnvd(
   "15") withWait() {new
   ExpertText(ByJQ("label[for='company_title']"))}
   withWait() {new
   ExpertText(ByJQ("label[for='org_type']"))} withWait()
   {new ExpertText(ByJQ("label[for='onvd_code']"))}
}
```

Кроки:

- спрацьовує повторюваний код — відкривається сторінка "Дані про організацію";
- натискання на кнопку "Далі";
- перевірка, що з'явилася валідація для полів назви організації, ОПФ, ОВД;
- вибирається зі списку ОВД значення;
- перевіряється, що валідація з усіх трьох полів зникає.

Розглянемо ще один сценарій: користувач заповнює всі поля на сторінці "Дані про організацію", натискає кнопку "Далі" і переходить на наступну сторінку. Після цього він натискає на кнопку "Назад" і потрапляє на попередню сторінку, при цьому всі поля повинні бути заповнені старими значеннями.

Код:

```
"Fill all fields, click next than return back - all fields" should "be filled" in {
    val companyInfoPage = setup
    companyInfoPage.companyTitle.text =
    OrgName.name
    companyInfoPage.orgType.select(OrgTypes.ODO)
companyInfoPage.onvdCodeSelectorLink.proceed().checkOnvd("15").nextPage.proceed().bac
kwardLin k.proceed()
```

```
    companyInfoPage.companyTitle.text should be(OrgName.name)
    companyInfoPage.orgType.getSelected should be(OrgTypes.ODO.toString())
    companyInfoPage.onvd.text should be("15. Виробництво харчових продуктів,
    включаючи напої")
  }
```

Кроки сценарію:

- спрацьовує повторюваний код — відкривається сторінка "Дані про організацію";
- заповнюються поля "Назва організації", ОПФ, ОВД;
- натискається кнопка "Далі";
- після переходу на нову сторінку натискається кнопка "Назад";
- відбувається перевірка, що всі поля заповнені старими значеннями.

Весь набір тест-кейсів для даної сторінки можна оглянути в Додатку Б.

Розроблений проект значно прискорив регресійне тестування і випуск оновлень. Ефективність тестування та швидкість проходження тестів була збільшена майже в два рази. Перед кожним апдейтом програмної системи тести проганяються на тестовому майданчику і тільки після їх успішного виконання відбувається оновлення на головному робочому сервері компанії-розробника.

Для створення ТНД застосована Scala — мультипарадигмова мова програмування, що поєднує властивості об'єктно-орієнтованого та функційного програмування. Програми мовою Scala виконуються на віртуальній машині Java за умови приєднання до дистрибутиву файлу `scala-library.jar`. Scala сумісна із існуючими програмами на мові Java, тобто код Scala може викликатися з Java-програм і навпаки. Scala пристосована для того, щоб для кожної предметної області можна було побудувати жорстко типізовану модель та виразити її у мовних конструкціях, які будуються в статично-типизованому середовищі.

Код опису класів на Scala втричі коротший у порівнянні з аналогічним кодом на Java, що дозволило досягти компактного запису тест-кейсів. Сьогодні Scala використовується для back-end розробки багатьох веб-сайтів відомих компаній.

Використання цієї мови в робочому проекті дозволило вдвічі підвищити ефективність автоматизованого тестування web-сервісів ПС, що розробляються.

ВИСНОВКИ

У ході виконання дипломної роботи у Розділі 1 та у Розділі 2 був досліджений процес тестування програмного забезпечення в ІТ-компаніях. Детально розглянуті більшість видів дефектів ПС, причини виникнення та способи їх відстеження за допомогою системи стеження за помилками. Також розглянуті способи створення і використання тест-кейсів для застосування їх в стратегії тестування “чорного ящика”.

Розроблений матеріал можна використовувати як для навчальних цілей, так і для успішного проходження технічної співбесіди під час влаштування на роботу в якості тестувальника. Також, користуючись цим матеріалом, можна сформувати методичний посібник для студентів напряму “Комп’ютерні науки” і включити його в програму навчання в якості додаткового матеріалу по дисципліні "Тестування ПЗ".

У Розділі 3 дипломної роботи були розглянуті програмні платформи для виконання навантажувального тестування. За допомогою CASE-засобу JMeter проведено навантажувальне тестування веб-сервісу, під час якого була виявлена слабка ланка у функціональній підсистемі фінансового аналізу веб-сайту.

У Розділі 4 були створені 2 проекти з автоматизованого функціонального тестування. Перший проект навчальний і несе ознайомлювальний характер. Він націлений на користь початківцю-тестеру для засвоєння ним принципів автоматизованого тестування, з метою допомогти йому швидко створити перші авто-тести і продовжити автоматизувати тестування в подальшій практиці.

Другий проект створює авто-тести для веб-сервісів, причому нині він використовується на головному робочому сервері компанії. Розроблений у дипломній роботі проект значно прискорив регресійне тестування і випуск оновлених версій програмних систем за рахунок використання мови Scala (на платформі Java). Використання особливостей мови Scala дозволило значно скоротити підготовку ТНД і підвищити ефективність тестування. Перед кожним апдейтом чергової версії системи тести прогоняються на тестовому майданчику і тільки після успішного їх виконання відбувається оновлення ПС на сервері компанії-розробника.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Бібліотека MSDN. Джерело інформації для розробників, які використовують засоби, продукти, технології та служби корпорації Майкрософт. [Електронний ресурс]. - Режим доступу: http://msdn.microsoft.com
2. КіберФорум. Форум початківців і професійних програмістів, системних адміністраторів, адміністраторів баз даних. [Електронний ресурс]. - Режим доступу: http://cyberforum.ru
3. ІНТУЇТ. Інтернет університет інформаційних технологій. [Електронний ресурс]. –Режим доступу: http://intuit.ru
4. Клуб програмістів. [Електронний ресурс]. - Режим доступу: http://www.cyberguru.ru/
5. Вікіпедія. Вільна енциклопедія. [Електронний ресурс]. - Режим доступу: http://ru.wikipedia.org/wiki
6. Тестування і якість ПЗ. [Електронний ресурс]. - Режим доступу: http://software-testing.ru/
7. Про тестинг. [Електронний ресурс]. Режим доступу: http://www.protesting.ru/
8. SQA Days. [Електронний ресурс]. Режим доступу: http://sqadays.com/
9. Software testing training and software testing services. [Електронний ресурс]. Режим доступу: http://www.rbcs-us.com/
10. Уиттакер Д., Арбон Д., Каролло Д. Как тестирует Google.: Пер. з англ. - Спб.: Издательский дом "Питер", 2014.-320с
11. Савин Р. Тестирование Дот Ком, или Пособие по жесткому обращению с багами в интернет- стартапах. - М.: Дело, 2007. - 312с
12. Хабрахабр. [Електронний ресурс]. Режим доступу: http://habrahabr.ru/
13. Code Project. Сообщество разработки программного обеспечения. [Электронный ресурс]. – Режим доступу: http://codeproject.com
14. Apache Software Foundation. [Електронний ресурс]. - Режим доступу: http://jmeter.apache.org/
15. SeleniumHQ Browser Automation. [Електронний ресурс]. - Режим доступу: http://docs.seleniumhq.org/

ДОДАТОК А

Програмний код, що описує інфраструктуру сторінки "Дані про організацію":

```
package pages.fa_pages

import pages._
import intro_pages._
import org.openqa.selenium.WebDriver
import queries._
import expert_elements._ import
org.scalatest.Matchers import
common._
import popups_and_lightboxes._
import utils.StringEnumeration
import pages.pages_constants.SkipperConstants
import org.scalatest.selenium.WebBrowser import
pages.pages_constants._

object OrgTypes extends StringEnumeration {
  val empty = StringValue("…")
  val OOO = StringValue("Общество с ограниченной ответственностью")
  val  OAO  =  StringValue("Открытое  акционерное  общество")  val
  ZAO = StringValue("Закрытое акционерное общество") val UP  =
  StringValue("Унитарное предприятие")
  val ODO = StringValue("Общество с дополнительной ответственностью")
  val PT  = StringValue("Полное товарищество")
  val TV  = StringValue("Товарищество на вере")
  val PK  = StringValue("Производственный кооператив")
  val KKH = StringValue("Крестьянское (фермерское) хозяйство")
  val NKO = StringValue("Некоммерческая организация")
  val IP  = StringValue("Индивидуальный предприниматель")
  val OBPUL = StringValue("Организация без прав юридического лица")
}


trait FaCompanyInfoPageBackground[Intro <: ExpertPage, Result <: ExpertPage] extends
BackgroundPageComponent {
  type BackgroundPage = FaCompanyInfoPage[Intro, Result]
  val backgroundPageConstructor = implicitly[Constructable[FaCompanyInfoPage[Intro,
Result]]].construct
}

trait FaCompanyInfoPageSaved[Intro <: ExpertPage, Result <: ExpertPage] extends
SavedPageComponent {
  type SavedPage = FaCompanyInfoPage[Intro, Result]
  val savedPageConstructor = new FaCompanyInfoPage[Intro, Result](_ : String)
}

class FaOnvdSelector[Intro <: ExpertPage, Result <: ExpertPage]()(implicit val driver :
WebDriver)
      extends OnvdSelectorLightBoxComponent
      with FaCompanyInfoPageBackground[Intro, Result]
      with FaCompanyInfoPageSaved[Intro, Result] {
  val lightbox = new OnvdSelectorLightBox(_)
}
```

```
class FaCompanyInfoPage[Intro <: ExpertPage, Result <: ExpertPage]
       (url : String, val isPeriodSelected : Boolean = false)(implicit val driver: WebDriver)
       extends FaFormPage(url) with BreadCrumbs[Intro] with Matchers
       with IncompleteSaver with ExpertEventually {
  val companyTitle = new ExpertInput(ByJQ("input#company_title"))
  val orgType: ExpertSelect[OrgTypes.type] = new ExpertSelect(ByJQ("select#org_type"))
  val onvd = new ExpertText(ByJQ("#selected_business_type_title"))

  protected lazy val faOnvdSelector = new FaOnvdSelector[Intro, Result]
  lazy val onvdCodeSelectorLink = new ExpertLink(ById("select_business_type")) with
Connector[faOnvdSelector.OnvdSelectorLightBox] {
    val construction = faOnvdSelector.lightbox
  }
  lazy val onvdSelectedCodeSelectorLink = new ExpertLink(ById("change_business_type")) with
Connector[faOnvdSelector.OnvdSelectorLightBox] {
    val construction = faOnvdSelector.lightbox
  }

  val wizardStepPage = "Данные об организации"
  val companyNameValidation = "Укажите название"
  val orgTypeValidation = "Заполните поле"
  val onvdCodeValidation = "Укажите основной вид деятельности"
  lazy val moneyUnitValidation = "Переключение единиц измерения невозможно – значения должны
быть целыми числами"

  def checkCompanyNameValidation() = checkText(ByJQ("label[for='company_title']"),
companyNameValidation)

  def checkOrgTypeValidation() = checkText(ByJQ("label[for='org_type']"), orgTypeValidation)

  def checkOnvdCodeValidation() = checkText(ByJQ("label[for='onvd_code']"),
onvdCodeValidation)

  private def checkText(textElementSelector: ExpertQuery, expectedText: String) {
    new ExpertText(textElementSelector).text should be(expectedText)
  }

  override def accept() {  checkText(ByJQ("div#company_info
    h1"), wizardStepPage) currentStep should be("Данные об
    организации")
  }

  lazy val moneyUnitElement = new ExpertText(ByJQ(".units-error"))
  lazy val unitOfMeasure = new ExpertSelect[UnitsOfMeasure.type](ById("units"))


  def chooseThousandsUnit : FaCompanyInfoPage[Intro, Result] = {
    unitOfMeasure.select(UnitsOfMeasure.thousands)
    new FaCompanyInfoPage(url)
  }


  def chooseMillionsUnit : FaCompanyInfoPage[Intro, Result] = {
    unitOfMeasure.select(UnitsOfMeasure.millions)
    new FaCompanyInfoPage(url)
  }

  def checkMoneyValidation() {
    moneyUnitElement.text should be(moneyUnitValidation)
  }
```

```
  lazy val nextPageInactive = new ExpertButton(ByJQ("#buttons_pane a.b-button-small")) }
     object FaCompanyInfoPage extends WebBrowser {
  type CompanyPageTypeConstructor[Intro <: ExpertPage, ResultPage[_ <: ExpertPage] <:
ExpertPage] =
    FaCompanyInfoPage[Intro, ResultPage[Intro]]


  def apply[Intro <: ExpertPage, Result <: ExpertPage, YearDependentPage <: ExpertPage](url :
String)
           (implicit driver : WebDriver) =
      new FaCompanyInfoPage[Intro, Result](url)

  implicit def FaCompanyInfoPageToConstructable[Intro <: ExpertPage, Result <: ExpertPage]
             (implicit driver : WebDriver) =
    new Constructable[FaCompanyInfoPage[Intro, Result]] {
      def construct = new FaCompanyInfoPage[Intro, Result](_ : String)
    }


  def FaCompanyInfoPageDefault[Intro <: ExpertPage, Result <: ExpertPage](url : String)
      (implicit driver: WebDriver) =
    new FaCompanyInfoPage[Intro, Result](url)(driver)

  type DefaultType[Intro <: ExpertPage, Result <: ExpertPage] = FaCompanyInfoPage[Intro,
Result]


  implicit def FaCompanyInfoPageToNextStep[Intro <: ExpertPage, Result <: ExpertPage]
              (current : FaCompanyInfoPage[Intro, Result])(implicit driver : WebDriver) =
    new FaCompanyInfoPage[Intro, Result](current.url)
    with WizardNextStep[FaCompanyInfoPage[Intro, Result], FaBalancePage[Intro, Result]] {
      override val nextPageConstructor = implicitly[Constructable[FaBalancePage[Intro,
Result]]].construct
    }


  implicit def FaCompanyInfoPageToPrevStep[Intro <: ExpertPage : Constructable, Result <:
ExpertPage]
              (current : FaCompanyInfoPage[Intro, Result])(implicit driver : WebDriver) =
    new FaCompanyInfoPage[Intro, Result](current.url)
    with WizardPrevStep[FaCompanyInfoPage[Intro, Result], Intro] {
      override protected val prevPageConstructor = implicitly[Constructable[Intro]].construct
    }
}
```

ДОДАТОК Б

Програмний код, який містить набір тест-кейсів для сторінки "Дані про організацію":

```
package test_cases.fa_tests

import  test_cases._
import pages.intro_pages._
import pages.fa_pages._
import expert_elements._
import queries._
import pages.pages_constants._

class FaCompanyInfoSpec extends ExpertSpec {
  protected def setup = {
    IntroPage().new_fa.proceed().chooseManual.selectPeriod(Years.year2019, PeriodTypes.year)
  }

 "Click next with empty fields than fill onvd code - all fields" should "be valid" in {
   val companyInfoPage = setup
   companyInfoPage.nextPageInactive.click()
   companyInfoPage.checkCompanyNameValidation()
   companyInfoPage.checkOnvdCodeValidation()
   companyInfoPage.checkOrgTypeValidation()
   companyInfoPage.onvdCodeSelectorLink.proceed().checkOnvd("15")
   withWait() {new ExpertText(ByJQ("label[for='company_title']"))}
   withWait() {new ExpertText(ByJQ("label[for='org_type']"))}
   withWait() {new ExpertText(ByJQ("label[for='onvd_code']"))}
 }

 "Fill all fields except onvd - validation" should "appear on onvd field" in {
   val companyInfoPage = setup
   companyInfoPage.companyTitle.text = OrgName.name
   companyInfoPage.orgType.select(OrgTypes.ODO)
   companyInfoPage.nextPageInactive.click()
   companyInfoPage.checkOnvdCodeValidation()
 }
  "Fill all fields, click next than return back - all fields" should "be filled" in {
    val companyInfoPage = setup
    companyInfoPage.companyTitle.text = OrgName.name
    companyInfoPage.orgType.select(OrgTypes.ODO)
companyInfoPage.onvdCodeSelectorLink.proceed().checkOnvd("15").nextPage.proceed().backwardLin
k.proceed()
    companyInfoPage.companyTitle.text should be(OrgName.name)
    companyInfoPage.orgType.getSelected should be(OrgTypes.ODO.toString())
    companyInfoPage.onvd.text should be("15. Производство пищевых продуктов, включая
напитки")
  }

  "Click next with empty fields, fill OrgType than reset and select again OrgType - all
fields" should "be valid" in {
    val companyInfoPage = setup
```

```
companyInfoPage.nextPageInactive.click()
companyInfoPage.orgType.select(OrgTypes.ODO)
 withWait() {new ExpertText(ByJQ("label[for='company_title']"))}
 withWait() {new ExpertText(ByJQ("label[for='org_type']"))}
 withWait() {new ExpertText(ByJQ("label[for='onvd_code']"))}

 companyInfoPage.orgType.select(OrgTypes.empty)
 companyInfoPage.nextPageInactive.click()

 companyInfoPage.orgType.select(OrgTypes.ODO)
 withWait() {new ExpertText(ByJQ("label[for='company_title']"))}
 withWait() {new ExpertText(ByJQ("label[for='org_type']"))}
 withWait() {new ExpertText(ByJQ("label[for='onvd_code']"))}
 }
```