math\_2018\_004

**Міністерство освіти і науки України**

**Чернівецький національний університет**

**імені Юрія Федьковича**

Факультет математики та інформатики

(повна назва інституту/факультету)

Кафедра математичного моделювання

(повна назва кафедри)

**Інформаційна система судових рішень**

**Дипломна робота**

**Рівень вищої освіти - другий (магістерський)**

Виконав:

студент 6 курсу, групи 601

спеціальності

122 – Комп’ютерні науки\_\_\_\_\_\_\_\_\_\_\_\_\_

(назва спеціальності)

Махов Сергій Олегович\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(прізвище, ім’я та по-батькові)

Керівник канд.ф.-м.н., асистент Перцов А.С.

(науковий ступінь, вчене звання, прізвище та ініціали)

**До захисту допущено:**

**Протокол засідання кафедри № 5**

від „ 12 ” грудня 2018 р.

зав. кафедри \_\_\_\_\_\_\_\_\_ доц. Піддубна Л.А.

Чернівці – 2018

# Анотація

Дипломна робота присвячена розробці системи зберігання та пошуку судових рішень судів України. У програмному продукті реалізовано схему бази даних для збереження інформації та пошукову систему.

Зміст

[Вступ 2](#_Toc531438645)

[1. Основні принципи та підходи ручного тестування програмного забезпечення 4](#_Toc531438646)

[1.1 Документація в роботі тестувальника 5](#_Toc531438647)

[1.2 Класифікація помилок в програмному забезпеченні. Їх серйозність на пріоритет. 11](#_Toc531438648)

[1.3 Рівні, види та принципи тестування 14](#_Toc531438649)

[1.4 Переваги й недоліки ручного тестування 21](#_Toc531438650)

[2. Основні принципи та підходи автоматизованого тестування програмного забезпечення 22](#_Toc531438651)

[2.1 Визначення доцільності впровадження автоматизованого тестування для програмних продуктів 23](#_Toc531438652)

[2.2 Види автоматизованого тестування 29](#_Toc531438653)

[2.3 Переваги та недоліки автоматизованого тестування 31](#_Toc531438654)

[3. Структура проекту та використанні технології для розробки збереження та обробки даних 32](#_Toc531438655)

[3.1 Характеристика бази даних 32](#_Toc531438656)

[3.2 Використані технології для розробки 35](#_Toc531438657)

[Список використаної літератури 38](#_Toc531438658)

# Вступ

Комп’ютери та інформаційні системи є важливою частиною нашого суспільства. Вони використовуються майже в кожній сфері нашого життя, включаючи наші роботи, для автоматизації послуг та продуктів; в сфері охорони здоров’я, для покращення якості лікування; в освіті, щоб допомогти дітям здобувати нові знання; у фінансовій сфері, для підтримки переказу коштів та операцій над власними рахунками; у сфері розваг, включаючи Інтернет, фільми, музику та інші мистецтва; у транспортній системі, для контролю над системами автомобілів та публічного транспорту, наприклад метро.

Технології розвиваються з феноменальним темпом і не збираються сповільнюватись. Технології вже допомогли робітникам безліч індустрій та ринків працювати якісніше. Інформаційні технології також дозволили покращити канали комунікації що також вплинуло на продуктивність роботи. У більшості сфер індустрії ІТ-технології є невід’ємною частиною процесу роботи.

Так як і у сферах нашого життя, так і у судові владі інформаційна ера розпочала революцію. Пропонуючи можливість опубліковувати величезні кількості судових рішень в мережі, судова влада може істотно збільшити власну прозорість. Саме можливість публікації рішень дасть змогу людям які не були задіяні в конкретному судовому засіданні швидко використати його як прецедент для власних потреб, як то: винести вирок, подати апеляцію. Опубліковані судові рішення є важливим базовим матеріалом для професій пов’язаних з юриспруденцією, академічних дослідників, журналістів та приватних компаній що працюють на ринку юридичної інформації.

Найбільш важливим аргументом проти публікації таких рішень в оригінальній формі є незаконне втручання в особисте життя, тобто порушення статті 8 «Хартії основних прав Європейського Союзу», оскільки зазвичай судові рішення часто містять особисту інформацію. Але цей аргумент може вдоволений якщо публікувати рішення у анонімній формі, тобто з вилученою усією особистою інформацією.

Функції які виконують публікації судових рішень вже існували навіть до появи інтернету: публічна перевірка справедливості судді та публічне знання про розвиток права. До початку поширення інтернету, перша функція, хоча й іноді не ідеально, здійснювалась за допомогою публічного оголошення рішення суду. Друга ж, здійснювалась самим судами або комерційними видавцями які пропонували друковані рішення що були важливими. З швидким розвитком інформаційних технологій дані функцій можуть бути ще більше покращені, оскільки швидкість поширення та кількість інформації значно збільшується.

В Україні, вже існує практика публікацій судових рішень, яка навіть підкріплена законом (Закон України про доступ до судових рішень) який регулює необхідність публікацій до судових рішень (ст.2 п.2) та доступ до них (ст.3, 4) за допомогою реєстру судових рішень. Однак, існуюча система є нестабільною та ненадійною, що призводить до порушення цього ж закону (ст.4, п.1).

Кроком для подолання цієї проблеми є розробка альтернативного рішення що не буде мати вищезгаданих недоліків. Саме тому розробка інформаційної системи реєстру судових рішень і є моє завдання на магістерську роботу.

# Постановка задачі

У 2002 році Верховна Рада схвалила Концепцію Загальнодержавної програми адаптації України до законодавства Європейського Союзу, де передбачалося створення національної інформаційної мережі судових рішень з метою доступу до судової практики та можливості їх публічного обговорення.

Однак перша спроба втілити її у життя була зроблена аж 18 січня 2005 року, коли у Верховній Раді був зареєстрований законопроект «Про забезпечення гласності при здійсненні господарського судочинства». Авторами законопроекту були Леонід Антоненко та Іван Крулько. Законодавча ініціатива належала Борису Тарасюку та Миколі Катеринчуку.

Законопроект отримав підтримку уряду і реалізацію закладених ідей було затверджено 4 лютого 2005 року. Майже одночасно з урядом ідея була підтримана і судовою владою. Так, постановою Президії Верховного Суду України, Президії Ради суддів України, колегії Державної судової адміністрації України «Про стан здійснення судочинства у 2004 р. і завдання на 2005 р.» від 18 лютого 2005 року, Державній судовій адміністрації України доручено запровадити реєстр судових рішень зі створенням у комп'ютерній мережі відповідної бази даних і забезпечити в установленому законом порядку доступ до неї.

31 серпня 2005 року в парламенті був зареєстрований законопроект «Про доступ до судових рішень». Законодавча ініціатива належала Василю Онопенко, текст проекту був розроблений Центром політико-правових реформ.

22 грудня 2005 року цей альтернативний проект було схвалено у другому читанні і в цілому конституційною більшістю — 303 голосами народних депутатів.

Закон набрав чинності 1 червня 2006 року. Саме з цієї дати усі ухвалені судами України процесуальні документи (крім рішень місцевих загальних судів) повинні були заноситись до реєстру і з'являтися на Інтернет-запит будь-якого користувача, а рішення місцевих загальних судів — з 1 січня 2007 року. Проте, перші документи почали з'являтися на запит користувачів з затримкою в принаймні два тижні.

Унаслідок вищезгаданих подій було розроблено Єдиний державний реєстр судових рішень, однак на сьогоднішній день він є малоефективним інструментом для пошуку та перегляду судових рішень, оскільки його пошуковий апарат є неефективним.

Метою магістерської роботи є розробка системи яка дозволить шукати та переглядати судові рішення.

Для реалізації поставленого завдання потрібно:

* знайти набори даних що представляють собою судові рішення,
* розробити концептуальну, логічну та фізичну схеми бази даних,
* визначити та реалізувати фільтри для здійснення ефективного пошуку,
* реалізувати серверну частину системи,
* реалізувати клієнтську частину системи

# Основні принципи та підходи організації даних програмного забезпечення

Сучасне інформаційне поле складається з маси подій, об’єктів та явищ. Воно охоплює такі об’єми, що без чітко працюючої визначеної системи, збереження всіх цих даних було б хаотичним та некерованим.

Визначною рисою бази даних від інших сховищ є те, що інформація в ній обов’язково піддається обробці електронними носіями з наступною систематизацією, яка підпорядкована певним правилам. Вона є важливою для неперервної взаємодії систем що містять інформацію.

### 2.1 Проектування бази даних

**Проектування бази даних –** організація даних згідно з моделлю бази даних. Розробник визначає які дані повинні бути збережені та як дані відносяться один до одного. Проектування баз даних включає в себе класифікацію даних та визначення взаємовідносин. Теоретичне відображення даних називається онтологією.

Основними задачами проектування баз даних є:

* Забезпечення збереження всієї необхідної інформації
* Забезпечення можливості отримання даних
* Зменшення надмірності та дублювання даних
* Забезпечення цілісності даних

Для визначення даних які мають бути збережені в базі даних, зазвичай залучають людину яка займається проектуванням бази даних, це швидше розробник що має експертизу в області теорії баз даних, більше аніж знання про предметну область данні якої будуть зберігатись. Більш того, дані які будуть зберігатись в базі даних повинні бути визначені у взаємодії з людиною яка має експертизу в даній предметній області та хто знає, які дані повинні бути присутні в системі.

Цей процес зазвичай називають «аналізом вимог», він вимагає від розробника бази даних навичку виявляти необхідну інформацію в тих хто володіє знанням про предметну область. Це необхідно, оскільки ті хто володіє предметною областю зазвичай не можуть чітко виразити які є системні вимоги для бази даних оскільки для них є незвичним думати в термінах дискретних даних які мають бути збережені. Також, дані які мають зберігатись можуть бути визначенні в специфікації вимог.

Коли проектувальник бази даних обізнаний про дані які мають зберігатись в базі даних, він повинен визначити залежності між даними. Іноді коли дані змінюються можуть змінитись і дані що не є видимими. Наприклад, нехай в списку імен з адресами декілька людей можуть мати однакову адресу, але одна людина не може мати більш аніж одну адреси – адреса залежна від імені. За допомогою імені можна однозначно визначити адресу, але не навпаки – за адресом не можна однозначно визначити ім’я, оскільки декілька людей можуть відноситись до однієї адреси. Оскільки адреса визначається по імені то вона вважається залежною від імені.

Коли відношення та залежності між частинами інформації визначено стає можливо організувати дані в логічні структури які можуть бути зіставлені в об’єкти сховища що підтримуються СУБД. У випадку реляційної бази даних, такими об’єктами являються таблиці де дані збережені у вигляді рядків з колонками. У об’єктно-орієнтовній базі даних об’єкти збереження прямо відповідають до об’єктів що використовуються в об’єктно-орієнтовних мовах програмування. Відношення можуть бути описані як атрибути класів об’єкту або ж як методи які оперують об’єктами іншого класу.

Кожна таблиця може відповідати реалізації або ж логічного об’єкту або відношенням що об’єднує один чи більше прикладів одного чи багатьох логічних об’єктів. Зв’язок між таблицями може бути збережений як посилання що пов’язує підпорядковані таблиці з батьківськими.

**Концептуальна схема** **–** опис інформаційних потреб бізнесу на високому рівні абстракції. Зазвичай включає лише основні ідеї та зв’язки між ними. Чернова модель, яка недостатньо деталізована для того щоб побудувати справжню базу даних. Вона описує семантику організації і відображає серію припущень про її природу. Конкретно, вона описує речі що є значимими для організації (сутності), про які вона намагається збирати інформацію, характеристики атрибутів сутностей та асоціації між парами цих сутностей.

Оскільки концептуальна схеми відображає семантику організації та не дизайн бази даних, вона може існувати на різних рівнях абстракцій.

**Логічна схема** – модель даних для специфічної задачі об’єктної області описана незалежно від конкретної СУБД, але в термінах структур даних, таких як реляційні таблиці та колонки, об’єктно орієнтовані класи або ж теги XML. Що і відрізняє її від концептуальної схеми, яка описує семантику організації даних без зв’язків з якими-небудь технологіями.

Логічна схема відображає абстрактну структуру предметної області. Досить часто вона є схематичною за природою і часто використовується в бізнес процесах що прагнуть охопити речі які є важливими для організації. Після валідації та підтвердження, логічна схема може стати базою для фізичної схеми.

Логічна схема повинна бути побудована на базі попередньої концептуальної схеми, яка за своїм походженням описує семантику та інформаційний контекст, це має також відображатись в логічній схемі.

**Фізична схема** **–** відображення дизайну даних яка реалізована, або призначена для реалізації у системі управління базами даних. У життєвому циклу проекту вона зазвичай походить від логічної схеми, також може бути отримана внаслідок «реверс-інженерінгу» від існуючої реалізації бази даних. Закінчена фізична схема буде містити всі елементи бази даних для створення зв’язків між таблицями або ж для покращення швидкодії, наприклад індекси, визначення обмежень, розділені таблиці чи кластери. Аналіз зазвичай може використати фізичну схему для оцінки вимог до сховища. Також схема може містити специфічні деталі до розміщення даної системи бази даних.

**Проектування ділиться на три етапи:**

*Концептуальне (інфологічне) проектування* – побудова семантичної схеми предметної області, тобто інформаційної моделі найвищого рівня абстракції. Така модель створюється без орієнтації на яку-небудь конкретну систему управління базами даних (СУБД) чи модель даних. Терміни «семантична модель», «концептуальна схема» та «інфологічна модель» є синонімами.

Частіше за все концептуальна схема містить в собі:

* Опис інформаційних об’єктів чи понять предметної області та зв’язків між ними.
* Опис обмежень цілісності, тобто вимог до допустимих значень даних і до зв’язків між ними.

*Логічне (даталогічне) проектування* – створення схеми бази даних на основі конкретної схемі даних, наприклад, реляційної схеми даних. Для реляційної схеми даних логічна схема – набір схем відношень, зазвичай з вказанням первинних ключів, а також зв’язків між відношеннями, що являють собою зовнішні ключі.

Перетворення концептуальної схеми в логічну схему, як правило, виконується за формальними правилами. Цей етап може бути в більшій мірі оптимізований.

На етапі логічного проектування враховується специфіка конкретної моделі даних, але може не враховуватись специфіка конкретної системи управління базами даних.

*Фізичне проектування –* створення схеми бази даних для конкретної СУБД. Специфіка СУБД може включати в себе обмеження на іменування об’єктів бази даних, обмеження на підтримуванні формати і т. д. Окрім того, специфіка конкретної СУБД при фізичному проектуванні включає в себе вибір рішень пов’язаних з фізичним середовищем збереженням даних (розподілення БД по файлам та пристроям, вибір методів керування дисковою пам’яттю), створення індексів і т. д.

Результатом фізичного проектування на мові SQL зазвичай є сценарій що створює необхідні об’єкти у СУБД.

### 2.2 Масштабування баз даних

**Масштабування баз даних** – одна з найбільш складних задач під час розвитку системи.

В основі масштабування даних лежить той же принцип, що і в основі масштабування Web-додатків. Це розподілення даних на групи та виділення їх на окремі сервери. Існує дві основні стратегії – реплікація та шардінг.

*Реплікація –* реплікація до

**Тестування програмного забезпечення** — перевірка відповідності між реальним і очікуваною поведінкою програми, здійснювана на кінцевому наборі тестів, обраному певним чином. У більш широкому сенсі, тестування — це одна з технік контролю якості, що включає в себе активності по плануванню робіт (Test Management), проектування тестів (Test Design), виконання тестування (Test Execution) та аналізу отриманих результатів (Test Analysis).

Однією із кінцевих цілей тестування є визначення якості програмного забезпечення. Якістю програмного забезпечення (Software Quality) називають сукупність характеристик програмного забезпечення, що відносяться до його здатності задовольняти встановлені і передбачувані потреби.   
Кожне програмне забезпечення повинно мати специфікацію(документацію), що його описує та є відображенням вимог та очікувань кінцевого користувача від даного програного продукту, то не варто плутати наступні два поняття.

Верифікація (verification) — це процес оцінки системи або її компонентів з метою визначення, чи задовольняють результати поточного етапу розробки умов, сформованим на початку цього етапу. Тобто виконуються наші цілі, терміни, завдання щодо розроблення проекту, визначені на початку поточної фази.  
Валідація (validation) — це визначення відповідності розробляється З очікуванням і потребам користувача, вимогам до системи. Також можна зустріти іншу інтерпретацію: Процес оцінки відповідності продукту явним вимогам (специфікацій) і є верифікація (verification), в той же час оцінка відповідності продукту очікуванням і вимогам користувачів — є валідація (validation).   
  
**Цілями тестування є:**  
- підвищити ймовірність того, що додаток, призначений для тестування, буде - працювати правильно при будь-яких обставин;  
- підвищити ймовірність того, що додаток, призначений для тестування, буде відповідати всім зазначеним вимогам;  
- надання актуальної інформації про стан та якість продукту на даний момент.  
  
**Етапами тестування є:**  
1. Аналіз  
2. Розробка стратегії тестування  
і планування процедур контролю якості  
3. Робота з вимогами  
4. Створення тестової документації  
5. Тестування прототипу  
6. Основне тестування  
7. Стабілізація  
8. Експлуатація

## 1.1 Документація в роботі тестувальника

Набір документів в роботі тестувальника варіюється від проекту до проекту, від компанії до компанії. Оскільки це залежить від прийнятих стандартів та усталених порядків в компанії. Проте він обов’язково повинен містити наступні документи.

*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 .

Для написання хорошого тест плану необхідно відповісти на наступні запитання:

-Що треба тестувати? (опис об'єкта тестування: системи, додатки, обладнання);

-Що будете тестувати? (список функцій і опис тестованої системи і її компонент окремо);

-Як будете тестувати? (стратегія тестування, а саме: види тестування і їх застосування по відношенню до об'єкта тестування);

-Коли будете тестувати? (послідовність проведення робіт: підготовка (Test Preparation), тестування (Testing), аналіз результатів (Test Result Analisys) в розрізі запланованих фаз розробки);

-Критерії початку тестування:

1. закінченість розробки необхідного функціоналу

2. наявність всієї необхідної документації

-Критерії закінчення тестування:

1. результати тестування задовольняють критеріям якості продукту;

2. вимоги до кількості відкритих багів виконані;

3. витримка певного періоду без відкриття нових багів (Zero Bug Bounce);

Відповівши в своєму тест плані на ці запитання, можна вважати, що у вас на руках вже є хороша чернетка документа з планування тестування.

*Рецензія і Затвердження*

Для збільшення цінності вашого тест плану рекомендується проводити його періодичне рецензування з боку учасників проектної групи. Це можна зробити просто домовившись між собою або ж реалізувати в вигляді "процедури затвердження". Як приклад, наведемо список учасників проектної групи, затвердження яких ми вважаємо за необхідне:

-Ведучий тестувальник

-Тест менеджер (менеджер з якості)

-Керівник розробки

-Менеджер проекту

Кожен з перелічених учасників проекту, перед затвердженням, проведе рецензію і внесе свої коментарі і пропозиції, які допоможуть зробити Ваш тест план повнішим і якісним.

*Тест дизайн* - це етап процесу тестування ПЗ, на якому проектуються і створюються тестові випадки.

План роботи над тест дизайном:

•аналіз наявних проектних артефактів: документація (специфікації, вимоги, плани), моделі, виконуваний код ;

• написання специфікації по тест дизайну (Test Design Specification);

• проектування і створення тестових випадків (Test Cases);

Ролі, відповідальні за тест дизайн:

• Тест аналітик - визначає "ЩО тестувати?";

• Тест дизайнер - визначає "ЯК тестувати?".

Попросту кажучи, завдання тест аналітиків і дизайнерів зводиться до того, щоб використовуючи різні стратегії і техніки тест дизайну, створити набір тестових випадків, що забезпечує оптимальне тестове покриття тестової програми. Однак, на більшості проектів ці ролі не виділяються, а це довіряється звичайним тестувальникам, що не завжди позитивно позначається на якості тестів, тестування і, як з цього випливає, на якості програмного забезпечення (кінцевого продукту).

*Техніки тест дизайну*  
• *Еквівалентне Поділ (Equivalence Partitioning — EP).* Як приклад, у вас є діапазон допустимих значень від 1 до 10, ви повинні вибрати одне вірне значення всередині інтервалу, скажімо, 5, і одне невірне значення поза інтервалу — 0.  
• *Аналіз Граничних Значень (Boundary Value Analysis — BVA).* Якщо взяти приклад вище, в якості значень для позитивного тестування виберемо мінімальну та максимальну межу (1 і 10), і значення, більше і менше кордонів (0 і 11). Граничний аналіз значень може бути застосований до полів записів, файлів, або до будь-якого роду сутностей мають обмеження.  
• *Причина / Наслідок (Cause/Effect — CE).* Це, як правило, введення комбінацій умов (причин), для отримання відповіді від системи (Наслідок). Наприклад, ви перевіряєте можливість додавати клієнта, використовуючи певну екранну форму. Для цього вам необхідно буде ввести кілька полів, таких як «Ім'я», «Адреса», «Номер Телефону» а потім, натиснути кнопку «Додати» — ця «Причина». Після натискання кнопки «Додати», система додає клієнта в базу даних і показує його на екрані — це «Наслідок».  
• *Передбачення помилки (Error Guessing — EG).* Це коли тест аналітик використовує свої знання системи і здатність до інтерпретації специфікації на предмет того, щоб «вгадати» при яких вхідних умов система може видати помилку. Наприклад, специфікація каже: «користувач повинен ввести код». Тест аналітик, буде думати: «Що, якщо я не введу код?», «Що, якщо я введу неправильний код? », і так далі. Це і є передбачення помилки.  
• *Вичерпне тестування (Exhaustive Testing — ET)* — це крайній випадок. В межах цієї техніки ви повинні перевірити всі можливі комбінації вхідних значень, і в принципі, це повинно знайти всі проблеми. На практиці застосування цього методу не представляється можливим, з-за величезної кількості вхідних значень.

Для того, щоб зрозуміти чи програмне забезпечення оптимально покрито тестовими випадками, використовують поняття матриці відповідності вимог. *Матриця відповідності вимог(Traceability matrix)* — це двовимірна таблиця, що містить відповідність функціональних вимог (functional requirements) продукту і підготовлених тестових сценаріїв (test cases). У заголовках колонок таблиці розташовані вимоги, а в заголовках рядків — тестові сценарії. На перетині — позначка, яка означає, що вимога поточної колонки вкрите тестовим сценарієм поточного рядка. Ця матриця використовується QA-інженерами для валідації покриття продукту тестами і є невід'ємною частиною тест-плану.

*Тестовий випадок (Test Case)* - це артефакт, що описує сукупність кроків, конкретних умов і параметрів, необхідних для перевірки реалізації тестованої функції або її частини.

Кожен тест кейс повинен мати 3 частини:

PreConditions - 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

PostConditions - список дій, які переводять систему в первинний стан (стан до проведення тесту - initial state).

Види тестових випадків

Тест кейси поділяються за очікуваного результату на позитивні і негативні:

• *Позитивний тест кейс* використовує тільки коректні дані і перевіряє, що додаток правильно виконало функцію, що викликається.

• *Негативний тест кейс* оперує як коректними так і некоректними даними (мінімум 1 некоректний параметр) і ставить за мету перевірку виняткових ситуацій (спрацьовування валідаторів), а також перевіряє, що викликається додатком функція не виконується при спрацьовуванні валідатора.

На деяких проектах не вимагається глибокий рівень деталізації тестування чи просто недостатньо часу на реалізацію, тому в таких випадках використовують чек-лист. *Чек-лист (check list)* — це документ, що описує що повинно бути протестовано. При цьому чек-лист в свою чергу теж може бути абсолютно різного рівня деталізації. На скільки детальним буде чек-лист залежить від вимог до звітності, рівня знання продукту співробітниками та складності продукту. Як правило, чек-лист містить тільки дії (кроки), без очікуваного результату. Чек-лист менш формалізований ніж тестовий сценарій. Його доречно використовувати тоді, коли тестові сценарії будуть надлишкові. Також чек-лист асоціюються з гнучкими підходами у тестуванні.

Наступним документом в роботі тестувальника є звіт про дефект(баг репорт).  
*Баг Репорт (Bug Report)*— це документ, що описує ситуацію або послідовність дій призвела до некоректної роботи об'єкта тестування, із зазначенням причин і очікуваного результату. Він має наступну структуру:  
- Заголовок, що містить короткий опис проблеми, явно вказує на причину і тип помилкової ситуації;  
- Проект -назва досліджуваного проекту;  
- Компонент програми - назва частини або функції досліджуваного продукту;  
- Номер версії, в якій була знайдена помилка;  
- Серйозність ( S1 – блокуючий, S2 – критичний, S3 – значний, S4 – незначний, S5 – тривіальний);  
- Пріоритет ( P1 – високий, P2 – середній, P3 - низький;  
- Статус (залежить від використовуваної процедури і життєвого циклу бага;

- Автор;  
- Призначений на ім'я працівника, який буде вирішувати проблему;  
- Оточення (ОС, сервіс пак, назва браузера, версія браузера і т.д.);

- Опис: а)кроки відтворення, за якими можна легко відтворити ситуацію, що призвела до помилки; б) фактичний результат, отриманий після проходження кроків для відтворення; в)очікуваний результат - правильний результат  
- Доповнення ( Прикріплений файл з логами, скріншот або інший документ, який може допомогти прояснити причину помилки або вказати на спосіб вирішення проблеми).

## 1.2 Класифікація помилок в програмному забезпеченні. Їх серйозність на пріоритет.

Існує наступна класифікація помилок в програмному забезпеченні:  
**Дефект (він же баг)**— це невідповідність фактичного результату виконання програми очікуваного результату. Дефекти виявляються на етапі тестування програмного забезпечення (ПЗ), коли тестувальник проводить порівняння отриманих результатів роботи програми (компоненту або дизайну) з очікуваним результатом, описаних у специфікації вимог.  
**Error**— помилка користувача, тобто він намагається використовувати програму іншим способом. Приклад — вводить букви в поля, де потрібно вводити цифри (вік, кількість товару тощо).  
В якісній програмі такі ситуації передбачено. Вони коректно обробляються програмою та видаються повідомлення про помилку (error message).  
**Bug (defect)** — помилка програміста (або дизайнера або ще когось, хто приймає участь в розробці), тобто коли в програмі, що щось іде не так як планувалося і програма виходить з-під контролю. Наприклад, коли ніяк не контролюється введення користувача, в результаті невірні дані викликають падіння у роботі програми. Або всередині програма побудована так, що спочатку не відповідає тому, що від неї очікується.   
**Failure**— збій (причому не обов'язково апаратний) в роботі компонента, всієї програми або системи. Тобто, існують такі дефекти, що призводять до збоїв і існують такі, які не призводять. Дефекти інтересу користувача, наприклад. Але апаратний збій, ніяк не пов'язаний з програмним забезпеченням, теж є failure.

**Сер**й**озність та пріоритет дефекту**  
Серйозність (Severity) — це атрибут, що характеризує вплив дефекту на працездатність програми.  
Пріоритет (Priority) — це атрибут, що вказує на черговість виконання завдання або усунення дефекту. Можна сказати, що це інструмент менеджера по плануванню робіт. Чим вище пріоритет, тим швидше потрібно виправити дефект.  
Severity визначається тестувальником  
Priority — менеджером, лідером команди або замовником.  
  
**Градація Серйозності дефекту (Severity)**  
**S1 Блокуюча (Blocker)**  
Блокуюча помилка, що приводить додаток в неробочий стан, в результаті якого подальша робота з досліджуваної системою або її ключовими функціями стає неможлива. Рішення проблеми необхідно для подальшого функціонування системи.  
**S2 Критичний (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 досліджуваної системою.  
**S3 Значна (Major)**  
Значна помилка, частину основної бізнес логіки працює некоректно. Помилка не критична чи є можливість для роботи з досліджуваної функцією, використовуючи інші вхідні точки.  
**S4 Незначна (Minor)**   
Незначна помилка, що не порушує бізнес логіку досліджуваної частини програми, очевидна проблема користувальницького інтерфейсу.  
**S5 Тривіальна (Trivial)**  
Тривіальна помилка, не стосується бізнес логіки додатка, погано відтворена проблема, малопомітна допомогою користувальницького інтерфейсу, проблема сторонніх бібліотек або сервісів, проблема, не надає ніякого впливу на загальну якість продукту.

**Градація Пріоритету дефекту (Priority)**  
**P1 Високий (High)**  
Помилка повинна бути виправлена як можна швидше, оскільки 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 вимагає термінового вирішення.

## 1.3 Рівні, види та принципи тестування

Тестування на різних рівнях проводиться протягом усього життєвого циклу розробки і супроводу ПЗ. Рівень тестування визначає те, над чим виробляються тести: над окремим модулем, групою модулів або системою в цілому. Проведення тестування на всіх рівнях системи – це запорука успішної реалізації та здачі проекту.

**1. Модульне тестування (Unit Testing)**  
Компонентне (модульне) тестування перевіряє функціональність і шукає дефекти в частинах програми, які доступні і можуть бути протестовані по-окремо (модулі програм, об'єкти, класи, функції і т. д.).  
**2. Інтеграційне тестування (Integration Testing)**  
Перевіряється взаємодія між компонентами системи після проведення компонентного тестування.  
**3. Системне тестування (Testing System)**  
Основним завданням системного тестування є перевірка як функціональних, так і не функціональних вимог у системі в цілому. При цьому виявляються дефекти, такі як невірне використання ресурсів системи, непередбачені комбінації даних користувацького рівня, несумісність з оточенням, непередбачені сценарії використання, відсутня або неправильна функціональність, незручність використання і т. д.  
**4. Операційне тестування (Release Testing).**  
Навіть якщо система задовольняє всім вимогам, важливо переконатися в тому, що вона задовольняє потреб користувача і виконує свою роль у середовищі своєї експлуатації, як це було визначено в бізнес моделі системи. Слід врахувати, що і бізнес модель може містити помилки. Тому так важливо провести операційне тестування як фінальний крок валідації. Крім цього, тестування в середовищі експлуатації дозволяє виявити і нефункціональні проблеми, такі як: конфлікт з іншими системами, суміжними в області бізнесу або в програмних та електронних середовищах; недостатня продуктивність системи в середовищі експлуатації та ін. Очевидно, що знаходження подібних речей на стадії впровадження — критична і дорога проблема. Тому так важливо проводити не тільки верифікації, але і валідації, з самих ранніх етапів розробки ПЗ.  
**5. Приймальне тестування (Acceptance Testing)**  
Формальний процес тестування, який перевіряє відповідність системи вимогам і проводиться з метою:  
• визначення задовольняє система приймальним критеріям;  
• винесення рішення замовником або іншою уповноваженою особою приймається додаток чи ні.

**Види / типи тестування**  
**Функціональні види тестування**  
• Функціональне тестування (Functional testing)  
• Тестування безпеки (Security and Access Control Testing)  
• Тестування взаємодії (Interoperability Testing)  
**Нефункціональні види тестування**  
• Всі види тестування продуктивності:  
o навантажувальне тестування (Performance and Load Testing)  
o стресове тестування (Stress Testing)  
o тестування стабільності та надійності (Stability / Reliability Testing)  
o об'ємне тестування (Volume Testing)  
• Тестування установки (Installation testing)  
• Тестування зручності використання (Usability Testing)  
• Тестування на відмову і відновлення (Failover and Recovery Testing)  
• Конфігураційне тестування (Configuration Testing)  
**Пов'язані із змінами види тестування**  
• Димове тестування (Smoke Testing)  
• Регрессионное тестування (Regression Testing)  
• Повторне тестування (Re-testing)  
• Тестування збірки (Build Verification Test)  
• Санітарний тестування або перевірка узгодженості/справності (Sanity Testing)  
  
**Функціональне тестування** розглядає заздалегідь зазначене поведінка і ґрунтується на аналізі специфікацій функціональності компонента або системи в цілому.  
**Тестування безпеки** — це стратегія тестування, використовувана для перевірки безпеки системи, а також для аналізу ризиків, пов'язаних з забезпеченням цілісного підходу до захисту додатки, атак хакерів, вірусів, несанкціонованого доступу до конфіденційних даних.  
**Тестування взаємодії** — це функціональне тестування, що перевіряє здатність програми взаємодіяти з одним і більше компонентами або системами і включає в себе тестування сумісності і інтеграційне тестування  
**Навантажувальне тестування** — це автоматизоване тестування, що імітує роботу певної кількості бізнес користувачів на якому-небудь загальному (поділюваному ними) ресурсі.  
**Стресове тестування** дозволяє перевірити наскільки додаток і система в цілому працездатні в умовах стресу і також оцінити здатність системи до регенерації, тобто до повернення до нормального стану після припинення дії стресу. Стресом у даному контексті може бути підвищення інтенсивності виконання операцій до дуже високих значень або аварійне зміна конфігурації сервера. Також одним із завдань при стресовому тестуванні може бути оцінка деградації продуктивності, таким чином цілі стресового тестування можуть перетинатися з цілями тестування продуктивності.  
**Об'ємне тестування.** Завданням об'ємного тестування є отримання оцінки продуктивності при збільшенні обсягів даних в базі даних програми  
**Тестування стабільності та надійності**. Завданням тестування стабільності (надійності) є перевірка працездатності програми при тривалому (багатогодинному) тестуванні з середнім рівнем навантаження.  
**Тестування установки** направлено на перевірку успішної інсталяції та налаштування, а також оновлення або видалення програмного забезпечення.  
**Тестування зручності користування**— це метод тестування, спрямований на встановлення ступеня зручності використання, навченості, зрозумілості і привабливості для користувачів розроблюваного продукту в контексті заданих умов. Сюди також входить:  
Тестування користувальницького інтерфейсу (англ. UI Testing) — це вид тестування дослідження, що виконується з метою визначення, чи зручний певний штучний об'єкт (такий як веб-сторінка, інтерфейс або пристрій) для його передбачуваного застосування.  
User eXperience (UX) — відчуття, що випробовується користувачем під час використання цифрового продукту, в той час як User interface — це інструмент, що дозволяє здійснювати інтеракцію «користувач — веб-ресурс».  
**Тестування на відмову і відновлення (Failover and Recovery Testing)** перевіряє досліджуваний продукт з точки зору здатності протистояти і успішно відновлюватися після можливих збоїв, що виникли у зв'язку з помилками програмного забезпечення, відмовами обладнання або проблемами зв'язку (наприклад, відмова мережі). Метою даного виду тестування є перевірка систем відновлення (або дублюючих основний функціонал систем), які, у разі виникнення збоїв, забезпечать збереження і цілісність даних досліджуваного продукту.  
**Конфігураційне тестування (Configuration Testing)**— спеціальний вид тестування, спрямований на перевірку роботи програмного забезпечення при різних конфігураціях системи (заявлених платформах, підтримуваних драйвери, при різних конфігураціях комп'ютерів і т. д.)  
**Димове (Smoke)** тестування розглядається як короткий цикл тестів, виконуваний для підтвердження того, що після складання коду (нового або виправленого) встановлюється додаток, стартує і виконує основні функції.  
**Регресійне тестування** — це вид тестування спрямований на перевірку змін, зроблених в додатку або навколишньому середовищі (лагодження дефекту, злиття коду, міграція на іншу операційну систему, бази даних, веб-сервер або сервер програми), для підтвердження того факту, що існуюча раніше функціональність працює як і раніше. Регресійними можуть бути як функціональні, так і нефункціональні тести.  
**Повторне тестування** — тестування, під час якого виконуються тестові сценарії, які виявили помилки під час останнього запуску, для підтвердження успішності виправлення цих помилок.  
У чому різниця між regression testing і re-testing?   
Re-testing — перевіряється виправлення багів  
Regression testing — перевіряється те, що виправлення багів не вплинуло на інші модулі і не викликало нових багів.  
**Тестування складання або Build Verification Test** — тестування спрямоване на визначення відповідності, випущеної версії, критеріям якості для початку тестування. По своїм цілям є аналогом Димового Тестування, спрямованого на прийняття нової версії подальше тестування або експлуатацію. Вглиб воно може проникати далі, в залежності від вимог до якості випущеної версії.  
**Санітарне тестування** — це вузьконаправлене тестування достатня для доказу того, що конкретна функція працює згідно заявленим у специфікації вимог. Є підмножиною регресійного тестування. Використовується для визначення працездатності певної частини програми після змін вироблених в ній або навколишньому середовищу. Зазвичай виконується вручну.  
**Передбачення помилки (Error Guessing — EG)**. Це коли тест аналітик використовує свої знання системи і здатність до інтерпретації специфікації на предмет того, щоб «вгадати» при яких вхідних умов система може видати помилку. Наприклад, специфікація каже: «користувач повинен ввести код». Тест аналітик, буде думати: «Що, якщо я не введу код?», «Що, якщо я введу неправильний код? », і так далі. Це і є передбачення помилки.  
  
**Підходи до інтеграційного тестування:**  
• **Знизу вгору " (Bottom Up Integration)**  
Всі низькорівневі модулі, процедури або функції збираються воєдино і потім тестуються. Після чого збирається наступний рівень модулів для проведення інтеграційного тестування. Даний підхід вважається корисним, якщо все або практично всі модулі, що розробляється рівня, готові. Також даний підхід допомагає визначити за результатами тестування рівень готовності програми.  
• **Зверху вниз (Top Down Integration)**  
Спочатку тестуються всі високорівневі модулі, і поступово один за іншим додаються низькорівневі. Всі модулі більш низького рівня симулюються заглушками з аналогічною функціональністю, потім по мірі готовності вони замінюються реальними активними компонентами. Таким чином ми проводимо тестування зверху вниз.  
• **Великий вибух («Big Bang» Integration)**  
Все або практично всі розроблені модулі збираються разом у вигляді закінченої системи або її основної частини, і потім проводиться інтеграційне тестування. Такий підхід дуже вдалий для збереження часу. Однак якщо тест кейси та їх результати записані не вірно, то сам процес інтеграції сильно ускладниться, що стане перешкодою для команди тестування при досягненні основної мети інтеграційного тестування.

**Принципи тестування**  
**Принцип 1** — Тестування демонструє наявність дефектів (Testing shows presence of defects)  
Тестування може показати, що дефекти присутні, але не може довести, що їх немає. Тестування знижує ймовірність наявності дефектів, що знаходяться в програмному забезпеченні, але, навіть якщо дефекти не були виявлені, це не доводить його коректності.

**Принцип 2** — Вичерпне тестування недосяжно (Exhaustive testing is impossible)  
Повне тестування з використанням всіх комбінацій вводів і передумов фізично неможливо, за винятком тривіальних випадків. Замість вичерпного тестування повинні використовуватися аналіз ризиків і розстановка пріоритетів, щоб більш точно сфокусувати зусилля по тестуванню.  
**Принцип 3** — Раннє тестування (Early testing)  
Щоб знайти дефекти як можна раніше, активності тестування повинні бути розпочаті як можна раніше в життєвому циклі розробки програмного забезпечення або системи, і повинні бути сфокусовані на визначених цілях.  
**Принцип 4** — Скупчення дефектів (Defects clustering)  
Зусилля тестування повинні бути зосереджені пропорційно очікуваній, а пізніше реальної щільності дефектів по модулям. Як правило, більша частина дефектів, виявлених при тестуванні або спричинили за собою основне кількість збоїв системи, міститься в невеликій кількості модулів.  
**Принцип 5** — Парадокс пестициду (Pesticide paradox)  
Якщо одні і ті ж тести будуть прогоняться багато разів, в кінцевому рахунку цей набір тестових сценаріїв більше не буде знаходити нових дефектів. Щоб подолати цей «парадокс пестициду», тестові сценарії повинні регулярно рецензуватись і коригуватися, нові тести повинні бути різнобічними, щоб охопити всі компоненти програмного забезпечення, або системи, та знайти як можна більше дефектів.  
**Принцип 6**— Тестування залежить від контексту (Testing is concept depending)  
Тестування виконується по-різному в залежності від контексту. Наприклад, програмне забезпечення, в якому критично важлива безпека, тестується інакше, ніж сайт електронної комерції.   
**Принцип 7** — Оману про відсутність помилок (Absence-of-errors fallacy)  
Виявлення і виправлення дефектів не допоможуть, якщо створена система не підходить користувачеві і не відповідає його очікуванням і потребам.

**Статичне і динамічне тестування**  
Статичне тестування відрізняється від динамічного тим, що виробляється без запуску програмного коду продукту. Тестування здійснюється шляхом аналізу програмного коду (code review) або скомпільованого коду. Аналіз може здійснюватися як вручну, так і за допомогою спеціальних інструментальних засобів. Метою аналізу є раннє виявлення помилок і потенційних проблем в продукті. Також до статичного тестування відноситься тестування специфікації та іншої документації.  
  
**Вимоги** — це специфікація (опис) того, що повинно бути реалізовано в програмному продукті. Вимоги описують те, що необхідно реалізувати, без деталізації технічної сторони рішення. — Що, а не як.  
**Вимоги повинні задовольняти наступні характеристики:**  
• Коректність;  
• Недвозначність;  
• Повнота набору вимог;  
• Несуперечність набору вимог;  
• Придатність до тестування;  
• Покроковість;  
• Зрозумілість.

## 1.4 Переваги й недоліки ручного тестування

Плюси ручного тестування:

* увесь звіт тестувальника може бути розглянутий як зворотний зв’язок від потенційного користувача.
* у наш час призначений для користувача інтерфейс грає величезну роль, і тому повністю протестувати його можна тільки вручну;
* нижча вартість: у короткостроковій перспективі ручне тестування дешевше, ніж інструменти автоматизованої перевірки;
* тестування в реальному часі: незначні зміни можуть бути досліджені відразу, без написання коду і його виконання;
* можливість дослідницького тестування, метою якого є перевірка різноманітних можливостей додатка. Важливо, що використовуються не заздалегідь складені тест-кейси, а придумані на льоту сценарії.

Мінуси ручного тестування:

* людський чинник: хоча інтерфейс користувача і може бути протестований тільки вручну, люди часто схильні до неефективності. Деякі помилки можуть залишитися непоміченими;
* трудомісткість повторного використання: провести серію стандартних автоматичних тестів простіше, ніж протестувати проект вручну після внесення навіть невеликих змін;
* неможливість тестування навантаження: не можливо змоделювати велику кількість користувачів вручну.

# Основні принципи та підходи автоматизованого тестування програмного забезпечення

Перші спроби «автоматизації» з'явилися в епоху операційних систем [DOS](https://uk.wikipedia.org/wiki/DOS) і [CP/M](https://uk.wikipedia.org/wiki/CP/M). Тоді вона полягала у видачі додатком команд через командний рядок і аналізі результатів. Трохи пізніше додалися віддалені виклики через [API](https://uk.wikipedia.org/wiki/API) для роботи з мережі. Вперше про автоматизоване тестування згадується в книзі [Фредеріка Брукса](https://uk.wikipedia.org/wiki/%D0%A4%D1%80%D0%B5%D0%B4%D0%B5%D1%80%D1%96%D0%BA_%D0%91%D1%80%D1%83%D0%BA%D1%81) «Міфічний людино-місяць», де йдеться про перспективи використання модульного тестування. Але по-справжньому автоматизація тестування стала розвиватися тільки в 1980-х роках.

[*Автоматизоване тестування*](https://uk.wikipedia.org/wiki/%D0%90%D0%B2%D1%82%D0%BE%D0%BC%D0%B0%D1%82%D0%B8%D0%B7%D0%BE%D0%B2%D0%B0%D0%BD%D0%B5_%D1%82%D0%B5%D1%81%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F) використовується для зменшення витрат часу на тестування та його вартості. Комп'ютер може виконувати записану послідовність кроків швидше, ніж людина, а також може запускати тести вночі, щоб отримати результати в першій половині дня. Проте праця, заощаджена на виконанні автоматизованих тестів, має витрачатися на написання тестових програм. Залежно від типу програм, які мають бути протестовані, і обраних засобів автоматизації, це може потребувати більшої кількості робочої сили, ніж ручний підхід. Крім того, деякі інструменти тестування видають дуже велику кількість даних, потенційно створюючи завдання інтерпретації результатів, яке може забрати забагато часу.

Щоб програмне забезпечення безпомилково працювало в призначеному середовищі, необхідне тестування, за допомогою якого мають не тільки ефективно знаходитися дефекти, але також, наскільки можливо, швидко і дешево виконуватись тести. Автоматизація тестування дасть змогу значно зменшити зусилля, необхідні для адекватного тестування або значно покращити тестування, проведення якого обмежено часом.

## 2.1 Визначення доцільності впровадження автоматизованого тестування для програмних продуктів

Автоматизоване тестування вигідно застосовувати, коли продукт часто змінюється, але має велику функціональність. Так можна відслідковувати стабільність роботи всіх функцій після внесення змін у програмний продукт. Автоматизоване тестування вигідно застосовувати після створення кожної нової версії програмного продукту в циклі розроблення ужитку. При цьому участь людини не є необхідною, адже програмний скрипт може запускатись навіть вночі. Автоматизоване тестування займає окреме місце серед наведених вище типів тестування, тому що автоматизувати можна як функціональне, так і навантажувальне тестування. Використання автоматизованого тестування – це інвестиції в майбутнє для підприємств-виробників програмного забезпечення: воно значно підвищує якість продуктів, що випускаються, а це, своєю чергою, виводить підприємство на новий, набагато вищий рівень виробництва програмного забезпечення. Правда, при цьому можуть виникати і певні труднощі та проблеми, адже перехід від “ручного” до автоматизованого тестування може забрати багато фінансових і часових ресурсів, а також потребувати переналаштування бізнес-процесів фірми:

− початок кожного автоматизованого тестування починається з ручного тестування, коли людина повинна показати роботу, що, де і як тестувати;

− створення автоматичних тестів вимагає від тестувальника програмістських знань, тому що професійна автоматизація неможлива без безпосередньої роботи з кодом програмного скрипта та знань об’єктно-орієнтованого програмування;

− чутливість автоматизованих тестів до програмного і апаратного середовища;

− автоматизацію недоцільно використовувати, коли об’єкт може протестувати тільки людина. Засіб автоматизованого тестування – це робот з певним штучним інтелектом. А відомо, що найскладнішою задачею штучного інтелекту є розпізнавання складних об’єктів і моделей поведінки. Тобто, протестувати програмний продукт на ергономіку за допомогою автоматизованого тестування неможливо. Всі вищеперераховані складні випадки необхідно брати до уваги.

Проте, якщо жоден з них не стане перепоною на шляху впровадження автоматизованого тестування і фірма почне випускати якісне автоматично протестоване програмне забезпечення набагато швидше, ніж конкуренти, то інвестування в автоматизацію виправдається не раз. Існує єдине правило, коли недоцільно застосовувати автоматизоване тестування: коли немає впевненості, що тестується один і той самий об’єкт в однакових умовах. Процес розроблення програмного забезпечення повинен передбачати такі періоди, коли зміни до коду програмного продукту не вводяться, і тестувальники можуть спокійно працювати. Найдоцільніше автоматизувати функціональне (Function testing) та навантажувальне тестування (Load testing), але і то не завжди. Як і в будь-якій справі, приймаючи рішення про автоматизацію, необхідно керуватись принципом економічності. Якщо компанія впевнена, що випускається і в майбутньому випускатиметься багато версій програмного продукту або програмний продукт є складним для “ручного” тестування, то в такому випадку автоматизація є доцільною. Процес автоматизації функціонального тестування програмного продукту забезпечить отримання прямих фінансових вигод за рахунок економії ресурсів, а також підвищення якості робіт, які виконуються для тестування програмного забезпечення. Автоматизація зробить процес тестування інформаційної системи багаторазовим та повторюваним без участі реального тестувальника, тим самим звільняючи ресурси тестувальників від рутинних, монотонних “ручних” перевірок функціональності.

Для того, щоб застосувати технологію автоматизованого тестування, необхідно виконати такі обов’язкові дії:

− провести аудит всієї існуючої проектної документації програмного продукту чи веб-ужитку для визначення доцільності автоматизованого тестування;

− вибрати з проектної документації вимоги, які необхідно перевірити в тестуванні;

− розробити план і порядок тестування;

− створити покрокові тест-кейси для перевірки всіх вимог;

− визначити необхідні умови та ресурси для проведення тестування (програмно-апаратна платформа та людські ресурси);

− вибрати засіб чи інструментарій для автоматизованого тестування згідно з визначеними умовами;

− створити за допомогою вибраного інструментарію програмний скрипт для проходження всіх тест-кейсів;

− запустити програмний скрипт на виконання;

− результати виконання тестів зберегти для звітування.

Найістотнішими перевагами автоматизованого тестування є такі:

− автоматизоване тестування економить час, тому що програма-робот швидше виконує операції чи відсилає запити за протоколом, ніж людина;

– це дає змогу швидше знайти помилки і виправити їх;

− автоматизоване тестування виключає людський фактор, адже програма-робот виконуватиме постійно повторювані рутинні операції без збоїв;

− автоматизоване тестування можна використовувати в програмних продуктах, які не мають графічного користувацького інтерфейсу – це доцільно на ранніх стадіях розробки, коли тестуватимуться виконавчі модулі чи функції, які повинні правильно працювати; − автоматизоване тестування дає змогу емулювати роботу багатьох користувачів, що є дуже важливим для навантажувального тестування; − всі засоби автоматизованого тестування мають інструментарій для фіксації знайдених помилок і результатів, допомагаючи моделювати різноманітні помилкові ситуації, заповнювати звіти, будувати діаграми. Зберігання створених наборів тестів є дуже важливим аспектом в автоматизованому тестуванні.

До тестів необхідно ставитися як до вихідного коду, тобто використовувати версії для можливості відтворення попередніх тестів, для повторного використання вже існуючих. Отже, основні переваги і недоліки автоматизованого тестування визначено. Окремо можна сказати про людський фактор під час цього складного процесу. Як вже говорилося вище, автоматизоване тестування виконує програма-робот. Не треба забувати, що будь-який засіб автоматизованого тестування – це тільки інструмент. Тому велику увагу треба приділити плануванню, а саме створенню тестового плану і тест-кейсів. Акуратне і адекватне планування – запорука успішного автоматизованого тестування. Засіб автоматизованого тестування не писатиме тестовий план і тест-кейси, а тільки їх виконуватиме. В тест-кейсах не повинно бути нічого, крім перевірки вимог (явних і неявних), які необхідно поділити за пріоритетністю. Неадекватний тест- план і тест-кейси зведуть автоматизацію до даремної втрати часу. Тому автотест, як і будь-яка інша звичайна програма, потребує акуратного створення та постійної підтримки. А це все потребує часу, деколи не меншого, ніж на розроблення самого програмного продукту. Коли вже зрозуміло, що і навіщо тестувати, створено тестовий план і тест-кейси, необхідно з’ясувати, як тестування зробити ефективнішим, швидшим, якіснішим. Застосування “ручного” тестування до складного програмного забезпечення є дорогим, а результати після кожного “ручного” виконання тестів зникають і їх важко відтворити. Для того, щоб збільшити об’єм перевірок і підвищити якість тестування, забезпечити можливість повторного виконання тестів при внесенні змін в програмне забезпечення, застосовують засоби автоматизації. Тому тестувальник повинен володіти знаннями про можливі засоби автоматизованого тестування, знати їх переваги і недоліки для того, щоб вміло вибрати для своїх потреб найкращий інструмент. Адже не всі вони однаково працюють з усіма різноманітними платформами і протоколами, технологіями створення ужитків і обміну даними.

Здебільшого засоби автоматизації тестування актуально застосувати в наступних випадках:

* для рутинних операцій — людині вони не цікаві, у людини настає втома, розсіяність. Наприклад, форми, в яких міститься велика кількість полів для набору даних (перебір даних). Автоматизований тестовий процес дозволяє автоматично виконувати безпомилкове заповнення полів, а також після збереження внесених даних здійснювати їх перевірку. Заповнюючи некоректними даними форму, одночасно проводиться перевірка різної валідації;
* коли необхідне повторне використання тестів при коригуванні ПЗ;
* довгих сценаріях (від кінця до кінця);
* з бекендом, що є найбільш важкодоступним;
* при роботі із базами даних, логуванням файлів, даними бази де присутній високий потенційний ризик припуститися помилки, і ці помилки вкрай небажані;
* тестування даних, для яких необхідні точні математичні розрахунки;
* добра практика автоматизовувати пошук даних у додатку, бо так гарантовано забезпечується швидке знаходження помилок.

Навпаки не варто застосовувати автоматизоване тестування:

* для перевірки нової функціональності програми — очевидно, ми не знаємо ще її поведінки;
* у випадках, де обов’язково необхідна участь людини — автоматизоване тестування — відмінний інструмент для перевірки коду, але для комп’ютера і його коду залишаються незбагненними, «просте людське співчуття», наприклад, тестування користувацького інтерфейсу чи «почуття прекрасного» у поєднані кольорів верстки (всі чергові розробки нейронних мереж це підтверджують);
* на малесенькому проекті вкладення коштів в автоматизацію 100% буде невиправданою розкішшю. Ручне тестування у цьому випадку організувати легше і швидше. Теж саме на великому проекті, якщо  проект «з’їдає» занадто ресурсів, то це вірна ознака того, що щось не те відбувається з автоматизацією на проекті;
* якщо проект має часті релізи, теж цілком можлива ситуація, що здійснити ручне тестування можна швидше, ніж написати тести. Автоматизоване тестування вимагає фахівців, які у ньому тямлять, часу щоб вони вникнули у суть проекту, плюс саме написання скрипту вимагагає часу.

## Види автоматизованого тестування

Існує два основних підходи до автоматизації тестування: тестування на рівні коду і [GUI](https://uk.wikipedia.org/wiki/GUI)-тестування. До першого типу належить, зокрема, модульне тестування. До другого — імітація дій користувача за допомогою спеціальних тестових [фреймворків](https://uk.wikipedia.org/wiki/%D0%A4%D1%80%D0%B5%D0%B9%D0%BC%D0%B2%D0%BE%D1%80%D0%BA).

[*Модульне тестування*](https://uk.wikipedia.org/wiki/%D0%9C%D0%BE%D0%B4%D1%83%D0%BB%D1%8C%D0%BD%D0%B5_%D1%82%D0%B5%D1%81%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F)*([англ.](https://uk.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D1%96%D0%B9%D1%81%D1%8C%D0%BA%D0%B0_%D0%BC%D0%BE%D0%B2%D0%B0) Unit testing*) — це метод тестування програмного забезпечення, який полягає в окремому тестуванні кожного модуля коду програми. Модулем називають найменшу частину програми, яку може бути протестованою. У процедурному програмуванні модулем вважають окрему функцію або процедуру. В об'єктно-орієнтованому програмуванні — інтерфейс, клас. Модульні тести, або unit-тести, розробляються в процесі розробки програмістами та, іноді, тестувальниками білої скриньки (white-box testers).

Найпоширенішою формою автоматизації є тестування додатків через графічний інтерфейс користувача. Популярність такого виду тестування пояснюється двома факторами: по-перше, додаток тестується тим же способом, яким його буде використовувати людина, по-друге, можна тестувати додаток, не маючи при цьому доступу до вихідного коду.

GUI-автоматизація розвивалася протягом 4 поколінь інструментів і технік:

* **Утиліти запису і відтворення (capture/playback tools)** — записують дії тестувальника під час ручного тестування. Вони дозволяють виконувати тести без прямої участі людини протягом тривалого часу, значно збільшуючи продуктивність і усуваючи «безглузде» повторення одноманітних дій під час ручного тестування. У той же час, будь-яка мала зміна ПЗ, що тестується вимагає перезапису ручних тестів. Тому це перше покоління інструментів не ефективне і не масштабоване.
* **Сценарії (Scripting)** — форма програмування на мовах, спеціально розроблених для автоматизації тестування ПЗ — пом'якшує багато проблем capture/playback tools. Але розробкою займаються програмісти високого рівня, які працюють окремо від тестувальників, що безпосередньо запускають тести. До того ж скрипти найбільше підходять для тестування GUI і не можуть бути впровадженими, пакетними або взагалі будь-яким чином об'єднані в систему. Нарешті, зміни в ПЗ, яке тестується вимагають складних змін у відповідних скриптах, і підтримка все більше зростаючої бібліотеки тестуючих скриптів стає зрештою непереборним завданням.
* **Data-driven testing** — методологія, яка використовується в автоматизації тестування. Особливістю є те, що тестові скрипти виконуються і верифікуються на основі даних, які зберігаються в центральному сховищі даних або БД. Роль БД можуть виконувати ODBC-ресурси, csv або xls файли і т. д. Data-driven testing — це об'єднання декількох взаємодіючих тестових скриптів та їх джерел даних в фреймворк, який використовується в методології. У цьому фреймворку змінні використовуються як для вхідних значень, так і для вихідних перевірочних значень: у тестовому скрипті зазвичай закодовані навігація по додатком, читання джерел даних, ведення логів тестування. Таким чином, логіка, яка буде виконана в скрипті, також залежить від даних.
* **Keyword-based** автоматизація передбачає поділ процесу створення кейсів на 2 етапи: етап планування та етап реалізації.

## 2.3 Переваги та недоліки автоматизованого тестування

Однією з головних проблем автоматизованого тестування є його трудомісткість: попри те, що воно дозволяє усунути частину рутинних операцій і прискорити виконання тестів, великі ресурси можуть витрачатися на оновлення самих тестів. Це відноситься до обох видів автоматизації. При рефакторінгу часто буває необхідно оновити і модульні тести, і зміна коду тестів може зайняти стільки ж часу, скільки і зміна основного коду. З іншого боку, при зміні інтерфейсу програми необхідно заново переписати всі тести, які пов'язані з оновленими вікнами, що при великій кількості тестів може відняти значні ресурси.

Плюси автоматизованого тестування

* Можливість тестування навантаження. Можна досить швидко змоделювати велику кількість користувачів.
* Заощадження часу. Ручне тестування великих застосувань – довгий і трудомісткий процес, тоді як сценарії пишуться лише один раз.
* Можливість повторного використання. Тестовий сценарій, написаний один раз, може бути використаний і в майбутньому при черговому оновленні проекту.

Мінуси автоматизованого тестування

* Висока ціна. Інструменти автоматизованого тестування, а також навчання їх використанню коштують недешево, тому треба ретельно оцінювати бюджет.
* UI-тестування. Автоматизоване тестування не може повною мірою покрити вимоги до призначеного для користувача інтерфейсу.
* Відсутність “людського погляду”. Можливе існування помилок, які помітить тільки людина.

Отже, можна зробити висновок, що обидва види тестування(ручне та авоматизоване) мають як переваги, так і недоліки. Комбінація обох – ідеальний спосіб отримати від тестування максимальний результат.

# Структура проекту та використанні технології для розробки збереження та обробки даних

Задача полягала у розробці веб порталу електронної бібліотеки наукових робіт, курсових та магістерських робіт студентів кафедри «Математичного моделювання».

Метою магістерської роботи є організувати збереження інформації про захищену курсову чи дипломну роботу.

Для реалізації поставленого завдання необхідно:

* розробити схему бази даних,
* визначити набір фільтрів для здійснення пошуку,
* підібрати технології та ресурси для забезпечення розгортання програмного продукту обраної роботи.

## 3.1 Характеристика бази даних

Для збереження даних в електронній бібліотеці використовується реляційна база даних, яка слугує для збереження таких мета даних наукових праць як:

* автор роботи,
* дата публікації,
* назва,
* тематика роботи,
* інша додаткова інформація, як розроблений додаток до магістерських, курсових робіт.

Схема бази даних:

Згідно малюнка схеми даних проекту «elibrary» який зображений на рис.1, схема бази даних складається з наступних реляційних таблиць:

* User
* Role
* Publication
* Them
* Work\_type
* Storage

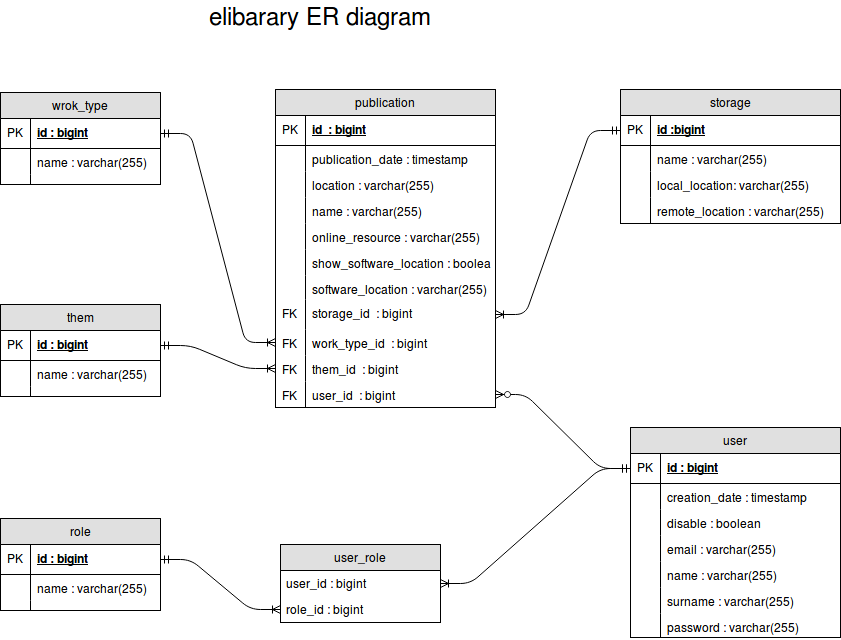


Рис.1 Схема бази даних

Структура таблиці User складається з наступних полів: імені користувача, прізвища користувача, електронної адреси, паролю, дати внесення користувача у базу даних, та на сам кінець прапорця чи даний користувач є активним в даний момент. Описана вище таблиця використана для внесення автора публікації так і для визначення системою аутентифікації та авторизації разом із даними таблички Role. Як можна побачити зі схеми зображеної на рис.1, таблиці User та Roles поєднані між собою зв’язком багато до багатьох, адже в нашій системі користувач може мати від однієї ролі до багатьох, тобто користувач може виступати як у ролі звичайного автора , так і у ролі адміністратора який має права на внесення та редагування матеріалів до електронної бібліотеки.

Наступною таблицею є Storage, яка складається з наступних полів: айді для унікальної ідентифікації, імені, локального шляху, глобального шляху. Два останніх поля використовуються для правильного розміщення файлів як у локальній файловій системі так і для роздачі їх по HTTP протоколу. Дана таблиця була введена для покриття вимог до проекту, що проект повинен мати можливість зберігати файли як на локальній файловій системі так і на хмарних сервісах.

Наступна таблиця у схемі це – Theme, яка складається тільки з двох полів: айді та ім’я.

Також схожою є таблиця Work\_type, яка містить такі ж стовпці: айді та ім’я. В даній таблиці будуть зберігатись такі дані як тип публікації: наукова стаття, курсова робота, магістерська і т.д.

Головною таблицею проекту є таблиця Publication, адже в ній міститься основна інформація, яка використовується в проекті: ім’я наукової роботи, дата публікації, відносний шлях до файлу у файловій системі, посилання на онлайн ресурс, якщо це якийсь існуючий веб проект тощо, відносний шлях на посилання до коду імплементації(якщо такий існує), прапорець, який вказує чи можна надавати посилання на код в залежності від ліцензування коду, айді сховища, айді теми, типу роботи та айді автора.

Як засіб СКБД в цьому проекті було використано СКБД PostgreSQL, основним критерієм вибору стало:

* + - PostgreSQL - це клієнт серверна система керування базами даних.
    - не потребує жодних ліцензій на використання,
    - На даний момент актуальна, тобто підтримується розробниками та виходять патчі з правками помилок і нові версії продукту.
    - PostgreSQL має дуже велику кількість драйверів для розробки програмних продуктів різними сучасними мовами програмування.

## 3.2 Використані технології для розробки

Згідно вимог поставлених замовником, а саме кафедрою «Математичного моделювання» факультету «Математики та Інформатики» Чернівецького національного Університету, проект має мати такі характеристики:

* + - Проект не повинен залежати від платформи (операційної системи).
    - Повинен мати можливість зберігати данні як на локальній файловій системі сервера так і можливість розміщати дані в хмарному сервісі.
    - Проект повинен відповідати найсучаснішим тенденціям розробки програмного забезпечення та архітектури.

Тому при розробці даного продукту було вирішено що це буде веб додаток, який імплементує REST архітектуру, а саме складається з двох під проектів.

Перший відповідає за відображення інформації про публікації у веб браузері, для реалізації даного проекту використаний фреймворк Angular 4. Це фреймворк побудований компанією Google для створенні односторінкових аплікацій, який дає змогу динамічно змінювати контент в залежності від дій користувача. Даний фреймворк – це імплементація шаблону розробки MVC, тобто це є реалізація зв’язки модель - вигляд - контролер. Він зараз набирає дуже великої популярності в зв’язку зі зручністю використання і можливостей побудови динамічних аплікацій. Основною мовою розробки є мова JS; також в даному фреймворці широко застосовується нещодавно розроблена мова програмування Type Script, яка є надбудовую над JS і дає можливість знаходити велику кількість помилок на етапі компіляції коду у JS.

Другим під проектом є проект розроблений мовою програмуванна Java; він також імплементує REST архітектуру – це веб додаток розроблений за допомогою найновішого фреймворку Spring boot, який в свою чергу є надбудовою над загальновідомим фреймворком Spring який в свою чергу складається з великої кількості таких підфреймворків:

- Spring Core це основний фреймворк для розробки спринг додатків, адже саме він реалізує такі основні функціональності як dependency injection - це властивість, яка дає можливість розробнику абстрагуватись від процесу створення об’єктів і реалізації їх зв’язків, а описати конфігурацію, яка в свою чергу дасть команду для контейнера створити всі потрібні об’єкти на той час скільки вони нам потрібні і також організувати зв'язок між цими об’єктами,

- Spring AOP - це також дуже корисна функціональність, яка дає можливість розробнику створити обгортки над будь-яким створеним об’єктом, яка в свою чергу дає можливість інтегрувати легування виконання кожного методу чи модифікація, або ж фільтрування даних до запуску власне методу, який має обрацювати ці дані,

- SpringWeb – це фреймворк який дає змогу легко зконфігурувати проект та, використовуючи стандартні анотації, легко визначити класи та сервіси, які зможуть комунікувати із зовнішнім світом за допомогою протоколу HTTP. Даний фреймворк надає розробнику набір анотацій для швидкого конфігурування Rest сервісів, контролерів та багато інших корисних функціональностей.

- SpringData - це фреймворк для роботи з різними джерелами збереження даних, який був розроблений для полегшення розробнику написання коду для комунікації між різними базами даних, даний фреймворк використовує фреймворки мапування зв’язків таблиць бази даних з об’єктами мови Java , до прикладу, Hibernate реалізує всі інтерфейси та специфікації JPA.

- Spring Boot дає можливість використовувати такі бібліотеки для складання та компіляції проекту як Maven, що є проект генератором, який використовує плагіни та надає можливість програмісту зібрати проект будь-якої складності, допомагаючи управляти залежностями та маючи один з найбільших у світі репозиторіїв для збереження бібліотек з відкритим кодом. Прописуючи всю структуру в одному файлі pom.xml програміст має змогу інформувати мейвен коли і які дію треба виконати.

Так як проект –це веб портал, то відповідно для розробки інтерфейсу було використано наступні засоби HTML5, CSS, JS. Для побудови користувацького інтерфейсу було обрано новітній і один з найпопулярніших в даний момент фреймворк –Angular4.

# Список використаної літератури

1. <http://ua.textreferat.com/referat-8464-1.html>
2. [https://uk.wikipedia.org/wiki/Програмне\_забезпечення](https://uk.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BD%D0%B5_%D0%B7%D0%B0%D0%B1%D0%B5%D0%B7%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%BD%D1%8F)