**Лекція 17 Тестове покриття**

**Тестування програмного забезпечення** це метод перевірки відповідності фактичного програмного продукту очікуваним вимогам і переконання, що програмний продукт відповідає вимогам якості. Він передбачає виконання компонентів програмного забезпечення/системи за допомогою ручних або автоматизованих інструментів для оцінки однієї чи кількох властивостей. Метою тестування програмного забезпечення є виявлення помилок, прогалин або відсутніх вимог на відміну від фактично заданих вимог.

**Тестування програмного забезпечення – це важливо** тому що якщо в програмному забезпеченні є якісь помилки, їх можна виявити на ранній стадії та вирішити до доставки програмного продукту. Належним чином перевірений програмний продукт забезпечує надійність, безпеку та високу продуктивність, що додатково призводить до економії часу, ефективності витрат і задоволення клієнтів.

Тестування є важливим, оскільки помилки програмного забезпечення можуть бути дорогими або навіть небезпечними. Помилки програмного забезпечення потенційно можуть призвести до грошових і людських втрат, і історія повна таких прикладів.

* Автомобілі Nissan відкликали з ринку понад 1 мільйон автомобілів через збій програмного забезпечення сенсорних детекторів подушок безпеки. Повідомлялося про дві аварії через програмний збій.
* Starbucks була змушена закрити близько 60 відсотків магазинів у США та Канаді через збій програмного забезпечення в системі POS. Одного разу в магазині подали каву безкоштовно, оскільки вони не змогли обробити транзакцію.
* Дещо з Amazon Сторонні роздрібні продавці побачили, що ціна їхніх продуктів знижена до 1 пенса через програмний збій. Вони зазнали великих втрат.
* У 2015 році винищувач F-35 став жертвою програмної помилки, через яку він не міг правильно виявляти цілі.
* Airbus A300 авіакомпанії China Airlines розбився через програмну помилку 26 квітня 1994 року, в результаті чого загинули 264 невинних людини.
* У 1985 році канадський апарат для променевої терапії Therac-25 вийшов з ладу через помилку в програмному забезпеченні та доставляв смертельні дози опромінення пацієнтам, в результаті чого 3 людини загинули та 3 отримали серйозні поранення.
* У квітні 1999 року помилка програмного забезпечення спричинила збій $1.2 bill запуск військового супутника ion, найдорожча аварія в історії
* У травні 1996 року помилка програмного забезпечення призвела до того, що на банківські рахунки 823 клієнтів великого банку США було зараховано 920 мільйонів доларів США.

**Переваги використання тестування програмного забезпечення**:

* **Економічно ефективним:** Це одна з важливих переваг тестування програмного забезпечення. Своєчасне тестування будь-якого ІТ-проекту допоможе зберегти гроші в довгостроковій перспективі. У випадку, якщо помилки були виявлені на ранній стадії тестування програмного забезпечення, виправлення коштує менше.
* **Безпека:** Це найбільш вразлива та чутлива перевага тестування програмного забезпечення. Люди шукають перевірені продукти. Це допомагає раніше усунути ризики та проблеми.
* **Якість продукту:** Це важлива вимога до будь-якого програмного продукту. Тестування гарантує доставку якісного продукту клієнтам.
* **Задоволеності клієнтів:** Основна мета будь-якого продукту - задовольнити своїх клієнтів. Тестування UI/UX забезпечує найкращу взаємодію з користувачем.

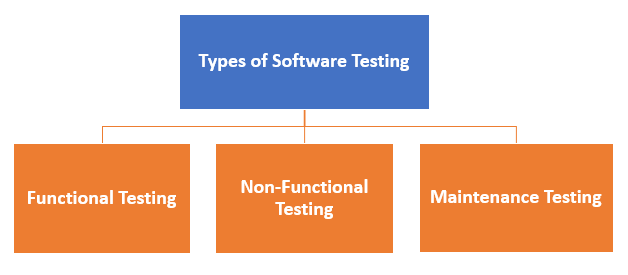
**Тестування в програмній інженерії**

Відповідно до ANSI/IEEE 1059, **Тестування в програмній інженерії** це процес оцінки програмного продукту, щоб визначити, чи відповідає поточний програмний продукт необхідним умовам чи ні. Процес тестування передбачає оцінку функцій програмного продукту на вимоги з точки зору будь-яких відсутніх вимог, помилок або помилок, безпеки, надійності та продуктивності.

**Види тестування ПЗ**

Зазвичай тестування поділяють на три категорії.

* Функціональне тестування
* Нефункціональне тестування або Тестування продуктивності
* Технічне обслуговування (регресія та технічне обслуговування)

[](https://www.guru99.com/images/2/061920_1310_Whatissoftwaretesting1.png)

Типи тестування програмного забезпечення в програмній інженерії

|  |  |
| --- | --- |
| **Категорія тестування** | **Види тестування** |
| Функціональне тестування | * Unit Testing * Інтеграційне тестування * Дим * UAT (тестування прийнятності користувача) * Локалізація * Глобалізація * Interoperaздатність * Так і далі |
| Нефункціональне тестування | * продуктивність * Витривалість * Навантаження * Об'єм * масштабованість * Юзабіліті * Тощо |
| технічне обслуговування | * Регресія * технічне обслуговування |

Це не повний список, оскільки їх більше ніж 150 видів тестування типів і ця кількість зростає. Не всі типи тестування можна застосувати до всіх проектів, це залежить від характеру та обсягу проекту.

**Стратегії тестування в інженерії програмного забезпечення**

**Модульне тестування:** Цей базовий підхід до тестування програмного забезпечення використовується програмістом для тестування одиниці програми. Це допомагає розробникам знати, чи окрема одиниця коду працює правильно чи ні.

**Інтеграційне тестування:** Він зосереджений на створенні та дизайні програмного забезпечення. Ви повинні бачити, чи працюють інтегровані блоки без помилок чи ні.

**Тестування системи:** У цьому методі ваше програмне забезпечення компілюється в цілому, а потім тестується в цілому. Ця стратегія тестування перевіряє, серед іншого, функціональність, безпеку, портативність.

**Тестування програми**

**Тестування програми** у тестуванні програмного забезпечення — це метод виконання фактичної програми програмного забезпечення з метою тестування поведінки програми та пошуку помилок. Програма виконується з тестовими даними для аналізу поведінки програми або відповіді на тестові дані. Гарне тестування програми – це те, яке має високі шанси виявити помилки.

# Покриття коду: розгалуження, тестування операторів і рішень

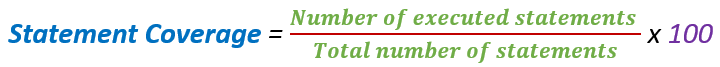
Покриття коду – це показник, який описує ступінь перевірки вихідного коду програми. Це одна з форм тестування методом "білого ящику", яке знаходить області програми, які не виконуються набором тестів. Він також визначає кількість тестів для збільшення охоплення та визначення кількісної міри охоплення коду. У більшості випадків система покриття коду збирає інформацію про запущену програму. Він також поєднує це з інформацією про вихідний код, щоб створити звіт про покриття коду набору тестів.

**Головні причини для використання покриття коду**:

* Це допомагає вам виміряти ефективність виконання тесту
* Метод пропонує кількісне вимірювання.
* Метод н визначає ступінь перевірки вихідного коду.

## Методи покриття коду

**Покриття коду** це техніка тестування білого ящика, при якій усі виконувані оператори у вихідному коді виконуються принаймні один раз. Він використовується для обчислення кількості операторів у вихідному коді, які були виконані. Основна мета Statement Coverage — охопити всі можливі шляхи, рядки та оператори у вихідному коді. Покриття операторів використовується для отримання сценарію на основі структури тестованого коду.



В технології білого ящика, тестувальник зосереджується на тому, як працює програмне забезпечення, на внутрішній роботі вихідного коду щодо графів потоку керування або блок-схем. Як правило, у будь-якому програмному забезпеченні, якщо ми подивимося на вихідний код, буде широкий спектр таких елементів, як oпeрaтори, функції, цикли, виняткові обробники тощо. На основі вхідних даних у програму деякі оператори коду можуть не виконуватися. Метою Покриття операторів (Statement coverage) є охоплення всіх можливих шляхів, рядків і операторів у коді.

Приклад.

Сценарій для розрахунку Statement Coverage для заданого вихідного коду. Тут ми беремо два різні сценарії, щоб перевірити відсоток охоплення заяви для кожного сценарію.

**Вихідний код:**

Prints (int a, int b) { ------------ Printsum is a function

int result = a+ b;

If (result> 0)

Print ("Positive", result)

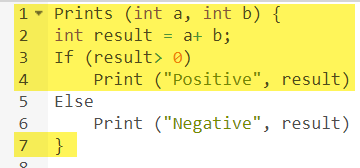
Else

Print ("Negative", result)

} ----------- End of the source code

**Сценарій 1:**

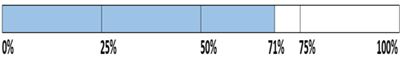
Якщо A = 3, B = 9



Жовтим кольором позначені оператори, які виконуються за сценарієм

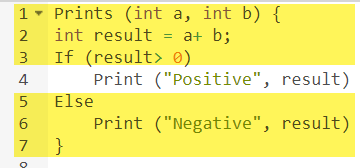
Кількість виконаних операторів = 5, Загальна кількість операторів = 7

Охоплення орераторів: 5/7 = 71%



**Сценарій 2:**

Якщо A = -3, B = -9



Жовтим кольором позначені оператори, які виконуються за сценарієм.

Кількість виконаних операторів = 6

Загальна кількість операторів = 7

Охоплення орераторів: 6/7 = 85%

Всі оператори охоплюються обома сценаріями. Таким чином, ми можемо зробити висновок, що загальне покриття становить 100%.

**Що охоплює Statement Coverage?**

1. Невикористані орератори
2. Мертвий код
3. Невикористані гілки коду
4. Відсутні твердження

## Тестування покриття рішень

**Покриття рішень** це техніка тестування білого ящика, яка повідомляє про істинні або хибні результати кожного логічного виразу вихідного коду. Метою тестування покриття рішення є охоплення та перевірка всього доступного вихідного коду шляхом перевірки та забезпечення того, що кожна гілка кожної можливої точки прийняття рішення виконується принаймні один раз.

З цього погляду вирази іноді можуть бути складними. Тому досягти 100% покриття дуже важко. Ось чому існує багато різних методів звітування про цей показник. Усі ці методи спрямовані на охоплення найважливіших комбінацій. Це дуже схоже на покриття рішень, але пропонує кращу чутливість для керування потоком.

https://www.guru99.com/images/1/102518_1122_CodeCoverag12.jpg

### Приклад покриття рішень

**Розглянемо код-**

Demo(int a) {

If (a> 5)

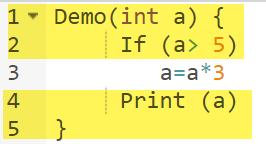
a=a\*3

Print (a)

}

**Сценарій 1:**

Значення a дорівнює 2

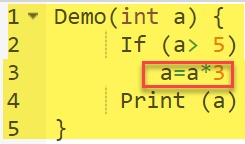


Код, виділений жовтим кольором, буде виконано. Тут перевіряється результат «Ні» рішення Якщо (a>5).

Покриття рішень = 50%

**Сценарій 2:**

Значення a дорівнює 6



Код, виділений жовтим кольором, буде виконано. Тут перевіряється результат «Так» рішення Якщо (a>5).

Покриття рішень = 50%

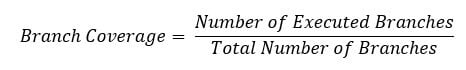
|  |  |  |  |
| --- | --- | --- | --- |
| **Тестовий випадок** | **Значення А** | **Вихід** | **Покриття рішень** |
| 1 | 2 | 2 | 50% |
| 2 | 6 | 18 | 50% |

## Тестування охоплення гілок

**Покриття гілок** це метод тестування білого ящика, у якому перевіряється кожен результат модуля коду (вираз або цикл). Мета покриття гілок полягає в тому, щоб забезпечити виконання кожної умови рішення з кожної гілки принаймні один раз. Це допомагає виміряти частки незалежних сегментів коду та знайти розділи, які не мають розгалужень.

Наприклад, якщо результати є двійковими, вам потрібно перевірити як True, так і False результати.

**Формула для розрахунку покриття гілок:**



### Приклад покриття гілок

**Розглянемо код-**

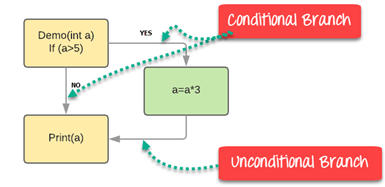
Demo(int a) {

If (a> 5)

a=a\*3

Print (a)

}



Покриття гілок також розглядатиме безумовні гілки

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Тестовий випадок** | **Значення А** | **Вихід** | **Покриття рішень** | **Покриття філій** |
| 1 | 2 | 2 | 50% | **33%** |
| 2 | 6 | 18 | 50% | **67%** |

**Переваги покриття гілок:**

* Дозволяє перевірити всі гілки в коді
* Допомагає переконатися, що жодне розгалуження не призводить до будь-якої аномалії дії програми
* Метод покриття гілок усуває проблеми, які виникають через тестування покриття операторів
* Дозволяє знайти ті області, які не перевіряються іншими методами тестування
* Це дозволяє знайти кількісну міру покриття коду
* Покриття гілок ігнорує гілки всередині булевих виразів

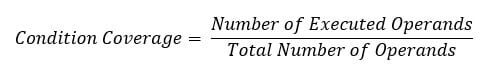
## Тестування покриття умов

**Покриття умов** або покриття виразу — це метод тестування, який використовується для перевірки та оцінки змінних або підвиразів в умовному операторі. Метою покриття умов є перевірка окремих результатів для кожної логічної умови. Покриття умов забезпечує кращу чутливість до потоку керування, ніж охоплення рішень. У цьому висвітленні розглядаються лише вирази з логічним операндами.

Наприклад, якщо вираз має логічні оператори як AND, OR, XOR, що вказує на загальну кількість можливостей.

Покриття умов не дає гарантії щодо повного покриття рішень.

**Формула для розрахунку покриття умов:**



**приклад:**

https://www.guru99.com/images/1/102518_1122_CodeCoverag11.png

**Для наведеного вище виразу ми маємо 4 можливі комбінації**

* TT
* FF
* TF
* FT

**Розглянемо наступний вхід**

|  |  |  |  |
| --- | --- | --- | --- |
| X = 3  Y = 4 | (х | ІСТИНА | Покриття умов становить ¼ = 25% |
| A = 3  B = 4 | (a>b) | ПОМИЛКОВИЙ |

## Покриття кінцевого автомата

Покриття кінцевого автомата, безумовно, найбільше комплексний тип методу покриття коду тому, що він працює на поведінку дизайну. У цьому методі охоплення потрібно шукати, скільки пройдено конкретних за часом станів відвідано. Він також перевіряє, скільки послідовностей включено в кінцевий автомат.

Щоб вибрати метод покриття, тестувальник повинен перевірити, чи

* тестований код має один або кілька невиявлених дефектів
* вартість потенційної невиявленої помилки
* вартість втраченої репутації
* вартість втраченого продажу тощо.

Чим вища ймовірність того, що дефекти призведуть до дорогих збоїв у виробництві, тим суворіший рівень покриття потрібно вибрати.

## Покриття коду проти функціонального покриття

|  |  |
| --- | --- |
| **Покриття коду** | **Функціональне покриття** |
| Покриття коду говорить про те, наскільки добре ваш тестовий сценарій використав вихідний код. | Функціональне охоплення вимірює, наскільки добре функціональні можливості дизайну були охоплені вашим тестовим стенарієм. |
| Ніколи не використовуйте специфікацію дизайну | Використовуйте специфікацію конструкції |
| Зроблено розробниками | Виконано тестувальниками |

|  |  |
| --- | --- |
| **Переваги покриття коду** | **Недоліки кодового покриття** |
| Корисно оцінити кількісну міру покриття коду | Навіть якщо будь-яка конкретна функція не реалізована в дизайні, покриття коду все одно повідомляє про 100% покриття. |
| Це дозволяє створювати додаткові тестові випадки для збільшення покриття | Неможливо визначити, чи ми протестували всі можливі значення функції за допомогою покриття коду |
| Це дозволяє вам знаходити області програми, які не виконуються набором тестів | Охоплення коду також не говорить про те, наскільки добре ви охопили свою логіку |
|  | У випадку, коли зазначена функція не реалізована або не включена в специфікацію, методи на основі структури не можуть знайти цю проблему. |

**Тестове покриття в тестуванні програмного забезпечення**

**Що таке тестове покриття?**

Тестове покриття визначається як показник у тестуванні програмного забезпечення, який вимірює обсяг тестування, виконаного набором тестів. Він включатиме збір інформації про те, які частини програми виконуються під час запуску набору тестів, щоб визначити, які гілки умовних операторів були використані.

Простіше кажучи, це техніка, яка дозволяє переконатися, що ваші тести перевіряють ваш код або те, яку частину коду ви використали під час виконання тесту.

**Що робить Test Coverage?**

* Пошук області вимоги, не реалізованої набором тестів
* Допомагає створити додаткові тестові випадки для збільшення покриття
* Визначення кількісної міри охоплення тестом, яка є непрямим методом перевірки якості
* Виявлення безглуздих тестових випадків, які не збільшують покриття

**Як можна здійснити тестове покриття?**

* Тестове покриття можна здійснити за допомогою методів статичної перевірки, таких як експертні оцінки, інспекції та покрокове керівництво
* Шляхом перетворення спеціальних дефектів у виконувані тестові випадки
* На рівні коду або модульного тестування охоплення тестом можна досягти, скориставшись інструментами автоматизованого покриття коду або модульного тестування
* Функціональне покриття тесту можна здійснити за допомогою відповідних інструментів керування тестуванням

**Переваги тестового покриття в розробці програмного забезпечення**

* Це може гарантувати якість тесту
* Це може допомогти визначити, які частини коду були фактично застосовані для випуску чи виправлення
* Це може допомогти визначити шляхи у вашій програмі, які не перевірялися
* Запобігатидефектам витоку пам‘яті
* Час, обсяг і вартість можна тримати під контролем
* Запобігання дефектам на ранній стадії життєвого циклу проекту
* Цке може визначати всі точки прийняття рішень і шляхи, які використовуються в додатку, що дозволяє збільшити покриття тесту
* Прогалини у вимогах, тестових випадках і дефекти на рівні модуля та коду можна знайти простим способом

**Які основні відмінності між покриттям коду та покриттям тесту?**

Покриття коду і тестове покриття — це методи вимірювання, які дозволяють оцінити якість коду програми.

Ось деякі критичні відмінності між кабінами цих методів покриття:

| **параметри** | **Покриття коду** | **Покриття тесту** |
| --- | --- | --- |
| Визначення | Термін покриття коду, який використовується, коли код програми виконується під час роботи програми. | Тестове покриття означає загальний план тестування. |
| Мета | Показники покриття коду можуть допомогти команді контролювати автоматизовані тести. | Тестове покриття надає деталі про рівень, до якого було перевірено кодування програми. |
| Підтипи | Покриття коду поділено на такі підтипи, як покриття операторів, покриття умов, покриття гілок, покриття Toogle, покриття FSM. | Немає підтипу методу тестового покриття. |

**Формула тестового покриття**

Щоб розрахувати тестове покриття, вам потрібно виконати наведені нижче дії.

**Крок 1)** Загальна кількість рядків коду в якості програмного забезпечення для тестування

**Крок 2)** Кількість рядків коду, які зараз виконуються всіма тестами

Тепер вам потрібно знайти (X, поділене на Y), помножене на 100. Результатом цього обчислення є % вашого тестового покриття.

**Наприклад:**

Якщо кількість рядків коду в компоненті системи дорівнює 500, а кількість рядків, виконаних у всіх існуючих тестах, дорівнює 50, то ваше тестове покриття дорівнює:

(50 / 500) \* 100 = 10%

**Приклади тестового покриття**

**Приклад 1:**

Наприклад, якщо «ніж» — це предмет, який ви хочете перевірити. Потім вам потрібно зосередитися на перевірці, чи правильно він нарізає овочі чи фрукти. Однак є й інші аспекти, на які слід звернути увагу, наприклад, користувач повинен мати можливість зручно ним керувати.

**Приклад 2:**

Наприклад, якщо ви хочете перевірити програму «Блокнот». Тоді перевірка його основних функцій є обов’язковою справою. Однак вам потрібно охопити інші аспекти, оскільки програма «Блокнот» відповідає очікуваним чином під час використання інших програм, користувач розуміє використання програми, не виходить з ладу, коли користувач намагається зробити щось незвичайне тощо.

**Недоліки тестового покриття:**

* Більшість завдань у тестовому покритті виконуються вручну, оскільки немає інструментів для автоматизації. Тому для аналізу вимог і створення тестових випадків потрібно багато зусиль.
* Тестове покриття дозволяє підрахувати функції, а потім порівняти з кількома тестами. Однак завжди є простір для помилкових суджень.

**Візуалізація тестового покриття**

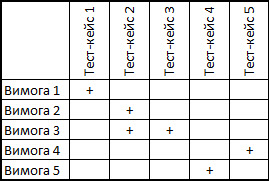
Візуалізація тестового покриття дозволяє оцінювати якість тестування: наскільки добре і ретельно був або буде перевірений продукт, яким областям не вистачає уваги, де основні ризики і який загальний прогрес.

**Навіщо оцінювати тестове покриття?** Будь-які метрики оцінки – трата часу. У цей час можна тестувати, заводити баги, готувати автотести. Яку ж користь ми можемо отримати завдяки метрикам тестового покриття, пожертвувавши часом на тестування?

1. **Пошук своїх слабких зон.** Це потрібно для того, щоб дізнатися, де потрібні поліпшення, виявити функціональні області не покриті тестами, знайти неперевірені блоки і виявити області з найбільшим ризиком пропуску помилок.
2. **Рідко за результатами оцінки покриття ми отримуємо 100%.** Що покращувати? Куди йти? Який зараз відсоток? Як ми його підвищимо будь-яким завданням? Як швидко ми дійдемо до 100? Всі ці питання вносять прозорість і зрозумілість нашого процесу, а відповіді на них дає оцінка покриття.
3. **Фокус уваги.** Припустимо, в нашому продукті 50 різних функціональних зон. Виходить нова версія і ми починаємо тестувати першу з них, знаходимо там помилки, зміщені на пару пікселів кнопки та інші дрібниці... І ось час на тестування завершено і ця функціональність перевірена детально... А решта 49? Оцінка покриття дозволяє нам пріоритезувати завдання виходячи з поточних реалій і термінів.

**Матриця відповідності вимог**

Для валідації і візуалізації покриття продукту тестами QA-інженери використовують **Матрицю відповідності вимог** («**Traceability Matrix**») – двовимірну таблицю, яка містить відповідності функціональних вимог (functional requirements) продукту і підготовлених тестових сценаріїв (test cases). У заголовках рядків таблиці розташовані вимоги, а в заголовках колонок – тестові сценарії або навпаки. На перетині – відмітка, яка означає, що вимога поточної колонки покрита тестовим сценарієм поточного рядка. Матриця зазвичай зберігається у вигляді електронної таблиці.



Таблиця дає візуальне відображення двох параметрів:

* наявність в системі вимог, які ще не покриті (якщо у вимоги немає жодного перетину з тест-кейсами (достатня умова));
* чи є в системі надмірне тестування – якщо вимоги мають кілька перетинів (необхідна умова).

На деяких проєктах матриці трасування можуть бути використані не тільки для оцінки покриття, але і для визначення зв'язків між завданнями на розробку, вимогами і тестовими артефактами.

Тому матриця має вигляд таблиці, кожен рядок якої містить:

* номер та опис завдання на розробку з таск-трекера (№, issue for development);
* логічний блок, до якого належить завдання (опціонально) (block of feature);
* атомарна вимога або приймальний критерій (acceptance criteria);
* пріоритет (priority);
* номер та опис відповідного тестового артефакту (test-case).



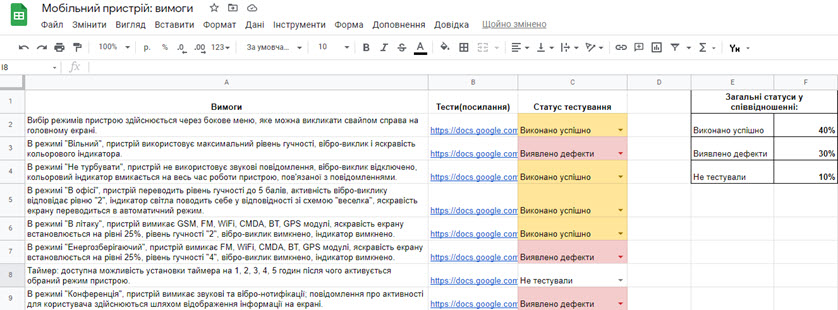
Таке трасування дозволяє:

* візуалізувати актуальний стан реалізації;
* розбивати вимоги на більш атомарні та структурувати їх;
* відслідковувати, чи є вимоги, на які ще не запланована розробка (пропуск реалізації);
* відслідковувати, чи реалізована вимога на даний момент;
* відслідковувати, чи покрита вимога тест-кейсом (пропуск тестування);
* наочно відображати пріоритезацію вимог.

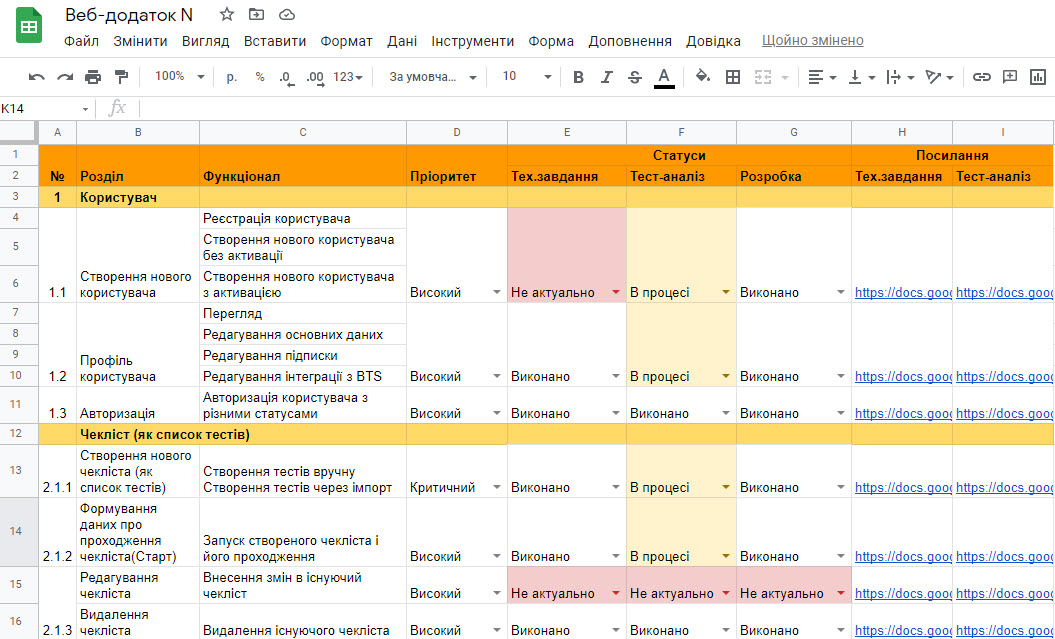
**Приклади візуалізації тестування і покриття**

Нижче, для можливості оцінити зручність і користь візуалізації покриття, наведено приклади з реальних проєктів:

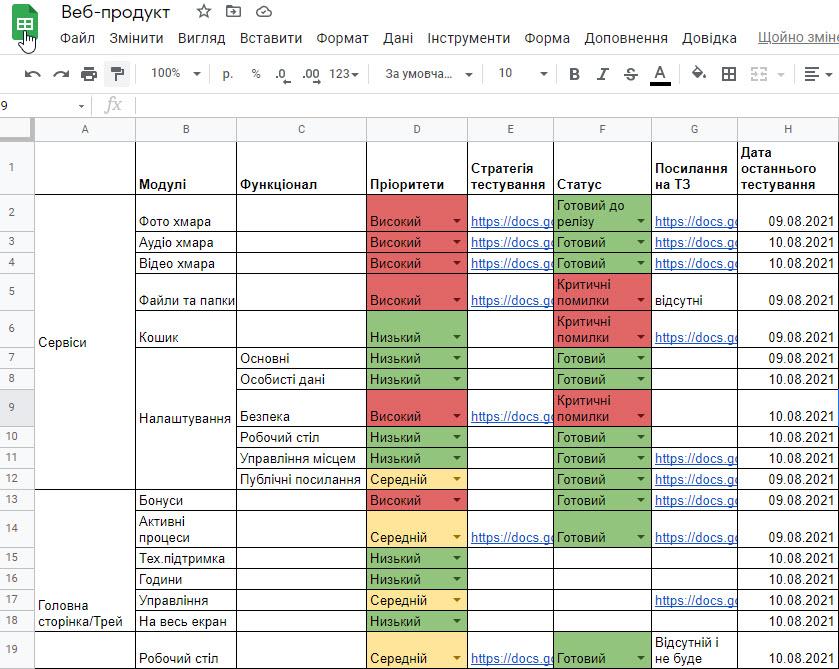
* матриця трасування вимог до мобільного пристрою зі співвідношенням статусів



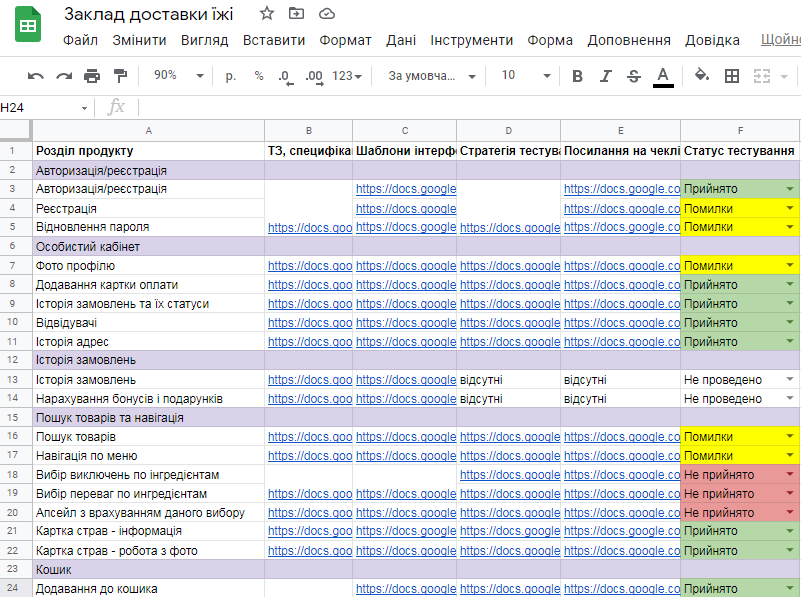
* матриця трасування веб-додатків з поділом за розділами



* матриця трасування веб-продукту за модулями з ітераціями



* матриця трасування інтернет-магазину з доставки їжі



**Висновки**

Візуалізація тестового покриття дозволяє об'єктивно оцінити, яка частина продукту покрита тестами, а яка ні. Це необхідна умова, щоб оцінити, який обсяг роботи вже виконано і що ще залишилося зробити – і по частині створення, і по частині виконання тестів.

В цій статті було розглянуто такий тестовий артефакт, як «Матриця відповідності вимог». Одна з основних її переваг – наочність. Якщо вона підтримується в актуальному стані, то можна відразу побачити «білі плями» і зосередитися на них.

Крім того, матриця трасування дозволяє зв'язати знайдені баги до вимог, що дозволяє тестувальникам більш ефективно «захищати» дефекти. Якщо ви можете вказати, яка саме вимога порушена в конкретному випадку, розробник вже не зможе заявити, що «це не баг, це фіча» :)