**А взагалі, для чого нам потрібні функціональні тести?**

Автоматизовані функціональні тести перевіряють код програми на коректну роботу. Ключове слово “автоматизовані” означає, що розробник чи адміністратор може сам проганяти їх, коли це необхідно. Без допомоги ззовні від тестера чи будь-кого іншого.

Таким чином, функціональні тести дозволяють швидко перевірити, чи новий код розробника працює та чи не поламав старий функціонал. Також вони дозволяють зробити швидкий тест перед тим, як закидати новий реліз програми на кінцевий сервер. Тести допомагають швидше виявити винуватця помилки по історії комітів в репозиторії коду. Тести допомагають уникати очевидних людських помилок. Функціональні тести підвищують загальний рівень якості коду.

Також тести служать документацією для складного коду і часто допомагають програмісту розібратись у чужому коді. Крім того, при рефакторингу існуючого коду можна легко виявити і відлагодити поламаний раніше працюючий код. Функціональні тести також дозволяють швидко протестувати програму на різних платформах і версіях середовища.

Після усіх вищеперечислених плюсів функціональних автоматизованих тестів складається враження, що їх варто писати завжди. Проте, після того, як я почав працювати напряму із клієнтами та обговорювати бізнес цілі проекту, а не лише його технічну сторону, почав розуміти, що тести – це не лише користь, але й додатковий бюджет для замовника.

*Автоматизовані тести* – це завжди код, на який треба витратити додатковий час. Часом навіть 30-50% від загального часу проекту, в залежності від його складності та кількості задіяних зовнішніх компонент та сервісів. Відповідно, на цей процент зростає вартість проекту. Крім того, тести – це річ, яка потребує часу на подальшу підтримку та оновлення відповідно до змін специфікацій проекту.

Так! *Тести* – це інвестиція в майбутнє проекту. З цим не можна не погодитись. Саме з цього місця і постають наступні запитання:

* Якого роду проект? Разовий чи довготерміновий?
* Яке можливе майбутнє проекту? Невизначене і з’ясується по ходу справи? А чи він точно буде затребуваним у найближчі кілька років?

В залежності від відповідей на вищенаведені запитання потреба в тестах буде різною. Ось кілька правил, якими я керуюсь, коли рекомендую або ж, навпаки, не раджу працювати над тестами в проекті:

* якщо проект невеликий і разовий (тобто подальші розробки будуть мінімальними і не буде глобального рефакторингу коду), тоді можна обійтись без тестів; в такому проекті вони будуть не інвестицією, а витратою часу та коштів;
* якщо великий і планується, довготривалий проект пишеться з нуля та при цьому вся бізнес модель (ідея) не протестована, тоді проект, швидше за все, потребуватиме частих та кардинальних експериментів у функціоналі; тоді тести не варто писати спочатку; лише на фазі, коли бізнес модель запрацювала і вже є достатньо даних, щоб зафіксувати основні специфікації функціоналу та покрити їх тестами; в протилежному випадку, якщо одразу почати писати тести в такому проекті, тести можуть переписуватись з нуля (так само як і код самого проекту) по кілька разів;
* в попередньому випадку (великий і довготривалий проект), якщо проект має хороше фінансування (із запасом) і точно потребуватиме неодноразового рефакторингу, варто одразу писати тести, але лише тести чорного ящика; в такому випадку, навіть повністю змінивши код проекту, розробник матиме швидший доказ того, що кінцевий функціонал не постраждав;
* якщо проект реалізується вже під тестовану бізнес ідею, яка працює, і є чітко визначені потреби, що не змінюватимуться, тоді тести також можна сміливо писати одразу; в майбутньому вони лише повертатимуть дивіденди у вигляді зекономлених годин на баг-фікси.

Ну і взагалі, ваша справа, як технічного спеціаліста, надати вищій ланці проекту усі плюси та мінуси, “за” і “проти” тестів саме в рамках даного проекту. І вже або ви разом, або вища ланка самостійно вирішує доцільність тестів. Головне, щоб в кінці кінців тести виявились у колонці інвестицій, а не в колонці витрат.

\*\*\*

Тепер по періоду написання тестів в житті проекту. Є кілька варіантів, коли варто починати роботу над тестами:

* тести пишемо ще до написання самого коду програми; так зване TDD (Test Driven Development, детальне пояснення у наступній секції);
* тести пишемо паралельно до написання нового коду;
* тести пишемо наприкінці проекту, вже коли узгоджені кінцеві специфікації, які могли змінюватись в процесі розробки;
* є ще такий тип тесту, як тест на знайдену помилку; відповідно, його додають, якщо в існуючому функціоналі програми на кінцевому середовищі (сервері) знайдено помилку.

Загалом, краще наповнювати проект тестами раніше, ніж пізніше. Але, знову ж таки, все залежить від бізнес цілей проекту і від запитань, наведених вище про доцільність написання тестів.

Варіант написання тестів до написання самого коду ми розглянемо пізніше. Він має свої плюси та мінуси. Він не особливо відрізняється від написання тестів паралельно до написання коду з точки зору бюджету проекту.

А от написання тестів лише після тесту бізнес ідеї і затвердження остаточного функціоналу проекту в першому релізі може мати свій великий плюс: економія часу на кількаразове переписування існуючих тестів.

Підсумовуємо: в проекті із точними кінцевими специфікаціями можна одразу приступати до створення тестів. В іншому випадку, краще зекономити трохи часу і коштів та додати тести після того, як функціонал програми “устаканиться”.

**Якими повинні бути функціональні тести?**

При написанні тестів варто пам’ятати кілька важливих моментів, які допоможуть вам зберегти трохи часу та підвищити їхню якість:

* краще, коли один тест тестує лише одну річ (функцію, клас, блок коду);
* 100% покриття коду тестами не гарантує відсутності помилок;
* з попереднього пункту випливає, що більше число тестів не означає кращий результат; краще написати кілька тестів, що покриють критичний функціонал програми;
* також з певного моменту збільшення кількості тестів стає малоефективним; тому варто в проекті додати лише найбільш необхідні тести; 20% тестів дозволять виявити 80% помилок;
* таким чином, не варто намагатись покривати увесь код і кожний умовний оператор тестами; краще використайте цей час на розробку нового функціоналу.

\*\*\*

Наприкінці теоретичної частини даної глави розглянемо ще кілька популярних англомовних абревіатур і слів у світі тестування:

**TDD, Continuous Integration, Code Test Coverage**

В цій теоретичній секції розберемось із трьома термінами, які наведені вище у заголовку. Останнім із них (Test Coverage) ми навіть скористаємось наприкінці даної глави.

***TDD***

*Test Driven Development*  ([TDD](http://bit.ly/vptddtest)) – розробка програмного забезпечення через написання тестів.

При такому підході програміст спочатку пише тест на не існуючий функціонал чи зміну до існуючого функціоналу. Спочатку тест має провалюватись. Далі програміст додає відповідний код і після цього написані ним тести проходять успішно.

Переваги TDD підходу:

* в результаті TDD підходу отримуємо дійсно якісні тести, а не просто галочку про їх наявність;
* TDD дозволяє краще продумати архітектуру програми ще на етапі написання тестів.

Недоліки TDD підходу:

* для початківця такий підхід забере більше часу, ніж написання тестів на вже існуючий код;
* при зміні специфікацій проекту прийдеться витратити додатковий час на оновлення тестів також.

TDD підхід можна і варто використовувати у всіх проектах, в яких було вирішено писати код одразу з тестами. Даний підхід підвищить якість коду проекту і зменшить кількість помилок. Звісно, в деяких випадках TDD може забирати ще більше часу, ніж написання тестів після написання коду. Тут, знову ж таки, варто обирати підхід відповідно до бізнес-цілей проекту, його бюджету, часових рамок, а також рівня компетентності програмістів.

***Continuous Integration***

*Continuous Integration* (з англ. [безперервна інтеграція](http://bit.ly/vpcinteg)) – це практика розробки програмного забезпечення, при якій проводяться регулярні збірки коду проекту з метою виявлення помилок інтеграції.

Дана практика полягає в автоматизації наступних процесів:

* отримання коду проекту з репозиторія;
* збірка проекту на окремому сервері, в середовищі максимально наближеному до кінцевого;
* запуск тестів;
* розгортання готового проекту;
* та підготовка звітів у тому чи іншому вигляді.

Вище описана процедура може виконуватись, наприклад, раз на добу. Або при кожному пуші (коміті) розробника на віддалений сервер. Таким чином, можна легко побачити, що після коміту одного із членів розробницької команди тести програми поламались, швидко виявити дану проблему та виправити.

Тобто можливість постійно моніторити стан системи на наявність помилок і є основною задачею практики Безперервної Інтеграції (Continuous Integration).

На практиці, щоб реалізувати даний підхід, використовують один із існуючих інструментів. Ось невеликий список одних із найпопулярніших опенсорсних систем Безперервної Інтеграції: [Hudson](http://hudson-ci.org/), [Jenkins](https://jenkins-ci.org/), [CruiseControl](https://en.wikipedia.org/wiki/CruiseControl), [CruiseControl.NET](http://www.cruisecontrolnet.org/).

Ну і, звичайно, код проекту обов’язково повинен знаходитись в репозиторії коду, щоб була можливість реалізувати дану практику.

***Code Test Coverage***

*Code Test Coverage* – [рівень покриття коду](http://bit.ly/vpcoverage) програми тестами. Код вважається *покритим*, якщо він був запущеним протягом прогонки тестів.

Покриття коду тестами прийнято вважати показником якості коду проекту. Хоча це не можна вважати однозначною характеристикою, адже 100% покриття коду тестами ще не означає, що код не містить помилок. До того ж кількість тестів ще не означає, що самі тести є якісні.

Відповідно, не варто намагатись покривати ваш код на всі 100% тестами. Зазвичай, достатньо написати кілька тестів, що покриватимуть лише найбільш критичні місця програми.