Многие разработчики говорят о юнит-тестах, но не всегда понятно, что они имеют в виду. Иногда неясно, чем они отличаются от других видов тестов, а порой совершенно непонятно их назначение.

Доказательство корректности кода

Автоматические тесты дают уверенность, что ваша программа работает как задумано. Такие тесты можно запускать многократно. Успешное выполнение тестов покажет разработчику, что его изменения не сломали ничего, что ломать не планировалось.

Провалившийся тест позволит обнаружить, что в коде сделаны изменения, которые меняют или ломают его поведение. Исследование ошибки, которую выдает провалившийся тест, и сравнение ожидаемого результата с полученным даст возможность понять, где возникла ошибка, будь она в коде или в требованиях.

Отличие от других видов тестов

Все вышесказанное справедливо для любых тестов. Там даже не упомянуты юнит-тесты как таковые. Итак, в чем же их отличие?

Ответ кроется в названии: «**юнит**» означает, что мы тестируем не всю систему в целом, а небольшие ее части. Мы проводим тестирование с **высокой гранулярностью**.

Это основное отличие юнит-тестов от системных, когда тестированию подвергается вся система или подсистема, и от интеграционных, которые проверяют взаимодействие между модулями.

Основное преимущество независимого тестирования маленького участка кода состоит в том, что если тест провалится, ошибку будет легко обнаружить и исправить.

И все-таки, что такое юнит?

Часто встречается мнение, что юнит — это класс. Однако это не всегда верно. Например, в C++, где классы не обязательны.

«Юнит» можно определить как маленький, связный участок кода. Это вполне согласуется с основным принципом разработки и часто юнит — это некий класс. Но это также может быть набор функций или несколько маленьких классов, если весь функционал невозможно разместить в одном.

> Юнит — это маленький самодостаточный участок кода, реализующий определенное поведение, который часто (но не всегда) является классом.

Это значит, что если вы жестко запрограммируете зависимости от других классов в тестируемый, ошибку, которая вызвала падение теста, будет сложно локализовать, и высокая гранулярность, которая необходима для юнит-тестов, будет потеряна.

> Отсутствие сцепления необходимо для написания юнит-тестов.

Другие применения юнит-тестов

Кроме доказательства корректности, у юнит-тестов есть еще несколько применений.

Тесты как документация

Юнит-тесты могут служить в качестве документации к коду. Грамотный набор тестов, который покрывает возможные способы использования, ограничения и потенциальные ошибки, ничуть не хуже специально написанных примеров, и, кроме того, его можно скомпилировать и убедиться в корректности реализации.

Я думаю, что если тесты легко использовать (а их должно быть легко использовать), то другой документации (к примеру, комментариев doxygen) не требуется.

Тем не менее, в этом обсуждении после поста про комментарии видно, что не все разделяют мое мнение на этот счет.

Разработка через тестирование

При разработке через тестирование *(test-driven development, TDD)* вы сначала пишете тесты, которые проверяют поведение вашего кода. При запуске они, конечно, провалятся (или даже не скомпилируются), поэтому ваша задача — написать код, который проходит эти тесты.

Основа философии разработки через тестирование — вы пишете только тот код, который нужен для прохождения тестов, ничего лишнего. Когда все тесты проходят, код нужно отрефакторить и почистить, а затем приступить к следующему участку.

Об этой технике разработки можно много дискутировать, но ее неоспоримое преимущество в том, что TDD не бывает без тестов, а значит она предостерегает нас от написания жестко сцепленного кода.

И, поскольку TDD предполагает, что нет участков кода, не покрытых тестами, все поведение написанного кода будет документировано.

Возможность лучше разобраться в коде

Когда вы разбираетесь в плохо документированном сложном старом коде, попробуйте написать для него тесты. Это может быть непросто, но достаточно полезно, так как:

- они позволят вам убедиться, что вы правильно понимаете, как работает код;
- они будут служить документацией для тех, кто будет читать код после вас;
- если вы планируете рефакторинг, тесты помогут вам убедиться в корректности изменений.