Основные положения тестирования

Что такое тестирование

Сначала попробуем понять, чем тестирование НЕ является.

Тестирование не разработка,

даже если тестировщики умеют программировать, в том числе и тесты (автоматизация тестирование = программирование), могут разрабатывать какие-то вспомогательные программы (для себя).

Тем не менее, тестирование — это не деятельность по разработке программного обеспечения.

Тестирование не анализ,

и не деятельность по сбору и анализу требований.

Хотя, в процессе тестирования иногда приходится уточнять требования, а иногда приходится их анализировать. Но эта деятельность не основная, скорее, это приходится делать просто по необходимости.

Тестирование не управление,

несмотря на то, что во многих организациях есть такая роль, как «тест-менеджер». Конечно же, тестировщиками надо управлять. Но само по себе тестирование управлением не является.

Тестирование не техписательство,

однако тестировщикам приходится документировать свои тесты и свою работу.

Тестирование нельзя считать ни одной из этих деятельностей просто потому, что в процессе разработки (или анализа требований, или написания документации для своих тестов) всю эту работу тестировщики делают **для себя**, а не для кого-то другого.

Деятельность значима только тогда, когда она востребована, то есть тестировщики должны что-то производить «на экспорт». Что они делают «на экспорт»?

Можно считать, что отчуждаемыми результатами работы тестировщиков являются дефекты, описания дефектов, или отчеты о тестировании.

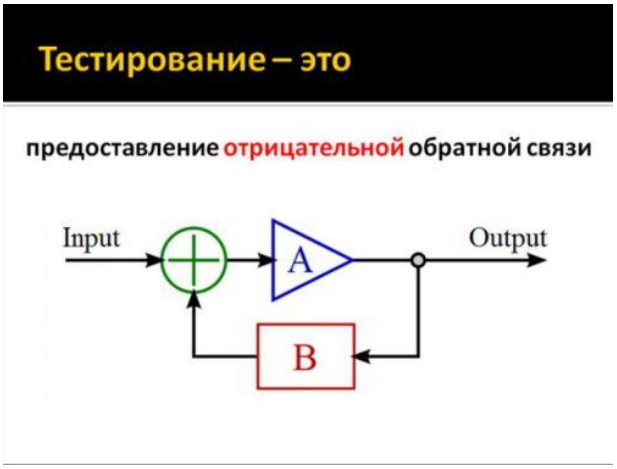
Это наивно.

Хотя частично это правда.

Но это не вся правда.

Главная деятельность тестировщиков

заключается в том, что они предоставляют участникам проекта по разработке программного обеспечения отрицательную обратную связь о качестве программного продукта.



«Отрицательная обратная связь» не несет какой-то негативный оттенок, и не означает, что тестировщики делают что-то плохое, или что они делают что-то плохо. Это просто технический термин, который обозначает достаточно простую вещь.

Но эта вещь очень значимая, и, наверное, единственная наиболее значимая составляющая деятельности тестировщиков.

Существует наука — «теория систем». В ней определяется такое понятие как «обратная связь»:

«Обратная связь» это некоторые данные, которые с выхода попадают обратно на вход, или какая-то часть данных, которые с выхода попадают обратно на вход. Эта обратная связь может быть положительной и отрицательной.

Считается, что положительная обратная связь прибавляется к входному сигналу, то есть, она усиливает входной сигнал. А отрицательная обратная связь входной сигнал ослабляет.

И та, и другая разновидности обратой связи равноценно важны.

Отрицательная обратная связь стабилизирует систему благодаря тому, что она ослабляет входной сигнал.

Положительная обратная связь, усиливая входной сигнал, приводит к тому, что он становится все сильнее и сильнее, и может возрастать неограниченно. Неограниченное возрастание входного сигнала может привести к разрушению системы.

Постоянное ослабление входного сигнала может привести к тому, что система просто затухает и становится стабильной, но входной сигнал становится равным нулю.

Для того, чтобы система приносила какую-то пользу, у нее должна быть одновременно и положительная обратная связь, которая усиливает входной сигнал, и отрицательная обратная связь, которая регулирует его мощность и не дает ему стать слишком сильным, иначе система разрушится.

В разработке программных систем положительной обратной связью, конечно же, является какая-то информация, которую мы получаем от конечных пользователей. Это запросы на какую-то новую функциональность, это увеличение объема продаж (если мы выпускаем качественный продукт).

Отрицательная обратная связь тоже может поступать от конечных пользователей в виде каких-то негативных отзывов. Либо она может поступать от тестировщиков.

Чем раньше предоставляется отрицательная обратная связь, тем более слабый сигнал ей еще нужно модифицировать, и поэтому тем меньше энергии необходимо для модификации этого сигнала. Именно поэтому тестировать нужно начинать как можно раньше, на самых ранних стадиях проекта, и предоставлять эту обратную связь и на этапе проектирования, и еще, может быть, раньше, еще на этапе сбора и анализа требований.

Синонимы термина «тестирование»

С точки зрения того, что тестирование — это предоставление отрицательной обратной связи, всемирно известная аббревиатура QA (англ. Quality Assurance — Обеспечение качества) синонимом термина «тестирование» уж совершенно точно НЕ является.

Нельзя считать обеспечением качества предоставление отрицательной обратной связи. Обеспечение — это некоторые позитивные меры. Подразумевается, что в этом случае мы именно обеспечиваем качество, своевременно предпринимаем какие-то меры для того, чтобы качество разработки ПО повысилось.

А вот «контроль качества» — Quality Control, можно считать в широком смысле синонимом для термина «тестирование», потому что контроль качества это и есть предоставление обратной связи в самых разных ее разновидностях, на самых разных этапах программного проекта.

Иногда тестирование подразумевается как некоторая отдельная форма контроля качества.

Путаница приходит из истории развития тестирования. В разное время под термином «тестирование» подразумевались различные действия.

1980

Процесс выполнения программы с намерением найти ошибки.
[Г.Майерс. Надежность программного обеспечения. М:Мир, 1980]

1987

Процесс наблюдения за выполнением программы в специальных условиях и вынесения на этой основе оценки каких-либо ее аспектов.
[ANSI/IEEE standard 610.12-1990: Glossary of SE Terminology. NY:IEEE, 1987]

1990

Это не действие. Это интеллектуальная дисциплина, имеющая целью получение надежного программного обеспечения без излишних усилий на его проверку.
[B. Beizer. Software Testing Techniques, Second Edition. NY:van Nostrand Reinhold, 1990]

1999

Техническое исследование программы для получения информации о ее качестве с точки зрения определенного круга заинтересованных лиц.

[C. Kaner, 1999]

2004

Проверка соответствия между реальным поведением программы и ее ожидаемым поведением на конечном наборе тестов, выбранном определенным образом.

[IEEE Guide to Software Engineering Body of Knowledge, SWEBOK, 2004]

Может создаться впечатление, что с 1980-го по 2004 теория тестирования эволюционировала настолько, что суть обсуждаемой темы в каком-то смысле даже изменилась.

Но разгадка в другом. Процитированные определения можно разделить на 2 больших класса: внешние и внутренние.

Внешние определения

Определения, которые в разное время дали Майерс, Бейзер, Канер, описывают тестирование как раз с точки зрения его ВНЕШНЕЙ значимости. То есть, с их точки зрения, тестирование — это деятельность, которая предназначена ДЛЯ чего-то, а не состоит из чего-то. Все три этих определения можно обобщить как предоставление отрицательной обратной связи.

Внутренние определения

Это определения, которые приведены в стандарт терминологии, используемой в программной инженерии, например, в стандарт де-факто, который называется SWEBOK.

Такие определения конструктивно объясняют, ЧТО представляет из себя деятельность по тестированию, но не дают ни малейшего представления о том, ДЛЯ ЧЕГО нужно тестирование, для чего потом будут использоваться все полученные результаты проверки соответствия между реальным поведением программы и ее ожидаемым поведением.

Итак,

тестирование — это

- проверка соответствия программы требованиям,
- осуществляемая путем наблюдения за ее работой
- в специальных, искусственно созданных ситуациях, выбранных определенным образом.

Отсюда и далее будем считать это рабочим определением «тестирования».

Общая схема тестирования примерно следующая:

1. Тестировщик на входе получает программу и/или требования.
2. Он с ними что-то делает, наблюдает за работой программы в определенных, искуственно созданных им ситуациях.
3. На выходе он получает информацию о соответствиях и несоответствиях.
4. Далее эта информация используется для того, чтобы улучшить уже существующую программу. Либо для того, чтобы изменить требования к еще только разрабатываемой программе.

Это весьма близко к определению, данному в SWEBOK, хотя есть несколько отличий. Например, в нашем определении нет слова «тест».

Определение тестирования по SWEBOK

звучит следующим образом:

Тестирование – это проверка соответствия между реальным поведением программы и ее ожидаемым поведением на конечном наборе тестов, выбранных определенных образом.

А мы с вами говорили о некоторых специальных искусственно созданных **ситуациях**, выбранных определенным образом. Вот эти специальные, искусственно созданные ситуации, и есть ТЕСТЫ. Чуть позже мы это сформулируем еще более точно в виде определения термина «тест», а пока пойдем дальше.

Определение по SWEBOK плохо по-следующей причине: оно говорит нам, что набор тестов должен быть КОНЕЧНЫМ. А он конечным будет всего лишь по определению. Он не может быть бесконечным, потому что у нас есть только ограниченное количество времени, ограниченные ресурсы, объем памяти компьютера конечен, поэтому это слово «конечный» ничего нам не дает осмысленного, правильно использовать термин «ограниченный», на ограниченный набор тестов. Мы его сами некоторым образом ограничиваем.

И это как раз очень важно, потому что это и есть то, чем занимается разработчик тестов. Он из огромного потенциально бесконечного набора тестов выбирает некоторый ограниченный набор.

Что такое тест

- Это специальная, искусственно созданная ситуация, выбранная определенным образом,
- и описание того, какие наблюдения за работой программы нужно сделать
- для проверки ее соответствия некоторому требованию.

Не нужно считать, что ситуация – это нечто одномоментное. Тест может быть достаточно длинным, например, при тестировании производительности вот эта искусственно созданная ситуация это может быть продолжающаяся в течение достаточно продолжительного времени нагрузка на систему. А наблюдения, которые нужно при этом делать, это набор различных графиков или метрик, которые мы измеряем в процессе выполнения этого теста.

Ну и таким образом мы можем заключить, что тестировщик делает в процессе тестирования две вещи.

1.Во-первых, он управляет выполнением программы и создает эти самые искусственные ситуации, в которых мы собираемся проверять поведение программы.

2.И, во-вторых, он наблюдает за поведением программы и сравнивает то, что он видит с тем, что ожидается.

Разумеется, иногда мы отклоняемся от этого определения, например, при тестировании удобства использования тестировщик может наблюдать не только за

поведением программы, но и за поведением специального человека, испытуемого, которому дается некоторое задание. Он выполняет задание, а мы смотрим, справляется он с ним или не справляется, за какое время он справляется.

Если тестировщик автоматизирует тесты, то он не сам наблюдает за поведением программы — он делегирует эту задачу специальному инструменту или специальной программе, которую он сам написал. Именно она наблюдает, она сравнивает наблюдаемое поведение с ожидаемым, а тестировщику выдает только некоторый конечный результат — совпадает ли наблюдаемое поведение с ожидаемым, или не совпадает.

Основные задачи тестирования

Еще несколько терминов, которые связаны с упомянутыми двумя задачами, которыми занимается тестировщик, это стимулы, реакции и оракул.

Стимулы – это данные, которые подаются на вход программе.

Реакции — это то, что получается на выходе.

Оракул — это способ проверки наблюдаемого результата, совпадает он с некоторыми ожиданиями или не совпадает.

Программа представляет собой механизм по переработке информации. На вход поступает информация в каком-то одном виде, на выходе информация в некотором другом виде. При этом входов и выходов у программы может быть много, они могут быть разными, то есть у программы может быть несколько разных интерфейсов, и эти интерфейсы могут иметь разные виды:

- Пользовательский интерфейс (UI)
- Программный интерфейс (API)
- Сетевой протокол
- Файловая система
- Состояние окружения
- События

Наиболее распространенные интерфейсы это

- графический,
- текстовый,
- консольный,
- и речевой.

Через пользовательский интерфейс компьютер взаимодействует с человеком, с пользователем.

Через программный интерфейс программы взаимодействуют друг с другом (человек тут не нужен).

Ну, и можно выделить такие еще разновидности как сетевой протокол, чаще всего тоже для взаимодействия программ друг с другом, но через сеть, а не непосредственно, как это происходит через программный интерфейс.

Это файловая система, программы могут писать данные на диск и читать данные с диска.

Это состояние окружения, которое могут программы модифицировать и, соответственно, тоже читать.

Это события, в частности, таймер. То есть некоторые механизмы отслеживания времени.

Используя все эти интерфейсы, тестировщик каким-то образом создает искусственные ситуации, и проверяет в этих ситуациях как программа себя ведет. **Вот это и есть тестирование**.

Виды тестирования

Теперь мы поговорим о разных видах тестирования, а также о разных видах контроля качества, а также о том, что такое качество вообще.

Говоря о качестве компьютерных программ, я придерживаюсь трактовки, которая изложена в стандарте ISO 9126. Этот стандарт выделяет 6 аспектов качества, у каждого из которых еще выделяется некоторое количество подхарактеристик.

Вот эти 6 аспектов качества верхнего уровня:

- функциональность,
- надежность,
- практичность,
- эффективность,
- сопровождаемость
- переносимость.

Функциональность

это, наверное, основной аспект качества. И наиболее важной его подхарактеристикой является пригодность к использованию (suitability). То есть программа должна делать то, что она должна, то, для чего она предназначена. Это во-первых.

Во-вторых, она должна делать это правильно (accuracy). Она должна вычислять с определенной точностью, она должна правильно конвертировать данные и так далее.

В-третьих, программа должна обладать способностью к взаимодействию с другими программами (interoperability), с операционной системой, с другими версиями той же самой программы, в частности, она должна поддерживать какие-то стандарты (compliance), удовлетворять определенным правилам.

Есть одна очень интересная особенность этого стандарта, защищенность или безопасность (security), в этом стандарте не выделяется как отдельный аспект верхнего уровня, а считается подхарактеристикой подфункциональности.

То есть мало того, что программа должна делать то, что она обязана, она должна еще НЕ делать ничего другого. Она не должна разрушать ваши данные, она не должна мешать работать другим программам, она не должна предоставлять доступ к данным тем, кому этот доступ не разрешен.

Надежность

это второй аспект качества, он достаточно часто неявно включается в какие-то другие аспекты, либо в функциональность, либо в производительность. Но в этом стандарте он выделяется отдельно.

К нему относятся такие подхарактеристики, как зрелость (maturity), которая является обратной величиной к частоте отказов, и устойчивость к отказам (fault tolerance), то есть способность системы не реагировать на какие-то внутренние проблемы. В том числе сюда относится транзакционная целостность и способность к восстановлению работоспособности при отказах (recoverability). То есть если у вас сервер упал, то он должен самостоятельно восстановиться, вернуть все нужные данные, ничего не потерять, и продолжить работу.

Практичность

Это понятность программы (understandability) то есть пользователь должен понять, как воспользоваться ею для достижения своих целей.

Это удобство обучения (learnability) или изучения. В частности, у программы, наверное, должна быть какая-то документация.

Это работоспособность (operability) или управляемость, то есть пользователь должен иметь возможность управлять поведением программы, она должна реагировать на его действия.

И привлекательность (attractiveness), эстетическая привлекательность, что тоже, конечно же, немаловажно.

Эффективность

Она же – производительность, четвертый аспект качества.

Сюда относятся временные характеристики (time behaviour), время отклика, скорость работы программы, скорость обработки определенных данных и так далее.

И использование ресурсов (resource utilisation), использование дисковых ресурсов, использование ресурсов процессора, использование оперативной памяти, использование сетевых ресурсов.

Сопровождаемость

Этот аспект качества в большей степени не внешний, а внутренний.

Он важен не столько конечным пользователям, не столько потребителям программы, сколько самим разработчикам и ее тестировщикам.

Является единственным аспектом качества, который плохо совместим с тестированием, и является сопровождаемость.

Все его подхарактеристики тестированию практически не поддаются, и для них применяются какие-то другие способы контроля качества, как правило, аналитические. Статический анализ кода, код ревью, анализ документации и так далее — все это не решается средствами тестирования.

Подразумевает анализируемость кода (analyzability), изменяемость (changeability), то есть удобство внесения изменений в программный код, риск возникновения неожиданных эффектов после того, как мы эти изменения внесли (stability), а также контролируемость (testability), или же – удобство тестирования программы.

Переносимость (или мобильность)

Программа должна уметь работать в различных окружениях (adaptibility), она должна быть достаточно проста в установке (installability), она должна работать одновременно с другими программами и не мешать им (coexistence), и они должны не мешать ей, конечно же.

И, наконец, хорошая программа должна предоставлять возможность пользователю перейти с какого-то другого аналогичного программного обеспечения на использование этой программы – замена другого ПО данным (replaceability). То есть предоставить какие-то возможности по миграции, чаще всего по миграции данных.

Соответственно, если исходить из вот этой классификации, то для каждого аспекта качества можно выделить соответствующий вид тестирования.

И получаем разные виды тестирования:

- тестирование функциональности,
- тестирование надежности,
- тестирование эффективности,
- тестирование практичности,
- тестирование сопровождаемости,
- тестирование переносимости.

Можно даже выделять какие-то более детальные виды тестирования, если опускаться до уровня отдельных подхарактеристик качества. То есть можно говорить о тестировании защищенности, можно говорить о тестировании устойчивости к отказам, можно говорить о тестировании ее привлекательности…

Другие классификации видов тестирования

Чаще всего используется разбиение на три уровня, это

1. модульное тестирование,
2. интеграционное тестирование,
3. системное тестирование.

Под модульным тестированием обычно подразумевается тестирование на достаточно низком уровне, то есть тестирование отдельных операций, методов, функций.

Под системным тестированием подразумевается тестирование на уровне пользовательского интерфейса.

Иногда используются также некоторые другие термины, такие, как «компонентное тестирование», но я предпочитаю выделять именно эти три, по причине того, что технологическое разделение на модульное и системное тестирование не имеет большого смысла. На разных уровнях могут использоваться одни и те же инструменты, одни и те же техники. Разделение условно.

Если посмотреть на какое-нибудь типичное веб-приложение, то можно заметить, что оно представляет собой своеобразную матрешку.

Оно состоит из клиентской и серверной части, клиентская часть включает в себя помимо браузера некоторый набор java-скрипт-библиотек, каждая из них состоит из набора функций.

Серверная часть, в свою очередь, еще может быть разделена на несколько крупных кусков. Один выполняется на сервере приложений, другой выполняется на стороне базы данных, на сервере приложений имеется целый ряд библиотек. Некоторые из них разработаны самостоятельно, некоторые уже готовые используются.

Библиотеки состоят из классов, классы состоят из методов, на стороне баз данных тоже есть пакеты, состоящие из хранимых процедур.

Само по себе приложение тоже может являться частью какой-то более крупной информационной системы.

В этой матрешке мы должны понять, где, на каком уровне у нас должно находиться модульное тестирование, а на каком должно находиться системное тестирование. И найти еще место для интеграционного.

Глядя на эту матрешку мы можем понять, что разделение на системное и модульное тестирование является чисто условным.

То есть у нас система состоит из каких-то модулей. Модули в свою очередь состоят из других более мелких модулей. Эти мелкие модули состоят еще из более мелких, и так далее.

Между двумя ближайшими слоями в этой матрешке нет практически никакой разницы ни с точки зрения используемых инструментов, ни с точки зрения используемых техник тестирования.

Да, разумеется, есть большая разница между самым верхним уровнем и самым нижним уровнем, но два самых ближних друг к другу слоя отличаются очень мало.

Кроме того, практика показывает, что инструменты, которые позиционируются производителем как инструменты модульного тестирования, с равным успехом могут применяться и на уровне тестирования всего приложения в целом.

А инструменты, которые тестируют все приложение в целом на уровне пользовательского интерфейса иногда хотят заглядывать, например, в базу данных или вызывать там какую-то отдельную хранимую процедуру.

То есть разделение на системное и модульное тестирование вообще говоря чисто условное, если говорить с технической точки зрения.

Используются одни и те же инструменты, и это нормально, используются одни и те же техники, на каждом уровне можно говорить о тестировании различного вида.

Комбинируем:

То есть, можно говорить о модульном тестировании функциональности.

Можно говорить о системном тестировании функциональности.

Можно говорить о модульном тестировании, например, эффективности.

Можно говорить о системном тестировании эффективности.

Либо мы рассматриваем эффективность какого-то отдельно взятого алгоритма, либо мы рассматриваем эффективность всей системы в целом. То есть технологическое разделение на модульное и системное тестирование не имеет большого смысла. Потому что на разных уровнях могут использоваться одни и те же инструменты, одни и те же техники.

Наконец, при интеграционном тестировании мы проверяем, если в рамках какой-то системы модули взаимодействуют друг с другом корректно. То есть, мы фактически выполняем те же самые тесты, что и при системном тестировании, только еще дополнительно обращаем внимание на то, как именно модули взаимодействуют между собой. Выполняем некоторые дополнительные проверки. Это единственная разница.

Конечно же, эта разница может влиять на то, что мы выполним некоторые дополнительные тесты для проверки каких-то особенностей взаимодействия модулей, которые мы не выполнили бы, если бы смотрели только на всю систему в целом. Но об этом мы с вами поговорим чуть позже, когда речь пойдет о тестировании методом черного и методом белого ящика.

А пока давайте еще раз попытаемся понять разницу между системным и модульным тестированием. Поскольку такое разделение встречается достаточно часто, эта разница должна быть.

И разница эта проявляется тогда, когда мы выполняем не технологическую классификацию, а классификацию по целям тестирования.

Классификацию по целям удобно выполнять с использованием «магического квадрата», который был изначально придуман Брайаном Мариком и потом улучшен Эри Тенненом.

В этом магическом квадрате все виды тестирования располагаются по четырем квадрантам в зависимости от того, чему в этих тестах больше уделяется внимания.

По вертикали — чем выше располагается вид тестирования, тем больше внимания уделяется некоторым внешним проявлениям поведения программы, чем ниже он находится, тем больше мы внимания уделяем ее внутреннему технологическому устройству программы.

По горизонтали — чем левее находятся наши тесты, тем больше внимания мы уделяем их программированию, чем правее они находятся, тем больше внимания мы уделяем ручному тестированию и исследованию программы человеком.

В частности, в этот квадрат можно легко вписать такие термины как приемочное тестирование, Acceptance Testing, модульное тестирование именно в том понимании, в котором оно чаще всего употребляется в литературе. Это низкоуровневое тестирование с большой, с подавляющей долей программирования. То есть это все тесты программируются, полностью автоматически выполняются и внимание уделяется в первую очередь именно внутреннему устройству программы, именно ее технологическим особенностям.

В правом верхнем углу у нас окажутся ручные тесты, нацеленные на внешнее какое-то поведение программы, в частности, тестирование удобства использования, а в правом нижнем углу у нас, скорее всего, окажутся проверки разных нефункциональных свойств: производительности, защищенности и так далее.

Так вот, исходя из классификации по целям, модульное тестирование у нас оказывается в левом нижнем квадранте, а все остальные квадранты — это системное тестирование.

Тестирование методом черного и белого ящика

Наконец, третья широко распространенная классификация — разделение тестирования на два больших класса: тестирование методом черного ящика и тестирование методом белого ящика. Эта классификация связана с таким понятием как «полнота тестирования». Поэтому сначала мы поговорим именно о ней.

Полнота тестирования

Когда мы говорим о полноте тестирования, то это понятие достаточно близко к понятию полноты в музейном смысле или в смысле коллекционирования. Мы пытаемся

собрать некоторую полную коллекцию тестов, но это не означает, что мы собираемся собрать все-все тесты. Мы хотим собрать только некоторых характерных типовых представителей.

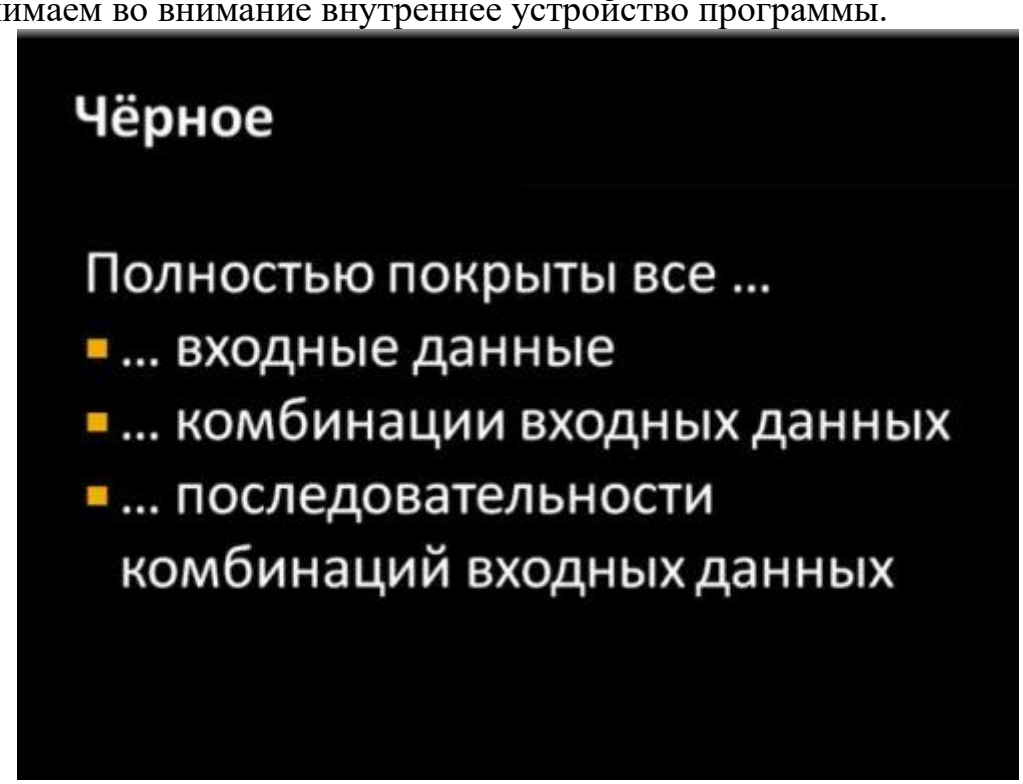
Например, если мы собираем полную коллекцию бабочек, то мы хотим, чтобы туда попали по одной бабочке каждого вида. Нам не нужно много бабочек одного и того же вида. И, конечно же, у нас нет цели переловить всех бабочек, которые живут на земле.

Когда мы собираем полное собрание сочинений, то у нас нет цели скупить все книги, полностью весь тираж, нам нужно чтобы в нашу коллекцию попало по одной книжке каждого вида.

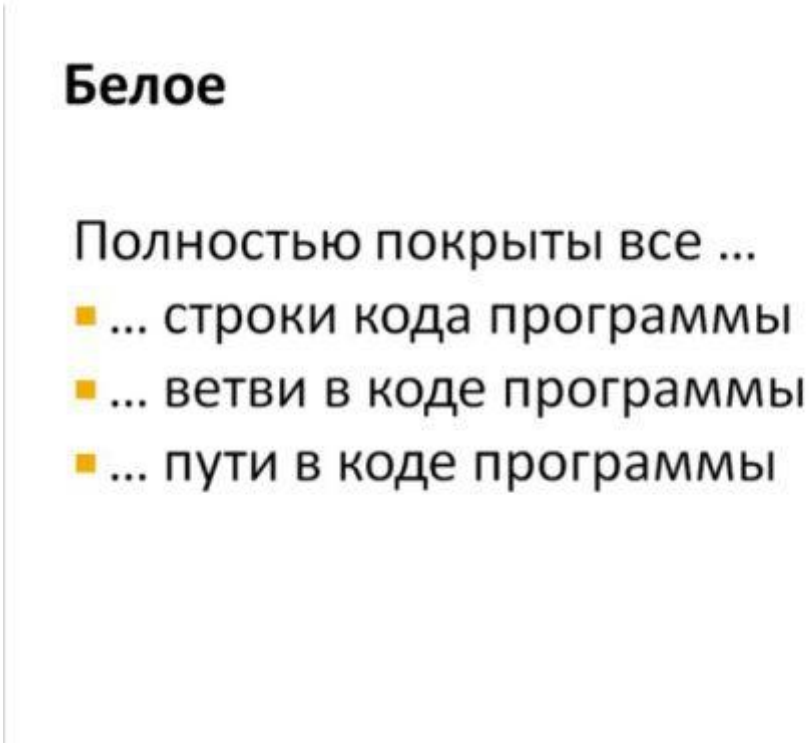
И вот когда мы собираем с вами эту коллекцию бабочек, мы можем, конечно же, ориентироваться только на раскраску крыльев. При этом мы можем не принимать во внимание внутреннее устройство бабочек. И может так оказаться, что у нас есть два разных вида бабочек, которые выглядят совершенно одинаково, но друг с другом не скрещиваются (то есть, с биологической точки зрения, представляют собой два разных вида). И тогда мы должны были бы их тоже включить в коллекцию, мы же пытаемся собрать все виды бабочек. Но одного рисунка нам недостаточно. Нам нужно еще принимать во внимание внутреннее устройство.

И вот это как раз и есть разница между тестированием методом черного ящика и тестированием методом белого ящика.

При тестировании методом черного ящика мы не видим, что внутри ящика, мы не принимаем во внимание внутреннее устройство программы.



При тестировании методом белого ящика, или правильнее говорить, наверное, «прозрачного ящика», мы смотрим, как программа устроена внутри, и эту информацию используем при выполнении и особенно при проектировании тестов.



Когда мы выполняем тестирование методом черного ящика, мы пытаемся полностью покрыть все входные данные или их комбинации, или даже последовательности комбинаций входных данных. Но этого может оказаться недостаточно для того, чтобы покрыть все строки кода программы, все ветви в коде программы или все пути в коде программы.

То есть, при тестировании методом белого ящика нам, как правило, нужно просто больше тестов, потому что у нас есть больше информации о том, какие разные варианты поведения программы могут быть. Мы принимаем во внимание и ее внешнее поведение, и ее внутреннее устройство. Коллекция тестов увеличивается.

С другой стороны, если мы снова посмотрим на наше приложение матрешку, мы можем увидеть, что это приложение состоит из каких-то частей, на картинке они отмечены голубеньким цветом, которые мы разработали сами, свои собственные библиотеки, свои собственные пакеты хранимых процедур в базе данных, а также в него включаются какие-то библиотеки, которые разработаны не нами.

Кроме того, наше приложение может взаимодействовать с какими-то веб-сервисами или другими приложениями. Может быть, даже с приложениями, которые работают совсем в другой организации. То есть мы к ним имеем очень ограниченный доступ. Наши приложения взаимодействуют с браузером, с операционной системой, то есть наша возможность контролировать, что там внутри очень сильно ограничена. Мы можем контролировать только программный код своих собственных библиотек, и можем стремиться достичь его покрытия тестами.

С такой точки зрения получается, что наш ящик, даже если мы к этому очень сильно будем стремиться, будет только чуть-чуть, слегка прозрачным. Где-то мы можем получить исходный код, где-то мы не можем получить исходный код, где-то мы можем вместо исходного кода получить достаточно подробные спецификации, где-то даже таких спецификаций нет.

Так что, правильнее говорить о разных степенях прозрачности, а может быть даже вообще о разных цветах ящика, а не о тестировании методом черного и методом белого ящика. Важно только то, какую информацию мы принимаем во внимание когда проектируем тесты. Либо мы используем информацию о внутреннем устройстве программы, либо не используем, вот и всё.