Конспект лекций по дисциплине
«Тестирование программного обеспечения»
студента группы И-33 Анистратенко Н. В,
преподаватель Строганов В. А.

Основные понятия

Целью любой разработки является создание качественного программного продукта. Понятие качество можно определить, как набор свойств, которыми долен обладать этот самый программный продукт. При этом различные участники процесса разработки могут по-разному понимать качество продукта. Так вот, тестирование — это наиболее традиционный способ определения показателей качества программного продукта, кроме того, тестирование является формальным процессом, поэтому позволяет делать однозначные выводы о техническом качестве программного продукта.

С технической точки зрения, тестирование — это выполнение программы на каком-то тестовом наборе данных и сравнение результатов, полученных с ожидаемыми. Если рассмотреть процесс разработки, это проектирование, кодирование и тестирование, то тестирование на самом деле довольно затратная по времени часть разработки, примерное распределение времени это 40%, 20% и 40%. В общем, тестирование занимает почти что половину времени. По стоимости, конечно, тестирование дешевле проектирования.

Формальное определение тестирования.

Во-первых, программу можно рассматривать как аналог формулы в обычной математике.

$$F = F_1 * F_2 * \ldots * Fn$$

Можно представить так, в том смысле, что программа F содержит n подпрограмм Fi.

Fi — это либо операторы, либо подпрограммы. Тогда формально задача тестирования заключается в доказательстве этого тождества. Есть два основных способа доказательства тождества, это формальных подход или доказательство.

Первый подход — в этом случае из исходных утверждений с помощью известных правил вывода нужно получить производные утверждения. Сам процесс доказательства тождества сводится к тому, что одна часть выражения с помощью формальных преобразований приводится к другой части выражения. Основное преимущество такого подхода, заключается в том, что мы избегаем подстановки большого количества переменных. Подход хороший, но в тестировании практически неприменим.

Второй подход — интерпретационный. В этом случае в формулы подставляются конкретные значения переменных и на всех наборах конкретных переменных проверяется работа тождества. Есть проблема, эти наборы могут быть очень большими, основная проблема этого метода. Именно этот подход доказательства тождеств лежит в основе процедур тестирования и отладки программного обеспечения.

Отладка программы (debug) — поиск, локализация и исправление ошибок в программе.

Это определение на самом деле не с потолка, в природе есть стандарты, есть международные. Так вот, IEEE Std. 610-12.1990, здесь содержится описание стандарта отладки. Под тестирование понимается процедура выявления фактов расхождения программы с требованиями технического задания. Если коротко, то тестирование — это обнаружение ошибок. Если программа не содержит синтаксических ошибок, то она может быть скомпилирована и выполнена, при выполнении она в любом случае вернет какой-нибудь результат. То есть в любом случае произойдёт отображение входных данных на выходные. Таким образом, при выполнении программы происходит доопределение не полностью определенной функции. Принимать решение о правильности работы программы можно сравнивания результаты вычисления функции с требованиями технического задания на эту функцию.

Тестирование бывает двух видов: *статическое* и *динамическое*. Статическое тестирование не требует запуска программы, оно заключается в анализе исходного кода программы. Второй подход — это динамическое тестирование, оно предполагает запуска и выполнение программы, это может быть более привычная нам процедура тестирования.

Организация тестирования.

Опять малость математики. Тестирование выполняется на заданном заранее множестве входных данных, множестве X. И на множестве ожидаемых выходных данных Y. Эти два множества задают график желаемой функции, которая описывает правильную работы программы. В результате тестирования мы получаем некоторое множество выходных данных Yb, полученное из исходного множества X. Поэтому процедура тестирования заключается в проверке принадлежит ли каждая точка (x_i, y_bi) графику (X, Y). Может быть обнаружено, что некоторая пара не принадлежит желаемому множеству, однако не может быть получено каких-либо данных о причинах этого несоответствия. Таким образом, тестирование определяет факт наличия ошибки, но не решает вопрос локализации ошибки, это уже задача отладки. После того как выявлено несоответствие, запускается процедура исправления ошибки. Процедура исправления ошибки предполагает во в первых анализ промежуточных результатов, которые привели к этой ошибке. Основные подходы которые здесь применяются, это выполнение программы в уме, либо пошаговая отладка, либо запись

промежуточных результатов. Результатом этой процедуры должна стать локализация этой ошибки, дабы было ясно что исправлять.

В общем, тестирование включат в себя три основных фазы: это создание тестового набора входных данных, такой набор может создаваться либо вручную, либо генерироваться автоматически по определенным правилам. Следующая фаза заключается в прогоне программы на этих тестовых данных, программный инструмент с помощью которого выполняется этот тап, называется тестовый драйвер, либо тестовый монитор, в результате этой фазы получается протокол тестирования, либо же как он ещё называется test log. Третья фаза — это оценка результатов тестирования, на этом этапе анализируются протоколы тестовых данных, полученные ранее протоколы, принимается решение тестировать дальше, либо хватит.

Основная проблема тестирования заключается в задаче определения минимально достаточного набора тестов для полной проверки работоспособности программы. Тестировать программу на всех доступных входных данных невозможно. Невозможно тестирование всех веток алгоритма программы. Отсюда следствие, что необходимо определить минимальный набор тестов, пользуясь часто некоторыми интуитивными соображениями. Из этого возникает задача оценки качества тестов, построение критерия тестов.

Требование к идеальному критерию тестирования.

Во-первых, критерий должен быть достаточен, то есть должен показывать, что некоторое конечное количество множеств достаточно для тестирования данной программы. Далее, критерий должен быть полным, то есть, если в программе есть ошибка, то должен найтись хотя бы один тест, который обнаруживает эту ошибку. Следующее требование, критерий должен быть надежным, то есть любые два множества тестов, которые удовлетворяют этому критерию, должны одновременно либо обнаружить ошибку в программе, либо нет. И ещё одно требование, критерий должен быть легко проверяемым. Идеальный критерий для нетривиальных программ невозможен.

Классы критериев тестирования.

1. **Структурные критерии** — их особенность в том, что они учитывают информацию о внутренней структуре программы, так называемым метод белого ящика.
2. **Функциональные критерии** — основываются на техническом задании, на спецификации функциональных требований программного продукта. Основная идея заключается в том, чтобы функция, описанная в техническом задании тестировалась хотя бы раз.
3. **Стохастические критерии** — основная их идея в том, что если входных данных много, то легче исследовать распределение выходных данных и сравнивать его с ожидаемым.

4. **Мутационные критерии** — идея в том, что мы пишем сразу относительно работающую программу, основные проблемы и основные ошибки — это мелкие опечатки. Как работает тест — в программу вводятся специальные изменения, так называемые мутации.

Структурные критерии тестирования.

Структурные критерии предполагают модель программы в виде белого либо прозрачного ящика. Предполагается, что при составлении тестов известен весь код программы, либо же её структура в виде графа потоков управления.

1. Критерий тестирования команд, тест предполагает выполнение каждой команды хотя бы один раз.
2. Критерий тестирования ветвей, тест удовлетворяющий этому критерию должен обеспечить прохождение каждой ветви графа хотя бы один раз.
3. Критерий тестирования путей, предполагается что тест этого критерия должен обеспечить прохождения каждого пути этого графа не менее одного раза, если программа содержит циклы, то число итераций ограничивается константой.

Функциональные критерии тестирования

Функциональный критерий тестирования предполагает проверку выполнения каждого пункта технического задания, то есть правильность работы каждой функции, которые перечислены в задании. При функциональном тестировании применяется принцип чёрного ящика. Основная проблема, связанная с использованием функциональных критерием — это трудоемкость (большие затраты времени), поскольку в техническом задании много функций, а проверить нужно каждую.

- **Первый вариант**: тестирование пунктов спецификации, где каждый элемент спецификаций требований должен быть протестирован не менее одного раза, поскольку спецификация может содержать очень много функций, такой подход очень трудоёмок.
- **Второй вариант**: тестирование классов входных данных, основная идея в том, чтобы разделить всё множество входных данных на подмножество, обрабатываемое программой одинаково, тогда для тестирования достаточно одного теста для каждого класса входных данных (для каждой области эквивалентности), при использовании этого подхода возможны следующие трудности — могут быть заданы довольно сложные ограничения на входные данные; при работе программы могут возникать ошибочные ситуации, которые тоже нужно обрабатывать.
- **Третий вариант**: тестирование правил — если входные и выходные данные описываются набором правил некоторой грамматики, то набор тестов должен обеспечить проверку каждого правила.

- **Четвёртый вариант**: тестирование классов выходных данных.
- **Пятый вариант**: тестирование функций. Каждая функция должна выполниться не менее одного раза. Самый полезный способ. Ограничения: проблемы с тестовым покрытием некоторых структурных и поведенческих свойств программы, которые не локализованы в рамках одной функции.
- **Шестой вариант:** комбинированные критерии. В этом случае строится набор тестов, где проверяются все входные данные и все элементы спецификации.

Стохастические критерии тестирования

При тестировании сложных программ может возникнуть ситуация, когда множество входных данных слишком велико. О переборе всех возможных решений речи быть не может. Данные, которые нельзя разбить на классы эквивалентности. В этом случае применяется следующий подход — последовательность входных данных генерируется случайным образом. Для каждого случайно полученного икса (входной сигнал) мы каким-то детерминированным способом вычисляем правильное значение выходного сигнала. В результате мы получаем тестовый набор — множество входных данных и набор ожидаемых выходных данных. После того как получено тестовое множество, выполняется тестирование.

- **Первый вариант**: детерминированный контроль, для каждого элемента входного множества получаем выходное значение, далее просто сравниваем данные с полученными от программы.
- **Второй вариант:** стохастический контроль, в этом случае вычисляется функция вероятности для выходных значений игрик и проверяется её соответствие с заранее вычисленной функцией вероятности от правильных значений выходных данных. Этот вариант может использоваться тогда, когда известна только функция распределения для набора данных.

Критерии стохастического тестирования

- **Первый вариант**: статистические методы окончания тестирования. Применяются стохастические методы о совпадении гипотез распределения случайных величин. Сюда относится метод Стьюдента и метод x^2.
- **Второй критерий стохастического тестирования**: метод оценки скорости выявления ошибок. Этот метод основан на модели выявления скорости ошибок, согласно этой модели тестирование прекращается в том случае, если оценка интервала времени между текущей ошибкой и следующей слишком велика для фазы тестирования.

Мутационный критерий тестирования

Основывается на следующем предположении — профессиональные программисты почти всегда пишут правильные программы, это значит, что там нет логических и алгоритмических ошибок, ошибки лишь заключаются в мелких опечатках. Мутация — мелкие ошибки в программе. Мутанты — это программы, отличающиеся друг от друга наличием мутации, то есть мелких изменений. Основной принцип — в исходную программу П искусственно вносятся мутации, добавляются мелкие ошибки, получаются программы-мутанты П_1, П_2 … исходная программа П и программы-мутанты тестируются на одном и том же наборе тестов. Если на наборе тестов (икс, игрик) подтверждается правильность работы программы П и обнаруживаются все мутации в программах П_1, П_2 и так далее, то считается, что набор тестов соответствует мутационному критерию тестирования. Если же этот тест выявил не все мутации, то просто расширяем множество входных (тестовых) данных.

Модульное тестирование программного обеспечения.

Модульное тестирование — это тестирование отдельных модулей программы. Под модулем может пониматься либо класс, либо метод, либо отдельная функция — то есть отдельно компилируемая часть программы. Каждый модуль при этом тестируется изолировано, не учитываются никакие другие взаимодействия с другими модулями. Модульное тестирование выполняется по окончанию разработки каждого модуля. Применятся подход белого ящика. Поскольку каждый модуль взаимодействует с другими модулями, а этим модули на этапе тестирования могут быть ещё не готовы, поэтому модульное тестирование предполагает создание тестового окружения — создаются заглушки, которые имитируют работу других модулей.

На этапе модульного тестирования выявляются алгоритмические ошибки и ошибки кодирования (опечатки и тому подобное). Какие ошибки не выявляются при модульном тестировании: общие системные ошибки, связанные с проектированием, не выявляются ошибки, связанные с неверной трактовкой данных, не выявляются ошибки, связанные с не правильной реализацией интерфейса, а также дефекты, связанные с общесистемными показателями — расход ресурсов, скорость работы.

Основная задача модульного тестирования — убедиться в том, что функция возвращает ожидаемый результат. Модульные тесты реализуются в рамках структурного тестирования, в соответствии с концепцией структурного тестирования, то есть мы учитываем внутреннею структуру модуля.

Первый принцип: организация структурного тестирования на основе потока данных; второй: на основе потока управления. Структурное тестирование на основе потоков данных предполагает, что тестовое покрытие программы организуется на основе структурных критериев тестирования. Кроме рассмотренных выше критерием добавляются следующие (две лекции назад) — критерий покрытия

условий программы, который предполагает, что все логические условия в программе должны проверяться хотя бы один раз. Можем применяться комбинированный критерий — тестирование ветвей и условий, может применяться критерий покрытия функций программы — это уже не структурный критерий, это уже функциональный, согласно которому каждая функция должна выполниться хотя бы один раз в ходе тестирования — вызовется хотя бы один раз за тестирование. Есть ещё третий критерий: критерий покрытия вызовов функций, в этом случае каждый вызов функции должен выполниться хотя бы один раз — может быть вызвана несколько раз с разными параметрами.

Тестирование на основе потока данных: этот способ тестирования направлен на выявление таких ошибок как: не инициализированные переменные, избыточные присваивания, обращение по не инициализированному указанию. Строится граф преобразования данных. Предполагает следующие три этапа: первый — построение графа потока управления (управляющий граф программы — УГП); второй — выбор тестовых путей, либо это независимые ветви, либо это другие ветки; третий — генерация тестов, соответствующих тестовым путям.

Первый этап подробно: выполняется статический анализ программы с целью получить множество элементов, которые должны быть покрыты тестами, в соответствии с выбранным критерием тестирования.

На втором этапе выполняется поиск тестовых путей, здесь есть разные подходы: тестовые пути на втором этапе могут анализироваться, могут находиться на основе во в первых статических методов, динамических методов и методов реализуемых путей.

Статические методы: анализируется граф потоков управления, путь на графе строится — каждый путь удлиняется на одну дугу, пока не будет достигнута выходная вершина графа. Основной недостаток в том, что могут быть построены нереализуемые пути.

Динамические методы: предполагают построение полной системы тестов, удовлетворяющих заданному критерию, путём одновременного решения построения, покрывающего множество путей и тестовых данных. Есть возможность автоматически учитывать реализуемость или не реализуемость путей.

Метод построения следующий: строится некоторый реализуемый отрезок пути на графе потоков управления, затем к этому пути добавляется очередной отрезок (очередной элемент), таким образом, чтобы не терялась во в первых реализуемость этого суммарного пути, во в вторых — чтобы покрывались требуемые структуры программы.

Методы реализуемых путей, последний: основаны на том, что в множестве всех возможных путей выделяется подмножество реализуемых путей, затем из подмножества реализуемых путей строится

необходимое покрываемое множество путей, то есть реализующих тестовых покрытий программы в соответствии с тестовым критерием.

Статические методы хороши тем, что требует небольшого объема ресурсов, недостаток в том, что статические методы дают непредсказуемый процент бракованных (не реализуемых) путей, которые при тестировании не нужны. Кроме того, статические методы предполагают построение тестовых последовательностей вручную, это чрезвычайно трудоемкая задача (дорогая, скучная).

Динамические методы требуют гораздо больших ресурсов, так как необходимо проверять реализуемость каждого нового пути. Достоинства в том, что эти методы позволяют получить качественный скачок.

Методы реализуемых путей, это самые лучшие методы, они дают максимально качественный результат, по алгоритму видно, что они требуют ещё больше ресурсов, однако характеризуются ещё большей трудоемкостью.

Таким образом, для реализации модульного тестирования необходим сам тестируемый модуль, это может быть класс / функция / программа небольшая и так далее. Затем, необходимы программные заглушки, которые имитируют поведение модулей, взаимодействующих с тестируемым. Необходимо разработать тестовый драйвер, или же программный модуль, который выполняет тестирование.

Рассмотрим пример модульного тестирования классов. Класс, который просто реализует алгоритмические действия. Ладно, только суммирование:

```
Class Summator {
	Public:
	Int add (int a, int b) {
		Return a + b;
	}
}
Class SummatorTest {
	Summator s;
	addTest () {
		int result = s.add (1, 2);
		if (result = 3) {
			print (“Ok”);
		}
		Else
			Print (“Error”);
	}
	SumatorTest () {
		S = new Summator ();
	}
}
```

Особенности реализации тестовых для разных элементов класса: для тестирования публичных элементов можно тестовый драйвера реализовать в виде внешнего класса (по отношению к тестируемому). Для тестирования защищенных (протектед) элементов, тестовый драйвер нужно

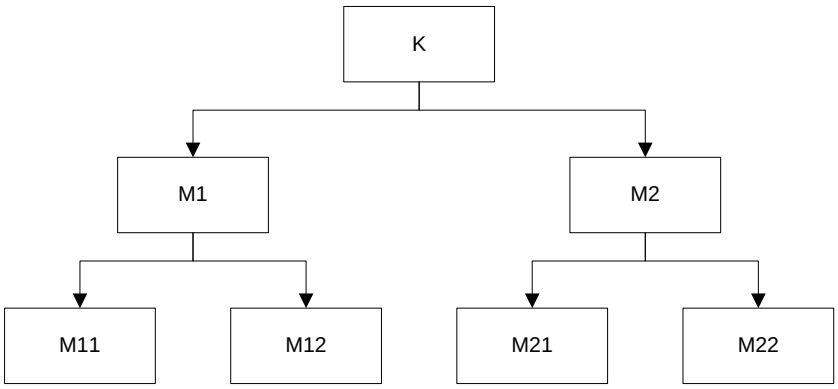
реализовать как наследника от тестируемого. Для тестирования приватных методов классов, тестовый драйвер реализуется внутри тестируемого класса.

Есть такой подход, как разработка через тестирование (TDD — Test Driven Development). Идея в чем: как работает классических подход к разработке понятно – получаем задание, разрабатываем задание, тестируем, эта методология говорит о том, а давайте попробуем наоборот: получили задание, разработали модульные тесты для каждого из классов, после того как разработали тесты начинаем программировать – разработали классы любым способом, только бы работало. Хватит программировать тогда, когда выполнился весь набор тестов. В результате получаем программу, которая как бы работает. После этого выполняется усовершенствование (рефакторинг) программного кода с целью улучшения программного кода, изменяя только внутреннюю структуру, не трогая внешнее взаимодействие. Такой подход позволяет сократить сроки разработки. Есть чёткий критерий, когда нужно завершить программировать, а это хорошо – как только тесты выполнились, значит всё хорошо.

Интеграционное тестирование программных продуктов.

Это тестирование части системы, состоящей из двух или большего количества модулей. Каждый из модулей предварительно прошёл тестирование и работает сам по себе правильно. Основная задача интеграционного тестирования — это обнаружение ошибок, связанных с реализацией и интерпретацией модульных интерфейсов. Интеграционное тестирование принципиально от модульного не отличается, просто является его количественным развитием. Как и модульное тестирование оно требует создание тестового окружения, написание программных заглушек.

Основное отличие от модульного тестирования: цели тестирования (типы обнаруживаемых дефектов), другие методы анализа / тестирования. Применяется покрытие модулей.



На модульное уровне интеграционное тестирование предполагает использование принципа белого ящика, в том смысле, что структура программы известна с точностью до точек вызова модулей. Можно выделить два подхода к интеграции и два подхода к тестированию.

Первый подход: метод большого взрыва (монолитный способ интеграции), все модули интегрируются в один момент времени.

Второй вариант: инкрементарный подход. Идея в том, что модули добавляются интегрируются по одному. Можно интегрировать модули сверху-вниз — метод нисходящего интеграционного тестирования. Второй вариант: интеграций снизу-вверх — восходящее модульное тестирование.

Монолитное тестирование обладает следующими особенностями: так как не все модули могут быть разработаны на момент тестирования, необходимо разрабатывать программные заглушки для замены ещё не реализованных модулей. Сложно идентифицировать и локализовать ошибки, так как сразу интегрируется несколько модулей, сложно определить взаимодействие каких модулей вызывало ошибку. Зато монолитное тестирование позволяет распараллелить тестирование.

Нисходящее тестирование.

Происходит следующим образом: тестирование выполняется начиная с модулей верхнего уровня, отсутствующие модули (нижнего уровня) могут заменяться программных заглушками.

1) K -> XYk;
2) M1 -> XY1;
3) M11 -> XY11; (модуль такой-то тестируется на таких-то входных данных);
4) M12 -> XY12;
5) M2 -> XY2;
6) M21 -> XY21;
7) M22 -> XY22;

Вся суть в том, что дочерние модули тестируются после родительского.

Вторая особенность. Сложность организации тестирования. Таким образом, чтобы модули выполнялись в необходимом порядке. Поскольку модули верхних и нижних уровней могут разрабатываться параллельно, они могут разрабатываться не совсем эффективно. Поскольку параллельно идет разработка модулей разных уровней, эта разработка выполняется не совсем эффективно, поскольку возникает задача подстройки модулей нижних уровней к модулям верхних уровней.

Особенности восходящего тестирования.

Главная особенность заключается в том, что сборка и тестирование модулей выполняется в порядке их реализации. Пример:

1) M11 –> XY11
2) M12 –> XY12
3) M1 -> XY1

4) M21 -> XY21
5) M2(M12, заглушка (м22)) -> XY2;
6) K (M1, M2 (M21, заглушка (M22))) -> XYk;
7) M22 -> XY22;

Недостаток: откладывается проверка концептуальных особенностей всего программного комплекса.

Особенности интеграционного тестирования процедурного программного обеспечения.

Процедурное программирование предполагает написание императивного программного кода. При процедурном проектировании применяется принцип функциональной декомпозиции — сложные действия разбиваются на более простые. Каждый модуль программы имеет несколько точек входа (в строгом смысле одну, но по факту несколько) и несколько точек выхода. Процедурная программа хорошо описывается графовой моделью программы. Построение графовой модули отдельного модуля — это простая тривиальная задача.

Для тестирования модулей процедурных программных продуктов можно использовать критерий покрытия ветвей: каждая дуга и каждая вершина графа проходится хотя бы в одном тесте. При интеграционном тестировании существенным является рассмотрение программы с помощью диаграмм потоков управления. Важным является проверка связей между модулями через данные, то есть тестирование межмодульной передачи данных. При этом каждая переменная модульного взаимодействия, межмодульного интерфейса проверяется на тождественность во взаимодействующих модулях.

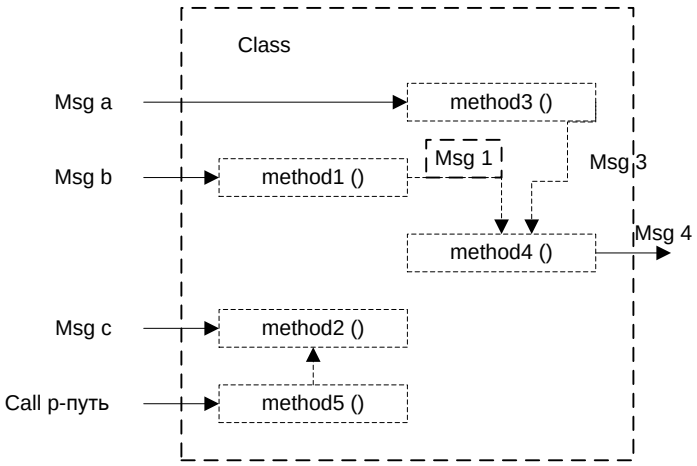
Есть два варианта: плоская или иерархическая модуль проекта, второй вариант: граф вызовов. Если применять **плоскую модуль проекта**, возникает одна проблема — слишком большая сложность графа, как следствие слишком большая сложность тестов => дорого и долго. Поэтому вместо иерархической модули проекта применятся второй вариант — **граф вызовов**. В этом случаем рассматривается не полная картина всех возможных взаимосвязей модуль, а только фактические имеющееся в программе вызовы модулей. То есть не надо тестировать то, что никогда не используется в программе. Построение графов вызовов вместо иерархической модули позволяет снизить сложность тестирования.

Основной вывод: при интеграционном тестировании процедурного кода используется критерий покрытия ветвей графовой модели программы, для построения графовой модели целесообразно использовать граф вызовов, а не полную модель иерархии модулей.

Особенности интеграционного тестирования объективно-ориентированных продуктов.

Графовая модель программы, которая строится для процедурно-ориентированной программы для ООП не подходит, объектно-ориентированная программа является событийно-управляемая, то есть управляется событиями, они могут быть двух видов: события вне вроде нажатия на кнопку или ввод данных; события могут инициироваться одним объектом и приниматься / обрабатываться другим. Объекты взаимодействуют путём передачи друг-другу сообщений. Вместо модели, описывающей структуру программы, применяется модель описывающая поведение программы.

На начальном этапе отдельные объекты проходят модульное тестирование, как правило в соответствии с структурным критерием тестирование ветвей. Объединение методов в класс, можно рассматривать как восходящую интеграцию, поэтому каждый класс можно рассмотреть, как субъект интеграционного тестирования. Графовая модель класса строится таким образом, что в роли узлов выступают методы, а дуги — это вызовы методов. Эти дуги обозначаются термином P – path, то есть процедурный путь. Второй вариант построения дуг: когда в качестве дуг используется обработка сообщений, в этом случае явного вызова метода нет. Для второго случая вызываемый метод можно породить в свою очередь другое сообщение. Это цепочка называется M/M (method / message), то есть метод-сообщение. Таком путь заканчивается, когда достигается метод, не вырабатывающий новых сообщений.



В ходе интеграционного тестирования должны быть проверены все возможные вызовы методов класса как прямые вызовы, так и вызовы через сообщения.

Тестирование интерфейсов модулей

Проводится, когда модули объединяются в более крупные структурные элементы. Каждая система имеет интерфейс для взаимодействия с другими объектами системы. Даже если модули написаны верно, могут возникнуть ошибки взаимодействия.

Ошибки интерфейсов, возникают, как правило, неправильным понимаем того, то делают модули. Тестирование интерфейсов очень важно в тестировании объектно-ориентированных программ.

Интерфейсы бывают (типы)

1) **Параметрические интерфейсы**. Ссылки на данные (иногда — указатели на функции) в виде параметра от одного компонента к другому.
2) **Интерфейсы разделяемой памяти.** Некоторый участок памяти совместно используется несколькими подсистемами (например, MPI).
3) **Процедурные интерфейс.** Одна подсистема инкапсулирует набор процедур, которые вызываются другими подсистемами. Пример: объекты классов.
4) **Интерфейсы передачи сообщений.** Одна подсистема, запрашивая сервис у другой, передавая ей сообщение и получая сообщение с результатом. Пример: Сервис-ориентированные архитектуры (SOA).

Классификация ошибок интерфейсов

1) **Неправильное использование интерфейсов.** Наиболее распространена эта ошибка про процедурные интерфейсы. Например, ошибки в типах аргументов; ошибки, связанные с порядком передачи параметров; ошибки, связанные с неправильным количеством параметров.
2) **Неправильное понимание интерфейсов.** Проблема заключается в следующем: вызывающий компонент, в котором заложена неправильная интерпретация интерфейса вызываемого компонента, предполагает определенное поведение этого компонента. Пример: мы ожидаем, что функция вернёт значение одной переменной, а она возвращает значение другой.
3) **Ошибки синхронизации.** Возникают в системах реального времени, которые используют интерфейсы разделяемой памяти. Причина: неправильное предположение при проектировании взаимодействующий подсистем или процессов. Например, предполагается на этапе проектирования что процесс, который «кладёт» данные в память, работает всегда быстрее, чем процесс, считывающий и обрабатывающий эти данные, как следствие: мы думаем, что никакой синхронизации нам не нужно. В результате, если это допущение нарушается во время работы программы, возникает ошибки синхронизации, которая заключается в том, что второй процесс пытается считаться ещё не записанные данные. Решение**:** не доверяй никому, используя синхронизацию!

Проблема при тестировании интерфейсов — ошибки интерфейсов проявляются в необычных условиях работы программы, которые трудно моделировать на этапе тестирования.

Ещё одна проблема — при тестировании интерфейсов ошибки могут обусловлены взаимодействием ошибок в нескольких модулях.

Общий порядок тестирования интерфейсов следующий:

1) Посмотреть тестируемый код и составить список вызовов, направленных к внешним компонентам.
2) Предусмотреть такие наборы тестовых данных, при которых параметры принимают свои предельные значения.
3) Если посредством интерфейса передаются указатели на данные, необходимо его протестировать с нулевыми указателями.
4) Если тестируется компонент с процедурным интерфейсом, который вызывает сбой в работе компонента.
5) В интерфейсах передачи сообщений следует использовать нагрузочное тестирование, то есть подавать тестовые данные со скоростью, превышающей предельно допустимую.
6) При тестировании интерфейсов памяти необходим менять время активации взаимодействующих компонентов, чтобы видеть неправильные допущения, которые были использованы при проектировании.

Для тестирования интерфейсов эффективным является статическое тестирование. В языках со строгим контролем типов многие ошибки интерфейсов выявляются ещё на этапе компиляции.

Для языков со слабым контролем типов (Си) необходимо применять специальные статические анализаторы. Кроме того, для проверки интерфейсов целесообразно проводить инспектирование программы (code-review).

Нагрузочное тестирование

После того, как завершено модульное тестирование всех модулей, выполнена сборка и интеграционное тестирование, можно говорить о том, что с функциональной точки зрения программа работает правильно.

Однако, кроме функциональных требований к программному продукту, предъявляются ещё и нефункциональные требования:

— надёжность;

— скорость работы;

— … и тому подобные, известные нам нефункциональные требования.

Чтобы проанализировать производительность программного продукта, необходимо выполнить особый вид тестирования, который заключается в том, что программа запускается

и на вход подаются данные сначала с предельно расчётной интенсивностью, а затем со всё увеличивающейся, до тех пор, пока не произойдёт сбой программы.

Такое тестирование — и есть нагрузочное тестирование.

У нагрузочного тестирования есть две основных функции:

1) Протестировать поведение системы во время сбоя. Сбой в системе не должен приводить к нарушению целостности данных и не должно происходить никаких аппаратных проблем.
2) Выявить дефекты, которые не проявляются в штатных режимах работы программы.

Особенности тестирования объектно-ориентированных систем

Объекты — тоже модули. Объект как программный компонент принципиально отличается программы или функции.

В случае процедурно-ориентированной программы говорить о иерархии модулей, в случае объектно-ориентированной программы объекты слабо связаны между собой, как следствие: нельзя говорить о иерархии объектов.

При тестировании объектно-ориентированных программы обнаруживается, что объекты могут использоваться повторно, а их исходный код может быть недоступен.

Порядок тестирования объектно-ориентированных программ

1) Тестирование отдельных методов;
2) Тестирование отдельных классов (модульное тестирование);
3) Тестирование кластеров объектов, то есть группы взаимодействующих объектов;
4) Системное тестирование.

Системное тестирование

Принципиальное отличается от интеграционного и модульного, потому что системное тестирование рассматривает систему в целом и выполняется на уровне интерфейсов пользователя.

Основные задачи системного тестирования — это не выявление дефектов, а анализ интегральных свойств системы, таких как надёжность, производительность, использование ресурсов и так далее.

Считается, что всё уже выполняется правильно. Проверяем, не что делает система, а как она это делает. Кроме программного продукта, системному тестированию подвергается и документация.

Критерии системного тестирования

1) Полнота решения функциональных задач;
2) Корректность использования ресурсов (анализируются утечки памяти, возврат ресурсов и так далее);
3) Оценка производительности программы;
4) Эффективность защиты от искажения данных и от некорректных действий пользователя;
5) Проверяется возможность инсталляции конфигурирования программного продукта на разных платформах.

Регрессионное тестирование

Регрессионное тестирование выполняется после внесения изменений в программный продукт.

Главный вопрос регрессионного тестирования — выбор минимально достаточного набора тестов, то есть поиск компромисса между полным перетестированием всей системы (долго и дорого) и тестированием только изменённых модулей (этого может не хватать).

Верификация и аттестация информационных систем.

Это процессы проверки и анализа, в ходе которых выявляется соответствие программного продукта спецификации и ожиданиям заказчика.

Верификация и аттестация — процессы схожие, но не совсем. Верификация отвечает на вопрос правильно ли создана система, проверяет спецификацию. Аттестация — отвечает на вопрос, правильно ли она работает, процесс более общий, не такой формальный, проверяет ожидание заказчика, проводится после верификации.

Используются два основных подхода для верификации аттестации.

1) Инспектирование программного продукта;
2) Тестирование программного продукта.

Инспектирование предполагает проверку и анализ различных представлений системы на всех этапах работы над проектом. Это процесс, в ходе которого инспектор или же группа инспекторов просматривает систему с целью обнаружения ошибок и недоработок.

Тестирование предполагает запуск данных с тестовыми данными.

Инспектирование можно считать статическим способом проверки требований, а тестирование — динамический способ.

Все методы тестирования можно разделить на две группы: это тестирование дефектов, статическое тестирование. Статическому тестированию подвергается правильно работящая программа, в которой нет дефектов, цель статического тестирования — анализ некоторых статических показателей системы, таких как производительность, надёжность, работа в различных режимах и так далее.

Инспектирование. Может выполняться на всех этапах жизненного цикла программного продукта. В том смысле, что инспектированию может подвергаться сама программа, так и документация — диаграмма и так далее. Инспектировать можно двумя основными способами: собственное инспектирование с привлечение группы экспертов-программистов; второй способ — это автоматический анализ.

Инспектирование программного обеспечения

Инспектирование применяется из-за ограниченных возможностей тестирования. Так как для тестирования программа должна быть рабочей и как минимум компилироваться, инспектирование позволяет проверять ещё не законченный продукт. **Инспектирование позволяет обнаруживать целые семейства взаимосвязанных ошибок, потенциальные ошибки и остальные неприятные моменты**. При тестировании ошибка в одном модуле может привести к неправильной работе другого модуля, разрушению данных, проще говоря — тестирование ограничено для выявления сложных ошибок. Инспектирование дешевле (если под тестированием понимать стоимость написания тестов). Статистика говорит, что более 60% ошибок в программе может быть выявлена с помощью инспектирования. При инспектировании с использованием математических методов (формализовать), то процент выявляемых ошибок можно довести до 90%.

Как организуется сам процесс.

Процесс инспектирования — это формализованный процесс, в том смысле, что есть разбивка на определённые участки и этапы. Классическая модель предполагает следующие роли в команде инспекторов: автор, рецензент, инспектор и координатор.

Для выполнения инспектирования должно выполняться следующие условие: должна быть точная спецификация кода, так как без неё мы не будем знать, что нам нужно инспектировать. Второе требование: при инспектировании представляется синтаксически корректная версия программного

кода. Участники инспекционной группы должны знать стандарты написания кода, которые приняты в данной команде.

Инспектирование включает в себя следующие этапы:

1) Планирование.

2) Этап предварительного просмотра.

3) Индивидуальная подготовка.

4) Собрание инспекционной группы.

5) Исправление ошибок.

6) Доработка.

Координатор составляет план инспектирование, подбирает участников инспекционной группы.

Координатор организует собрание, контролирует чтобы спецификация и синтаксически верная программа были представлены.

Программа, предоставленная на инспектирование, передаётся на рассмотрение инспекционной группы.

На этапе предварительного просмотра автор описывает назначение программы.

На этапе индивидуальной подготовки, каждый из участников инспекционной группы изучает программу и её спецификацию и выявляет дефекты.

Затем проводится собрание инспекционной группы. Это собрание должно быть по возможности коротким — не более двух часов. На этом этапе основное внимание обращается на обнаружение дефектов, аномалий и других отклонений от спецификации. Инспекционная группа не должна предлагать способов устранения дефектов.

После этого (на пятом этапе), автор программы изменяет и модифицирует программу, исправляя обнаруженные ошибки.

На последнем (шестом) координатор решает, нужно ли проводить инспектирование повторное, либо же хватит уже. Если принято решение, что повторно инспектировать не нужно, то все найденные ранее дефекты документально фиксируются, составленный документ предоставляется руководителям проекта.

На этапе предварительного просмотра — за один час просматривается около 500 операторов программы.

На этапе индивидуальной подготовки проверяется около 125 операторов, так как проверка выполняется более детально.

На собрании инспекционной группы проверяется от 90 до 125 операторов. Эти все цифры условны, всё зависит от многих факторов — классификации продукта, сложности, стиля и так далее.

Статический анализ программы

Выполняется с помощью специализированных программных инструментов. Анализатор анализирует код программы и выявляет потенциальные ошибки. Обычно выполняется перед компиляцией. Статический анализатор — программный инструмент, предназначенный для выявления ошибок, выполнение программы при этом не требуется. Цель автоматического статического анализа — привлечь внимание разработчика к аномалиям программы.

Статический анализ включает в себя следующие этапы:

1) Анализ потока управления. На этом этапе определяются циклы в программы, точки входа и выхода, обнаруживается по возможность неиспользуемый программный код.

2) Анализ использования данных. Здесь анализируется как используются переменные в программы, определить переменные, которые используются без инициализации, обнаруживаются переменные, значения которых нигде не используются. Здесь же выявляются условные операторы с избыточными условиями и так далее.

3) Анализ интерфейсов. На этом этапе анализируется согласованность отдельных модулей программы, проверяется правильность написания процедур. Проверяется также правильность вызова процедур. Здесь же могут обнаруживаться функции, которые нигде и никогда не вызываются, либо же функции, результаты которых нигде и никогда не используются.

4) Анализ потоков данных. Определяются зависимости между выходными данными (переменными). Ошибок такой не выявляет, но помогает понять, как работает программный модуль. Помогает понять, когда на вход поступают верные данные.

5) Анализ ветвей программы. Выполняется сематический анализ программы, определяются все ветки программы и операторы, выполняющееся в каждой ветке.

Показатели качества работы программного продукта

Программный продукт можно описать некоторыми количественными характеристиками.

Показатели можно разделить на контрольные и прогнозируемые

- **Контрольные показатели** обычно связаны с процессом разработки, а прогнозируемые характеризуют конечный программный продукт. Пример контрольного показателя: среднее время на исправление ошибки, длительность разработки.
- **Прогнозируемые показатели** — цикломатическая сложность модулей, средняя длина идентификаторов в программе и так далее.

Основная сложность, связанная с определение основных внешних показателей — нельзя измерить непосредственно. Эти показатели — внешние показатели. Применяется выход: измеряются внутренние показатели (цикломатическое число и так далее), на их основе измеряются внешние показатели.

Показатели бывают динамические и статические

- **Динамические показатели** — показатели, которые могут быть определены только результатам выполняя программы. Расход памяти сложно определить без выполнения программы и так далее.
- **Статические показатели** — определяются статическим представлением системы, они характеризуются структуры программы, документацию, для этого выполнение программы не требуется.

Главным образом именно динамические показатели характеризируют качество программного продукта.