Лекция

Тема: Отладка и тестирование программного продукта

ВВЕДЕНИЕ

Известно, что при создании типичного программного проекта около 50 % общего времени и более 50 % общей стоимости расходуется на тестирование разрабатываемой программы или системы. Эти цифры могут вызвать целую дискуссию, но, оставив вопрос точности оценки в стороне и основываясь на том, что тестирование является важным этапом в создании программного продукта, можно было бы предположить, что к настоящему времени тестирование программ поднялось до уровня точной науки. Увы, это не так. На самом деле тестирование программ освещено, пожалуй, меньше, чем любой другой аспект разработки программного обеспечения. К тому же тестирование является до сих пор «немодным» предметом, если иметь в виду спорность публикаций по этому вопросу.

Любой программист может похвастать «хорошо» написанным кодом, модулем, классом, но, как правило, он практически ничего не может сказать, насколько полно оттестирован этот код. Многие готовы ругать других разработчиков, указывая на их ошибки, сбои их программного обеспечения, забывая о своих.

Так что же делать разработчику, менеджеру проекта, руководителю фирмы? Как сократить расходы и повысить качество программного обеспечения? Сколько нужно тестировать программное обеспечение? Как построить эффективный процесс тестирования? Какие инструментальные средства использовать? Вопросов в области тестирования настолько много, что охватить их в одном пособии практически невозможно. Было решение создать пособие, с которого можно было бы начать изучение тестирования программного обеспечения. В качестве базы выбрана замечательная книга [1].

Материал книги значительно переработан, расширен современными публикациями и дополнен собственным опытом. Весь публикуемый материал неоднократно излагался автором на лекциях по дисциплинам «Технология программирования» и «Технологии программирования».

1. ФИЛОСОФИЯ ТЕСТИРОВАНИЯ

1.1. Тест для самооценки

Хотите испытать себя в тестировании? Задача состоит в том, чтобы проверить некоторый метод.

Данный метод получает в качестве параметров три целых числа, которые интерпретируются как длины сторон треугольника. Выходом метода является сообщение о том, является ли треугольник неравносторонним, равнобедренным или равносторонним [1].

Напишите на листе бумаги набор тестов (т. е. специальные последовательности данных), которые, как вам кажется, будут адекватно проверять этот метод. Построив свои тесты, проанализируйте их.

Следующий шаг состоит в оценке эффективности вашей проверки. Оказывается, что метод труднее написать, чем это могло показаться вначале. Были изучены различные версии данного метода и составлен список общих ошибок. Оцените ваш набор тестов, попытавшись с его помощью ответить на приведенные ниже вопросы. За каждый ответ «да» присуждается одно очко.

1. Составили ли вы тест, который представляет правильный неравносторонний треугольник? (Заметим, что ответ «да» на тесты, со значениями 1, 2, 3 и 2, 5, 10 не обоснован, так как не существует треугольников, имеющих такие стороны.)
2. Составили ли вы тест, который представляет правильный равносторонний треугольник?
3. Составили ли вы тест, который представляет правильный равнобедренный треугольник? (Тесты со значениями 2, 2, 4 принимать в расчет не следует.)
4. Составили ли вы, по крайней мере, три теста, которые представляют правильные равнобедренные треугольники, полученные перестановкой двух равных сторон треугольника (например, 3, 3, 4; 3, 4, 3 и 4, 3, 3)?
5. Составили ли вы тест, в котором длина одной из сторон треугольника принимает нулевое значение?
6. Составили ли вы тест, в котором длина одной из сторон треугольника принимает отрицательное значение?
7. Составили ли вы тест, включающий три положительных целых числа, сумма двух из которых равна третьему? (Другими словами, если программа выдала сообщение о том, что числа 1, 2, 3 представляют собой стороны неравностороннего треугольника, то такая программа содержит ошибку.)
8. Составили ли вы, по крайней мере, три теста с заданными значениями всех трех перестановок, в которых длина одной стороны равна сумме длин двух других сторон (например, 1, 2, 3; 1, 3, 2 и 3, 1, 2)?
9. Составили ли вы тест из трех целых положительных чисел, таких, что сумма двух из них меньше третьего числа (т. е. 1, 2, 4 или 12, 15, 30)?
10. Составили ли вы, по крайней мере, три теста из категории 9, в которых вами испытаны все три перестановки (например, 1, 2, 4; 1, 4, 2 и 4, 1, 2)?
11. Составили ли вы тест, в котором все стороны треугольника имеют длину, равную нулю (т. е. 0, 0, 0)?
12. Составили ли вы, по крайней мере, один тест, содержащий нецелые значения?
13. Составили ли вы хотя бы один тест, содержащий неправильное число значений (например, два, а не три целых числа)?
14. Описали ли вы заранее в каждом тесте не только входные значения, но и выходные данные метода?

Конечно, нет гарантий, что с помощью набора тестов, который удовлетворяет вышеперечисленным условиям, будут найдены все возможные ошибки. Но поскольку вопросы 1–13 представляют ошибки, имевшие место в различных версиях данного метода, адекватный тест для него должен их обнаруживать. Для сравнения отметим, что опытные профессиональные программисты и тестировщики набирают в среднем только 7–8 очков из 14 возможных. Выполненное упражнение показывает нам, что тестирование даже тривиальных программ, подобных приведенной, – непростая задача.

1.2. Определение термина «тестирование»

Тестирование как объект изучения может рассматриваться с различных чисто технических точек зрения. Однако наиболее важными при изучении тестирования представляются вопросы его экономики и психологии разработчика. Иными словами, достоверность тестирования программы в первую очередь определяется тем, кто будет ее тестировать и каков его образ мышления, и уже затем определенными

технологическими аспектами. Поэтому, прежде чем перейти к техническим проблемам, мы остановимся на этих вопросах.

Вопросы экономики и психологии до сих пор тщательно не исследованы. Однако, необходимо разобраться в общих моментах экономики и тестирования.

Поначалу может показаться тривиальным жизненно важный вопрос определения термина «тестирование». Необходимость обсуждения этого термина связана с тем, что большинство специалистов используют его неверно, а это в свою очередь приводит к плохому тестированию. Таковы, например, следующие определения: «Тестирование представляет собой процесс, демонстрирующий отсутствие ошибок в программе», «Цель тестирования – показать, что программа корректно исполняет предусмотренные функции», «Тестирование – это процесс, позволяющий убедиться в том, что программа выполняет свое назначение».

Эти определения описывают нечто противоположное тому, что следует понимать под тестированием, поэтому они неверны. Оставив на время определения, предположим, что если мы тестируем программу, то нам нужно добавить к ней некоторую новую стоимость (так как тестирование стоит денег и нам желательно возвратить затраченную сумму, а это можно сделать только путем увеличения стоимости программы). Увеличение стоимости означает повышение качества или возрастание надежности программы, в противном случае пользователь будет недоволен платой за качество. Повышение качества или надежности программы связано с обнаружением и удалением из нее ошибок. Следовательно, программа тестируется не для того, чтобы показать, что она работает, а скорее наоборот – тестирование начинается с предположения, что в ней есть ошибки (это предположение справедливо практически для любой программы), а затем уже обнаруживается их максимально возможное число. Таким образом, сформулируем наиболее приемлемое и простое определение:

Тестирование – это процесс исполнения программы с целью обнаружения ошибок.

Пока все наши рассуждения могут показаться тонкой игрой семантик, однако практикой установлено, что именно ими в значительной мере определяется успех тестирования. Дело в том, что верный выбор цели дает важный психологический эффект, поскольку для человеческого сознания характерна целевая направленность. Если поставить целью демонстрацию отсутствия ошибок, то мы подсознательно будем стремиться к этой цели, выбирая тестовые данные, на которых вероятность появления ошибки мала. В то же время, если нашей задачей станет обнаружение ошибок, то создаваемый нами тест будет обладать большей вероятностью обнаружения ошибки. Такой подход заметнее повысит качество программы, чем первый.

Из приведенного определения тестирования вытекает несколько следствий. Например, одно из них состоит в том, что **тестирование – процесс деструктивный** (т. е. обратный созидательному, конструктивному). Именно этим и объясняется, почему многие программисты и тестировщики считают его трудным. Большинство людей склонны к конструктивному процессу созидания объектов и в меньшей степени – к деструктивному процессу разделения на части. Из определения следует также, как нужно строить набор тестовых данных и кто должен (а кто не должен) тестировать данную программу.

Для усиления определения тестирования проанализируем два понятия «удачный» и «неудачный» и, в частности, их использование руководителями проектов при оценке

результатов тестирования. Некоторые руководители программных проектов называют тестовый прогон «неудачным» если обнаружена ошибка, и, наоборот, удачным, если он прошел без ошибок. Чаще всего это является следствием ошибочного понимания термина «тестирование», так как, по существу, слово «удачный» означает «результативный», а слово «неудачный» – «нежелательный», «нерезультативный». Но если тест не обнаружил ошибки, его выполнение связано с потерей времени и денег, и термин «удачный» никак не может быть применен к нему. Естественно, заранее неизвестно, будет ли тест удачным или неудачным, но построение удачных тестов – отдельная тема.

Тестовый прогон, приведший к обнаружению ошибки, нельзя назвать неудачным хотя бы потому, что, как отмечалось выше, это целесообразное вложение капитала. Отсюда следует, что в слова «удачный» и «неудачный» необходимо вкладывать смысл, обратный общепринятому. Поэтому в дальнейшем будем называть тестовый прогон удачным, если в процессе его выполнения обнаружена ошибка, и неудачным, если получен корректный результат.

Проведем аналогию с посещением больным врача. Если рекомендованное врачом лабораторное исследование не обнаружило причины болезни, не назовем же мы такое исследование удачным – оно неудачно: ведь счет пациента сократился на 500 рублей, а он все так же болен. Если же исследование показало, что у больного язва желудка, то оно является удачным, поскольку врач может прописать необходимый курс лечения. Следовательно, медики используют эти термины в нужном нам смысле. (Аналогия здесь, конечно, заключается в том, что программа, которую предстоит тестировать, подобна больному пациенту.)

Определения типа «тестирование представляет собой процесс демонстрации отсутствия ошибок» (например, в [5] и [6]) порождают еще одну проблему: они ставят цель, которая не может быть достигнута ни для одной программы, даже весьма тривиальной. Результаты психологических исследований показывают, что если перед человеком ставится невыполнимая задача, то он работает хуже. Например, если предложить кому-то решить кроссворд в воскресном номере «Нью-Йорк Таймс» за 15 минут, то через 10 минут не будет достигнут значительный успех; ведь понятно, что это невыполнимая задача. Если же на решение отводится четыре часа, то через 10 минут результат окажется лучше [1]. Иными словами, определение тестирования как процесса обнаружения ошибок переводит его в разряд решаемых задач и таким образом преодолевается психологическая трудность.

Другая проблема возникает в том случае, когда для тестирования используется следующее определение: «Тестирование – это процесс, позволяющий убедиться в том, что программа выполняет свое назначение», поскольку программа, удовлетворяющая данному определению, может содержать ошибки. Если программа не делает того, что от нее требуется, то ясно, что она содержит ошибки. Однако ошибки могут быть и тогда, когда она делает то, что от нее не требуется. Вспомните тест для самооценки, метод может допустить ошибку, если будет делать то, что он не должен делать (например, сообщать, что тройка 1, 2, 3 представляет неравносторонний треугольник, а тройка 0, 0, 0 – равносторонний). Ошибки этого класса можно обнаружить скорее, если рассматривать тестирование как процесс поиска ошибок, а не демонстрацию корректности работы.

Подводя итог вопросу определения термина «тестирование», можно сказать, что тестирование представляется деструктивным процессом попыток обнаружения ошибок в программе (наличие которых предполагается). Набор тестов,

способствующий обнаружению ошибки, считается удачным. Естественно, в конечном счете, каждый с помощью тестирования хочет добиться определенной степени уверенности в том, что его программа соответствует своему назначению и не делает того, для чего она не предназначена, но лучшим средством для достижения этой цели является непосредственный поиск ошибок. Допустим, кто-то обращается к вам с заявлением: «Моя программа великолепна» (т. е. не содержит ошибок). Лучший способ доказать справедливость подобного утверждения – попытаться его опровергнуть, обнаружить неточности, нежели просто согласиться с тем, что программа на определенном наборе входных данных работает корректно.

1.3. Экономика тестирования

Дав такое определение тестированию, необходимо на следующем шаге рассмотреть возможность создания теста, обнаруживающего все ошибки программы. Покажем, что ответ будет отрицательным даже для самых тривиальных программ. В общем случае невозможно обнаружить все ошибки программы. А это в свою очередь порождает экономические проблемы, задачи, связанные с функциями человека в процессе отладки, способы построения тестов [2].

1.3.1. Тестирование программы как черного ящика

Одним из способов изучения поставленного вопроса является исследование стратегии тестирования, называемой стратегией черного ящика, тестированием с управлением по данным, или тестированием с управлением по входу-выходу. При использовании этой стратегии программа рассматривается как черный ящик. Иными словами, такое тестирование имеет целью выяснение обстоятельств, в которых поведение программы не соответствует ее спецификации. Тестовые же данные используются только в соответствии со спецификацией программы (т. е. без учета знаний о ее внутренней структуре).

При таком подходе обнаружение всех ошибок в программе является критерием исчерпывающего входного тестирования. Последнее может быть достигнуто, если в качестве тестовых наборов использовать все возможные наборы входных данных. Необходимость выбора именно этого критерия иллюстрируется следующим примером. Если в той же задаче о треугольниках один треугольник корректно признан равносторонним, нет никакой гарантии того, что все остальные равносторонние треугольники так же будут корректно идентифицированы. Так, для треугольника со сторонами 3842, 3842, 3842 может быть предусмотрена специальная проверка и он считается неравносторонним. Поскольку программа представляет собой черный ящик, единственный способ удовлетворения приведенному выше критерию – перебор всех возможных входных значений.

Таким образом, исчерпывающий тест для задачи о треугольниках должен включать равносторонние треугольники с длинами сторон вплоть до максимального целого числа. Это, безусловно, астрономическое число, но и оно не обеспечивает полноту проверки. Вполне вероятно, что останутся некоторые ошибки, например, метод может представить треугольник со сторонами 3, 4, 5 неразносторонним, а со сторонами 2, A, 2 – равносторонним. Для того, чтобы обнаружить подобные ошибки, нужно перебрать не только все разумные, но и все вообще возможные входные наборы. Следовательно, мы приходим к выводу, что для исчерпывающего тестирования задачи о треугольниках требуется бесконечное число тестов.

Если такое испытание представляется сложным, то еще сложнее создать исчерпывающий тест для большой программы. Образно говоря, число тестов можно оценить «числом, большим, чем бесконечность». Допустим, что делается попытка тестирования методом черного ящика компилятора с языка Java. Для построения исчерпывающего теста нужно использовать все множество правильных программ на Java (фактически их число бесконечно) и все множество неправильных программ (т. е. действительно бесконечное число), чтобы убедиться в том, что компилятор обнаруживает все ошибки. Только в этом случае синтаксически неверная программа не будет компилирована. Если же программа имеет собственную память (например, операционная система, база данных или система распределенных вычислений), то дело обстоит еще хуже. В таких программах исполнение команды (например, задание, запрос в базу данных, выполнение расчета) зависит от того, какие события ей предшествовали, т. е. от предыдущих команд. Здесь следует перебрать не только все возможные команды, но и все их возможные последовательности.

Из изложенного следует, что построение исчерпывающего входного теста невозможно. Это подтверждается двумя аргументами: во-первых, **нельзя создать тест, гарантирующий отсутствие ошибок**; во-вторых, разработка таких тестов противоречит экономическим требованиям. Поскольку исчерпывающее тестирование исключается, нашей целью должна стать максимизация результативности капиталовложений в тестирование (иными словами, максимизация числа ошибок, обнаруживаемых одним тестом). Для этого мы можем рассматривать внутреннюю структуру программы и делать некоторые разумные, но, конечно, не обладающие полной гарантией достоверности предположения (например, разумно предположить, что если программа сочла треугольник 2, 2, 2 равносторонним, то таким же окажется и треугольник со сторонами 3, 3, 3).

1.3.2. Тестирование программы как белого ящика

Стратегия белого ящика, или стратегия тестирования, управляемого логикой программы, позволяет исследовать внутреннюю структуру программы. В этом случае тестирующий получает тестовые данные путем анализа логики программы (к сожалению, здесь часто не используется спецификация программы).

Сравним способ построения тестов при данной стратегии с исчерпывающим входным тестированием стратегии черного ящика. Непосвященному может показаться, что достаточно построить такой набор тестов, в котором каждый оператор исполняется хотя бы один раз; нетрудно показать, что это неверно. Не вдаваясь в детали, укажем лишь, что исчерпывающему входному тестированию может быть поставлено в соответствие исчерпывающее тестирование маршрутов. Подразумевается, что программа проверена полностью, если с помощью тестов удается осуществить выполнение этой программы по всем возможным маршрутам ее потока (графа) передач управления.

Последнее утверждение имеет два слабых пункта. Один из них состоит в том, что число не повторяющих друг друга маршрутов в программе – астрономическое. Чтобы убедиться в этом, рассмотрим представленный на рис. 1 граф передач управления в простейшей программе. Каждая вершина, или кружок, обозначают участок программы, содержащий последовательность линейных операторов, которая может заканчиваться оператором ветвления. Дуги, оканчивающиеся стрелками, соответствуют передачам управления. По-видимому, граф описывает программу из 10–20 операторов, включая цикл **WHILE** (или DO WHILE), который исполняется не менее 20 раз (на рисунке показан темным цветом). Внутри цикла имеется несколько

операторов **IF** (на рисунке соответствующие узлы графа изображены пустыми кружками). Для того чтобы определить число неповторяющихся маршрутов при исполнении программы, подсчитаем число неповторяющихся маршрутов из точки **A** в **B** в предположении, что все приказы взаимно независимы. Это число вычисляется как сумма $5^{20} + 5^{19} + ... + 5^{1} = 10^{14}$, или 100 триллионам, где 5 – число путей внутри цикла. Поскольку большинству людей трудно оценить это число, приведем такой пример: если допустить, что на составление каждого теста мы тратим пять минут, то для построения набора тестов нам потребуется примерно один миллиард лет.

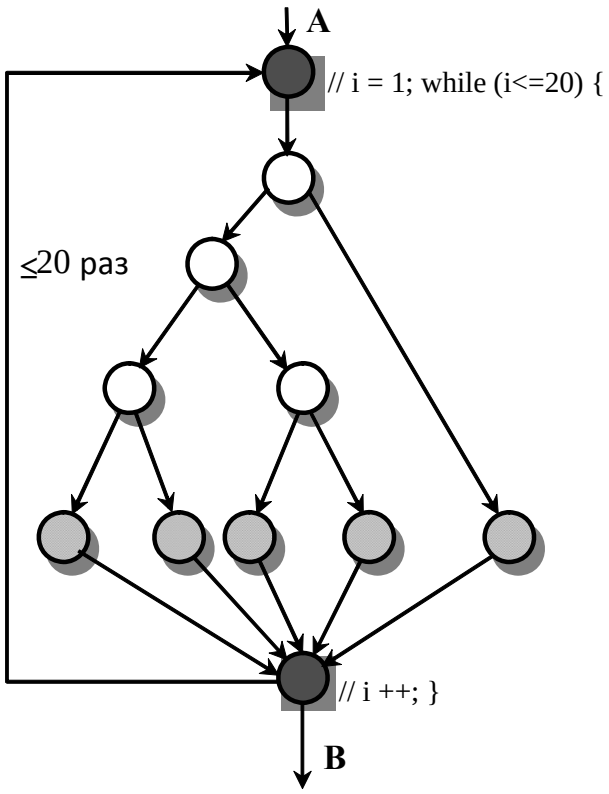

Рис. 1. Граф передач управления небольшой программы

Конечно, в реальных программах условные переходы не могут быть взаимно независимы, т. е. число маршрутов исполнения будет несколько меньше. С другой стороны, реальные программы значительно больше, чем простая программа, представленная на рис. 1. Следовательно, исчерпывающее тестирование маршрутов, как и исчерпывающее входное тестирование, не только невыполнимо, но и невозможно.

Второй слабый пункт утверждения заключается в том, что, хотя исчерпывающее тестирование маршрутов является полным тестом и хотя каждый маршрут программы может быть проверен, сама программа будет содержать ошибки. Это объясняется следующим образом. Во-первых, исчерпывающее тестирование маршрутов не может дать гарантии того, что программа соответствует описанию. Например, вместо требуемой программы сортировки по возрастанию случайно была написана программа сортировки по убыванию. В этом случае ценность тестирования маршрутов невелика, поскольку после тестирования в программе окажется одна ошибка, т. е. программа неверна. Во-вторых, программа может быть неверной в силу того, что пропущены некоторые маршруты. Исчерпывающее тестирование маршрутов не обнаружит их отсутствия. Втретьих, исчерпывающее тестирование маршрутов не может обнаружить ошибок, появление которых зависит от обрабатываемых данных.

Существует множество примеров таких ошибок. Приведем один из них. Допустим, в программе необходимо выполнить сравнение двух чисел на сходимость, т. е. определить, является ли разность между двумя числами меньше предварительно определенного числа. Может быть написано выражение IF ((a – b) < epsilon)…

Безусловно, оно содержит ошибку, поскольку необходимо выполнить сравнение абсолютных величин. Однако обнаружение этой ошибки зависит от значений, использованных для a и b, и ошибка не обязательно будет обнаружена просто путем исполнения каждого маршрута программы.

В заключение отметим, что, хотя исчерпывающее входное тестирование предпочтительнее исчерпывающего тестирования маршрутов, ни то, ни другое не могут стать полезными стратегиями, потому что оба они нереализуемы. Возможно, поэтому реальным путем, который позволит создать хорошую, но, конечно, не абсолютную стратегию, является сочетание тестирования программы как черного и как белого ящиков. Вопрос выбора методов тестирования и их описание будет рассмотрен в дальнейшем.

1.4. Принципы тестирования

Сформулируем основные принципы тестирования, используя главную предпосылку настоящей главы о том, что наиболее важными в тестировании программ являются вопросы психологии [4]. Эти принципы интересны тем, что в основном они интуитивно ясны, но в то же время на них часто не обращают должного внимания.

Описание предполагаемых значений выходных данных или результатов должно быть необходимой частью тестового набора.

Нарушение этого очевидного принципа представляет одну из наиболее распространенных ошибок. Ошибочные, но правдоподобные результаты могут быть признаны правильными, если результаты теста не были заранее определены. Здесь мы сталкиваемся с явлением психологии: мы видим то, что мы хотим увидеть. Другими словами, несмотря на то, что тестирование по определению – деструктивный процесс, есть подсознательное желание видеть корректный результат. Один из способов борьбы с этим состоит в поощрении детального анализа выходных переменных заранее при разработке теста. Поэтому тест должен включать две компоненты: **описание входных данных** и **описание точного и корректного результата**, соответствующего набору входных данных.

Необходимость этого подчеркивал логик Копи в работе [3]: «Проблема может быть охарактеризована как факт или группа фактов, которые не имеют приемлемого объяснения, которые кажутся необычными или которые не удается подогнать под наши представления или предположения. Очевидно, что если *что-нибудь* подвергается сомнению, то об этом должна иметься какая-то предварительная информация. Если нет предположений, то не может быть и неожиданных результатов».

Следует избегать тестирования программы ее автором.

К сожалению, реализация этого в целом верного принципа не всегда возможна в силу трех факторов:

1) людские ресурсы разработки, как правило, недостаточны;

2) для регулярного применения этого принципа к каждой программе требуется весьма высокая квалификация всех программистов или большой группы программистов, тестирующих все программы, что не всегда осуществимо;
3) необходим высокий уровень формализации ведения разработки; тщательные формализованные спецификации требований к программам и данным, тщательное описание интерфейса и формализация ответственности за качество продукта.

В настоящее время проводится значительная работа по созданию и внедрению формализованных методов в большинстве крупных разработок, но опыт подобного ведения разработок пока еще недостаточно массовый.

Этот принцип следует из того факта, что тестирование – это деструктивный процесс. После выполнения конструктивной части при проектировании и написании программы программисту трудно быстро (в течение одного дня) перестроиться на деструктивный образ мышления.

Многие, кому приходилось самому делать дома ремонт, знают, что процесс обрывания старых обоев (деструктивный процесс) не легок, но он просто невыносим, если не кто-то другой, а вы сами вчера их наклеивали. И вам не придет в голову срывать их, если где-то они чуть-чуть неровно легли на стену. Вот так же и большинство программистов не могут эффективно тестировать свои программы, потому что им трудно демонстрировать собственные ошибки.

Это действительно сильный психологический фактор при коллективной разработке. Программист, тщательно отлаживающий программу, невольно может работать медленнее, что становится известно другим участникам разработки. С другой стороны, он вынужден запрашивать дополнительное машинное время на отладку у своего непосредственного руководителя. Тем самым итоги тестирования оказываются уже не просто делом одного человека, тестирующего программу (пока в большинстве случаев ее автора), но и информацией, возбуждающей общественный интерес (и оценку!) участников разработки, в том числе ее руководителей. Перспектива создать о себе мнение как о специалисте, делающем много ошибок, не воодушевляет программиста, и он подсознательно снижает требования к тщательности тестирования. В такой ситуации от руководителей разработки всех рангов требуется большое чувство такта и понимание процессов, чтобы поощрять специалистов, проводящих тщательное тестирование, и уметь различить и ограничить деятельность программистов, прикрывающих свою нерадивость трудностями тестирования.

В дополнение к этой психологической проблеме следует отметить еще одну, не менее важную: программа может содержать ошибки, связанные с неверным пониманием постановки или описания задачи разработчиком. Тогда существует вероятность, что к тестированию разработчик приступит с таким же недопониманием своей задачи.

Тестирование можно уподобить работе корректора или рецензента над статьей или книгой. Многие авторы представляют себе трудности, связанные с редактированием собственной рукописи. Очевидно, что обнаружение недостатков в своей деятельности противоречит человеческой психологии.

Отсюда вовсе не следует, что программист не может тестировать свою программу. Многие программисты с этим вполне успешно справляются. Здесь лишь делается вывод о том, что тестирование является более эффективным, если оно выполняется кем-либо другим. Заметим, что все наши рассуждения не относятся к

отладке, т. е. к исправлению уже известных ошибок. Эта работа эффективнее выполняется самим автором программы.

Программирующая организация не должна сама тестировать разработанные ею программы.

Здесь можно привести те же аргументы, что и в предыдущем случае. Во многих смыслах проектирующая или программирующая организация подобна живому организму с его психологическими проблемами. Работа программирующей организации или ее руководителя оценивается по их способности производить программы в течение заданного времени и определенной стоимости. Одна из причин такой системы оценок состоит в том, что временные и стоимостные показатели легко измерить, но в то же время чрезвычайно трудно количественно оценить надежность программы. Именно поэтому в процессе тестирования программирующей организации трудно быть объективной, поскольку тестирование в соответствии с данным определением может быть рассмотрено как средство уменьшения вероятности соответствия программы заданным временным и стоимостным параметрам.

Как и ранее, из изложенного не следует, что программирующая организация не может найти свои ошибки; тестирование в определенной степени может пройти успешно. Мы утверждаем здесь лишь то, что экономически более целесообразно выполнение тестирования каким-либо объективным, независимым подразделением.

В некоторых организациях подобная практика существует, но только на этапах комплексной отладки. Подобный способ тестирования чрезвычайно сложно реализовать из-за организационных трудностей.

Необходимо досконально изучать результаты применения каждого теста.

По всей вероятности, это наиболее очевидный принцип, но и ему часто не уделяется должное внимание. В экспериментах, проверенных автором, многие испытуемые не смогли обнаружить определенные ошибки, хотя их признаки были совершенно явными в выходных листингах. Представляется достоверным, что значительная часть всех обнаруженных в конечном итоге ошибок могла быть выявлена в результате самых первых тестовых прогонов, но они были пропущены вследствие недостаточно тщательного анализа результатов первого тестового прогона.

Тесты для неправильных и непредусмотренных входных данных следует разрабатывать так же тщательно, как для правильных и предусмотренных.

При тестировании программ имеется естественная тенденция концентрировать внимание на правильных и предусмотренных входных условиях, а неправильным и непредусмотренным входным данным не придавать значения. Например, при тестировании задачи о треугольниках, лишь немногие смогут привести в качестве теста длины сторон 1, 2 и 5, чтобы убедиться в том, что треугольник не будет ошибочно интерпретирован как неравносторонний. Множество ошибок можно также обнаружить, если использовать программу новым, не предусмотренным ранее способом. Вполне вероятно, что тесты, представляющие неверные и неправильные входные данные, обладают большей обнаруживающей способностью, чем тесты, соответствующие корректным входным данным.

Необходимо проверять не только, делает ли программа то, для чего она предназначена, но и не делает ли она то, что не должна делать.

Это логически просто вытекает из предыдущего принципа. Необходимо проверить программу на нежелательные побочные эффекты. Например, программа расчета зарплаты, которая производит правильные платежные чеки, окажется неверной, если она произведет лишние чеки для работающих или дважды запишет первую запись в список личного состава.

Не следует выбрасывать тесты, даже если программа уже не нужна.

Эта проблема наиболее часто возникает при использовании интерактивных систем отладки. Обычно тестирующий сидит за терминалом, на лету придумывает тесты и запускает программу на выполнение. При такой практике работы после применения тесты пропадают. После внесения изменений или исправления ошибок необходимо повторять тестирование, тогда приходится заново изобретать тесты. Как правило, этого стараются избегать, поскольку повторное создание тестов требует значительной работы. В результате повторное тестирование бывает менее тщательным, чем первоначальное, т. е. если модификация затронула функциональную часть программы и при этом была допущена ошибка, то она зачастую может остаться необнаруженной.

Эту проблему почти полностью решают современные инструментальные средства тестирования, однако, она перешла в область организации труда разработчика.

Нельзя планировать тестирование в предположении, что ошибки не будут обнаружены.

Такую ошибку обычно допускают руководители проекта, использующие неверное определение тестирования как процесса демонстрации отсутствия ошибок в программе, корректного функционирования программы.

Вероятность наличия необнаруженных ошибок в части программы пропорциональна числу ошибок, уже обнаруженных в этой части.

Этот принцип, не согласующийся с интуитивным представлением, иллюстрируется рис. 2. На первый взгляд он лишен смысла, но, тем не менее, подтверждается многими программами. Например, допустим, что некоторая программа состоит из модулей или подпрограмм A и B. К определенному сроку в модуле A обнаружено пять ошибок, а в модуле B – только одна, причем модуль A не подвергался более тщательному тестированию.

Тогда из рассматриваемого принципа следует, что вероятность необнаруженных ошибок в модуле A больше, чем в модуле B. Справедливость этого принципа подтверждается еще и тем, что для ошибок свойственно располагаться в программе в виде неких скоплений. В качестве примера можно рассмотреть операционные системы IBM S/370. В одной из версий операционной системы 47 % ошибок, обнаруженных пользователями, приходилось на 4 % модулей системы.

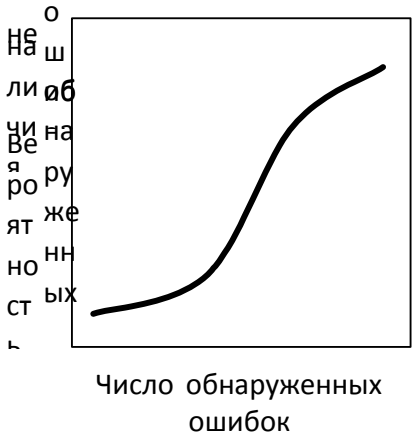

Рис. 2. Неожиданное соотношение числа оставшихся и числа обнаруженных ошибок

Интуитивно понятно, что ошибки могут группироваться в частях программы (модулях), разрабатываемых программистами низкой квалификации, или в модулях, в которых слабо проработана общая идея. Раннее выявление таких модулей – залог эффективного процесса тестирования.

Преимущество рассматриваемого принципа заключается в том, что он позволяет ввести обратную связь в процесс тестирования. Если в какой-нибудь части программы обнаружено больше ошибок, чем в других, то на ее тестирование должны быть направлены дополнительные усилия.

Тестирование — процесс творческий.

Вполне вероятно, что для тестирования большой программы требуется больший творческий потенциал, чем для ее проектирования. Выше было показано, что нельзя дать гарантию построения теста, обнаруживающего все ошибки. В дальнейшем будут обсуждаться методы построения хороших наборов тестов, но применение этих методов должно быть творческим.

1.5. Контрольные вопросы и задания

1. Сформулируйте характеристики хорошего теста.
2. Придумайте, каким образом два программиста, создающие одну программу, могут протестировать ее, не нарушая принципов тестирования.
3. Покажите и докажите, что абсолютное тестирование невозможно на конкретной программе.
4. Дайте определение понятию «тестирование».
5. Перечислите принципы тестирования.
6. Поясните, почему тестирование является деструктивным процессом.
7. Повторите эксперимент с кроссвордом, описанный в пункте 1.2.
8. Почему при тестировании необходимо желать, чтобы программа дала сбой?
9. Почему в участке кода, где было обнаружено больше всего ошибок, может содержаться еще большее их количество?
10. К чему ведет планирование теста в предположении отсутствия ошибок?
11. Приведите пример, когда отсутствие тестов, проверяющих, не делает ли программа лишних действий (что она не должна делать), может повлечь ошибки в работе?
12. Что дает проверка программы на неправильных входных данных?

13. Зачем необходимо хранить тесты, если программа уже выпущена?
14. В чем заключается принцип тестирования программы как белого ящика?
15. В чем заключается принцип тестирования программы как черного ящика?