Павлова Марианна Владимировна

ТЕСТИРОВАНИЕ ПРОГРАММ

Каждый из вас, конечно, слышал высказывание о том, что «В каждой программе есть, по крайней мере, одна ошибка». Каким бы странным ни казалось это высказывание из программистского фольклора, но в нем есть очень большая доля истины, конечно, если речь идет не об элементарной учебной программе, а о каком-то настоящем программном продукте. Чем длиннее и сложнее программа, тем менее мы можем быть уверены в том, что она работает правильно во всех случаях. Вопрос о том, как показать (или доказать), что программа «правильная», давно стал предметом изучения. Он волнует как создателей программ, так и их пользователей. Правильная программа не только должна делать то, для чего она предназначена, но и не должна делать того, что от нее не требуется. Хуже всего, если программа на неправильные данные выдает правдоподобные результаты. Обсудим некоторые вопросы, связанные с одним из видов проверки правильности программ – тестированием, и некоторые технологии тестирования, которые для этого применяются.

С начала 50-х годов, когда начали составляться первые программы, и до наших дней отмечен огромный рост стоимости программных продуктов. И если до начала 70-х годов в оценке труда программиста наблюдался явный оптимизм (была бы хорошая техническая база, вот тогда мы сможем решить все проблемы), то с начала 70-х годов все явственнее стал проявляться кризис, который затронул и область научных исследований, и производство. Этот кризис был обусловлен как усложнением программ, решающих реальные производственные задачи, так и уменьшением надежности этих программ. Появились уже не слухи, а статьи и официальные сообщения о «провалах» проектов, произошедших не из-за сбоя оборудования, а из-за ошибки программиста. Так из-за одной ошибки в программе на Фортране потерпела неудачу попытка запуска первого исследовательского американского корабля на Венеру, произошло несколько смертей в медицинских учреждениях, несколько авиакатастроф.

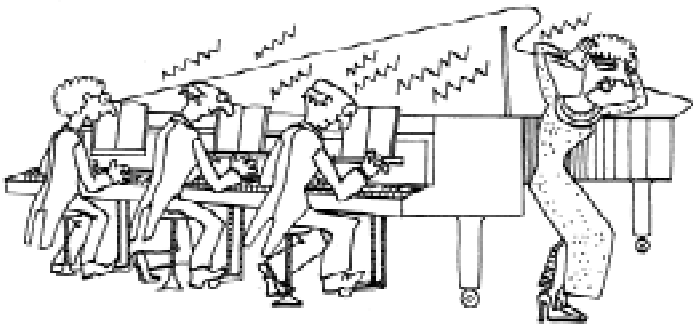
Вследствие этого встал вопрос о том, что нужно что-то менять в самом программировании. Эдсгер Дейкстра первым взглянул на программу не с точки зрения затрат времени и памяти, а обратил внимание на то, как программа написана, то есть на текст программы и действия, которые этот текст вызывает. Так появилось теоретическое направление в программировании и такое важное понятие как структурное программирование (structured programming). В структурном программировании параллельно рассматриваются два вопроса:

– как организовать работу над программой,

– как организовать документацию.

Оказалось, что при использовании структурного программирования можно решить многие проблемы:

– повышается в 5 раз производительность труда программиста;



В каждой программе есть, по крайней мере, одна ошибка.

– повышается ясность и читабельность программ;

– меньше требуется вносить изменений в уже созданную программу, а многочисленные исследования показали, что стоимость проверки на правильность одного изменения, внесенного в программу, в 100 раз выше, чем стоимость самого изменения;

– легче тестировать и документировать программы, а стоимость сопровождения программы экспоненциальна по отношению к длине программы. Если учесть, что программы постоянно усложняются и становятся все более длинными, то необходимость использования принципов структурного программирования становится очевидной. Большие программы, предназначенные для коммерческих целей или длительной эксплуатации многими пользователями принято называть *программными продуктами.*

Прежде всего необходимо разобраться в том, что такое тестирование и чем термин «тестирование программ» отличается от термина «отладка программ». Ясно, что и отладка, и тестирование - это некоторые мероприятия, которые проводятся для повышения надежности программного продукта. Несмотря на то, что проблема определения правильности программ стоит уже давно и ею занимались и продолжают заниматься многие ученые, до сих пор на вопрос, что какое тестирование можно услышать утверждение, что «тестирование – это процесс, подтверждающий правильность программы и демонстрирующий, что ошибок в программе нет». Однако любой человек, занимающийся программированием, знает, что продемонстрировать полное отсутствие ошибок в программе невозможно. Такое «определение» тестирования описывает что-то противоположное тому, что на самом деле следует понимать под термином «тестирование». С таким ошибочным представлением пытались бороться почти все классики программирования. Например, Никола Вирт писал «Экспериментальное тестирование программ может служить для доказательства наличия ошибок, но никогда не докажет их отсутствия». Большой специалист по тестированию Гленфорд Майерс дает такое определение:

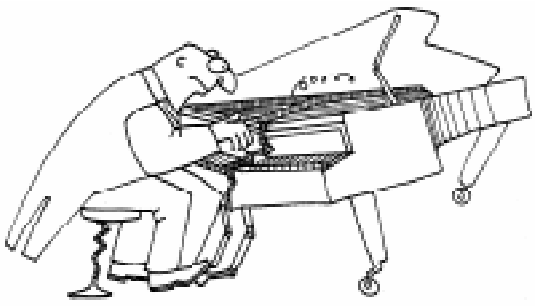
Тестирование – процесс выполнения программы с целью найти ошибки.

Тестирование процесс довольно необычный, а поэтому и трудный. Это процесс деструктивный (разрушительный), ведь цель тестировщика (иногда говорят «тестовика») – заставить программу сбиться, проработать неправильно. Трудность тестирования во многом психологическая. Тест считается успешным, если выявлена хотя бы одна ошибка. Однако, если программист сам составляет тесты для своих программ, то он подсознательно хочет показать, что его программа верна, поэтому его тесты с большой вероятностью будут выполняться правильно, «хитрые» тесты подсознательно будут блокироваться. Академик Андрей Петрович Ершов писал: «Слабый профессионализм большинства программистов затрудняет им преодоление наивного оптимизма в самооценке своего труда, при котором предусмотрительное ожидание ошибки постоянно заслоняется верой в близость конечной цели».

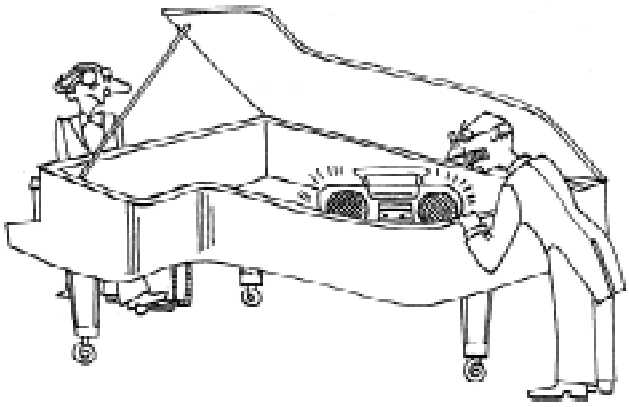
Из всего сказанного, следуя Г. Майерсу, сделаем основной вывод:

«Если ваша цель – показать отсутствие ошибок, вы их найдете не слишком много. Если же ваша цель – показать наличие ошибок, вы найдете значительную их часть».

В последние годы принято, чтобы при создании крупных программных продуктов тестирование проводил не сам программист, а специальная группа тестировщиков. При обнаружении ошибки тести-



Это процесс деструктивный (разрушительный)...



Если же ваша цель — показать наличие ошибок, вы найдете значительную их часть.

ровщик информирует программиста, тот исправляет ее и вновь программа тестируется. Этот ***процесс последовательного поиска и исправления ошибок называется отладкой программ.*** Если программа правильно ведет себя для солидного набора тестов, то хоть и нельзя утверждать, что она работает правильно всегда, но можно говорить, что неизвестно, когда она работает неправильно, и о некотором уровне уверенности в правильности программы.

Возникает вопрос, нельзя ли так составить наборы тестов, чтобы рассмотреть все возможные значения переменных, с которыми работает программа, и пройти по всем возможным путям, предусмотренным в программе?

К сожалению, такой подход совершенно неосуществим даже для самых простых программ.

Для доказательства невозможности перебрать все наборы допустимых данных, рассмотрим простой пример, предложенный Г. Майерсом.

Даны четыре целых числа. Требуется найти их среднее арифметическое. В результате работы такой программы должно получиться одно вещественное число. Как известно, на хранение одного целого числа отводится 2 байта или 16 бит. Таким образом, чтобы перебрать все возможные целые числа для тестирования этой программы, потребуется для одной переменной рассмотреть 2^{16} вариантов значений. Следовательно, для четырех переменных вариантов будет $2^{16} * 2^{16} * 2^{16} * 2^{16} = 2^{64}$. Если предположить, что программа исполняется 0,001 сек, то на исчерпывающее тестирование этой программы потребуется около 585 млн. лет.

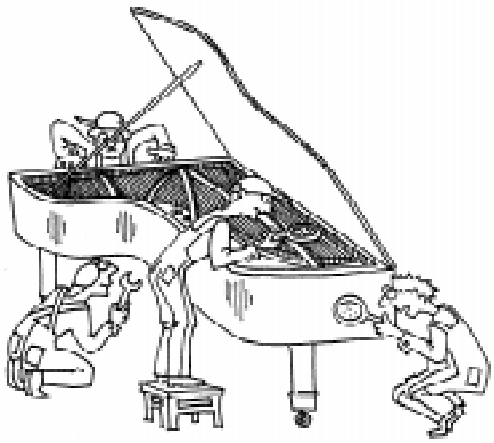
Аналогичная картина будет и при проверке всех переходов в программе.

Даже, если предположить, что мы когда-то сумеем провести исчерпывающее тестирование программы, то и тогда мы не сможем быть уверены в абсолютной правильности этой программы, так как, возможно, мы построили неправильный алгоритм ее решения. Например, если при поиске корня кубического из числа, мы воспользовались формулами, извлекающими корень квадратный, то программа не будет правильной, даже если мы проверим все выполнимые пути.

Отсюда следует основной вывод: ***исчерпывающее тестирование невозможно.***

Теперь рассмотрим относительную стоимость разработки программных продуктов на разных стадиях разработки.

Для относительно небольшой программы считается, что ее разработка включает задачи двух типов: проектирование и тестирование. При этом на этапе проектирования составляются некоторые виды тестов, которые неизбежно возникают на этом этапе. Остальные виды тестов касаются готового программного обеспечения.



... исчерпывающее тестирование невозможно...

Жизненным циклом программного продукта принято называть время от первого появления идеи о создании данного продукта до его ухода из эксплуатации. В настоящее время в связи с созданием больших программных систем большими группами разработчиков, внедрением этих систем в другие области человеческой деятельности, то есть в связи с их долгой эксплуатацией пользователями, принято выделять пять основных этапов жизненного цикла программного продукта. Они описываются так, как будто строго следуют один за другим – один заканчивается и начинается другой. На самом деле они в значительной степени перекрываются. Перечислим эти этапы:

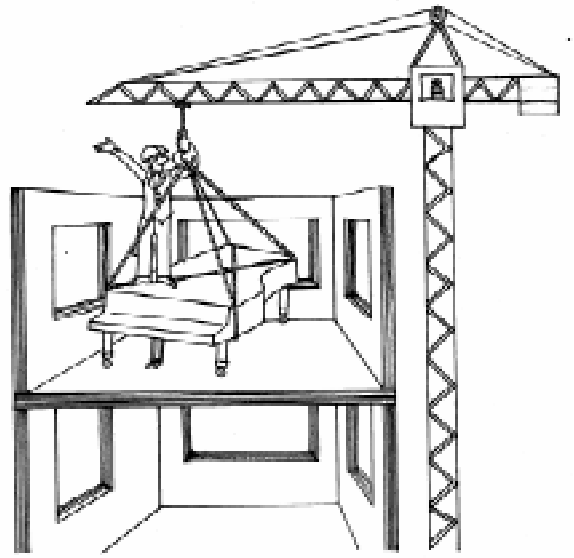
1. Планирование.
2. Проектирование.
3. Кодирование и написание документации.
4. Тестирование и исправление недостатков.
5. Сопровождение (после выпуска) и усовершенствование.

Рассмотрим относительную стоимость каждого из этих этапов:

Этапы стадии разработки:

1. Анализ требований	3%
2. Спецификация	3%
3. Проектирование	5%
4. Кодирование	7%
5. Тестирование	15%
Производственная стадия:	
Промышленное производство и сопровождение	67%.

Как видно их этих данных, дороже всего обходится сопровождение продукта после его выпуска. На втором месте по стоимости стоит тестирование – 45% от всей стоимости разработки. В процессе сопровождения продукта при исправлении ошибок и внесении усовершенствований в него большая часть затрат также приходится на тестирование. Продукт тестируют и исправляют практически на каждом этапе его жизненного цикла. При этом, чем дальше продвигается работа, тем дороже становятся и тестирование, и исправление ошибок.



...чем раньше найти и исправить ошибку, тем дешевле она обойдется.

Следовательно, ***чем раньше найти и исправить ошибку, тем дешевле она обойдется.***

Теперь рассмотрим перечисленные этапы разработки программного продукта, обращая особое внимание на тестирование.

ЭТАП ПЛАНИРОВАНИЯ

В команду планирования должны входить ведущие инженеры, маркетологи и руководители проекта. Они совместно вырабатывают общие характеристики продукта, в результате чего на свет появляется несколько документов, которые и определяют дальнейшую работу. В задачи этой группы входит: определение целей разработки, описание общего вида продукта, определение пользовательского интерфейса, выработка требований к надежности продукта и его производительности. Определяются стоимость продукта и затраты на его разработку. Документация этой группы появляется на свет, в основном, для того, чтобы поставить перед командой разработчиков конкретную цель. В процессе работы будут неизбежно вноситься изменения в этот план. Конечный результат может не соответствовать первоначальному описанию. Требования к

программному продукту, выдвигаемые на этом этапе, носят функциональный характер и должны быть выполнены обязательно, а сама практическая реализация этих требований – дело разработчиков. То, насколько подробно все описывается, зависит от разработчиков и от особенностей проекта. В требования на ранней стадии планирования входят и требования к аппаратному обеспечению. Требования к продукту определяемые отделом маркетинга, связаны с планируемой продажей продукта. Инженерам нужен перечень функций программного продукта и описание входных и выходных документов.

ТЕСТИРОВАНИЕ НА ЭТАПЕ ПЛАНИРОВАНИЯ

На данном этапе тестируются, естественно, не программы, а идеи. В этом случае к тестированию привлекаются все члены группы планирования, а вот группа тестировщиков в работе практически не участвует. При анализе и оценке требований к проекту и его функциональных характеристик стараются выяснить следующее:

- Адекватны ли эти требования?
- Полны ли они? Не упущены ли какие-то необходимые или просто полезные функции? Нельзя ли часть требований ослабить?
- Совместимы ли требования между собой?
- Выполнимы ли они? Не требуется ли более мощное аппаратное обеспечение, чем указано в документации?
- Разумны ли они? Найдено ли оптимальное равновесие между затратами и временем разработки и надежностью и производительностью? Здесь же расставляются приоритеты по требованиям.
- Поддаются ли они тестированию? Насколько легко можно определить, соответствует ли проектная документация требованиям к программному продукту?

ЭТАП ПРОЕКТИРОВАНИЯ

На этом этапе группа специалистов решает, как будут реализованы запланированные возможности продукта. Они разрабатывают внешний дизайн, то есть определяют интерфейс продукта для пользователя, и внутреннюю структуру. Эти части сильно связаны между собой и проектируются одновременно. К этому времени уже должен быть готов документ, который полностью описывает требования к создаваемому продукту. Кодирование может начаться только после того, как проектирование будет полностью завершено, однако некоторые низкоуровневые процедуры – те, к которым предъявляются особо строгие требования по скорости и ресурсам, – пишутся именно на этапе проектирования.

Внешний дизайн. Описание внешнего дизайна программного продукта включает полное описание его пользовательского интерфейса. Иногда его определяет пользователь, если у продукта есть конкретный заказчик. Работая над внешним дизайном, надо понимать, что даже при самой тщательно продуманной системе в процессе эксплуатации все равно обнаружатся некоторые недостатки.

Внутренняя структура. Внутренняя структура программного продукта определяет набор будущих программных модулей (программную архитектуру), структуру, взаимосвязи и принципы хранения и обработки данных (организацию данных) и алгоритмы работы программы.

- Проектирование программной архитектуры. Как правило, каждую задачу можно разбить на четкие подзадачи, которые, в свою очередь, разбиваются на более мелкие элементы и т.д. Такое разбиение называется *декомпозицией* и выполняется до тех пор, пока не будут выделены настолько мелкие, но самостоятельные элементы, что их можно реализовывать в виде процедур или программных модулей. Сложные программные продукты реализуются в виде набора полноценных программ, связанных между собой. Такие программы часто называют *процессами*. Хотя процессы могут работать и независимо, обычно, они взаимодействуют между собой. Например, используют общие данные.

Документацию, которая определяет принципы и правила взаимодействия процессов, называют *протоколом*.

• Проектирование организации данных. Разработчик структуры данных должен ответить на несколько принципиально важных вопросов:

– какие данные обрабатывает программа и какова их структура;

– как осуществляется доступ к данным;

– каковы принципы наименования данных;

– как хранятся данные.

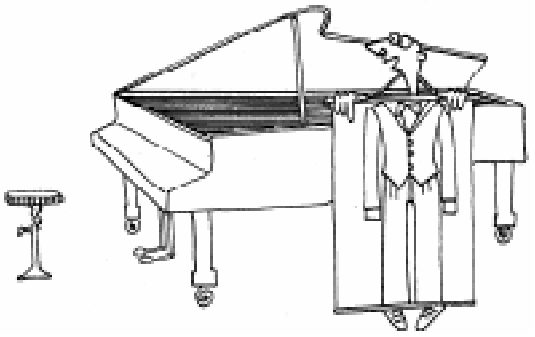
Если структура данных меняется, то все обращающиеся к ним модули при каждом изменении приходится переделывать. Поэтому полезно строить концепцию программы с точки зрения данных, к которым она обращается. В таком подходе программа рассматривается как то, что преобразует данные от входной информации до выходной. Модули же в этом случае рассматриваются как функциональные элементы, необходимые для различных операций.

• Описание алгоритмов. Группе программистов прежде всего необходимо продумать, как будут реализовываться поставленные задачи. Они выбирают оптимальные алгоритмы и описывают последовательность логических шагов, необходимых для решения каждой задачи.

• Моделирование. Чтобы лучше представить себе будущий программный продукт, на этапе проектирования может быть разработан его прототип – программная модель либо всего продукта, либо его части. Прототип строится быстро, с минимумом затрат, его легко модифицировать, и он не выполняет никакой реальной работы. Иногда моделирование выполняется для внешнего дизайна, иногда и для внутренней структуры системы. Цель создания прототипа – оценка функциональности будущей системы.

ТЕСТИРОВАНИЕ НА ЭТАПЕ ПРОЕКТИРОВАНИЯ

Как и на этапе планирования, кода еще нет, поэтому тестируются только идеи.



Тестирование на этапе проектирования.

Однако, на этом этапе идеи уже хорошо формализованы и подробно описаны. Специалисты по тестированию могут не участвовать в работе группы аналитиков, но для планирования системы будущих тестов, их участие очень полезно.

На этапе проектирования аналитики рассматривают следующие проблемы:

– действительно ли проект хорош, то есть будет ли созданный по нему продукт эффективным, хорошо тестируемым, легким в сопровождении и легко модернизируемым;

– соответствует ли проект требованиям;

– полон ли проект, описывает ли он все взаимосвязи между модулями и передачу данных между ними;

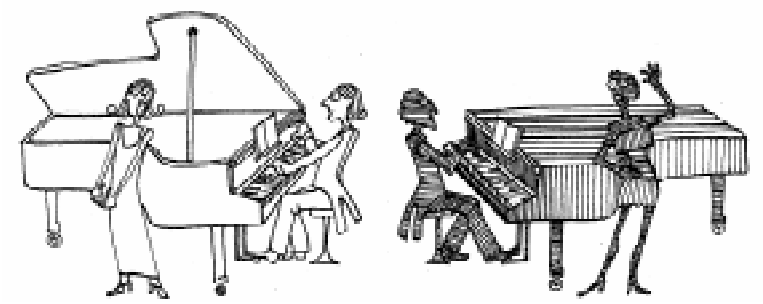
– реалистичен ли он, удовлетворяют ли имеющиеся аппаратные и программные средства потребностям проекта;

– хорошо ли описана в проекте подсистема обработки ошибок.

Особенно при нисходящем проектировании велико желание оставить обработку ошибок «на потом». Здесь и должны внимательно следить тестировщики за тем, чтобы правильно был определен уровень, на котором каждая из ошибок будет обрабатываться, чтобы ее возникновение в одном модуле не вело к ошибкам в других.

ТЕСТИРОВАНИЕ НА ЭТАПЕ КОДИРОВАНИЯ

На этом этапе программист сам пишет программы и сам их тестирует, поэтому методы тестирования, описанные в



...«тестированием белого ящика» ... в противоположность классическому понятию «черного ящика»...

этой части статьи, используются и для тестирования небольших программ.

Технология тестирования, которая применяется на этом этапе, называется *тестированием «стеклянного ящика»* (glass box). Иногда ее называют «тестированием белого ящика» (white box) в противоположность классическому понятию «черного ящика» (black box).

При тестировании «черного ящика» программа рассматривается как объект, внутренняя структура которого неизвестна. Тестировщик, не зная как работает сама программа, только вводит данные и анализирует результат. При построении тестов он выбирает такие данные и такие условия, которые, по его мнению, могут привести к нестандартным результатам. Интересны для него в этой ситуации такие представители в каждом классе входных данных, на которых с наибольшей вероятностью могут проявиться ошибки в тестируемой программе.

При тестировании «стеклянного ящика» ситуация совсем другая. Тестировщик (обычно это сам программист) разрабатывает тесты, основываясь на знании исходного кода, к которому он имеет неограниченный доступ. В результате использования этой технологии мы получаем следующие преимущества:

- Направленность тестирования. Программист может тестировать программу по частям, то есть написать специальные тестовые программки, которые вызывают тестируемый модуль и передают ему интересующие нас данные. Помодульно тестировать легче всего по этой технологии.
- Полный охват кода. Всегда хорошо видно, какие фрагменты участвуют в каждом тесте. Видно, какие ветви программы остались еще недостаточно оттестированными, следовательно, можно подобрать такие тесты, в которых они будут выполняться.
- Управление потоком. Программист всегда знает, какая функция должна выполняться следующей и каким должно быть ее текущее состояние. Чтобы убедиться в том, что программа работает так, как ему нужно, программист может включить в нее отладочные команды, которые отобразят информацию о ходе выполнения программы. Можно для этой цели воспользоваться отладчиком – специальным средством, которое позволяет отслеживать и менять последовательность выполнения команд программы, показывать значения переменных и их адреса в памяти и др.
- Отслеживание целостности данных. Программисту известно, какая часть программы должна изменять каждый из элементов данных. Отслеживая состояние данных (с помощью, например, отладчика), можно выявить такие ошибки, как изменение данных не теми модулями, их неверная или неудачная организация.

Рассмотрим основные концепции тестирования «стеклянного ящика», сравнивая их с соответствующими концепциями тестирования «черного ящика».

СТРУКТУРНОЕ ТЕСТИРОВАНИЕ – ФУНКЦИОНАЛЬНОЕ ТЕСТИРОВАНИЕ

Структурное тестирование является одним из видов тестирования «стеклянного ящика». Его главная идея – правильный выбор тестируемого программного пути. Этот вид тестирования имеет серьезную теоретическую базу, он легко поддается математическому моделированию.

В противоположность ему, функциональное тестирование относится к кате-

гории тестирования «черного ящика». Каждая функция программы тестируется путем ввода в нее входных данных и анализа выходных.

Нельзя сказать, что один метод хорош, а другой плох. Применение каждой из технологий позволяет выявить ошибки, пропущенные другой.

ТЕСТИРОВАНИЕ ПРОГРАММНЫХ ПУТЕЙ: КРИТЕРИИ ОХВАТА

Путь выполнения программы – последовательность команд, которые она выполняет от старта до завершения. Объектом тестирования не обязательно должен быть полный путь. Можно рассматривать его отдельные участки.

Как мы знаем, протестировать все пути невозможно, поэтому необходимо выделить из всех возможных путей те группы, которые нужно протестировать обязательно. Для отбора путей пользуются специальными критериями, называемыми *критериями охвата.* Эти критерии определяют реальное (хотя иногда и огромное) количество тестов. Критерии охвата иногда называют *логическими критериями охвата* или *критериями полноты.*

Рассмотрим три основных критерия полноты: охвата строк, охвата ветвлений и охвата условий. Когда тестирование организовано в соответствии с этими критериями, то о нем говорят как о *тестировании путей.*

Критерий охвата строк – наиболее слабый из всех. Его требование состоит в том, чтобы каждая строка программы выполнялась хоть один раз. Для серьезного тестирования программного продукта этого далеко недостаточно. Если строка содержит оператор принятия решения, то должны быть проверены все управляющие решением значения.

Рассмотрим простой пример, приведенный в [1]. Пусть есть фрагмент кода:

```
If (A<B) and (C=5)
then <сделать НЕЧТО>;
D:=5;
```

Чтобы проверить этот код, нужно проанализировать следующие 4 варианта:

A) A<B и C=5 {НЕЧТО выполняется и D присваивается значение 5}

B) A<B и C<>5 {НЕЧТО не выполняется и D присваивается значение 5}

C) A>=B и C=5 {НЕЧТО не выполняется и D присваивается значение 5}

D) A>=B и C<>5 {НЕЧТО не выполняется и D присваивается значение 5}

Для выполнения всех трех строк кода достаточно проверить только один вариант (A).

При более основательном тестировании – по *критерию охвата ветвлений* – программист проверяет вариант (A) и еще один из трех остальных вариантов. Смысл этого способа в том, что проверяются действие программы при выполнении условия в операторе if и при невыполнении. В этом случае программа проходит не только все строки кода, но и все возможные ветви.

Еще более строгим является *критерий охвата условий.* По этому критерию нужно проверить все составляющие каждого логического условия. В нашем примере это означает проверку всех четырех перечисленных вариантов.

Тестирование путей программы считается завершенным, когда выбранный критерий охвата полностью выполнен. Для автоматизации этого процесса разработаны специальные программы, анализирующие программный код и вычисляющие количество подлежащих тестированию путей, а затем подсчитывающие, сколько их уже проверено. Такие программы называют *средствами мониторинга охвата.*

Одного тестирования путей недостаточно для эффективного выявления ошибок. В 1975 году был приведен простой, но ставший классическим пример того, что проход строки кода еще не означает выявления имеющейся в ней ошибки. Рассмотрим строку программы: A:=B/C.

Если C<>0, то строка проработает нормально, если C=0, то программа либо сообщит об ошибке и прекратит работу, либо «зависнет». Разница же между вариантами не в пути, а в данных.

Программный путь называется *чувствительным к ошибкам*, если при его прохождении ошибки могут проявиться.

Программный путь называется *обнаруживающим ошибки*, если при его прохождении ошибки обязательно проявятся.

Любой путь, проходящий через приведенную строку, является чувствительным. Ошибка выявляется только при нулевом значении С. Для выявления таких ошибок технология «черного ящика» подходит больше, чем анализ программного кода.

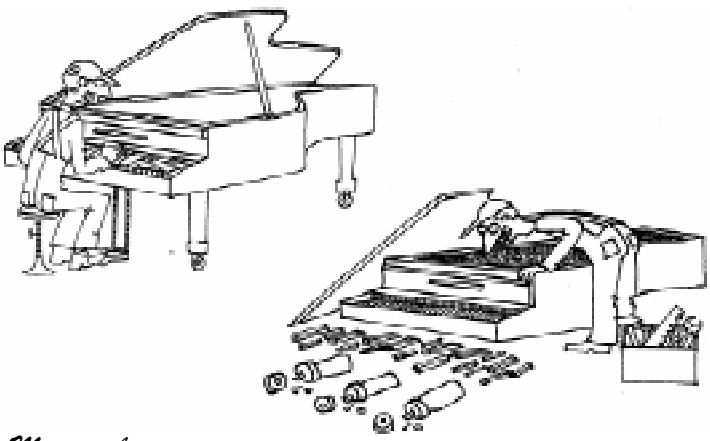
ТЕСТИРОВАНИЕ ЧАСТЕЙ – ТЕСТИРОВАНИЕ ЦЕЛОГО

Любая система разрабатывается по частям – как набор процессов или модулей. Ее можно таким же образом тестировать – сначала отдельные части, а затем их взаимодействие. Такая стратегия называется *восходящей*.

Выяснив, что отдельные части системы работают нормально, приступают к тестированию их совместной работы. Часто оказывается, что вместе работать части отказываются. Например, если перепутать (поменять местами) параметры вызываемой функции, то при выполнении вызова произойдет ошибка. Эта ошибка будет выявлена при проверке совместной работы двух функций – вызывающей и вызываемой. Тестирование совместной работы программных модулей называют *интеграционным*. В ходе такого тестирования модули сначала объединяются в пары, затем в тройки, затем в еще большие блоки, и так до тех пор, пока все модули не будут объединены в единую систему.

Восходящее тестирование – хороший способ локализации ошибок. Если ошибка обнаружена при тестировании единственного модуля, то она именно в нем, и искать ее нужно только в нем, не рассматривая код всей программы. Если же ошибка обнаружена при совместной работе двух ранее оттестированных модулей, значит дело в их интерфейсе. Вообще при восходящем тестировании программист концентрируется на конкретном модуле или на передаче данных между двумя модулями, поэтому тестирование проводится более тщательно.

Основным недостатком этого вида тестирования является необходимость написания специального кода-оболочки, вызывающего данный модуль. Если же этот модуль вызывает другой, то приходится писать *заглушку*. Заглушка – это имитация вызываемой функции. Она должна выдавать те же результаты, что и вызываемая функция, но больше ничего не делать. Конечно, написание заглушек и оболочек сильно замедляет работу, а для конечного программного продукта они не пригодятся. Если же все таки пойти на их написание, то мы получим прекрасный эффективный инструмент для тестирования, поскольку набор заглушек и оболочек можно будет использовать повторно при каждом изменении в тексте программы.



Тестирование частей — Тестирование целого.

В противоположность восходящему тестированию, *стратегия целостного тестирования* предполагает, что до полной интеграции системы ее отдельные модули не подвергаются тщательному тестированию. Преимущество этого подхода в том, что является недостатком для восходящего тестирования – отсутствие необходимости писать оболочки. Но в этом способе тестирования есть свои существенные недостатки:

1. Очень трудно выявить источник ошибки. Поскольку

ни один из модулей не проверяется детально, то в большинстве из них есть ошибки. Проблема в том, чтобы понять, какая из ошибок в вовлеченных в процесс модулях привела к полученному результату. Когда накладываются ошибки нескольких модулей, то ситуацию трудно повторить для проверки. Кроме того, ошибка в одном из модулей может полностью заблокировать для проверки другой.

2. Трудно организовать исправление ошибок. Это касается проектов, над которыми работает группа программистов.

3. Процесс тестирования плохо автоматизирован. То, что в начале казалось преимуществом, а именно отсутствие необходимости писать заглушки и оболочки, на самом деле становится недостатком. Программа постоянно меняется, ее приходится тестировать снова и снова.

НИСХОДЯЩЕЕ ТЕСТИРОВАНИЕ – ВОСХОДЯЩЕЕ ТЕСТИРОВАНИЕ

Существует еще один способ оттестировать программу по частям. Только направление движения меняется. Сначала тестируется самый высокий уровень иерархии модулей, а от него тестирование постепенно спускается все ниже и ниже. Такая технология называется *нисходящей.* При нисходящем тестировании отпадает необходимость писать оболочки, но заглушки остаются. По мере тестирования они постепенно заменяются на реальные модули.

СТАТИЧЕСКОЕ ТЕСТИРОВАНИЕ – ДИНАМИЧЕСКОЕ ТЕСТИРОВАНИЕ

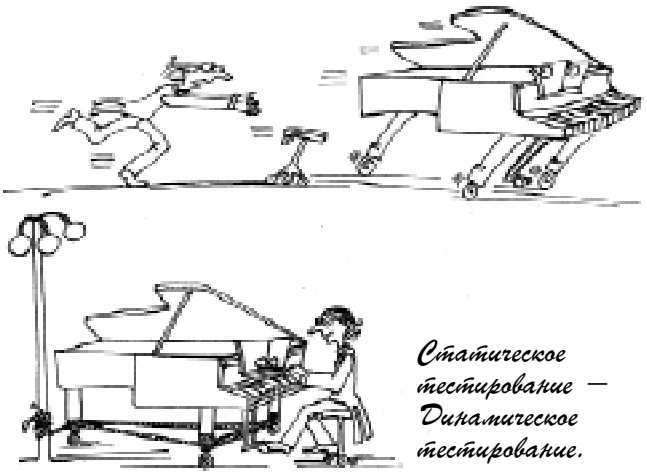
При статическом тестировании программный код вообще не исполняется – он тестируется только путем логического анализа.

Две описанные раньше стратегии – тестирование «стеклянного ящика» и тестирование «черного ящика» являются динамическими. Программа запускается, вводятся данные, программист или тестировщик анализирует результаты. Разница только в том, на какой информации основывается подбор тестов.

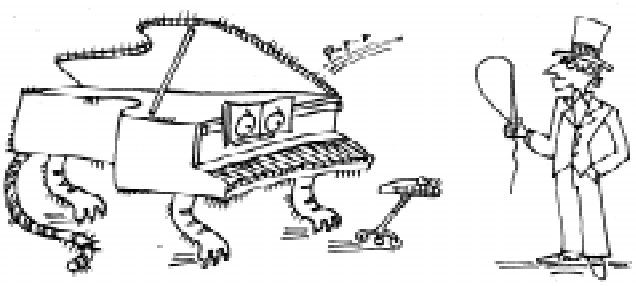
Для статического тестирования существует много инструментальных средств. Самое известное из них – компилятор. Встретив синтаксическую ошибку или недопустимую операцию, компилятор выдает соответствующее сообщение. Ряд полезных сообщений, например, о повторяющихся именах переменных и других объектов, ссылках на необъявленные переменные и функции, выдает компоновщик. Статический анализ программы может выполняться и людьми. Они читают исходный код, обсуждают его на рецензионных совещаниях или поручают анализ кода одному тестировщику. При внимательной проверке выявляется большое количество ошибок. Эта работа очень однообразная и скучная, но чрезвычайно полезная.

НАМЕРЕННЫЕ ОШИБКИ: ПСЕВДООТЛАДКА И МУТАЦИОННОЕ ТЕСТИРОВАНИЕ

Существует еще одна интересная технология тестирования, которая состоит в том, что в программу намеренно внедряется ряд ошибок. Зная, что ошибки есть наверняка, тестировщик будет более внимательным. Основной целью такого тестирования является оценка эффективности проводимых тестов. Если в програм-



Статическое тестирование — Динамическое тестирование.



...мутационное тестирование.

му внедрили 100 ошибок, а нашли лишь 20 из них, значит, в программе осталось 80 % невыявленных ошибок. Такая технология получила название *псевдоотладка.*

Еще одним способом оценки адекватности проводимых тестов является *мутационное тестирование.* При этом в программу вносится небольшое изменение (мутация). Если при тестировании это изменение не проявилось, значит, тесты плохо подобраны.

РЕГРЕССИОННОЕ ТЕСТИРОВАНИЕ

Основной работой тестировщиков является регрессионное тестирование. Этот термин имеет два значения, но идея одна – повторное исполнение разработанных тестов.

1. Первая идея состоит в тестировании при исправлении ошибок. Если один из тестов обнаружил ошибку, программист внес исправления, то нет уверенности, что теперь все будет в порядке. Нужно проверить, как будет работать программа на том же тесте, чтобы убедиться, что ошибки больше нет. Более того, можно провести серию тестов для окончательной проверки фрагмента, где была обнаружена ошибка. Это и есть регрессия. Часто в набор регрессионных тестов включают все тесты, на которых когда-либо проявились ошибки. Регрессионные тесты проводятся каждый раз после любого изменения в программе.

2. Вторая идея состоит в проверке того, чтобы исправления в одной части программы не испортили другую. В этом случае тестируется целостность всей программы.

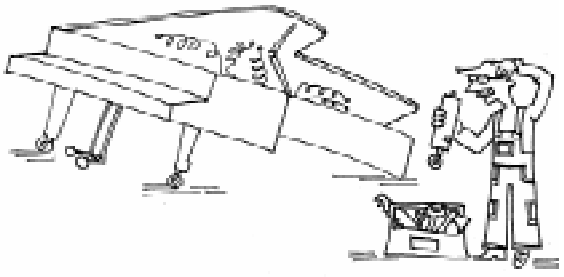
Некоторое время спустя после начала тестирования программного продукта формируется *библиотека регрессионных тестов*. Это полный набор тестов, охватывающий всю программу и выполняющийся каждый раз, когда программисты сдают ее очередную рабочую версию.

Лучше всего, если тесты полностью автоматизированы. В этом случае тоже, конечно, затрачивается некоторое время, но вся процедура тестирования гораздо менее трудоемка.

ДАЛЬНЕЙШАЯ РАБОТА НАД ПРОДУКТОМ

Дальнейшая работа над продуктом идет по следующей схеме.

Кодирование завершено. Программа переходит в руки тестировщиков. При поступлении каждой новой версии программного продукта нужно прежде всего проверить, достаточно ли она стабильна. Если версия не выдерживает малейшей провокации, то ее можно не тестировать вовсе. Такое первое беглое знакомство с версией называют *приемочным* или *квалификационным.* Приемочные тесты должны быть короткими. В них проверяются только основные функции и основные данные. Если уж программа не проходит такой тест, то она точно никуда не годится. Если приемочное тестирование прошло успешно, то начинается собственно тестирование. Тестировщики выявляют ошибки, составляют по ним отчеты, передают продукт программистам на исправление. Через некоторое время новая версия про-



...исправления в одной части программы не испортили другую.

граммы снова попадает к тестировщикам. На следующем круге тестирования выявляются старые ошибки, которые не были обнаружены на предыдущем этапе, и новые, появившиеся в процессе исправлений. Существуют интересные статистические данные об эффективности исправления найденных ошибок:

• Если для исправления ошибки нужно изменить не более 10 операторов, то вероятность того, что это будет сделано правильно с первого раза, составляет 50 %.

• Если для исправления ошибки требуется изменить около 50 операторов, то вероятность того, что это будет правильно сделано с первого раза, составляет лишь 20 %.

Проблема не только в том, что программист с первого раза может не исправить ошибку полностью. Хуже то, что при исправлении могут быть привнесены *побочные эффекты.* Исправление одной ошибки может привести к появлению другой или других. Возможна ситуация, когда одна ошибка скрывает другую, и та проявляется только после устранения первой.

ХАРАКТЕРИСТИКИ ХОРОШЕГО ТЕСТА

Хороший тест должен удовлетворять следующим критериям:

• Существует обоснованная вероятность выявления ошибки. Целью тестирования является поиск ошибок. Поэтому, придумывая тестовые примеры, необходимо постараться обнаружить все возможные случаи сбоя программы или ее неправильного поведения. Если в программе может произойти определенная ошибка, то должен быть тест, который позволит ее поймать.

• Набор тестов не должен быть избыточным. Если два теста предназначены для выявления одной и той же ошибки, не следует выполнять их оба.

• Тест должен быть лучшим в своей категории. В группе похожих тестов одни могут быть эффективнее других. Поэтому, выбирая тест, нужно взять тот, который с наибольшей вероятностью выявит ошибку. Ясно, что на границах каждого диапазона значений параметров программ ошибки проявляются чаще, чем на средних значениях этого же диапазона. Поэтому для тестов больше подходят граничные значения.

• Тест должен быть не слишком сложен и не слишком прост. Огромный и сложный тест долго создавать, трудно понять и трудно выполнить. Поэтому лучше всего придерживаться золотой середины, разрабатывая простые, но все же не совсем элементарные тестовые примеры.

Как узнать, прошла ли программа тест?

Даже неверные выходные данные на экране или на бумаге тестировщик может случайно пропустить, не говоря уже о том, что результат ошибки может оказаться скрытым.

• Разрабатывая тест, нужно подробно описать ожидаемые выходные данные или реакцию программы.

• Следует разрабатывать тесты так, чтобы объем выходных данных был минимальным. В огромной распечатке трудно найти неправильную цифру.

КЛАССЫ ЭКВИВАЛЕНТНОСТИ И ГРАНИЧНЫЕ УСЛОВИЯ

Одними из ключевых понятий теории тестирования являются классы эквивалентности и граничные условия.

Если от выполнения двух тестов ожидается один и тот же результат, они считаются эквивалентными. Группа тестов представляет собой один класс эквивалентности, если выполняются следующие условия:

– все тесты предназначены для выявления одной и той же ошибки;

– если один из тестов выявит ошибку, и остальные, скорее всего, тоже это сделают;

– если один из тестов не выявит ошибки, и остальные, скорее всего, тоже этого не сделают.

Кроме этих абстрактных критериев, необходимы еще и практические навыки,

позволяющие отнести к одному классу конкретную группу тестов. Вот на чем может основываться этот отбор.

• Тесты включают значения одних и тех же входных данных. Для их проведения выполняются одни и те же операции программы.

• В результате всех тестов формируются значения одних и тех же выходных данных.

• Либо ни один из тестов не вызывает выполнения блока обработки ошибок программы, либо выполнение этого блока вызывается всеми тестами группы.

ПОИСК КЛАССОВ ЭКВИВАЛЕНТНОСТИ

Поиск классов эквивалентности – процесс субъективный. Два человека, анализирующих одну и ту же программу, составят различные перечни классов.

Вот несколько рекомендаций для поиска классов эквивалентности:

• Не забывайте о классах, охватывающих заведомо неверные или недопустимые входные данные.

• Организуйте формируемый перечень классов в виде таблицы или плана.

• Определите диапазоны числовых значений.

• Для полей или параметров, принимающих фиксированные перечни значений, выясните, какие из значений входят в перечень.

• Проанализируйте возможные результаты выбора из списков и меню.

• Поищите переменные, значения которых должны быть равными.

• Поищите классы значений, зависящих от времени.

• Выявите группы переменных, совместно участвующих в определенных вычислениях, результат которых ограничивается конкретным набором или диапазоном значений.

• Посмотрите, на какие действия программа отвечает эквивалентными событиями.

Например, если программа должна принимать числа от 1 до 99, существует как минимум четыре класса эквивалентных тестов.

• Допусти́м ввод чисел от 1 до 99.

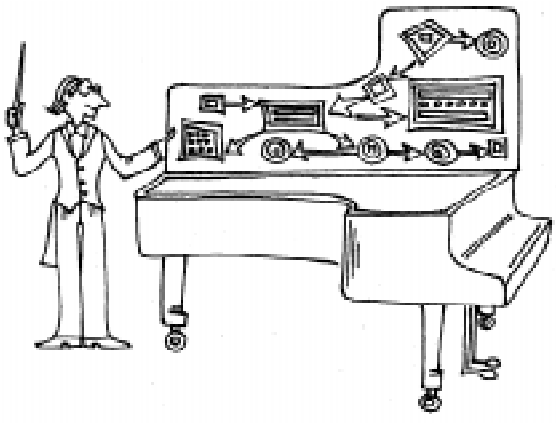
• Любое число меньше 1 слишком мало. Данный диапазон включает 0 и все отрицательные числа.

• Любое число больше 99 слишком велико.

• Если введена нечисловая информация, она не принимается.

Обычно классов эквивалентности оказывается много, поэтому не обойтись без удобного и продуманного способа организации собранной информации. Чаще всего на практике используются два подхода. Обычно вся информация либо сводится в большую таблицу, либо представляется в форме плана. Оба способа организации информации – и таблица, и план – достаточно удобны. У каждого из них есть преимущества и недостатки. Табличный формат информации более понятен, его легче читать, и одним взглядом можно охватить больше информации. Планы выглядят компактнее, в них легче разбивать информацию на составляющие. Однако и повторения при использовании планов случаются чаще.

Для каждого класса эквивалентности достаточно провести один-два теста. Лучшими из них будут те, которые проверяют значения, лежащие на границах клас-



...классов эквивалентности оказывается много, поэтому не обойтись без удобного и продуманного способа организации собранной информации.

са. Они могут быть наибольшими, наименьшими, быстрейшими, кратчайшими, самыми громкими, самыми красивыми – но в любом случае это должны быть предельные значения параметров класса. Неправильные операторы сравнения (например, > вместо >=) вызывают ошибки только на граничных значениях аргументов. В то же время программа, которая сбоит на промежуточных значениях диапазона, почти наверняка будет сбоить и на граничных. Необходимо протестировать каждую границу класса эквивалентности, причем с обеих сторон. Программа, которая пройдет эти тесты, скорее всего, пройдет и все остальные, относящиеся к данному классу.

В заключение отметим, что при разработке реальных коммерческих проектов проводятся также:

– бета-тестирование (продукт проверяется будущими пользователями);

– тестирование в условиях гонок (продукт заставляют работать при максимальном ускорении и максимальном замедлении техники);

– нагрузочные испытания (при максимальной загрузке файлов и техники);

– конфигурационное тестирование (проверяется работа с различным программно-аппаратным окружением, особенно с различными видами принтеров);

– тестирование документации и некоторые другие виды тестирования.

Литература:

1. *Сэм Канер и др.* Тестирование программного обеспечения. ДиаСофт, 2000.

2. *Майерс Г.* Искусство тестирования программ. М.: Финансы и статистика, 1982.

3. *Майерс Г.* Надежность программного обеспечения. М.: Мир, 1980.

4. *Ван Тассел.* Стиль, разработка, эффективность, отладка и испытание программ. М.: Мир, 1981.

5. *Гудман С., Хидетниеми С.* Введение в разработку и анализ алгоритмов. М.: Мир, 1981.

6. *Зиглер К.* Методы проектирования программных систем. М.: Мир, 1985.

7. *Брукс Ф. П. мл.* Как проектируются и создаются программные комплексы. М.: Наука, 1979.

8. *Фокс Дж.* Программное обеспечение и его разработка. М.: Мир, 1982.

9. *Безбородов Ю.М.* Индивидуальная отладка программ. М.: Наука, 1982.

Павлова Марианна Владимировна, старший преподаватель кафедры информатики математико-механического факультета Санкт-Петербургского Университета.