В. В. Пупышев, Пример тестирования программы о лабиринте, *Вестн. Удмуртск. ун-та. Матем.*, 2005, выпуск 1, 225–234

ВЕСТНИК УДМУРТСКОГО УНИВЕРСИТЕТА

МАТЕМАТИКА 2005. № 1

ИНФОРМАТИКА

УДК 519.681

В. В. Пупышев

ПРИМЕР ТЕСТИРОВАНИЯ ПРОГРАММЫ О ЛАБИРИНТЕ

Рассказывается об основных приёмах и методах тестирования в целях обнаружения ошибок. Приведён простой пример и указано, почему нет способа проверить его реализацию на наличие всех ошибок. На этом примере показаны основные проблемы тестирования программных систем.

Ключевые слова: тестирование, качество, пример, ошибка, лабиринт, программа.

Введение

На сегодняшний день существует огромное количество программных систем. Все пользователи знают, что любая программа содержит какие-то ошибки и недоработки. Говорят, что качество системы плохое.

Наиболее общее определение качества таково: *качество системы* — способность удовлетворять потребности.

Если программная система представляет собой отображение $P : W \to S$, где W — потребности, S — их удовлетворение, тогда P — отношение[1] между элементами множества потребностей[2] W и множества S возможных ответов на потребности[3].

Одним из основных моментов проверки качества системы является тестирование.

Тестирование — процесс выполнения программ в целях обнаружения факта наличия ошибок.

Это классическое определение тестирования, принадлежащее Майерсу [4]. Исходя из нашего понятия качества, получаем другое определение.

Тестирование — процесс проверки того, как система отвечает на потребности.

Есть много критериев качества, но все они косвенные. Единственным реальным критерием является «живая» эксплуатация, но далеко не всегда возможно передавать программу конечному пользователю без проверки. Для такой проверки и служит тестирование.

Тест — единичная потребность.

---

[1] Program.

[2] to want — хотеть.

[3] to satisfy the requirements — удовлетворять потребности.

Тестовый пример — пара: тест и ответ на него.

Реакция системы — пара: тест и ответ на него системы.

Тестовое покрытие — множество некоторых проверенных реакций системы.

Полное тестовое покрытие — множество всех реакций системы.

В работе будем рассматривать только системы с однозначной реакцией, где одинаковым потребностям соответствуют одинаковые ответы. Этому соответствует математическое понятие функции.

Задачи данной работы: показать основные современные способы создания тестов и предложить свои способы.

§ 1. Современное положение дел в тестировании

В технологии тестирования выделяют несколько уровней. На разных уровнях используют определённые виды тестирования, в которых применяются всевозможные методы. Рассмотрим подробнее методы, виды и уровни.

Методы тестирования

Тестирование методом **чёрного ящика** — тестируются только реакции на потребности, как устроена система не имеет значения.

Тестирование методом **белого ящика** — тестируется система с целью проверить правильную работу на всех программных путях.

Виды тестирования

Выделяют несколько различных видов тестирования:

- структурное — тестируются отдельные компоненты системы;
- сборочное — тестируются крупные части системы (подсистемы);
- функциональное — проверяется соответствие техническому заданию;
- регрессионное — проверка изменяемости, модифицируемости системы;
- нагрузочное — проверка системы при работе в реальных условиях эксплуатации;
- стрессовое — корректность работы системы при предельных нагрузках.

Для каждого вида тестирования нужны специальные тесты:

структурные — структурное и сборочное тестирование;

комплексные — функциональное тестирование;

регрессионные — регрессионное тестирование;

нагрузочные — нагрузочное и стрессовое тестирование.

Уровни тестирования

Принято разделять тестирование по уровням задач и объектов на разных стадиях и этапах разработки программного обеспечения (ПО):

- тестирование частей ПО (модулей, компонентов) в целях проверки правильности реализации алгоритмов;
- функциональное тестирование подсистем и ПО в целом в целях проверки степени выполнения функциональных требований к ПО;
- нагрузочное тестирование (в том числе стрессовое) для выявления характеристик функционирования ПО при изменении нагрузки (интенсивности обращений к нему, наполнения базы данных и т. п.).

У последних двух уровней есть ещё дополнительные отличительные черты не формального характера.

Функциональное тестирование рекомендуется проводить отдельной группой тестировщиков, не подчиненной руководителю разработки. Для выполнения нагрузочного тестирования требуются высококвалифицированные тестировщики и дорогостоящие средства автоматизации экспериментов.

§ 2. Получение тестов

Всё это многообразие уровней, видов и методов тестирования требуется только для того, чтобы обойти основную проблему тестирования: *Нет возможности проверить реакцию системы для каждой потребности.*

Получение тестов на этапе проектирования

На каждом этапе проектирования возможно получить набор тестов.

Вот краткая рекомендуемая последовательность действий по построению тестов [1].

Требования. На этом этапе создаются *комплексные тесты.* Они появляются как примеры для иллюстрации требований.

Проектирование. А комплексные тесты с предыдущего этапа используются для проверки адекватности проекта поставленной

задаче. Тестирование статическое ручное, методом белого ящика.

На этом этапе создаются *компонентные*, *нагрузочные тесты* и наращиваются *комплексные тесты*.

Реализация. Все тесты к этому времени готовы. Их прохождение есть единственное условие перехода к следующему этапу.

Эксплуатация и сопровождение. Выявление недостатков на этом этапе даёт новые **тесты**.

Такой метод построения тестов рекомендуется, поскольку при создании тестов проверяются спецификации и качество их увеличивается.

Получение тестов из спецификаций

Когда спецификации описаны достаточно точно, то из них можно получить неплохой набор тестов. К сожалению, написание хороших и полных спецификаций обычно работа более сложная, чем написание программы. Это ведёт к значительному удорожанию системы.

Получение тестов по программным путям

При реализации описанных спецификаций нередко возникают ошибки. Поэтому нужно проверять программу «изнутри». Тесты получаются при отслеживании всех логических путей в программе.

§ 3. Проблемы получения тестов

Даже применение всех описанных выше способов получения тестов вместе не даёт уверенности в отсутствии ошибок в программе. Разберёмся с этой проблемой тестирования глубже. Для этого рассмотрим следующий пример.

П р и м е р 1. **Задача**. Дан лабиринт, можно ли его пройти.

Технические требования: лабиринт задаётся квадратной матрицей 10×10, состоящей из 0 и 1.

Написать программу, которая выдает 1, если есть цепочка из 0 со следующими условиями:

- есть цепочка до соседа нижнего правого угла;
- в нижнем правом углу стоит 0;

либо выдать 0, если такой цепочки нет.

Сосед — ячейка, находящаяся в том же ряду или столбце рядом.

Движение начинается в левом верхнем углу.

Техническое задание: входные данные записываются в файле *input.txt*, ответ нужно сохранять в файле *output.txt*.

Пример:

```
input.txt
0000000001
0111111111
0000000001
1111111101
0000000001
1010101011
1110101111
0000100001
1011110100
0000000100

output.txt
1
```

Пусть есть программа, решающая эту задачу.

Рассмотрим тестирование методом чёрного ящика. Для создания полного покрытия тестами необходимо подать на вход все возможные лабиринты размером 10×10. Но их количество 2^{100} — это невообразимо огромное число.

Метод белого ящика тоже подходит не для всех случаев.

Это убедительно доказывает Канер с соавторами [2], приводя в качестве примера небольшую программу, описанную Майерсом еще в 1979 году и состоящую из цикла с несколькими операторами IF (примерно 20 строк кода), на полное тестирование которой ушел бы миллион лет (100 триллионов путей выполнения).

Вывод. Проверить реакцию системы на все потребности невозможно.

На практике дело обстоит ещё хуже, входных данных намного больше.

Вернёмся к примеру 1. Пусть в файле *input.txt* записано:

```
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
```

В спецификации задачи сказано, что должны быть только 0 и 1. Но как должна вести себя программа в этом случае? Традиционно

считается, что программа должна сообщить об ошибке во входных данных.

Вывод. В правильной системе множество потребностей W должно быть равно множеству всевозможных объектов — универсу $\mathbb{U}$. Поскольку удовлетворить все потребности нереально, множество ответов S будет содержать подмножество отказов E.

Иногда складывается впечатление, что можно в спецификации описать реакции на все потребности, но для программных систем это впечатление обманчиво.

Описать все потребности невозможно (потребности бесконечны, программные системы конечны и ограничены производительностью), поэтому существует подмножество неописанных потребностей $\overline{A}$, $A \cup \overline{A} = \mathbb{U}$, $A \cap \overline{A} = \varnothing$.

Значит, ограничиться только тестами, полученными из спецификаций, недостаточно.

Вывод. Для неописанных потребностей приходится смотреть шире множества спецификаций.

Из практики программирования известно, что при реализации системы возможно возникновение ошибок из-за ошибки программиста или ошибки в системе программирования.

Мы не можем проверить все входы по спецификации.

Вывод. Необходимо проверить множество «проблемных входов».

Проблемный вход — элемент спецификации, на котором достаточно часто делаются ошибки при реализации систем.

Формализовать понятие проблемного входа нельзя, поэтому тестирование — искусство даже в большей степени, чем программирование.

Существуют автоматы для тестирования. Но применять их как способ перебрать больше вариантов малоэффективно. И вот почему.

Пусть A' — множество проблемных входов, тогда $|A'| \ll |A|$. Значит, при автоматическом тестировании вероятность случайно проверить *проблемный вход* близка к 0.

Другая причина. Рассмотрим автомат для тестирования программы, решающей задачу примера 1. Пусть автомат работает следующим образом: подаёт программе на вход очередной лабиринт и проверяет ответ. Следующий вывод показывает бессмысленность таких автоматов.

Вывод. Если создать полный набор тестов (w_i, s_i) и автомат, проверяющий реакцию P для каждого i, то получим систему, которая содержит в себе полностью P. Поэтому бессмысленно стремиться сделать полное покрытие тестами.

Таким образом, полное тестовое покрытие в реальных системах — недостижимый идеал. И цель — получить тестовое покрытие со следующими условиями:

полнота: максимально полное покрытие;

объём: количество тестов, которое можно проверить за разумное время;

время: получение необходимых для проверки тестов должно происходить в приемлемые сроки (время).

§ 4. Построение тестового покрытия

К проблеме получения тестов можно подходить с другой стороны. Разбить множество потребностей на такие подмножества, что в них есть такие элементы, взяв которые, можно быть уверенным в правильной реакции остальных элементов. Попробуем выяснить принципы разбиения на подмножества.

По выходу

Если выходных данных немного, то можно для них построить прообраз. Поскольку рассматриваемые системы — функции, то множество выходных данных разобьётся на непересекающиеся множества.

Для примера 1 это будут три множества:

- лабиринты, которые можно пройти;
- лабиринты, которые нельзя пройти;
- неправильные лабиринты.

По спецификациям

Представив спецификации системы в виде логической формулы, можно извлечь разбиение на подмножества из доказательства правильности реализации. Например, если доказательство использует метод разбора случаев, то каждый случай даст отдельное подмножество тестов.

Подробное описание данного способа — тема отдельного исследования.

По реализации

Каждый программный путь должен давать своё подмножество тестов. Например, каждый оператор IF должен дать два подмножества.

Этот способ может быть автоматизирован. Правда, для каждого языка программирования будет необходим свой автомат для перебора программых путей.

Интуитивный

Опыт часто подсказывает, где в системе могут быть «больные места». Для работы интуиции нужен опыт. Опыт может складываться из знания различных методов и области их применимости. Часто ошибки случаются из-за использования метода решения, область применимости которого не совпадает с предметной областью задачи.

Для примера 1 интуиция подсказывает, что лабиринты

```
0000000000   0000000000   0000000000
0000000000   0101010010   1111111110
0000000000   0000000000   0000000010
0000000000   1010101010   0111111010
0000000000   0000000000   0100010010
0000000000   0101010101   0101000010
0000000000   0000000000   0101111110
0000000000   1010101010   0100000000
0000000000   0000000000   0111111111
0000000000   0101010100   0000000010
```

должны быть интересными. Первый лабиринт не имеет стенок, если программа рассчитывает искать стенки, то этот тест может стать для неё проблемой. Второй лабиринт не имеет тупиков, зато много петель. Как и первый, это неудобный вариант для поиска в ширину. Третий лабиринт для противодействия программам, которые идут вдоль одной стенки и хранят недостаточно информации.

§ 5. Тесты

Заранее ограничим количество тестов, поскольку полное тестовое покрытие недостижимо. Пусть нам требуется получить 10 тестов для примера 1.

Тест 1 — пример из постановки задачи.

```
0000000001
0111111111
0000000001
1111111101
0000000001
1010101011
1110101111
0000100001
1011110100
0000000100
```

Тест 2 — для получения ответа 0, немного исправленный пример из постановки задачи. Поскольку тестов мало, то они должны включать в себя проверку нескольких «больных мест». В данном случае пробуем «запутать» программу.

```
0000000001
0111111111
0000000001
1111111101
0000000001
1010101011
1110101111
0000101001
1011110100
0000000100
```

Тесты 3, 4, 5 — интуитивные.

```
0000000000    0000000000    0101010101
0000000000    0101010010    1010101010
0000000000    0000000000    0101010101
0000000000    1010101010    1010101010
0000000000    0000000000    0101010101
0000000000    0101010101    1010101010
0000000000    0000000000    0101010101
0000000000    1010101010    1010101010
0000000000    0000000000    0101010101
0000000000    0101010100    1010101010
```

Тесты 6, 7 — неверное задание лабиринта.

```
0123456789    0101010100
0123456789    0101010100
0123456789    0101010100
0123456789    0101010100
0123456789    0101010100
0123456789    0101010100
0123456789    0101010100
0123456789    0101010100
0123456789    0101010100
0123456789
```

Тест 7 содержит меньше строк, чем положено.

Тесты 8, 9, 10 — правильные лабиринты с ответом 1, которые должны проверять возможные ошибки в алгоритмах.

```
0100010000   0100000000   0000000000
0101010101   0101011110   0001111110
0101010100   0000000010   0100000010
0101010110   1010101010   0101111010
0101010100   1110111011   0100001010
0101010101   0100010101   0101101000
0101010100   0000000000   0101001110
0101010110   0010101010   0101000001
0101010100   0110101011   0111111000
0001000110   0010100000   0000000100
```

Как описывалось выше, ни этот, ни какой-то другой набор тестов не гарантирует правильности написанной программы для любых примеров. Но представленный набор тестов позволит говорить, что программа работает неплохо.

СПИСОК ЛИТЕРАТУРЫ

1. Грэтхем Д. Семь мифов о независимости спецификаций и тестирования // Открытые системы. СУБД. 2002. № 11. С. 56-58

2. Канер С., Фолк Дж., Нгуен Е. К. Тестирование программного обеспечения / Пер. с англ. М.: ДиаСофт, 2001. 544 с.

3. Канер С., Фолк Дж., Нгуен Е. К. Тестирование программного обеспечения / Пер. с англ. Киев: ДиаСофт, 2000. 416 с.

4. Майерс Г. Надежность программного обеспечения / Пер. с англ. М.: Мир, 1980. 360 с.

Поступила в редакцию 13.10.04

V. V. Pupyshev

Example of a Testing programm about maze

The task about maze passing is a simple example of impossible find all bugs in a program.

Пупышев Вячеслав Викторович
Удмуртский государственный университет
Кафедра математического обеспечения ЭВМ
426034, Россия, г. Ижевск,
ул. Университетская, 1(корп. 4)
e-mail: pvv@uni.udm.ru