Тестирования программного обеспечения: основы
или
Все, что вы всегда хотели знать о тестировании (но боялись спросить)

Software Testing 101

Марат Ахин

Санкт-Петербургский государственный политехнический университет

2012

Содержание

1 Прелюдия
- Software Testing 101
- Что такое тестирование?

2 Общая модель тестирования

3 Проблема тестовых входных данных

4 Проблема наблюдаемости

5 Постлюдия

Software Testing 101

- Основы тестирования программного обеспечения
  - То, что Вы точно должны знать, если хотите стать хорошим специалистом по тестированию ПО
  - А если я хочу быть разработчиком?

Знание основ тестирования ПО разработчику никогда не помешает

Software Testing 101

- Основы тестирования программного обеспечения
  - То, что Вы точно должны знать, если хотите стать хорошим специалистом по тестированию ПО
  - А если я хочу быть разработчиком?

> Знание основ тестирования ПО разработчику никогда не помешает

Почему разработчик должен уметь тестировать ПО?

- Потому что тестирование является основой практически любой методологии проектирования и разработки ПО, которые используются в настоящее время
- Потому что умение тестировать сделает вас лучшим разработчиком

Что за вопрос лежит в основе тестирования?

Работает ли это ПО правильно?

- НЕТ, тестирование никогда не может дать Вам ответа на этот вопрос

Работает ли это ПО неправильно?

Тестирование = Разрушение

Что за вопрос лежит в основе тестирования?

Работает ли это ПО правильно?

- **НЕТ**, тестирование никогда не может дать Вам ответа на этот вопрос

Работает ли это ПО неправильно?

Тестирование = Разрушение

Что за вопрос лежит в основе тестирования?

Работает ли это ПО правильно?

- **НЕТ**, тестирование никогда не может дать Вам ответа на этот вопрос

Работает ли это ПО неправильно?

- ДА, тестирование может (и должно) ответить на этот вопрос

Тестирование = Разрушение

Что за вопрос лежит в основе тестирования?

Работает ли это ПО правильно?

- **НЕТ**, тестирование никогда не может дать Вам ответа на этот вопрос

Работает ли это ПО неправильно?

- **ДА**, тестирование может (и должно) ответить на этот вопрос

Тестирование = Разрушение

Что за вопрос лежит в основе тестирования?

Работает ли это ПО правильно?

- **НЕТ**, тестирование никогда не может дать Вам ответа на этот вопрос

Работает ли это ПО неправильно?

- **ДА**, тестирование может (и должно) ответить на этот вопрос

Тестирование = Разрушение

Можем ли мы что-то гарантировать при тестировании?

- Данное ПО никогда не упадет по NPE
- Потоки никогда не заблокируются
- Вычисления всегда выполняются корректно
- Временные характеристики всегда выдерживаются

> Мы можем дать такие гарантии лишь в самых тривиальных случаях, когда обычно все ясно и без тестирования

Почему тестировать сложно?

Brian Kernighan

«Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.»

Massimo Arnoldi (feat. Kent Beck)

«Unfortunately at least for me (and not only) testing goes against human nature. If you realize the pig in you, you will see that you program without tests.»

Что же делать?

Содержание

1 Прелюдия

2 Общая модель тестирования
- Тестирование ПО с точки зрения дилетанта
- Модель программной ошибки
- Модель тестирования ПО

3 Проблема тестовых входных данных

4 Проблема наблюдаемости

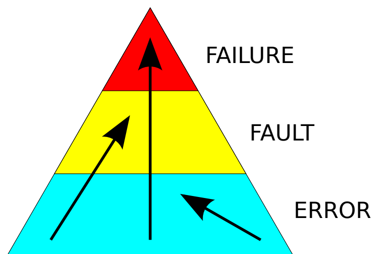
5 Постлюдия

Тестирование ПО с точки зрения дилетанта

- Запустили приложение
- Проверили результаты выполнения на предмет наличия в них ошибок
  - aka «багов»
  - aka «сбоев»
  - aka «дефектов»
  - aka «неудач»

Сперва надо разобраться, а что же такое «программная ошибка»?

Модель программной ошибки



- **Неудача** – наблюдаемое **снаружи** некорректное поведение программы
- **Сбой** – некорректное состояние программы из-за ошибки
- **Ошибка** – ошибка в самой программе, внесенная на этапе разработки

Рассмотрим данную модель на примере

Queue

```
public class Queue<E> {                                   1
  int head, tail, max;                                    2
  ArrayList<E> data;                                      3
                                                          4
  Queue() {                                               5
    this(3);                                              6
  }                                                       7
                                                          8
  Queue(int max) {                                        9
    this.head = this.tail = 0;                            10
    this.max = max;                                       11
    this.data = new ArrayList<E>(max);                    12
  }                                                       13
                                                          14
  void add(E e) {                                         15
    if (tail == max) tail = 0;                            16
    data.set(tail++, e);                                  17
  }                                                       18
...                                                       19
}                                                         20
```

Queue

```
Queue q = new Queue();    1
q.add(x);                 2
q.add(y);                 3
```

Нет ни сбоя, ни неудачи – программа работает корректно

Queue

```
Queue q = new Queue();    1
q.add(x);                 2
q.add(y);                 3
```

Нет ни сбоя, ни неудачи – программа работает корректно

Queue

```
Queue q = new Queue();    1
q.add(x);                 2
q.add(y);                 3
q.add(z);                 4
```

Нет ни сбоя, ни неудачи – программа работает корректно

Queue

```
Queue q = new Queue();    1
q.add(x);                 2
q.add(y);                 3
q.add(z);                 4
```

Нет ни сбоя, ни неудачи – программа работает корректно

Queue

```
Queue q = new Queue();   1
q.add(x);                2
q.add(y);                3
q.add(z);                4
q.add(a);                5
```

Сбой есть – программа проходит через некорректное состояние
Но неудачи нет – результат работы программы корректен

Queue

```
Queue q = new Queue();   1
q.add(x);                2
q.add(y);                3
q.add(z);                4
q.add(a);                5
```

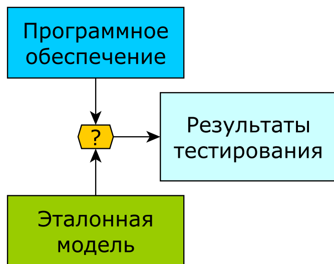
Сбой есть – программа проходит через некорректное состояние
Но **неудачи** нет – результат работы программы корректен

Queue

```
Queue q = new Queue();      1
q.add(x);                   2
q.add(y);                   3
q.add(z);                   4
q.add(a);                   5
System.out.println(q);      6
```

Сбой есть – программа проходит через некорректное состояние
Неудача тоже есть – результат работы программы неправильный

Модель тестирования ПО



Эталонная модель может быть представлена множеством различных способов

- неформальное представление того, «как ПО должно работать»
- формальная техническая спецификация
- набор тестовых примеров
- корректные результаты работы программы
- другая (априори корректная) реализация той же исходной спецификации

Свойства теста

Для того, чтобы найти ошибку, тест должен обеспечивать выполнение определенных свойств

Каких?

- Reachibility – тест должен выполнить место в исходном коде, где присутствует программная ошибка
- Corruption – при выполнении ошибки состояние программы должно испортиться с появлением сбоя
- Propagation – сбой должен распространиться дальше и вызвать неудачу в работе программы

Обеспечение этих свойств – одна из самых сложных проблем тестирования ПО

Свойства теста

Для того, чтобы найти ошибку, тест должен обеспечивать выполнение определенных свойств

Каких?

- **Reachibility** – тест должен выполнить место в исходном коде, где присутствует программная ошибка
- Corruption – при выполнении ошибки состояние программы должно испортиться с появлением сбоя
- Propagation – сбой должен распространиться дальше и вызвать неудачу в работе программы

Обеспечение этих свойств – одна из самых сложных проблем тестирования ПО

Свойства теста

Для того, чтобы найти ошибку, тест должен обеспечивать выполнение определенных свойств

Каких?

- **Reachibility** – тест должен выполнить место в исходном коде, где присутствует программная ошибка
- **Corruption** – при выполнении ошибки состояние программы должно испортиться с появлением сбоя
- **Propagation** – сбой должен распространиться дальше и вызвать неудачу в работе программы

Обеспечение этих свойств – одна из самых сложных проблем тестирования ПО

Свойства теста

Для того, чтобы найти ошибку, тест должен обеспечивать выполнение определенных свойств

Каких?

- **Reachibility** – тест должен выполнить место в исходном коде, где присутствует программная ошибка
- **Corruption** – при выполнении ошибки состояние программы должно испортиться с появлением сбоя
- **Propagation** – сбой должен распространиться дальше и вызвать неудачу в работе программы

Обеспечение этих свойств – одна из самых сложных проблем тестирования ПО

Свойства теста

Для того, чтобы найти ошибку, тест должен обеспечивать выполнение определенных свойств

Каких?

- **Reachibility** – тест должен выполнить место в исходном коде, где присутствует программная ошибка
- **Corruption** – при выполнении ошибки состояние программы должно испортиться с появлением сбоя
- **Propagation** – сбой должен распространиться дальше и вызвать неудачу в работе программы

Обеспечение этих свойств – одна из самых сложных проблем тестирования ПО

Содержание

1 Прелюдия

2 Общая модель тестирования

3 Проблема тестовых входных данных
- Обеспечение достижимости
- Тестовые данные
- Классы эквивалентности
- Неявные входные данные
- Обеспечение порчи внутреннего состояния

4 Проблема наблюдаемости

5 Постлюдия

Обеспечение достижимости

Какими способами можно управлять выполнением кода?

- Изменением входных данных
- Изменением самого исходного кода

Какой способ можно использовать для обеспечения достижимости (Reachability)?

Тестовые данные

Почему бы просто не перебрать все возможные варианты?

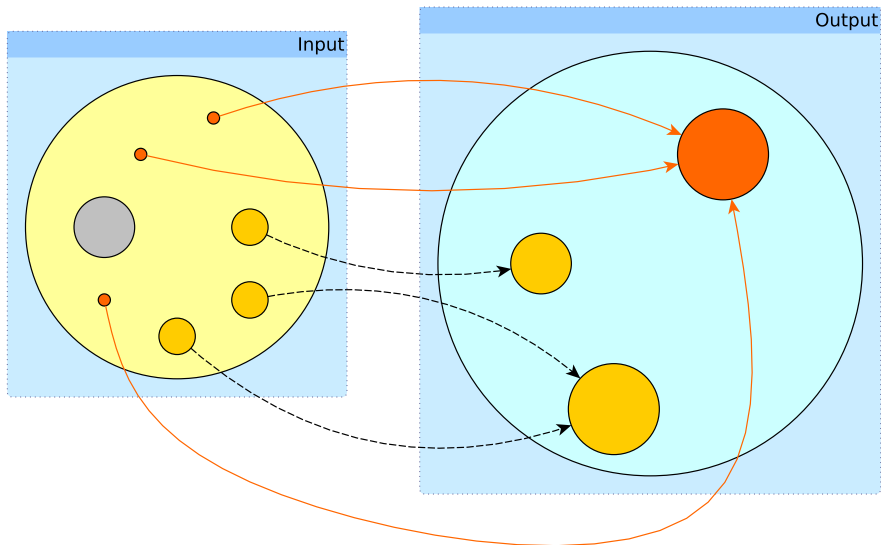
```
int add(int a, int b) { ... }
```

1

18,446,744,073,709,551,616 вариантов

А если у тестируемого модуля есть внутреннее состояние...

Тестовые данные



Классы эквивалентности

- Все пространство входных состояний можно разбить на множество классов эквивалентности
    - Каждый класс эквивалентности обрабатывается тестируемым модулем одинаково *с точки зрения спецификации*
    - Тестирование всех классов эквивалентности позволяет найти ошибки прямого нарушения спецификации

Как найти классы эквивалентности?

Классы эквивалентности

- Ad hoc testing
- Метод свободного поиска
- Анализ граничных значений
- Причинно-следственный анализ

Все они основываются на анализе спецификации

Примеры

- Queue
- atoi
- md5sum
- PDF reader

Неявные входные данные

Покрывает ли спецификация все множество входных данных?

- В некоторых случаях – да
- В большинстве случаев – нет

Почему?

- Проблема заключается в том, что на тестовый модуль, кроме явных, влияет множество *неявных* входных данных
- Неявные входные данные часто не находят отражения в спецификации

Неявные входные данные

- Текущая дата/время
- IP/MAC адрес
- Локаль пользователя
- Идентификаторы устройств
- Контекстные переключения/планирование нитей
- Скорость поступления IP пакетов
- Время нажатия клавиш на клавиатуре

Все это – примеры неявных входных данных

Therac-25

PATIENT NAME : TEST
TREATMENT MODE : FIX   BEAM TYPE: X   ENERGY (MeV): 25

	ACTUAL	PRESCRIBED	
UNIT RATE/MINUTE	0	200	
MONITOR UNITS	50 50	200	
TIME (MIN)	0.27	1.00	
GANTRY ROTATION (DEG)	0.0	0	VERIFIED
COLLIMATOR ROTATION (DEG)	359.2	359	VERIFIED
COLLIMATOR X (CM)	14.2	14.3	VERIFIED
COLLIMATOR Y (CM)	27.2	27.3	VERIFIED
WEDGE NUMBER	1	1	VERIFIED
ACCESSORY NUMBER	0	0	VERIFIED

DATE : 84-OCT-26   SYSTEM : BEAM READY   OP. MODE : TREAT   AUTO
TIME : 12:55: 8   TREAT : TREAT PAUSE   X-RAY   173777
OPR ID : T25V02-R03   REASON : OPERATOR   COMMAND:

Как учесть неявные входные данные?

Какими способами можно управлять выполнением кода?

- Изменением входных данных
- Изменением самого исходного кода

Simulate and Stub

- Заменяем части модуля заглушками, которые эмулируют соответствующее окружение
- Это позволяет сделать неявные входные данные явными

Simulate and Stub

- Данный подход позволяет:
  - имитировать возникновение редких ситуаций
  - внести детерминизм там, где его нет
  - обеспечить обратную масштабируемость
- Для того, чтобы эффективно использовать S&S, тестируемый модуль должен разрабатываться соответствующим образом

Обеспечение порчи внутреннего состояния

- Для того, чтобы обеспечить порчу внутреннего состояния (corruption), необходимо:
  - обеспечить достижимость
  - передать такие тестовые входные данные, которые вызывают нарушение целостности внутреннего состояния

Все то же самое, только хуже...

Содержание

1 Прелюдия

2 Общая модель тестирования

3 Проблема тестовых входных данных

4 Проблема наблюдаемости
- Обеспечение распространения сбоя
- Assertions
- Журналирование

5 Постлюдия

Обеспечение распространения сбоя

Какими способами можно управлять выполнением кода?

- Изменением входных данных
- Изменением самого исходного кода

Необходимо обнаружить сбой и распространить его, сделав наблюдаемым снаружи (propagation)

Assertions

Основной способ обеспечения наблюдаемости – использование assertions

```
assert this.size >= 0 && this.size <= this.max;            1
assert (this.head + this.size) % this.max == this.tail;    2
```

Assertions

Что дает использование assertions?

- Проверка корректности внутреннего состояния
- Неудача происходит ближе к причине ее возникновения
- Явное документирование пред-, пост-условий, инвариантов, и т.п.

Assertions

О чем необходимо помнить при использовании assertions?

- Assertions != обработка ошибок
- Побочные эффекты в assertions == ЗЛО!
- Очевидные assertions == ЗЛО!

Используются ли assertions на практике?

Журналирование

- Другой способ обеспечения наблюдаемости
- Запись процесса выполнения программы вместе со внутренним состоянием в журнал
- Позволяет изучить ход выполнения программы позднее

Что лучше – assertions или журналирование?

Журналирование

```
 INFO [http-thread-pool-8080(5)] Received token: e6749451
TRACE [http-thread-pool-8080(5)] Calling: AuthStorageBean.getAuthData
TRACE [http-thread-pool-8080(5)] Called: AuthStorageBean.getAuthData -> 2.0708E-5
 INFO [http-thread-pool-8080(5)] Authentication data found: AuthData { authToken:e6749451
      userId:1 firstName: lastName:Admin patrName: role:ru.korus.tmis.core.entity.model.
      Role[id=1] spec: }
TRACE [http-thread-pool-8080(5)] Calling: AuthStorageBean.getAuthDateTime
TRACE [http-thread-pool-8080(5)] Called: AuthStorageBean.getAuthDateTime -> 1.9825E-5
 INFO [http-thread-pool-8080(5)] Token is valid
TRACE [http-thread-pool-8080(5)] attempting to get session; create = false; session is
     null = true; session has id = false
TRACE [http-thread-pool-8080(5)] Authentication attempt received for token [ru.korus.tmis
     .core.auth.TmisShiroToken@37bd2b6]
DEBUG [http-thread-pool-8080(5)] Performing credentials equality check for
     tokenCredentials of type [java.lang.String and accountCredentials of type [java.lang
     .String]
DEBUG [http-thread-pool-8080(5)] Both credentials arguments can be easily converted to
     byte arrays. Performing array equals comparison
DEBUG [http-thread-pool-8080(5)] Authentication successful for token [ru.korus.tmis.core.
     auth.TmisShiroToken@37bd2b6]. Returned account [(admin,ru.korus.tmis.core.entity.
     model.Role[id=1])]
DEBUG [http-thread-pool-8080(5)] No SecurityManager available in subject context map.
     Falling back to SecurityUtils.getSecurityManager() lookup.
TRACE [http-thread-pool-8080(5)] get() - in thread [http-thread-pool-8080(5)]
TRACE [http-thread-pool-8080(5)] Context already contains a SecurityManager instance.
     Returning.
```

Содержание

1. Прелюдия
2. Общая модель тестирования
3. Проблема тестовых входных данных
4. Проблема наблюдаемости
5. Постлюдия
   - Что вы можете запомнить?
   - Что вы должны запомнить?

Что вы можете запомнить?

- Тестирование – это разрушение
- Хорошие тестовые входные данные – это сложно
- Изменять код для облегчения тестирования – это норма

Что вы должны запомнить?