Полнота тестирования ПО: оценка и обеспечение

Software Testing 102

Марат Ахин

Санкт-Петербургский государственный политехнический университет

2011

Quiz

Содержание

В чем отличие от тестирования черного ящика?

В цвете??? Вообще-то – да!

- Мы знаем то, как устроен тот «ящик», который мы тестируем
- Мы можем использовать это знание для того, чтобы проектировать тесты
- Этот метод также известен как структурное тестирование

Это дополнительное знание – хорошо или плохо?

Свойства тестирования белого ящика

- Дополнительная информация о тестируемом ПО не может быть лишней
  - Благодаря ей мы можем проектировать более эффективные и точные тесты
  - Мы знаем устройство «ящика» – и знаем то, какие его части требуют более тщательного тестирования, а какие можно практически не тестировать
  - За счет этого можно снизить стоимость проведения тестирования

Свойства тестирования белого ящика

- К сожалению, при тестировании ПО всегда есть один негативный фактор – человек
  - Обычно при тестировании белого ящика тесты пишет сам разработчик «ящика»
  - При разработке он думал о вполне определенных входных воздействиях на модуль
  - Скорее всего, при тестировании он будет подавать такие же входные воздействия
  - В то же время, ошибки появляются, когда в работе модуля возникают «неожиданности»

Свойства тестирования белого ящика

Как мы можем использовать знание о том, как устроен тестируемый модуль?

- Мы можем пытаться обеспечить определенный уровень покрытия исходного кода модуля
- В зависимости от того, как именно мы хотим покрыть исходный код, мы будем получать те или иные методы тестирования белого ящика
- Оценки покрытия также используются для оценки полноты тестирования ПО

Свойства тестирования белого ящика

Как мы можем использовать знание о том, как устроен тестируемый модуль?

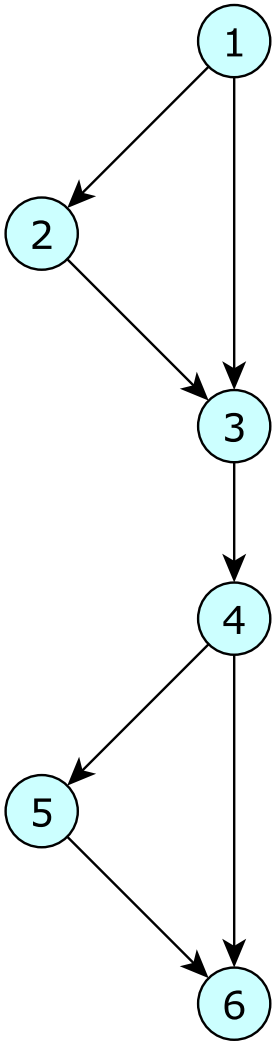
- Мы можем пытаться обеспечить определенный уровень **покрытия** исходного кода модуля
- В зависимости от того, как именно мы хотим покрыть исходный код, мы будем получать те или иные методы тестирования белого ящика
- Оценки покрытия также используются для оценки **полноты** тестирования ПО
  - Об этом мы поговорим чуть дальше

Виды тестового покрытия

- Выделяют два основных вида покрытия
  - Покрытие потока управления
  - Покрытие потока данных
- Они работают с такими понятиями, как граф потока управления (CFG) и граф потока данных (DFG)

Надеюсь, что все знают, что такое CFG и DFG?..

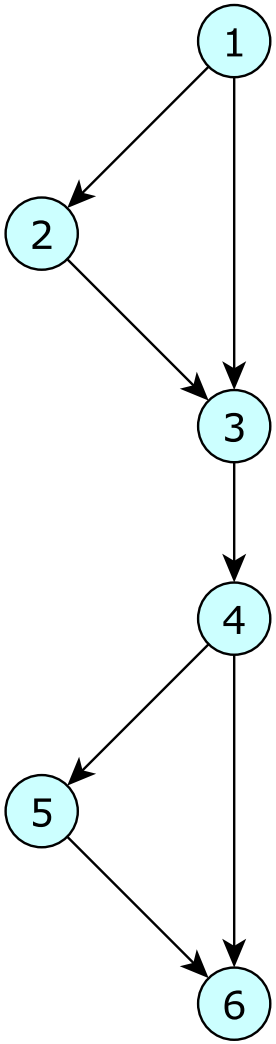
Покрытие потока управления



Как мы можем покрыть данный CFG?

- По узлам
- По дугам
- По условиям
- По путям
- ...

Покрытие потока управления

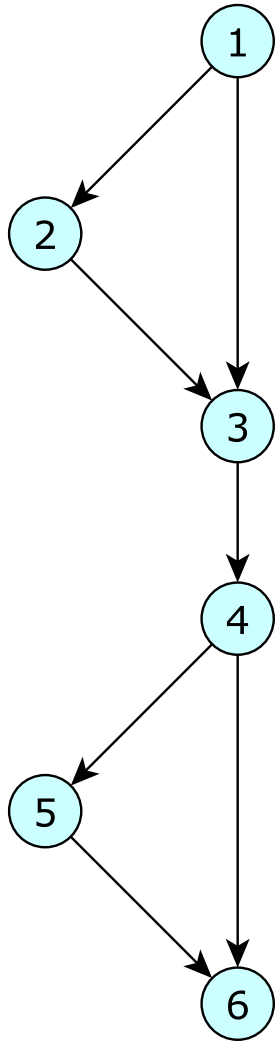


Как мы можем покрыть данный CFG?

- По узлам
- По дугам
- По условиям
- По путям
- ...

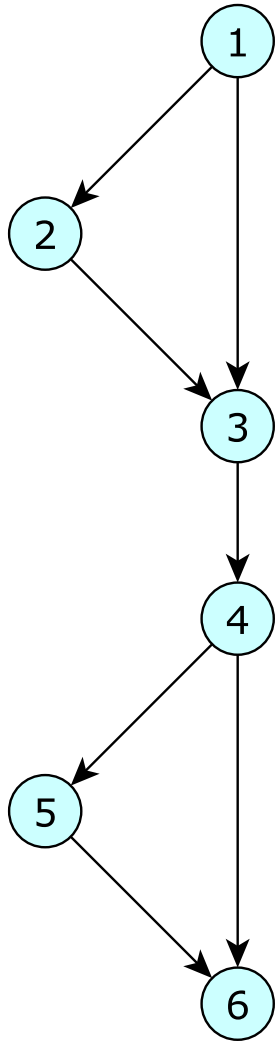
Покрытие операторов программы

- Каждый узел CFG был пройден в процессе тестирования хотя бы один раз
- Самый слабый способ оценки тестового покрытия
- Сколько тестов требуется для того, чтобы обеспечить полное покрытие операторов программы для следующего примера?



Покрытие ветвлений программы

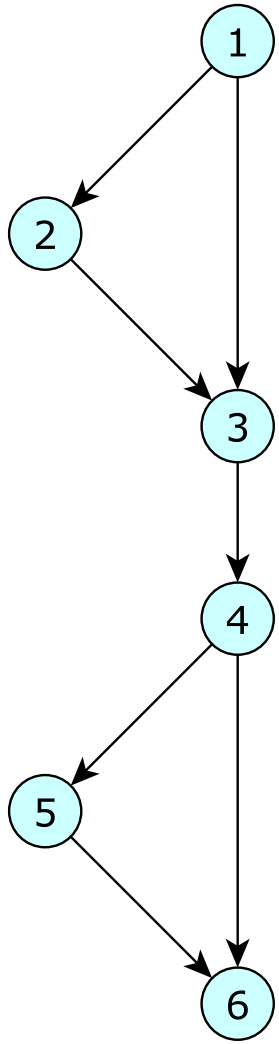
- Каждая ветка программы была пройдена хотя бы один раз
- Несколько более сильный способ оценки покрытия
- Сколько тестов требуется для того, чтобы обеспечить полное покрытие ветвлений программы для следующего примера?



Покрытие ветвлений программы

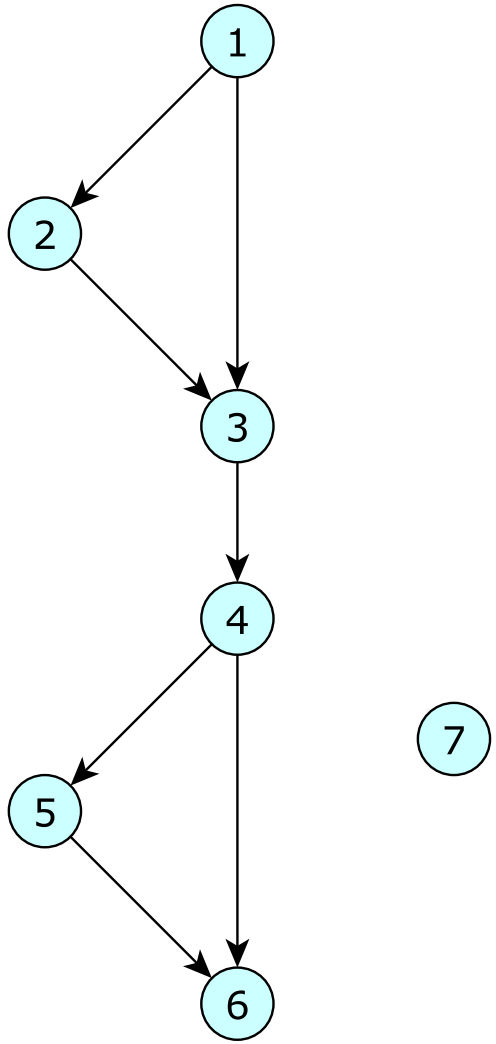
Как соотносятся между собой покрытия операторов и ветвлений?

1. Никак
2. Покрытие операторов включает покрытие ветвлений
3. Покрытие ветвлений включает покрытие операторов



Покрытие ветвлений программы

- Никак
- Почему покрытие ветвлений не включает в себя покрытие операторов?
  - Потому что в программе может присутствовать «мертвый код»



Покрытие условий программы

- Каждое ветвление может выполняться по различным причинам
- При покрытии условий программы мы требуем, чтобы все условия программы хотя бы один раз приняли значение «true» и «false»

```
1 if (p != q && (p == null || !p.equals(q))) {
2     ...
3 }
```

Как соотносятся между собой покрытия ветвлений и условий?

Покрытие ветвлений и условий программы

- Комбинация соответствующих покрытий
  - Полностью покрывает их
- Можно ли предложить что-то более сильное?

```
1 if (p != q && (p == null || !p.equals(q))) {
2     ...
3 }
```

Комбинационное покрытие условий программы

- Полный перебор всех возможных **комбинаций** условий всех возможных ветвлений
- Сколько вариантов необходимо перебрать, чтобы получить полное комбинационное покрытие для следующего примера?

```
1 if (p != q && (p == null || !p.equals(q))) {
2     ...
3 }
```

Покрытие путей программы

- Мы требуем, чтобы все возможные пути программы были выполнены хотя бы один раз
  - Как данное покрытие соотносится с комбинационным покрытием условий?
- Обычно считается самым сильным типом покрытия потока управления
- Его можно было бы использовать, если бы не...
  - Циклы и рекурсия

Покрытие путей программы

- Мы требуем, чтобы все возможные пути программы были выполнены хотя бы один раз
  - Как данное покрытие соотносится с комбинационным покрытием условий?
- Обычно считается самым сильным типом покрытия потока управления
- Его можно было бы использовать, если бы не...

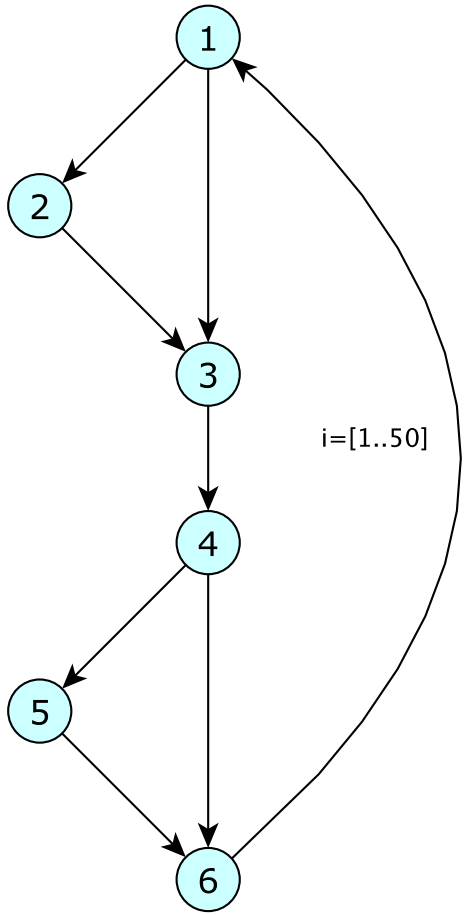
Циклы и рекурсия

Покрытие путей программы

- Мы требуем, чтобы все возможные пути программы были выполнены хотя бы один раз
  - Как данное покрытие соотносится с комбинационным покрытием условий?
- Обычно считается самым сильным типом покрытия потока управления
- Его можно было бы использовать, если бы не...

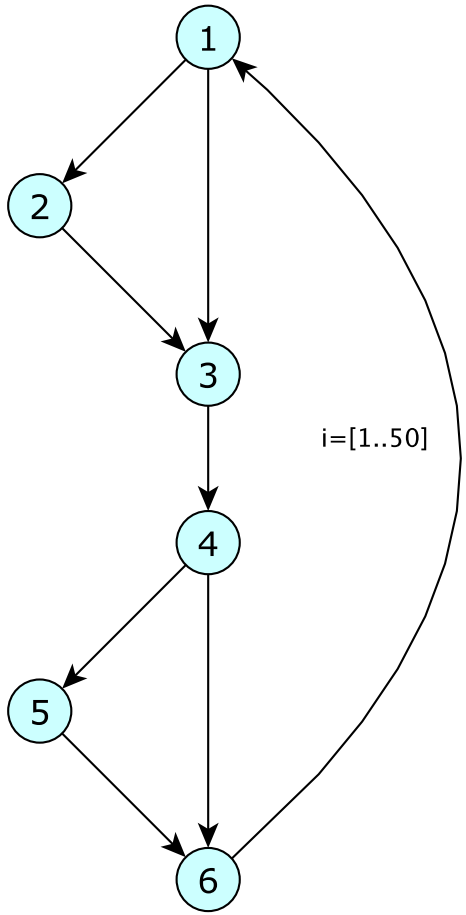
Циклы и рекурсия

Покрытие путей программы



4^{50} возможных путей

Покрытие путей программы



4^{50} возможных путей

Покрытие путей программы

- Для борьбы с этим используют несколько подходов
- Один из вариантов
  - Требуем, чтобы тело цикла было выполнено
    - 0
    - 1
    - 2
    - k
    - max
    - $max + 1$

Объясните, почему выбраны именно эти варианты

Покрытие потока данных

Что еще можно сделать?

- Можно вспомнить о том, что программы работают не просто так
- С этой точки зрения для тестирования представляет интерес анализ таких путей выполнения программы, на которых активно работают с данными

Сперва вспомним несколько определений

Покрытие потока данных

Что еще можно сделать?

- Можно вспомнить о том, что программы работают не просто так
  - Основная цель работы любой программы – работа с данными
- С этой точки зрения для тестирования представляет интерес анализ таких путей выполнения программы, на которых активно работают с данными

Сперва вспомним несколько определений

Покрытие потока данных

Что еще можно сделать?

- Можно вспомнить о том, что программы работают не просто так
  - Основная цель работы любой программы – работа с данными
- С этой точки зрения для тестирования представляет интерес анализ таких путей выполнения программы, на которых активно работают с данными

Сперва вспомним несколько определений

Определения и использования переменных

Определение переменной (Def)

Оператор программы, в котором значение переменной `v` может быть изменено

- `v = 42`
- `f(&v, ...)`
- `scanf(fmt, &v)`

Использование переменной (Use)

Оператор программы, в котором значение переменной `v` влияет на выполнение программы тем или иным способом

- `q = v + 2`
- `g(v, ...)`
- `if (v != NULL) ...`

Определения и использования переменных

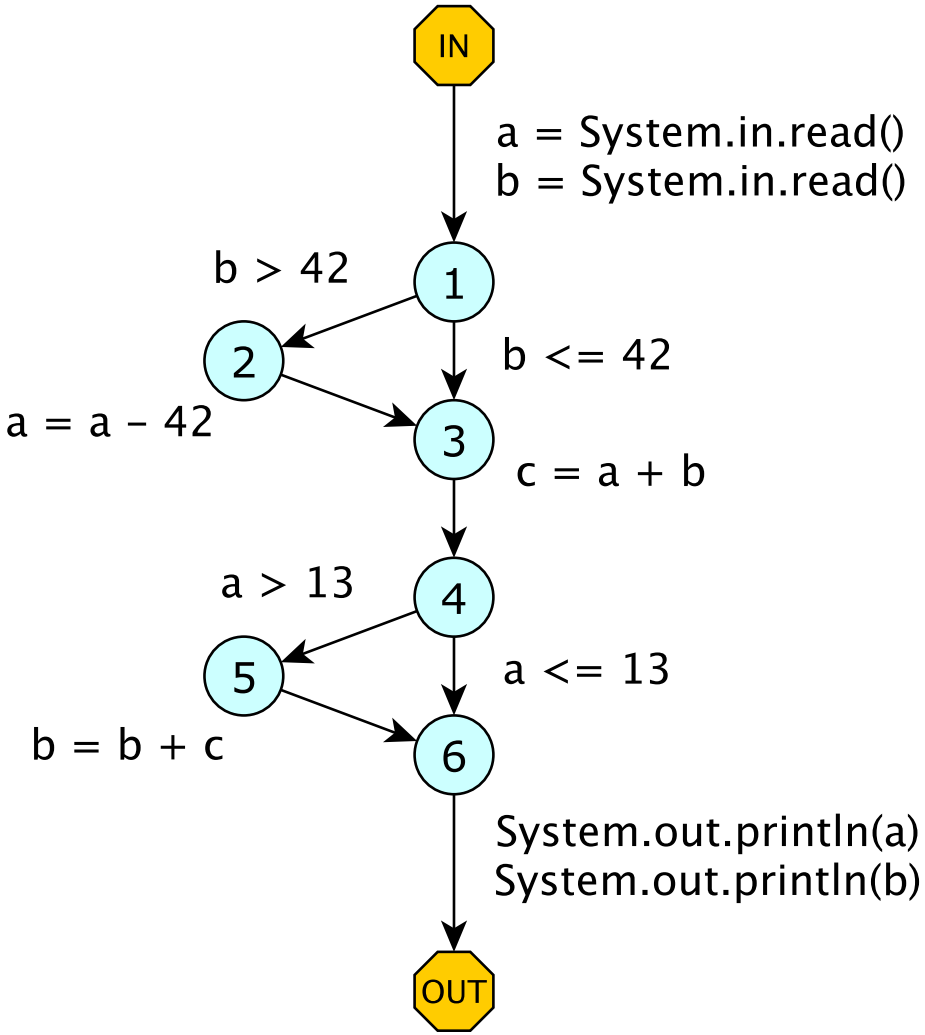
Def-Use Chain

Пара (d, u) операторов программы, для которой выполняются следующие условия

- d – определение переменной v
- u – использование переменной v
- между d и u существует хотя бы один путь, на котором переменная v не переопределяется

Рассмотрим данные определения на примере

Пример

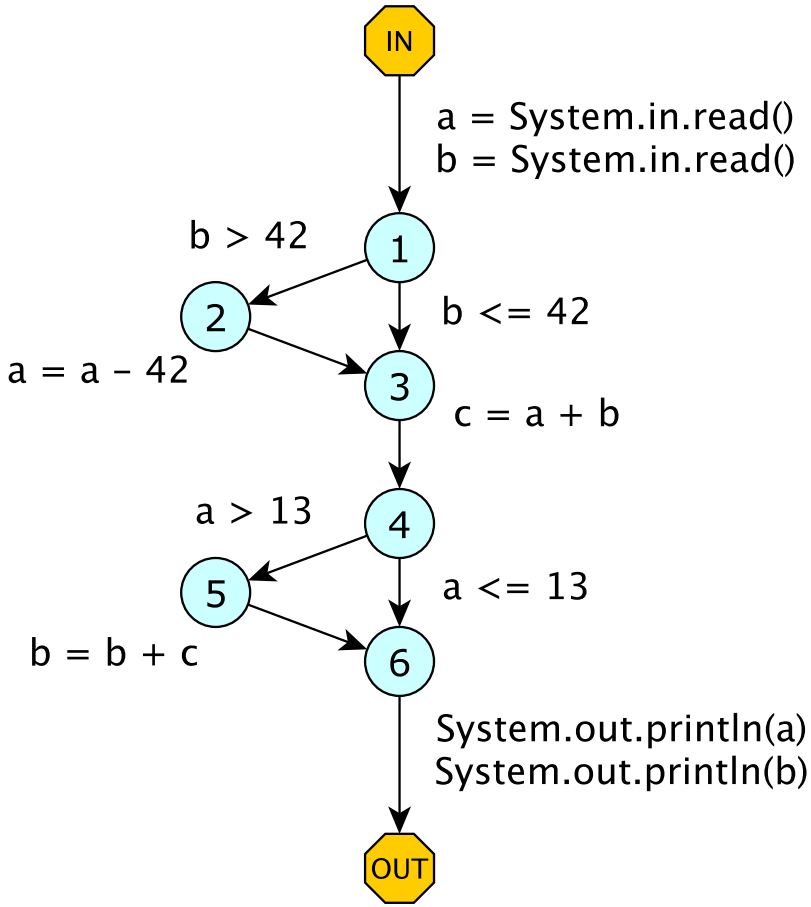

IN
a = System.in.read()
b = System.in.read()
1
b > 42
2
b <= 42
a = a - 42
3
c = a + b
4
a > 13
5
a <= 13
b = b + c
6
System.out.println(a)
System.out.println(b)
OUT

Покрытие потока данных

Какие варианты покрытий можно предложить с использованием этих понятий?

- Покрытие всех определений
  - Для каждой интересующей нас переменной v должна быть протестирована хотя бы одна *Def-Use Chain* от **каждого** определения v до хотя бы одного использования v

All-Def Coverage



Рассмотрим следующие тесты

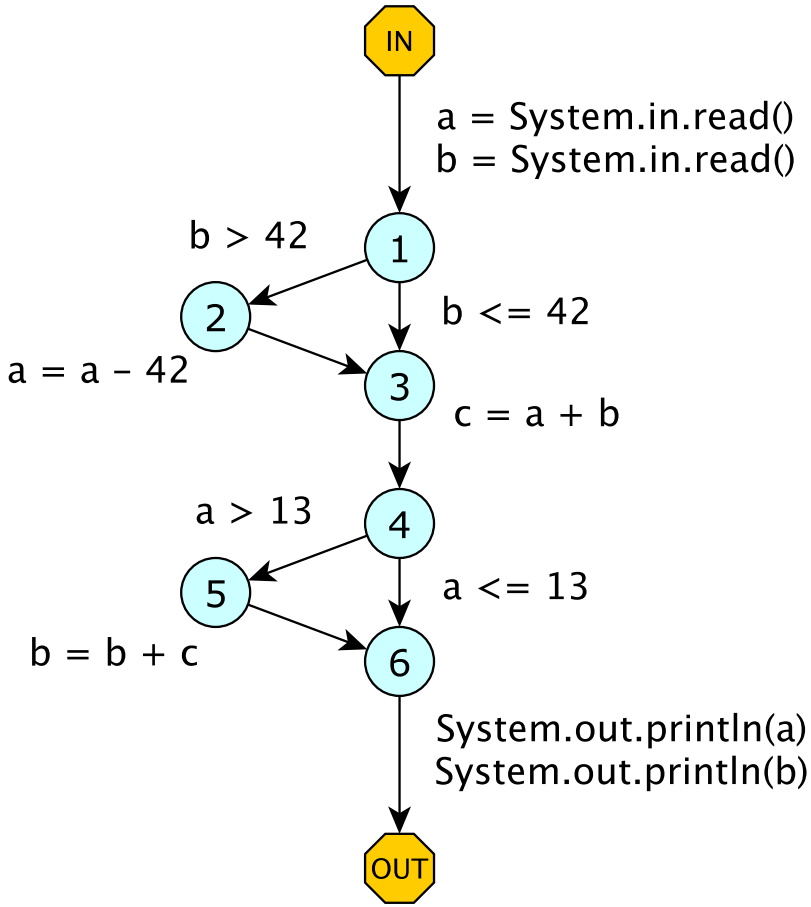
- $1 \to 2 \to 3 \to 4 \to 5 \to 6$

Покрытие потока данных

- Покрытие всех использований
  - Для каждой интересующей нас переменной `v` должна быть протестирована хотя бы одна *Def-Use Chain* от **каждого** определения `v` до **каждого** использования `v`

Является ли All-Use более сильным критерием по сравнению с All-Def?

All-Use Coverage



Рассмотрим следующие тесты

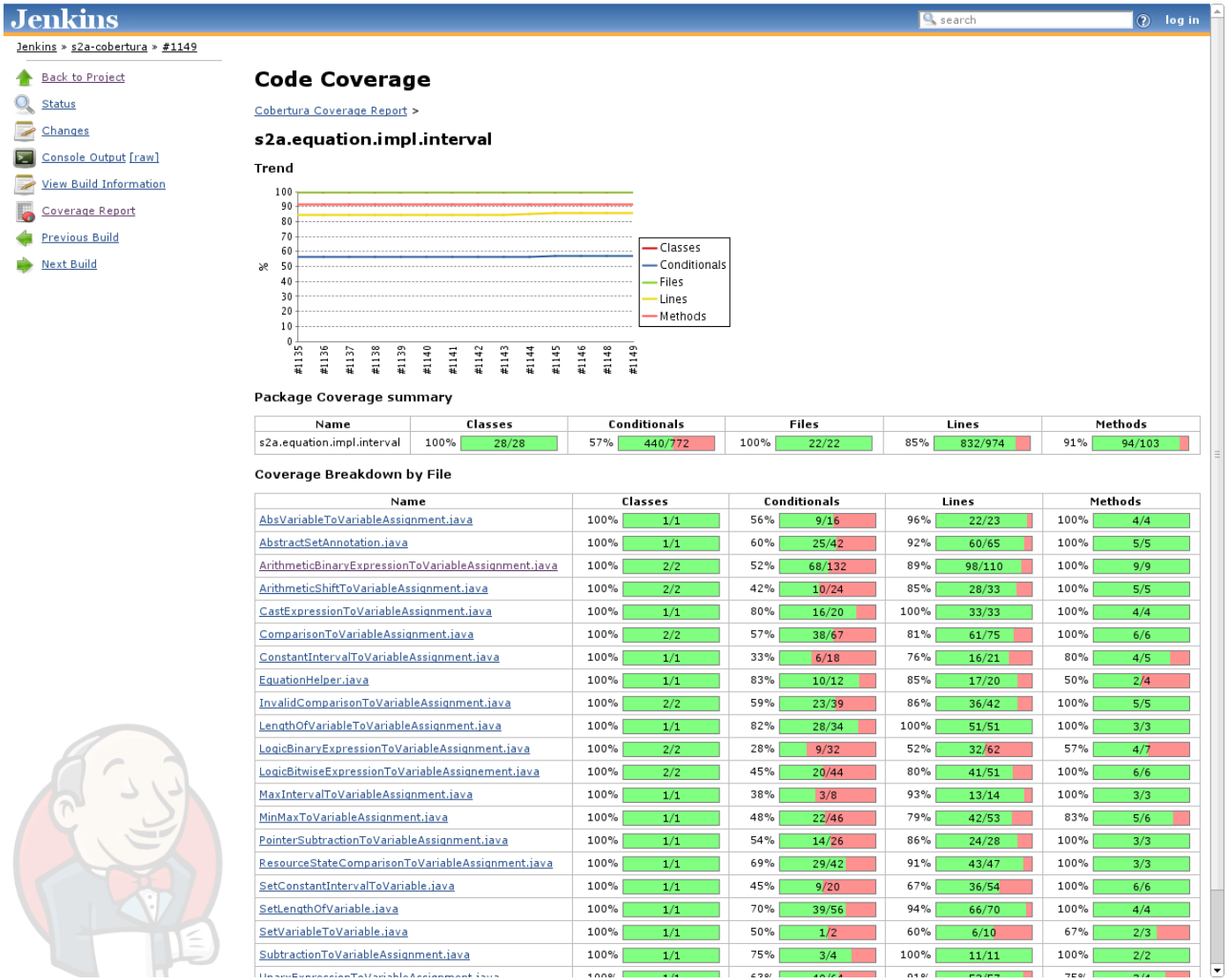
- $1 \to 2 \to 3 \to 4 \to 5 \to 6$
- $1 \to 2 \to 3 \to 4 \to 6$
- $1 \to 3 \to 4 \to 5 \to 6$

Покрытие потока данных

- Покрытие всех Def-Use Chain
  - Для каждой интересующей нас переменной `v` должны быть протестированы все возможные *Def-Use Chain* от **каждого** определения `v` до **каждого** использования `v`

Как соотносится All-Def-Use-Chain с покрытием всех путей программы?

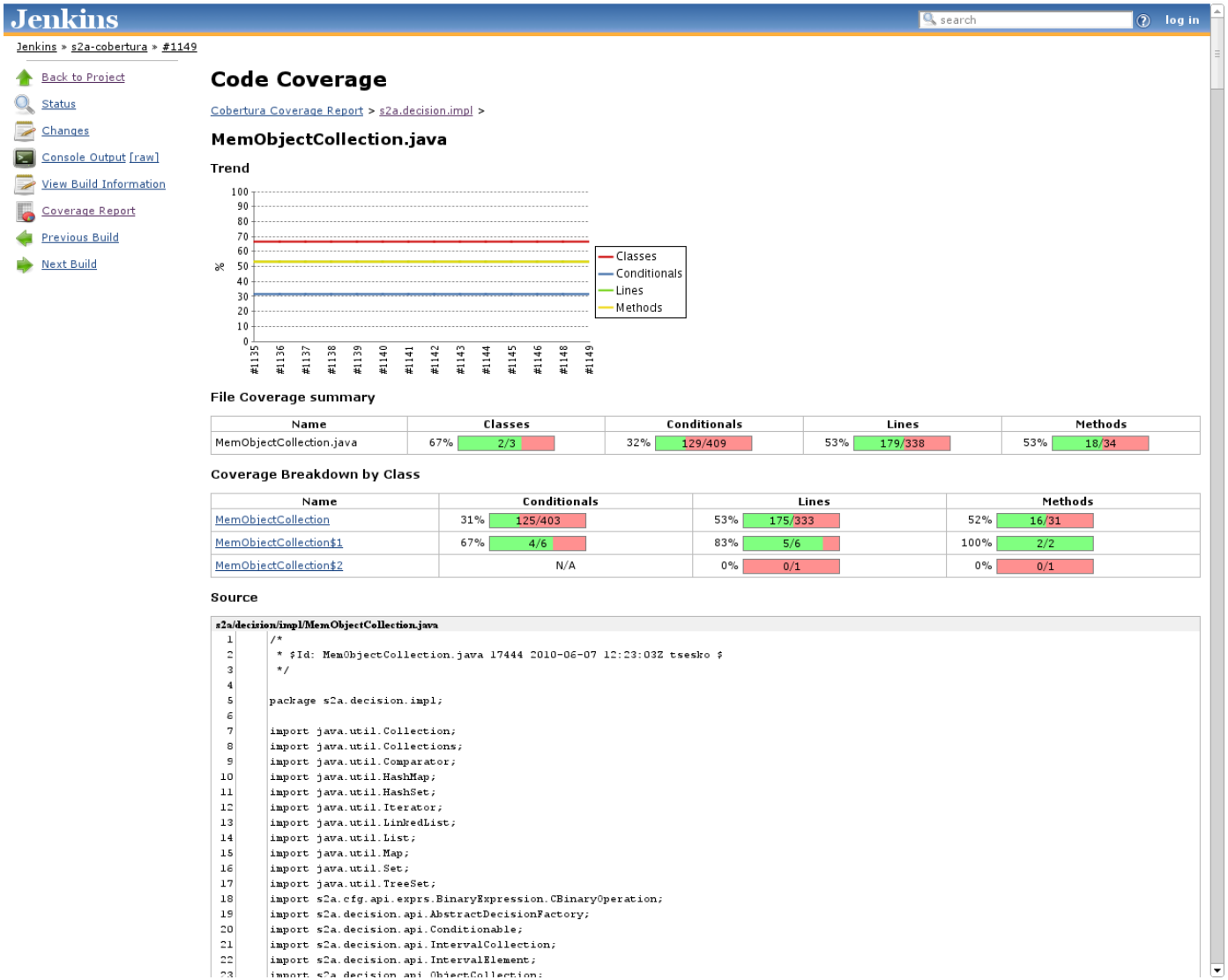
Пример



Name	Classes	Conditionals	Files	Lines	Methods
s2a.equation.impl.interval	100% 28/28	57% 440/772	100% 22/22	85% 832/974	91% 94/103

Name	Classes	Conditionals	Lines	Methods
AbsVariableToVariableAssignment.java	100% 1/1	56% 9/16	96% 22/23	100% 4/4
AbstractSetAnnotation.java	100% 1/1	60% 25/42	92% 60/65	100% 5/5
ArithmeticBinaryExpressionToVariableAssignment.java	100% 2/2	52% 68/132	89% 98/110	100% 9/9
ArithmeticShiftToVariableAssignment.java	100% 2/2	42% 10/24	85% 28/33	100% 5/5
CastExpressionToVariableAssignment.java	100% 1/1	80% 16/20	100% 33/33	100% 4/4
ComparisonToVariableAssignment.java	100% 2/2	57% 38/67	81% 61/75	100% 6/6
ConstantIntervalToVariableAssignment.java	100% 1/1	33% 6/18	76% 16/21	80% 4/5
EquationHelper.java	100% 1/1	83% 10/12	85% 17/20	50% 2/4
InvalidComparisonToVariableAssignment.java	100% 2/2	59% 23/39	86% 36/42	100% 5/5
LengthOfVariableToVariableAssignment.java	100% 1/1	82% 28/34	100% 51/51	100% 3/3
LogicBinaryExpressionToVariableAssignment.java	100% 2/2	28% 9/32	52% 32/62	57% 4/7
LogicBitwiseExpressionToVariableAssignement.java	100% 2/2	45% 20/44	80% 41/51	100% 6/6
MaxIntervalToVariableAssignment.java	100% 1/1	38% 3/8	93% 13/14	100% 3/3
MinMaxToVariableAssignment.java	100% 1/1	48% 22/46	79% 42/53	83% 5/6
PointerSubtractionToVariableAssignment.java	100% 1/1	54% 14/26	86% 24/28	100% 3/3
ResourceStateComparisonToVariableAssignment.java	100% 1/1	69% 29/42	91% 43/47	100% 3/3
SetConstantIntervalToVariable.java	100% 1/1	45% 9/20	67% 36/54	100% 6/6
SetLengthOfVariable.java	100% 1/1	70% 39/56	94% 66/70	100% 4/4
SetVariableToVariable.java	100% 1/1	50% 1/2	60% 6/10	67% 2/3
SubtractionToVariableAssignment.java	100% 1/1	75% 3/4	100% 11/11	100% 2/2

Пример



Name	Classes	Conditionals	Lines	Methods
MemObjectCollection.java	67% 2/3	32% 129/409	53% 179/338	53% 18/34

Name	Conditionals	Lines	Methods
MemObjectCollection	31% 125/403	53% 175/333	52% 16/31
MemObjectCollection$1	67% 4/6	83% 5/6	100% 2/2
MemObjectCollection$2	N/A	0% 0/1	0% 0/1

```
1   /*
2    * $Id: MemObjectCollection.java 17444 2010-06-07 12:23:03Z tsesko $
3    */
4
5   package s2a.decision.impl;
6
7   import java.util.Collection;
8   import java.util.Collections;
9   import java.util.Comparator;
10  import java.util.HashMap;
11  import java.util.HashSet;
12  import java.util.Iterator;
13  import java.util.LinkedList;
14  import java.util.List;
15  import java.util.Map;
16  import java.util.Set;
17  import java.util.TreeSet;
18  import s2a.cfg.api.exprs.BinaryExpression.CBinaryOperation;
19  import s2a.decision.api.AbstractDecisionFactory;
20  import s2a.decision.api.Conditionable;
21  import s2a.decision.api.IntervalCollection;
22  import s2a.decision.api.IntervalElement;
23  import s2a.decision.api.ObjectCollection;
```

Пример

```
53 1984     }
54
55          @Override
56          public String toString() {
57    96        final StringBuilder sb = new StringBuilder();
58
59    96        final List<ObjectElement> data = new LinkedList<ObjectElement>(this);
60   309        Collections.sort(data, new Comparator<ObjectElement>() {
61                  @Override
62                  public int compare(ObjectElement o1, ObjectElement o2) {
63   117                if (o1.isOrdinary() && o2.isOrdinary()) {
64   117                    if (o1.getObject() == o2.getObject()) {
65    32                        return o1.getShift() - o2.getShift();
66                          } else {
67    85                        return o1.getObject().toString()
68                                      .compareTo(o2.getObject().toString());
69                          }
70                      } else {
71     0                    return o1.toString().compareTo(o2.toString());
72                      }
73                  }
74              });
75
76    96        final Iterator<ObjectElement> iterator = data.iterator();
77   256        while (iterator.hasNext()) {
78   160            final ObjectElement e = iterator.next();
79   160            sb.append(e);
80   160            if (iterator.hasNext()) {
81    96                sb.append(", ");
82                  }
83   160        }
84
85    96        return sb.toString();
86          }
87
88          @Override
89          public boolean equals(final Object o) {
90   224        if (o == this) {
91    32            return true;
92              }
93
94   192        if (!(o instanceof Collection)) {
95     0            return false;
96              }
97
98              @SuppressWarnings("unchecked")
99   192        final Collection<ObjectElement> c = (Collection<ObjectElement>) o;
100
101  192        if (c.size() != size()) {
102   64            return false;
103             }
104
105             try {
106  128            return containsAll(c);
107    0        } catch (ClassCastException unused) {
108    0            return false;
109    0        } catch (NullPointerException unused) {
110    0            return false;
111             }
112         }
113
114         @Override
115         public int hashCode() {
116    0        return super.hashCode();
```

Пример

```
493       }
494
495       public IntervalCollection minus(
496               final ObjectCollection other) {
497   0       assert other != null;
498           assert !this.containsNonInit() && !other.containsNonInit():
499   0           "???? ?????????????????????? ?????????????????? ?????? ????????????????????????????????????????????????????? ??????????????????????";
500
501           // ?????? ???????? ????????????????????? ?????????????????? ??????????????????????? ?????????????????????????? ?????????????????
502   0       if (contains(o_invalid) || other.contains(o_invalid)) {
503   0           return factory.singleton(i_max);
504           }
505
506           // ???????????? ??????????????????????
507
508   0       final IntervalCollection result = factory.createIntervalCollection();
509
510   0       for (ObjectElement ii : this) {
511   0           for (ObjectElement jj : other) {
512   0               if (ii.getObject().equals(jj.getObject())) {
513   0                   final int shift = ii.getShift() - jj.getShift();
514   0                   result.add(factory.createInterval(shift, shift));
515   0               } else {
516                       // ???????? ??????????????????? ????????????????????? ???? ???????????? ?????????????
517   0                   return factory.singleton(i_max);
518               }
519           }
520       }
521
522   0       return result;
523       }
524
525       public IntervalCollection compare(final ObjectCollection other) {
526   0       boolean maximumIsNeeded = false;
527   0       boolean zeroIsNeeded = false;
528
529   0       final IntervalCollection result = factory.createIntervalCollection();
530
531           // ?????? ???????? ????????????????????? ?????????????????? ??????????????????????? ?????????????????????????? ?????????????????
532   0       if (contains(o_invalid) || other.contains(o_invalid)) {
533   0           maximumIsNeeded = true;
534           }
535
536           // ???????????? ??????????????????????
537   0       for (ObjectElement ii : this) {
538   0           if (ii.isSingle()) {
539   0               for (ObjectElement jj : other) {
540   0                   if (jj.isSingle()) {
541   0                       if (ii.getObject().equals(jj.getObject())) {
542   0                           zeroIsNeeded = true;
543                           } else {
544                               // ???????? ????????????????????? ????????????????????? ???? ???????????? ?????????????
545   0                           maximumIsNeeded = true;
546                           }
547                       }
548               }
549           }
550       }
551
552   0       if (maximumIsNeeded) {
553   0           result.add(i_max_point);
554           }
555
556   0       if (zeroIsNeeded) {
```

Что еще мы можем сделать с «белым ящиком»?

- Зная внутреннее устройство модуля, мы можем измерить то, насколько полно мы его тестируем
- А также можем оптимизировать написание тестов, используя информацию о тестовом покрытии

Может быть, кто-нибудь сможет предложить что-нибудь еще?

Что еще мы можем сделать с «белым ящиком»?

- Мы всегда можем забраться внутрь модуля и что-нибудь там изменить
  - Зачем?
- Чтобы оценить полноту нашего тестирования – но в другой плоскости
  - Идеальный тест работает только на тестируемой программе
  - При любом изменений он перестает проходить

В этом заключается основная идея «мутационного тестирования»

Мутационное тестирование

- Исходная программа подвергается мутации, в результате получается набор из N мутантов
- После этого имеющиеся тесты запускаются на этих мутантах
  - Если тест не проходит на мутанте, то говорят, что тест «убивает» этого мутанта
- Доля «убитых» мутантов показывает, насколько полно данный набор тестов покрывает нашу программу

Недостатки?

Мутационное тестирование

- Основная сложность заключается в том, что данный вид тестирования практически невозможно выполнять вручную
  - Количество необходимых для оценки покрытия мутантов пропорционально объему анализируемого ПО
  - Понятно, что даже для небольшой программной системы получение достаточного числа мутантов вручную является невозможным
- Кроме того, сильно возрастают затраты на проведение тестирования
  - Вместо одного запуска каждого теста требуется выполнить N запусков

Содержание

Что еще мы можем сделать с «белым ящиком»?

- Мы всегда можем забраться внутрь модуля и что-нибудь там изменить
  - Зачем?

Тест должен обеспечивать выполнение трех основных свойств

- Reachibility
- Corruption
- Propagation

Что еще мы можем сделать с «белым ящиком»?

- Мы всегда можем забраться внутрь модуля и что-нибудь там изменить
  - Зачем?

Тест должен обеспечивать выполнение трех основных свойств

- Reachibility
- Corruption
- Propagation

Как можно обеспечить выполнение этих свойств?

1. Можно надеяться на то, что данные свойства будут выполняться сами по себе
   - Как показывает многолетний опыт, если при разработке ПО не думают о RCP, то тестировать данное ПО будет очень сложно
   - Адаптировать ПО для тестирования, если RCP нет, также практически невозможно
2. Можно заранее обеспокоиться выполнением RCP

Design for Testability (D4T)

ПО разрабатывается

- управляемыми – разработчик может относительно просто заставить программу делать именно то, что ему требуется при тестировании
- наблюдаемыми – разработчику всегда доступна вся информация, необходимая для анализа поведения программы в каждый момент времени

Как можно обеспечить выполнение этих свойств?

- Способы D4T зависят от того, какое ПО мы разрабатываем

Test-Driven Development

- Наиболее известный способ обеспечения D4T
- Единственный способ, применимый практически к любому ПО
- Основной принцип – тесты разрабатываются перед исходным кодом

Зачем?

Test-Driven Development

Чтобы обеспечить управляемость и наблюдаемость

- Наблюдаемость обеспечивается за счет того, что каждый тест проверяет какой-либо набор предположений о тестируемой программе
  - Корректность этих предположений должна быть видна снаружи программы
  - Для каждого теста проверяется как успешное, так и неудачное выполнение теста
- Управляемость обеспечивается за счет того, что тесты пишутся для минимальных элементов функциональности
  - Никакой дополнительной функциональности, которая не покрыта тестами, нет
  - Любое возможное действие имеет соответствующий тест

Преимущества и недостатки?

Test-Driven Development

- TDD стимулирует к написанию маленьких, локальных модулей
- Ошибки обнаруживаются рано, что снижает стоимость их исправления
- Разработка идет от функциональности, что приводит к созданию более простых и понятных интерфейсов

- Разработка тестов требует дополнительного времени
- При исправлении ошибки, сделанной на ранних этапах разработки, требуется модифицировать или написать с нуля большое число тестов
- Эффект «розовых очков»

Способы обеспечения управляемости

- Мы можем практически полностью управлять тем кодом, что разработан нами
  - Этим вопросом занимаются такие дисциплины как Software Architecture, Software Analysis & Design, etc.
- Кроме этого, в ПО обычно есть значительная часть кода, с которым мы ничего сделать не можем
  - сторонние библиотеки
  - системные библиотеки ОС
  - legacy-код

Как управлять ими?

Simulate & Stub

Никак

- Критически важные сторонние компоненты необходимо эмулировать или заменять заглушками (Simulate & Stub → S&S)
  - Заглушки могут работать значительно быстрее, чем сторонний код
  - Заглушки являются полностью управляемыми элементами
    - С их помощью можно имитировать редкие ошибки (например, сбои в аппаратуре)
    - Заглушки могут «вносить в систему детерминизм» там, где его нет
  - S&S может обеспечить «обратную масштабируемость» (downwards scalability)

Simulate & Stub

- Пример S&S – mock-объекты
- Заглушки в ООП
- Могут быть легко заменены (при правильном проектировании) на реальные объекты при развертывании приложения
- Часто работают на порядок быстрее реальных объектов
- Стимулируют разработку модульного, слабо связанного кода

Способы обеспечения управляемости

А все же – что мы можем сделать с нашим кодом?

- Правило C2POF
  - Consistent test run results
  - Configuration is not needed
  - Partial testing is possible
  - Order of tests is not important
  - Fast run time

C2POF

- Результаты тестов должны быть повторяемыми
  - В тесте не должно быть недетерминированного поведения
  - Необходимо минимизировать зависимость теста от его окружения
  - S&S – но уже внутри нашего кода
- Тесты не должны требовать дополнительной конфигурации
  - Это касается как самих тестов, так и тестируемых модулей программы
  - Иначе в каждом тесте требуется выполнять рутинную работу по заданию каких-либо параметров
  - S&S – но уже внутри нашего кода
- Тесты должны выполняться быстро
  - Когда число тестов становится достаточно большим, Вы не можете тратить по 10 минут каждые полчаса на запуск тестов
  - Но тесты необходимо запускать часто – чтобы находить ошибки как можно раньше
  - S&S – но уже внутри нашего кода

C2POF

- Тесты должны допускать возможность частичного запуска
  - То есть между тестами не должно быть никаких зависимостей по выполнению
  - В противном случае возникают сложные, мало кому понятные ошибки при изменении отдельных тестов
- Тесты должны допускать возможность запуска в любом порядке
  - То есть между тестами не должно быть никаких зависимостей по выполнению
  - Если Вы не можете добиться этого – значит, Вы делаете что-то не так

- Какие из этих свойств важнее?
- Какие из этих свойств сложнее в обеспечении?

Способы обеспечения наблюдаемости

- Некоторые способы обеспечения наблюдаемости мы уже рассмотрели «между строк» ранее
  - Например, разработка на основе маленьких, локальных модулей

Что еще можно сделать?

- Мы всегда можем забраться внутрь модуля и что-нибудь там изменить
- Чтобы обеспечить выполнение свойства распространяемости сбоя
- Это, фактически, и означает обеспечение наблюдаемости

Способы обеспечения наблюдаемости

- Некоторые способы обеспечения наблюдаемости мы уже рассмотрели «между строк» ранее
  - Например, разработка на основе маленьких, локальных модулей

Что еще можно сделать?

- Мы всегда можем забраться внутрь модуля и что-нибудь там изменить
  - Зачем?
  - Чтобы обеспечить выполнение свойства распространяемости сбоя
  - Это, фактически, и означает обеспечение наблюдаемости

Способы обеспечения наблюдаемости

- Некоторые способы обеспечения наблюдаемости мы уже рассмотрели «между строк» ранее
  - Например, разработка на основе маленьких, локальных модулей

Что еще можно сделать?

- Мы всегда можем забраться внутрь модуля и что-нибудь там изменить
  - Зачем?
- Чтобы обеспечить выполнение свойства распространяемости сбоя
- Это, фактически, и означает обеспечение наблюдаемости

Способы обеспечения наблюдаемости

- Выделяют три основных способа обеспечения наблюдаемости
  - Использование assert'ов
  - Проверка состояния программы при выполнении
  - Журналирование

Assertions

- Контролируют некоторые факты о состоянии программы при необходимости
- Превращают некоторые сбои в неудачи
  - Причем по такой неудаче относительно просто понять причину ее возникновения
- Могут использоваться как при тестировании, так и во время реальной эксплуатации ПО

Способы обеспечения наблюдаемости

Invariant checkers

- Проверяют корректность состояния программы в какой-то точке выполнения
  - Могут рассматриваться как более масштабные assert'ы
- Корректность может выражаться как набор довольно сложных инвариантов
- Могут обнаруживать хитрые «скрытые» сбои
- Крайне ресурсоемки, поэтому используются только при тестировании

Способы обеспечения наблюдаемости

Logging

- Самый часто используемый способ обеспечения наблюдаемости
- Записывают трассы выполнения программы вместе с ее состоянием в удобной для разработчика форме
- Предоставляют разработчику очень много информации
  - Это позволяет разобраться в причинах некорректной работы практически в любом случае
  - Это может сделать работу с трассами практически невозможной

Тестирование некоторых классов приложений

- Простота обеспечения управляемости и наблюдаемости зависит от того, какое ПО мы разрабатываем
- Рассмотрим некоторые проблемы, которые возникают при тестировании
  - объектно-ориентированных приложений
  - приложений с GUI
  - Веб-приложений

Тестирование ОО-приложений

- В чем основные отличия ОО-приложений?
  - Наследование
  - Полиморфизм
  - Инкапсуляция

Как они влияют на управляемость и наблюдаемость?

Тестирование ОО-приложений

- В чем основные отличия ОО-приложений?
  - Наследование
  - Полиморфизм
  - Инкапсуляция

Как они влияют на управляемость и наблюдаемость?

Тестирование ОО-приложений

- Наследование и полиморфизм усложняют обеспечение управляемости
  - Необходимо знать не только то, что делает тестируемый модуль, но и все модули в его иерархии наследования
  - Изменения в модулях могут приводить к вызову других полиморфных методов, что, в свою очередь, может изменить результаты тестирования
- Инкапсуляция усложняет наблюдаемость
  - Собственно, ее основная задача – это скрыть подробности реализации
  - У нас нет стандартных средств определения внутреннего состояния объектов

Тестирование приложений с GUI

- В чем основные отличия приложений с GUI?
  - В GUI
  - Основной недостаток GUI для тестирования – крайне большое возможное число состояний
  - Кроме того, GUI обычно очень контекстно-чувствительны

Как это влияет на управляемость и наблюдаемость?

Тестирование приложений с GUI

- Управляемость GUI обычно крайне низка
  - Часто с этим ничего невозможно сделать
  - В некоторых случаях GUI имеет средства автоматизации
  - Для больших интерфейсов даже этого может быть мало
- Наблюдаемость GUI обычно крайне низка
  - Анализировать визуальные ошибки автоматически очень сложно
  - К сожалению, классические средства обеспечения наблюдаемости слабо подходят для работы с GUI

Тестирование Веб-приложений

- В чем основные отличия Веб-приложений?
  - Распределенная архитектура
  - Активная работа с базами данных
  - Крайне сложное взаимодействие с пользователем

Как они влияют на управляемость и наблюдаемость?

Тестирование Веб-приложений

- Распределенная архитектура несколько помогает при тестировании
  - ПО всегда можно «разрезать» по какой-либо плоскости и использовать S&S
  - Взаимодействие по сети просто отслеживать и анализировать
- БД усложняют обеспечение как управляемости, так и наблюдаемости
  - БД – это огромное хранимое состояние программы, которое влияет на ее работу
  - Понять, что и как хранится в БД, очень сложно

Содержание

Homework

Написать эссе объемом не менее 3000 символов на одну из следующих тем

- Какое из свойств C2POF важнее и почему?
- Какое из свойств C2POF сложнее в обеспечении и почему?
- Какой из способов обеспечения управляемости лучше и почему?
- Какой из способов обеспечения наблюдаемости лучше и почему?