Основы тестирования программного обеспечения

Software Testing 102

Марат Ахин

Санкт-Петербургский государственный политехнический университет

2011

Содержание

Методы обеспечения качества ПО

Статический анализ

Проверка моделей

Дедуктивная верификация

Software Testing 102

- 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 102

- 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 так сказал преподаватель!
- Потому что тестирование является основой практически любой методологии проектирования и разработки ПО, которые используются в настоящее время

Почему разработчик должен уметь тестировать ПО?

- Потому что так сказал преподаватель!
- Потому что тестирование является основой практически любой методологии проектирования и разработки ПО, которые используются в настоящее время

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

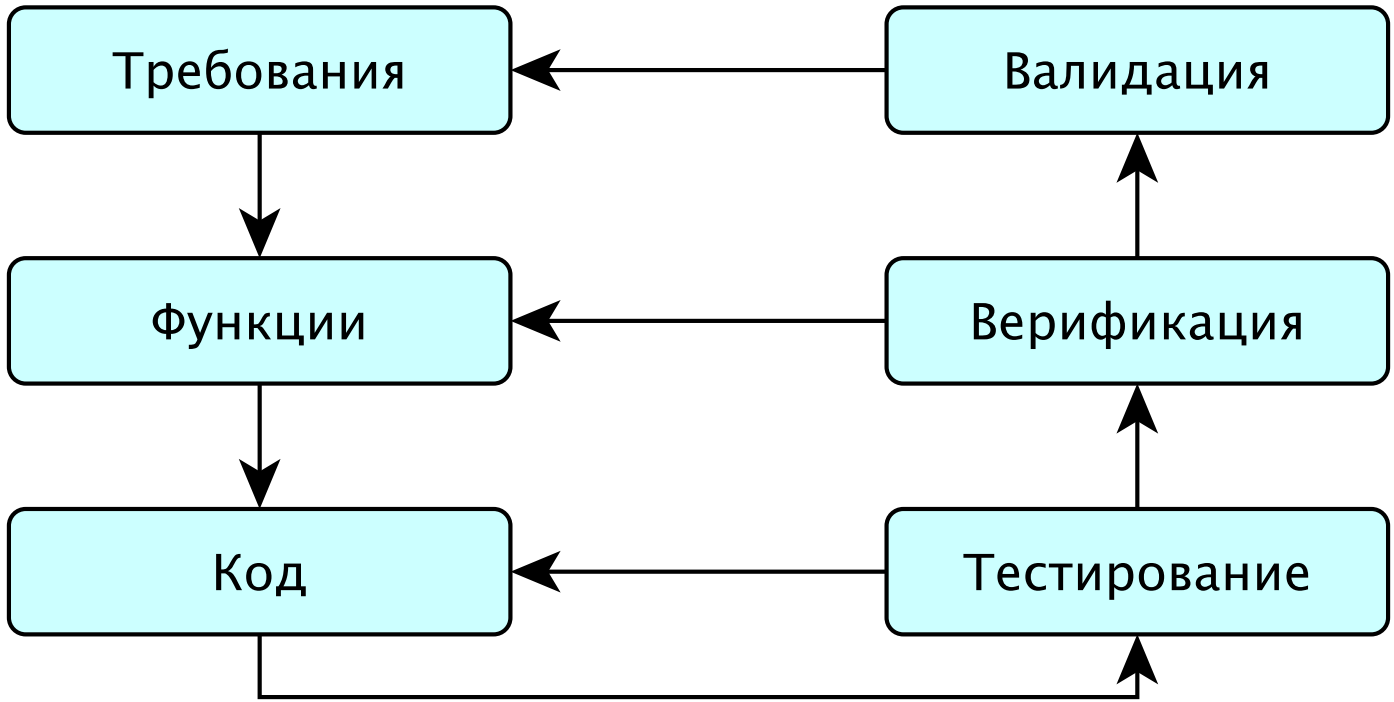
Работает ли это ПО неправильно?

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

Тестирование = Разрушение

Что такое тестирование?

- Лучший друг верификации и валидации ПО
- В чем разница?
  - Верификация – «мы сделали это правильно»
  - Валидация – «мы сделали то, что надо»



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 это так – мы рассмотрим в следующих лекциях

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.»

Чтобы сделать тестирование более приятным процессом, необходимо понять его характерные особенности и свойства

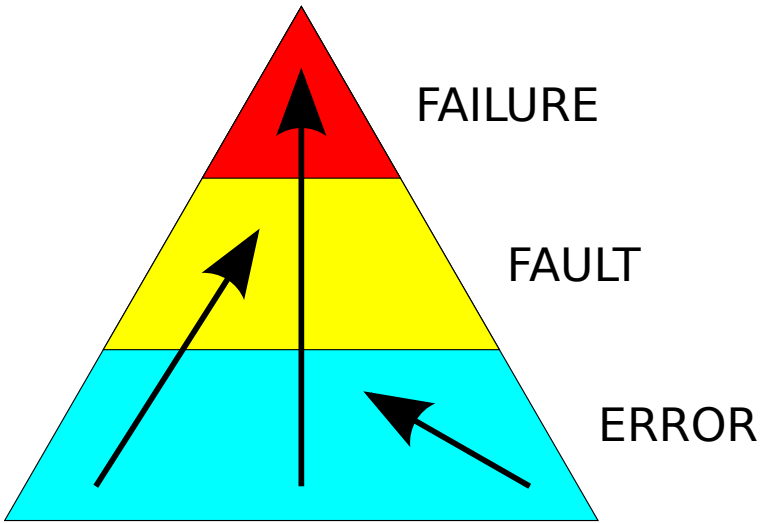
Содержание

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

Модель программной ошибки

Найдите ошибку в следующей программе на Java

```
1 int sumCollection(final Collection<Integer> c) {
2     int sum = 0;
3     for (int i : c) {
4         sum += i;
5     }
6     return sum;
7 }
```

Возможное переполнение в строке 4

Модель программной ошибки

Найдите ошибку в следующей программе на Java

```
1 int sumCollection(final Collection<Integer> c) {
2     int sum = 0;
3     for (int i : c) {
4         sum += i;
5     }
6     return sum;
7 }
```

Возможное переполнение в строке 4

Модель программной ошибки

c = {}

Что будет?

```
1 int sumCollection(final Collection<Integer> c) {
2     int sum = 0;
3     for (int i : c) {
4         sum += i;
5     }
6     return sum;
7 }
```

Нет ни сбоя, ни неудачи – программа работает корректно

Модель программной ошибки

c = {}

Что будет?

```
1 int sumCollection(final Collection<Integer> c) {
2     int sum = 0;
3     for (int i : c) {
4         sum += i;
5     }
6     return sum;
7 }
```

Нет ни сбоя, ни неудачи – программа работает корректно

Модель программной ошибки

```
c = {1, 2, 3, 5}
```

Что будет?

```
1 int sumCollection(final Collection<Integer> c) {
2     int sum = 0;
3     for (int i : c) {
4         sum += i;
5     }
6     return sum;
7 }
```

Нет ни сбоя, ни неудачи – программа работает корректно

Модель программной ошибки

```
c = {1, 2, 3, 5}
```

Что будет?

```
1 int sumCollection(final Collection<Integer> c) {
2     int sum = 0;
3     for (int i : c) {
4         sum += i;
5     }
6     return sum;
7 }
```

Нет ни сбоя, ни неудачи – программа работает корректно

Модель программной ошибки

```
c = {1, 2, 3, 5, +MAX_INT, -MAX_INT}
```

Что будет?

```
1 int sumCollection(final Collection<Integer> c) {
2     int sum = 0;
3     for (int i : c) {
4         sum += i;
5     }
6     return sum;
7 }
```

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

Модель программной ошибки

`c = {1, 2, 3, 5, +MAX_INT, -MAX_INT}`

Что будет?

```
1 int sumCollection(final Collection<Integer> c) {
2     int sum = 0;
3     for (int i : c) {
4         sum += i;
5     }
6     return sum;
7 }
```

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

Модель программной ошибки

`c = {1, 2, 3, 5, +MAX_INT}`

Что будет?

```
1 int sumCollection(final Collection<Integer> c) {
2     int sum = 0;
3     for (int i : c) {
4         sum += i;
5     }
6     return sum;
7 }
```

Сбой есть – программа проходит через некорректное состояние

Неудача тоже есть – результат работы программы неправильный

Модель программной ошибки

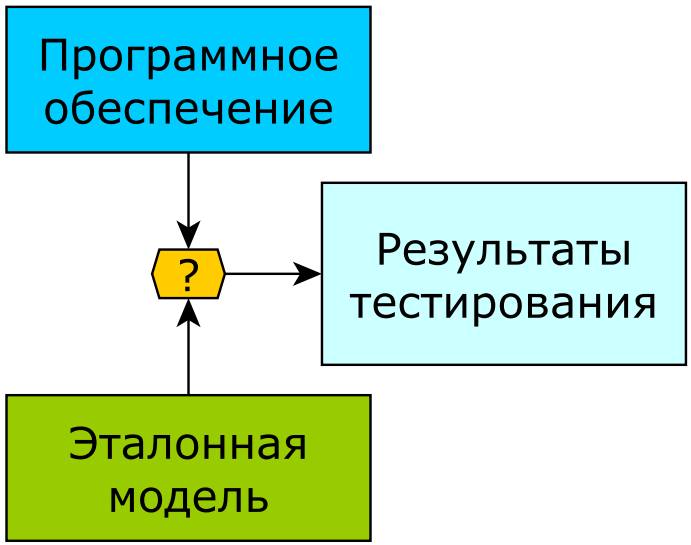
`c = {1, 2, 3, 5, +MAX_INT}`

Что будет?

```
1 int sumCollection(final Collection<Integer> c) {
2     int sum = 0;
3     for (int i : c) {
4         sum += i;
5     }
6     return sum;
7 }
```

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

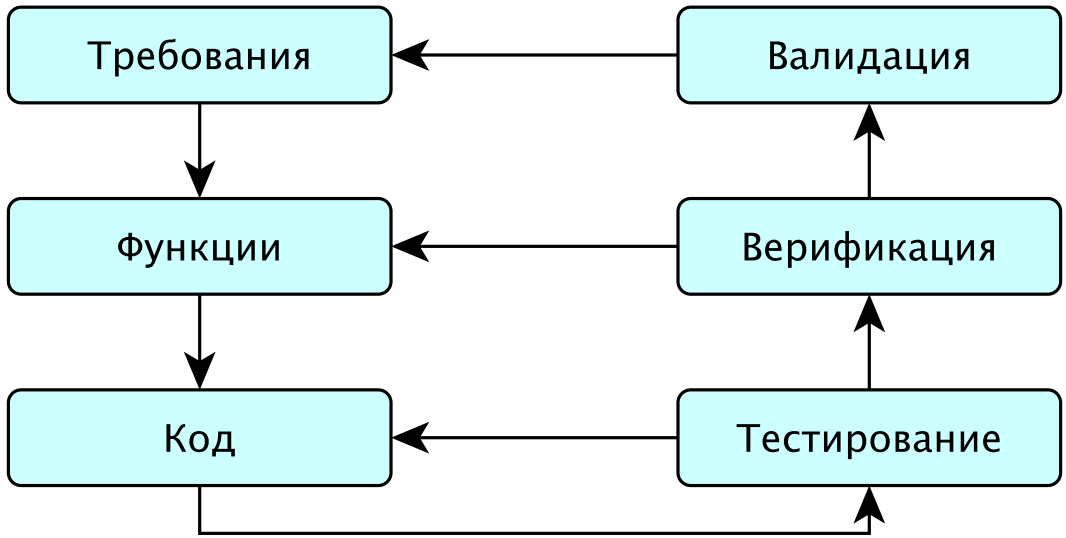
Свойства теста

О том, как обеспечить выполнение этих свойств, мы поговорим на следующей лекции

А сейчас...

Как именно мы можем наблюдать за ошибками в работе программы?

Можно выделить три основных уровня, на которых мы можем проверять корректность работы программы



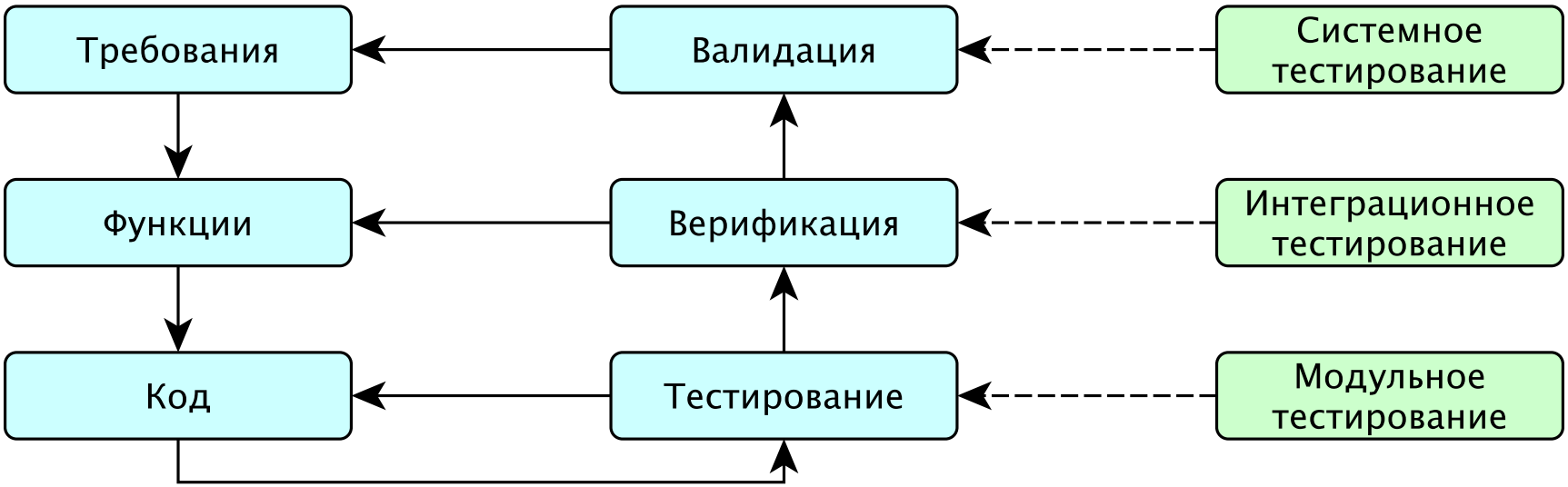
Свойства теста

О том, как обеспечить выполнение этих свойств, мы поговорим на следующей лекции

А сейчас...

Как именно мы можем наблюдать за ошибками в работе программы?

Можно выделить три основных уровня, на которых мы можем проверять корректность работы программы



Содержание

Модульное тестирование

- Модульное тестирование – проверка корректности работы отдельных модулей программы
- Модуль – это
  - Набор входов
  - Некий «ящик», выполняющий какие-либо преобразования
  - Набор выходов

Вопрос – что мы **вообще** можем сделать с модулем?

Модульное тестирование

Мы можем

- Подать модулю на вход какие-либо данные
- Считать с выходов результаты его работы

К сожалению, это **ВСЕ**, что мы можем сделать с модулем

Как можно тестировать ПО в таких ужасных условиях?

Модульное тестирование

Мы можем

- Подать модулю на вход какие-либо данные
- Считать с выходов результаты его работы

К сожалению, это **ВСЕ**, что мы можем сделать с модулем

Как можно тестировать ПО в таких ужасных условиях?

Модульное тестирование

Решение «в лоб»

- Протестировать модуль на всех возможных комбинациях входов
- Чтобы протестировать функцию, которая перемножает два 32-разрядных числа, необходимо перебрать 2^{64} вариантов.
- Если на проверку каждого варианта тратить по 1 наносекунде, нам понадобится *всего лишь* 584 года

Необходимо как-то уменьшить число перебираемых вариантов

Модульное тестирование

- Все зависит от того, знаем ли мы что-нибудь о том самом «ящике»
  - Тестирование черного ящика – известна только внешняя спецификация модуля
  - Тестирование белого ящика – известно полное внутреннее устройство модуля
  - Тестирование серого ящика – известны некоторые детали устройства модуля

Тестирование черного ящика

- Мы ничего не знаем об устройстве модуля
- Мы знаем только то, как он должен себя вести с точки зрения собственного окружения

Разбиение на классы эквивалентности

- Идея заключается в том, чтобы разбить пространство входных состояний программы на небольшое число классов эквивалентности таким образом, чтобы все элементы одного класса эквивалентности обрабатывались модулем одинаково **с точки зрения спецификации**
- В случае, если при написании модуля спецификация была полностью учтена, тестирование всех классов эквивалентности означает полное тестирование данного модуля

Тестирование черного ящика

Пример

- Вход – пара чисел (x, y)
- Выход – строка, характеризующая отношение между x и y
  - $x < y$
  - $x > y$
  - $x = y$
  - *ERROR* (если на вход модуля был подан объект, не являющийся парой чисел)

В данном случае все просто и понятно...

Тестирование черного ящика

Пример

- Вход – три числа x, y и z
- Выход
  - $x \cdot y \cdot z$, если x – целое число, $y > 0$, $z > 0$
  - $x \cdot y$, если x – целое число, $x < 42$, $y > z^2$
  - x^z, если x представимо рациональной дробью по основанию 13, $z < 42$
  - ...

В этом случае все уже не столь очевидно...

Тестирование черного ящика

Необходимо использовать какую-либо стратегию **комбинирования** ортогональных друг другу классов эквивалентности

Пример

Расчет городского налога

- Нерезиденты платят 1% от общего дохода
- Резиденты платят
  - 1% от дохода, если он не превышает 300,000 в год
  - 5% от дохода, если он не превышает 600,000 в год
  - 15% от дохода, если он превышает 600,000 в год

Тестирование черного ящика

Некоторые стратегии дают разную схему комбинирования классов эквивалентности для разных спецификаций, даже если эти спецификации задают одинаковое поведение модуля

Пример

Расчет городского налога

- Если доход не превышает 300,000 в год, налог составляет 1%
- Если доход не превышает 600,000 в год
  - Для нерезидентов налог составляет 1% от общего дохода
  - Для резидентов налог составляет 5% от общего дохода
- Если доход превышает 600,000 в год
  - Для нерезидентов налог составляет 1% от общего дохода
  - Для резидентов налог составляет 15% от общего дохода

Причинно-следственный анализ

Анализ причинно-следственных связей

- Систематический подход к перебору возможных причин (входов) и следствий (выходов) и отношений между ними
- Под причиной может пониматься как конкретное значение входа, так и определенный класс эквивалентности
- Под следствием обычно понимается конкретное значение выхода
  - Но иногда это может также быть какой-либо класс эквивалентности

Причинно-следственный анализ

1: Определение причин и следствий

- Самый важный и самый сложный этап анализа
- Крайне важно выбрать правильный уровень абстракции
- Причины не обязательно должны быть взаимоисключающими

2: Вывод логических отношений и ограничений

- Задаются при помощи логических операций И, ИЛИ, НЕ
- Описывают возможные комбинации причин с учетом анализируемой предметной области и здравого смысла

Причинно-следственный анализ

3: Выбор подходящей стратегии выбора тестов

- С помощью стратегии задается то, какие комбинации причин будут выбираться при проведении тестов
- В зависимости от выбранной стратегии будут получаться те или иные показатели полноты и стоимости тестирования

4: Составить план тестирования

- Требует внимательного анализа причинно-следственных зависимостей в соответствии со стратегией выбора тестов
- Обычно это **крайне** трудоемкий процесс
  - Что означает возможность ошибки при подготовке к тестированию
  - А это очень и очень плохо – поэтому желательно использовать средства автоматизации

Причинно-следственный анализ

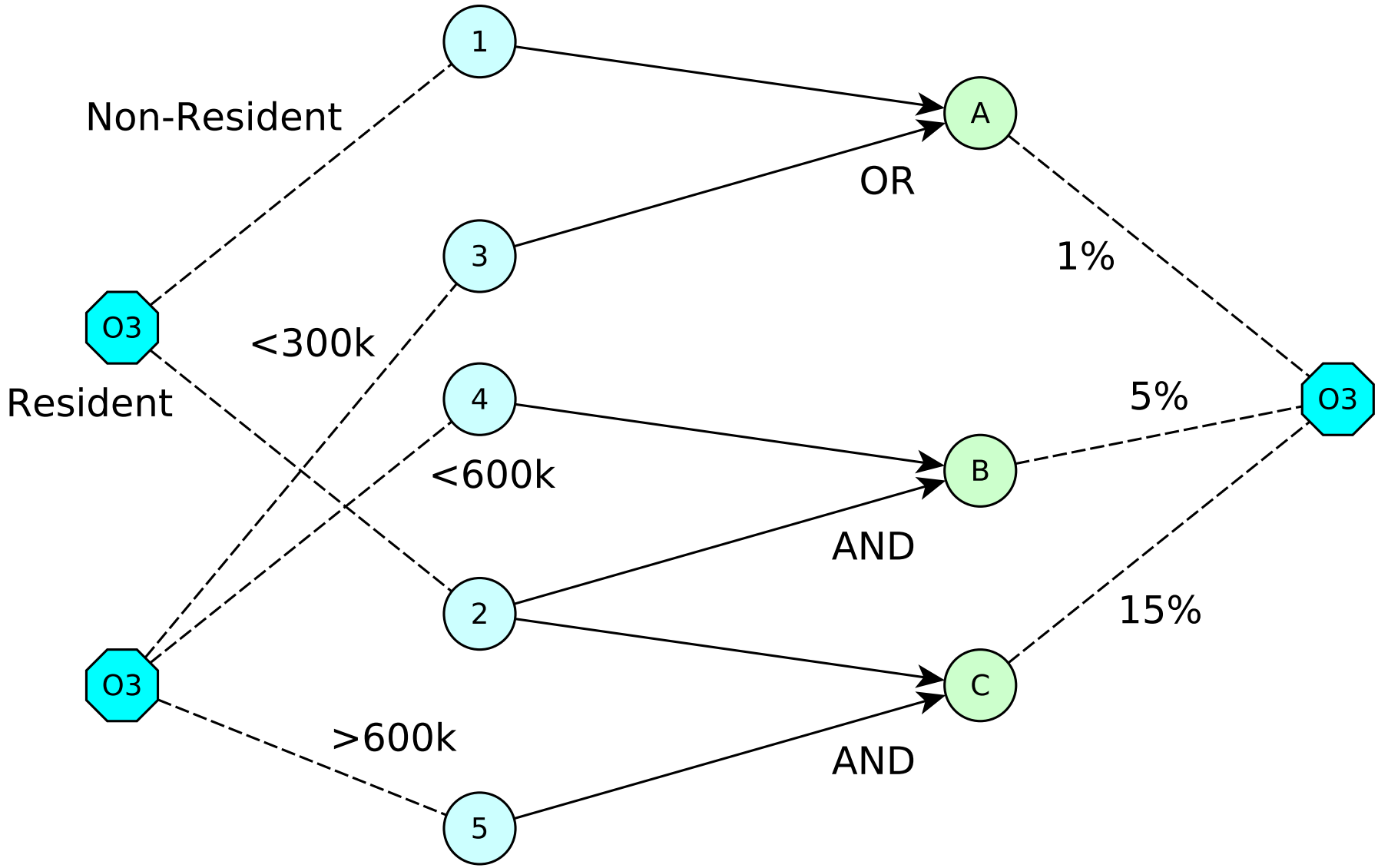
Пример

Расчет городского налога

- Нерезиденты платят 1% от общего дохода
- Резиденты платят
  - 1% от дохода, если он не превышает 300,000 в год
  - 5% от дохода, если он не превышает 600,000 в год
  - 15% от дохода, если он превышает 600,000 в год

Удобным способом представления причинно-следственных связей являются булевы графы

Причинно-следственный анализ



Причинно-следственный анализ

Возможные стратегии выбора тестов

1. Все возможные комбинации причин
2. Все возможные эффекты при минимальном общем числе тестов
3. Все возможные эффекты для всех возможных при этом комбинаций причин
4. Стратегия 3 с набором дополнительных правил
   - Для узлов типа И рассматриваем только такие комбинации причин, в которых
     - либо все входы истинны
     - либо лишь один вход ложен
   - Для узлов типа ИЛИ рассматриваем только такие комбинации причин, в которых
     - либо все входы ложны
     - либо лишь один вход истинен

Другие методы тестирования черного ящика

- Анализ граничных значений
- Exploratory testing
  - Метод *научного тыка*
  - Чертовски эффективный в случае правильного его применения
  - Необходимо выйти за рамки разумного и пытаться сломать систему настолько нестандартными способами, какие Вы только можете себе представить
- Ad hoc testing
  - Самый неформальный метод тестирования
  - Метод *научного тыка*, возведенный в абсолют
  - Нет ни предварительного планирования, ни документирования процесса тестирования

Exploratory testing

Exploratory testing is simultaneous learning, test design, and test execution

- Test design
- Careful observation
- Critical thinking
- Diverse ideas
- Rich resources

Exploratory testing

- Input/Output attacks
  - Attacks by input value
  - Attacks by input value combination
  - Attacks by input order
- Data attacks
  - Attacks by variable value
  - Attacks by data element size
  - Attacks by data access
- Computation attacks
  - Attacks by operand
  - Attacks by result
  - Attacks by feature interaction

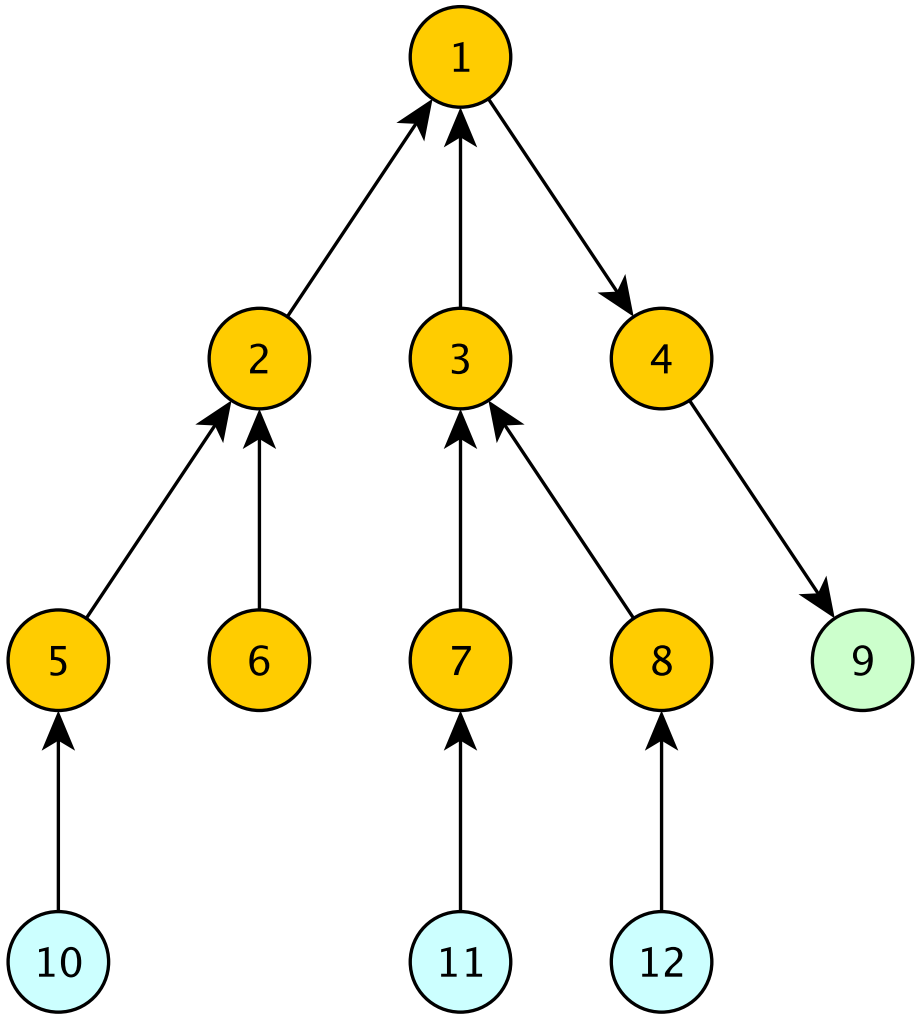
Тестирование белого ящика

Будет рассмотрено на следующей лекции

Интеграционное и системное тестирование

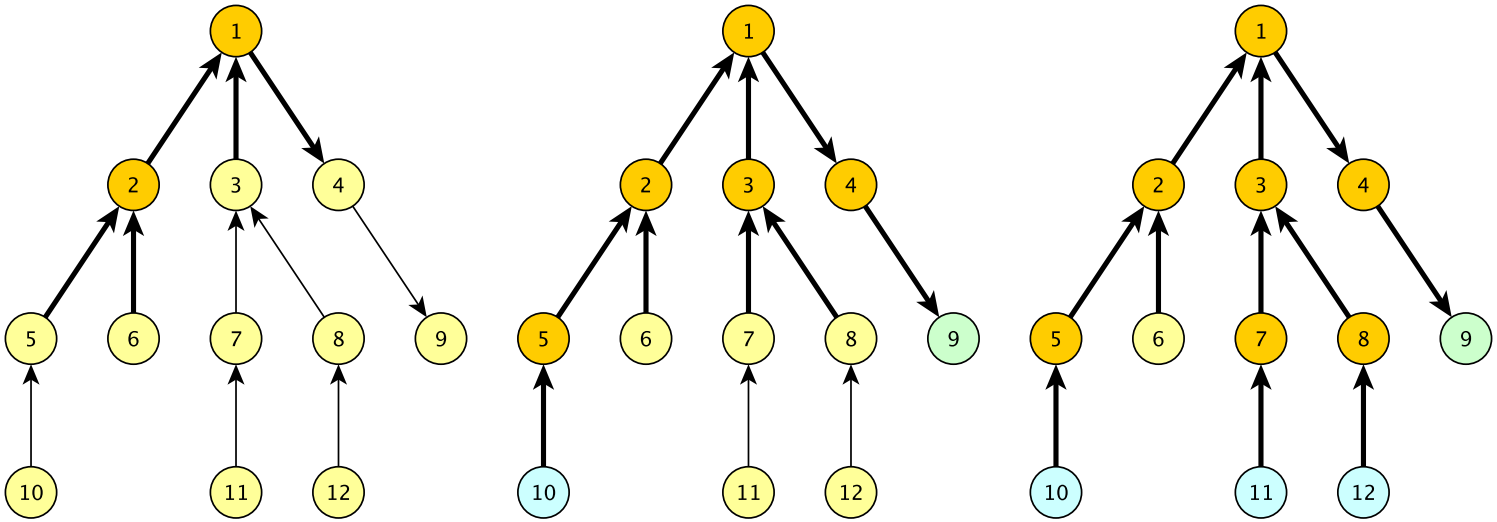
- Тестирование корректности работы нескольких модулей вместе в составе программной системы
- Два основных подхода
  - Моментальный
    - Тестирование взаимодействия начинается после написания всех отдельных модулей
    - Такое тестирование также известно как тестирование методом «Большого Взрыва»
  - Инкрементальный
    - Тестирование взаимодействия выполняется постоянно, в процессе разработки
    - Вопрос в том, о какой инкрементальности мы говорим

Интеграционное и системное тестирование



Нисходящее интеграционное тестирование

- Тестирование начинается с верхних уровней системы
- Отсутствующие на данный момент модули заменяются «заглушками»
- По мере реализации новых модулей они подключаются к системе вместо «заглушек»



Нисходящее интеграционное тестирование

Преимущества

- Возможность ранней проверки корректности высокоуровневого поведения
- Модули могут добавляться по одному, независимо друг от друга
- Не требуется разработка множества драйверов
- Можно разрабатывать систему как в глубину, так и в ширину

Недостатки

- Отложенная проверка низкоуровневого поведения
- Требуется разработка «заглушек»
- Крайне сложно корректно сформулировать требования ко входам/выходам частичной системы

Нисходящее интеграционное тестирование

Преимущества

- Возможность ранней проверки корректности высокоуровневого поведения
- Модули могут добавляться по одному, независимо друг от друга
- Не требуется разработка множества драйверов
- Можно разрабатывать систему как в глубину, так и в ширину

Недостатки

- Отложенная проверка низкоуровневого поведения
- Требуется разработка «заглушек»
- Крайне сложно корректно сформулировать требования ко входам/выходам частичной системы

Нисходящее интеграционное тестирование

Преимущества

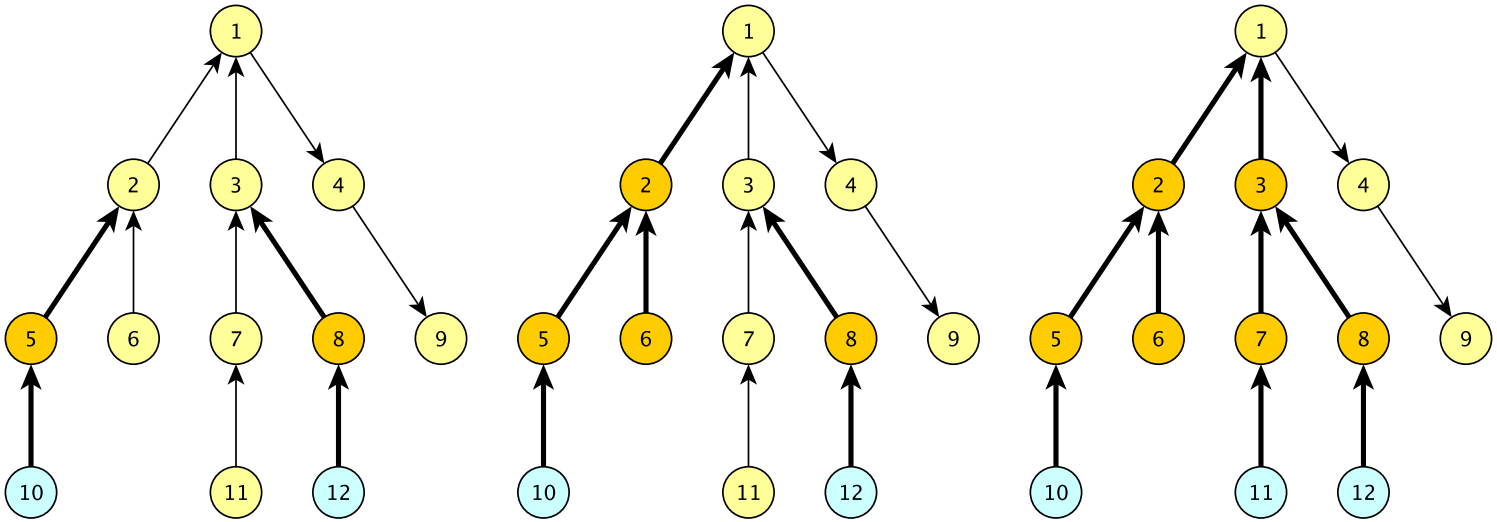
- Возможность ранней проверки корректности высокоуровневого поведения
- Модули могут добавляться по одному, независимо друг от друга
- Не требуется разработка множества драйверов
- Можно разрабатывать систему как в глубину, так и в ширину

Недостатки

- Отложенная проверка низкоуровневого поведения
- Требуется разработка «заглушек»
- Крайне сложно корректно сформулировать требования ко входам/выходам частичной системы

Восходящее интеграционное тестирование

- Тестирование начинается с нижних уровней системы
- Отсутствующие на данный момент модули заменяются драйверами
- При реализации всех модулей нижнего уровня драйвер может быть заменен на соответствующий модуль



Восходящее интеграционное тестирование

Преимущества

- Возможность ранней проверки корректности низкоуровневого поведения
- Не требуется написание заглушек
- Просто определить требования ко входам/выходам модулей

Недостатки

- Отложенная проверка высокоуровневого поведения
- Требуется разработка драйверов
- При замене драйвера на модуль высокого уровня может произойти «мини-Большой Взрыв»

Восходящее интеграционное тестирование

Преимущества

- Возможность ранней проверки корректности низкоуровневого поведения
- Не требуется написание заглушек
- Просто определить требования ко входам/выходам модулей

Недостатки

- Отложенная проверка высокоуровневого поведения
- Требуется разработка драйверов
- При замене драйвера на модуль высокого уровня может произойти «мини-Большой Взрыв»

Восходящее интеграционное тестирование

Преимущества

- Возможность ранней проверки корректности низкоуровневого поведения
- Не требуется написание заглушек
- Просто определить требования ко входам/выходам модулей

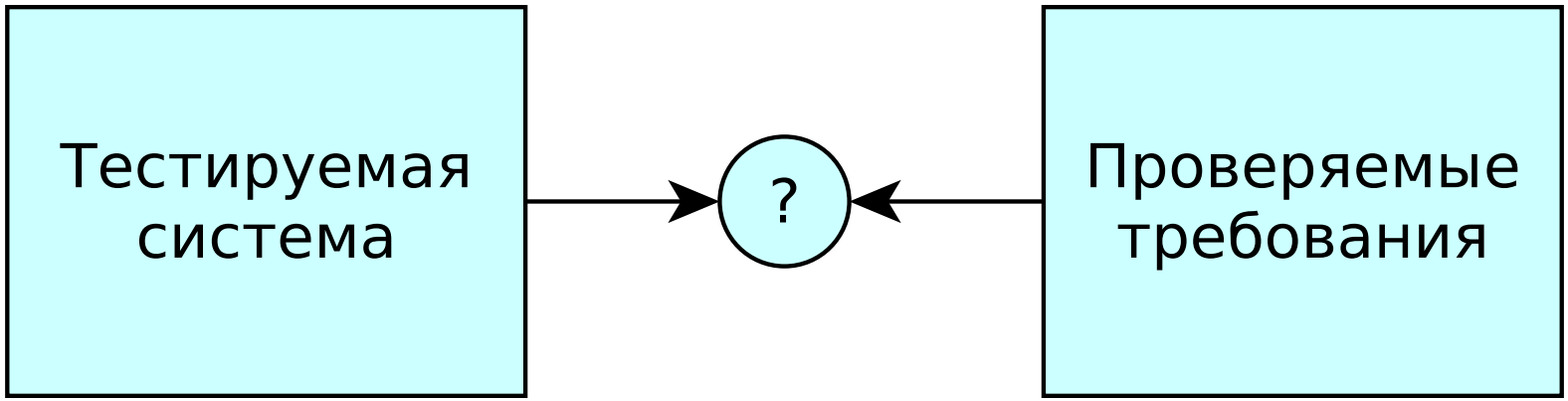
Недостатки

- Отложенная проверка высокоуровневого поведения
- Требуется разработка драйверов
- При замене драйвера на модуль высокого уровня может произойти «мини-Большой Взрыв»

Системное тестирование

- Модульное тестирование «очень большого и сложного ящика»
- Обычно системное тестирование выполняется с определенной целью
  - Тестирование производительности
  - Стресс тестирование
  - Тестирование безопасности
  - Тестирование совместимости
  - Тестирование переносимости
  - Тестирование восстанавливаемости
  - Тестирование устанавливаемости
  - Тестирование обслуживаемости
  - ...

Системное тестирование



Как проверить соответствие поведения системы неформальным требованиям?

Содержание

Homework

- Написать краткое эссе на тему: какой метод системного тестирования является самым важным с точки зрения обеспечения качества и почему?
- Оригинальность (в разумных пределах) приветствуется