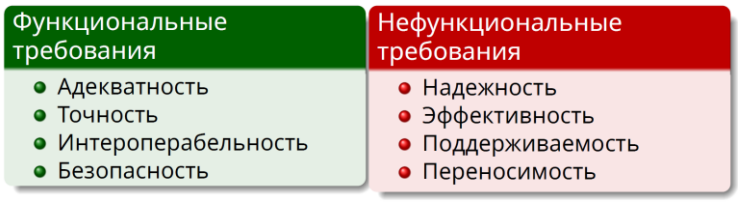
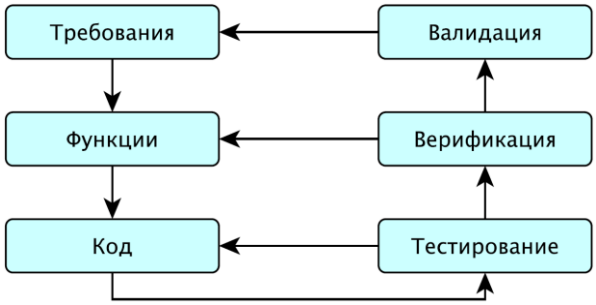
# 1. Тестирование ПО: основы

- Тестирование – это один из способов обеспечения качества ПО, в котором пытаются обеспечить функциональные и нефункциональные требования.




- Есть ли какой-то универсальный способ что-то проверить?
  - Запустить программу, если она собралась. Запуском можно проверить большинство требований.
  - Смотреть на исходный код программы (можно и через различные анализаторы).
- Посмотреть на код и понять, что он делает без запуска самой программы, конечно здорово, потому что мы не запуская код универсально проверяем, что он чего-то там никогда не сделает или обеспечит, однако это очень сложно и требует определённого набора «высшего знания» о том, в каком окружении будет запускаться программа, что умеет библиотека, ЯП и т. д. Это та самая причина, почему не в каждом IDE автоматически на запуске нашего кода запускается какой-то конвейер статического анализа, который обеспечивает, что никаких ошибок в нашем коде нет.
- Основной плюс тестирования в том, что программу можно всегда запустить и посмотреть, что происходит.
- Таким образом, **тестирование** – это один из методов обеспечения качества ПО, который запускает программу и проверяет, отвечает ли эта программа каким-то требованиям.
- По статистике тестирование занимает от 60 до 80% времени сборки.
- Ключевой вопрос тестирования: «Работает ли это ПО неправильно?»
- Фундаментальная проблема тестирования:
  - Тестированием занимаются сами же разработчики, из-за чего трудно переключиться из созидательного режима в разрушительный, чтобы тесты ломали программу.
- У тестирования есть два старших брата:
  - **Верификация** – «мы сделали это правильно»
  - Валидация – «мы сделали то, что надо»




- В силу того, что тестирование ограничено каким-то набором запусков, то мы никогда не можем гарантировать, что приложение не упадёт и прочее. Гарантию

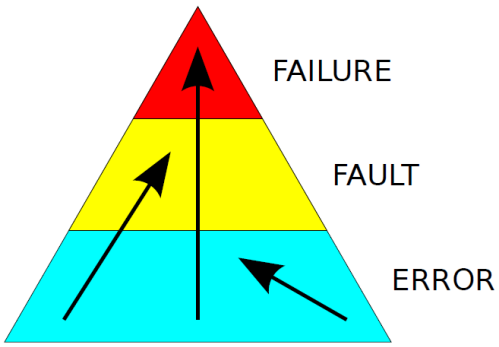
можно дать лишь в самых тривиальных случаях, когда обычно всё ясно и без тестирования.

- Тестированием нужно заниматься как можно раньше, так как цена ошибки с каждой стадией проекта становится всё дороже (выпустив релиз с багом придётся его фиксить так, чтобы ничего не развалилось у конечных пользователей при обновлении)


## 2. Модель программной ошибки. Модель тестирования ПО и место тестирования в процессе разработки ПО.

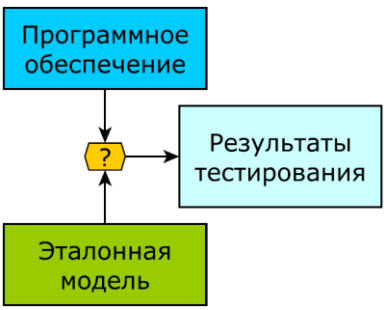
- Тестирование с точки зрения дилетанта:
  - Запустили приложение и проверили результаты выполнения на предмет наличия в них ошибок aka «багов»

### Модель программной ошибки




- **Неудача** – наблюдаемое **снаружи** некорректное поведение программы (упала, зависла)
- **Сбой** – некорректное состояние программы из-за ошибки (заедлочилась программа, что-то неправильно посчиталось)
- **Ошибка** – ошибка в самой программе, внесенная на этапе разработки (что-то написано в исходном коде так, что может привести к тому, что программа будет вести себя не так, как нужно)
- **Важная мысль тестирования**: большая часть того, чем занимается тестирование, это попытка сделать так, чтобы максимальное количество ошибок, разбросанные по программе, были залевелаплены аж до видимой снаружи неудачи.

### Модель тестирования ПО

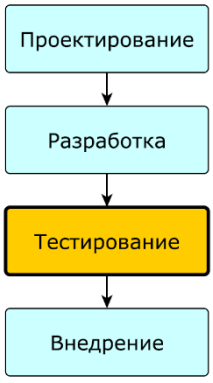


- Эталонная модель может быть представлена множеством различных способов:
  - неформальное представление о том, “как должна работать программа”
  - формальная техническая спецификация
  - набор тестовых примеров
  - корректные результаты работы программы
  - другая (априори корректная) реализация той же исходной спецификации
- Тест может не пройти по следующим причинам:
  - Проблема в программном обеспечении
  - Ошибка в эталонной модели (тест написан давно и не изменялся под новые требования)
  - Баг в тестовом окружении (другие компоненты, от которых зависит компонент тестирования)
  - Баг в программно-аппаратной платформе (что-то неправильно скомпилировалось или баг в ядре ОС) Этот пункт отличается от предыдущего тем, что в тестовом окружении мы можем что-то изменить, контролировать его, а программно-аппаратную платформу мы контролировать не можем.
- Рассмотренная модель не является самодостаточной:
  - Откуда брать эталонную модель?
  - Как сравнивать результаты работы программы и модели? (особенно UI)
  - Когда останавливать процесс тестирования?
  - Как подобрать входные данные, чтобы:
    - Дойти до места с программной ошибкой (**Reachability**)
    - Испортить состояние программы с появлением сбоя (**Corruption**)
    - Вызвать неудачу в работе программы (**Propagation**)

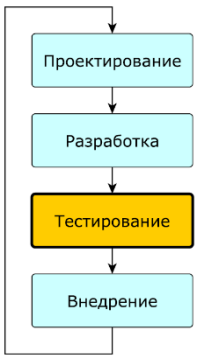
## Тестирование в процессе разработки ПО

### Модели разработки ПО

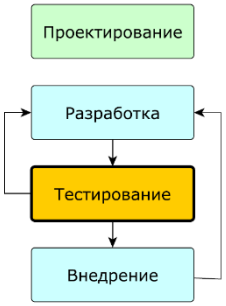
- Чем активнее используется тестирование в процессе разработки, тем важнее его правильное использование
- Водопадная модель:
  - Строго последовательная модель разработки
  - Тестирование выполняется над всей программой сразу
  - Имеется хорошая эталонная модель
  - Стоимость поиска и исправления ошибок очень высока
  - Плюсы:
    - В момент тестирования у нас есть всё, то есть вся программа уже реализована
    - В водопадной модели очень хорошо написана документация и техническое задание -> просто получить эталонную модель. Большой стоимостью исправления ошибок мы заплатили за то, что тестирование будет хорошо проверять всю программу сразу (тестирование проще).



- Инкрементальная модель:
  - Разработка проходит в несколько итераций
  - Тестируются отдельные версии программы
  - Имеется неплохая эталонная модель (она немного похуже, потому что можем что-то забыть в неё включить)
  - Стоимость поиска и исправления ошибок высока (уменьшилась по сравнению с водопадом)




- Гибкая модель:
  - Все этапы разработки неразрывно связаны друг с другом
  - Тестированию подвергаются как сама программа, так и ее компоненты
  - Эталонная модель есть не всегда (почти всегда чего-то не хватает из окружения, которое мы хотим протестировать относительно того, что мы тестируем на текущем спринте)
  - Стоимость поиска и исправления ошибок относительно низка
  - Минусы:
    - Если в процессе тестирования что-то построено не совсем правильно, то это может достаточно сильно повлиять на то, что дальше всё может немного заработать неправильно




### Проблемы тестирования ПО

- Розовые очки
  - Неправильное тестирование создает иллюзию, что всё хорошо, тогда как на самом деле всё может быть очень и очень плохо (тесты запускаются невалидно или не запускаются вовсе)
- Наводнение
  - Неправильное тестирование создает иллюзию, что всё плохо, тогда как на самом деле всё вполне себе ничего (эталонная модель неправильная или тестовое окружение)

## 3. Проблема тестовых входных данных

### Обеспечение достижимости

- Какими способами можно управлять выполнением кода?
  - Изменением входных данных
  - Изменением самого исходного кода

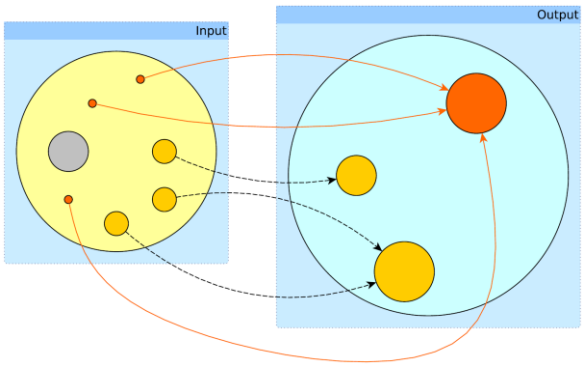
### Входные данные

- Входными данными может быть практически всё, что так или иначе приводит к исполнению того или иного фрагмента кода и может на него влиять:
  - Файлы
  - Фактические аргументы функций
  - Сетевые пакеты
  - Результаты запроса к БД
  - Последовательность вызовов функций
  - Конфигурация ПО

### Тестовые данные

- Почему бы просто не перебрать все возможные варианты?

    - Очень часто возможных вариантов может быть крайне много и даже если на каждый вариант будет тратиться одна микросекунда, то перебирать все варианты по-честному займёт огромное количество времени.
    - Если у тестируемого модуля есть внутреннее состояние, какая-то память и то, что она делает, зависит от последовательности каких-то предыдущих вызовов, то всё получается ещё хуже, потому что на каждом вызове все возможные варианты этого вызова со всеми возможными аргументами умножается на возможное пространство состояний этого модуля/метода.
- Всё пространство входа разбито на какие-то классы эквивалентности, и все тестовые данные из этого класса эквивалентности модулем скорее всего обрабатываются примерно одинаково, что позволяет нам проверить лишь некоторые данные из класса.
    - В этом случае проблема тестирования перерастает и становится с одной стороны проще, а с другой нет, так как нашей задачей теперь является правильное разбиение пространства входных данных на классы эквивалентности.




## Классы эквивалентности

- Всё 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:
    - Ad hoc testing (тестирование методом «научного тыка»)
        - Производим какие-то воздействия на код, или как-то пытаемся использовать библиотеку, не особо задумываясь, и смотрим, правильно работает или нет.
        - Проблемы:
            - Сложно восстановить состояние, при которого возник баг (пользователь прислал ошибку в приложении)
            - Трудный анализ граничных значений
    - Метод свободного поиска (exploratory testing) (доп. метод, а не отдельный)
        - Ad hoc testing + планирование
        - Перед тем, как начать что-то делать с софтом, мы составляем план наших действий и ожидаемые данные на выходе (приблизительные)
        - Например, в геймдеве в некоторых ситуациях трудно написать тест-план, потому что трудно описать, что нужно сделать в геймлпее, если

не делить то, что происходит в игре, на какие-то замкнутые подкомпоненты.
        - Плюсы:
            - Большая управляемость (в тест-плане описаны и входные данные и примерный результат)
        - Минусы:
            - Анализ граничных значений
    - Анализ граничных значений
        - Находим пограничные значения входных данных и проверяем ПО вокруг выбранных пограничных значений
        - Граничные значения могут приходить как из какой-то спецификации на софт, наше понимание того, что софт написан на каком-то ЯП, работает с какими-то типами данных, и при работе с этими типами данных обычно происходят ошибки.
        - Проблемы:
            - Всё равно можно куда-то не добраться в коде (кривой софт, лишняя функциональность)
- Примеры:
    - strtol
        - В документации уже описаны классы эквивалентности
    - sha1sum
        - Всего 2 класса эквивалентности: пустая строка и любые данные.
        - С точки зрения любых данных – это слишком большой класс эквивалентности, и нужно заглядывать в код программы, чтобы разделить на более мелкие.
    - PDF Reader
        - Очень много классов эквивалентности, которые невозможно перебрать. Поэтому мы берём какие-то общие черты классов эквивалентностей и тестируем некоторые из них.

# 4. Проблема неявных входных данных

### Неявные входные данные

- Покрывает ли спецификация все множество входных данных?
    - В некоторых случаях – да
    - В большинстве случаев – нет
- Проблема заключается в том, что на тестовый модуль, кроме явных, влияет множество **неявных** входных данных
- Неявные входные данные часто не находят отражения в спецификации (у нас нет прямого контроля над неявными данными, поэтому описывать их бессмысленно)
- Примеры:
    - Текущая дата/время
    - IP/MAC адрес
    - Идентификаторы устройств
    - Контекстные переключения/планирование нитей
    - Скорость поступления IP пакетов
    - Ритм нажатия клавиш на клавиатуре
- Как учесть неявные входные данные?
    - Изменяем сам исходный код: Simulate and Stub

        - Заменяем части модуля управляемыми заглушками
        - Это позволяет сделать неявные входные данные явными
        - Это также делает управление явными входными данными проще

### Использование заглушек

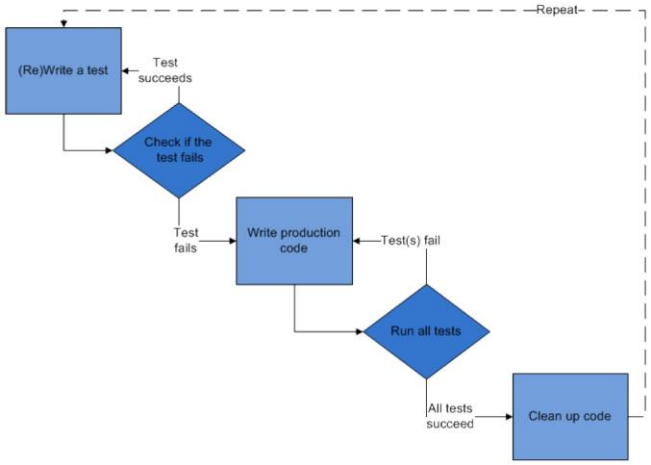
- Данный подход позволяет:
    - имитировать возникновение редких ситуаций
    - внести детерминизм там, где его нет
- Для того, чтобы можно было использовать S&S, тестируемый модуль должен разрабатываться соответствующим образом
- Пример S&S – mock-объекты
    - Повторяют внешний интерфейс тестируемого объекта
    - Могут демонстрировать любое требуемое поведение (неявные входные данные)
- Mock-объекты:
    - Dummies
        - Объект-муляж, не обладающий собственным поведением (чтобы сошлись типы в программе)
    - Fakes
        - Упрощенная реализация требуемой функциональности (фейк базы данных)
    - Stubs
        - Тестовая заглушка, способная отвечать на внешние запросы (то же самое, что и фейк, только имеет какие-то сторонние инструкции помимо тех, что были заложены в фейке, то есть их мы уже сами дописываем)
    - Mocks
        - Тестовая заглушка, способная отвечать на внешние запросы и проверять их корректность
- Что использовать на практике – сильно зависит от того, что мы тестируем.

## 5. Разработка через тестирование

### Test-driven development

- Mock-объекты используются в процессе разработки:
    - Для управления неявными входными данными
    - Для управления явными входными данными
- Заглушки не следует использовать, если:
    - Можно просто подождать, пока не будут разработаны все компоненты. Мы не потратим время на разработку и использование заглушек. Это получится проще, дешевле и лучше, чем с заглушками.
    - Если наш проект относится к категории проектов, где мы можем себе позволить подождать до того момента, когда будут разработаны все компоненты, тогда использовать заглушки не имеет смысла. Тем более заглушки – это тоже код, который нужно написать, в котором также могут быть ошибки.
- Однако далеко не всегда мы можем отложить тестирование до того момента, как все компоненты будут готовы, так как чем дольше итерация, тем сложнее и дороже исправление ошибок.

    - Чем быстрее будут написаны тесты для разрабатываемого компонента, тем проще найти ошибки
    - После разработки компонента пишем для него тест
    - Отсутствующие части системы заменяем заглушками
- **Идея TDD**:
    - Пишем тест для компонента **перед** его разработкой и заменяем сам компонент заглушкой.
    - То есть компонент должен работать так, как он описан в тестах.
    - Обычно в качестве эталонной модели подразумевается какой-то корпус автоматизированных тестов (обычно модульных).
    - Таким образом, мы начинаем с эталонной модели, описываем чего хотим, а затем заглушку заменяем на какую-то реальную реализацию.
    - TDD – это работа в цикле, то есть пишется тест, потом смотрим, что остальные тесты проходят, а новый тест валится. Таким образом, мы постоянно находимся в состоянии, когда все тесты зелёные кроме одного теста, который отвечает за ту функциональность, что мы сейчас будем реализовывать.




## Плюсы TDD

- Разработка ведётся небольшими контролируемыми фрагментами
    - В каждый момент времени разработчик думает об ограниченном фрагменте спецификации
    - Это упрощает анализ возможного пространства входных данных
    - Кроме того, код получается более модульным и расширяемым (так просто получается само по себе)
    - Тестируется даже самый тривиальный код, который мог бы обойтись без тестов
- Минимальная цена ошибки
    - В случае возникновения ошибки очень просто вернуть систему в рабочее состояние (всегда можно откатиться)
    - Практически отсутствует необходимость в отладке (так как каждый тест написан для маленького куска кода, и если что-то не работает, то можно по истории коммитов посмотреть с какими изменениями в коде связан тест)
    - Крайне просто использовать метод бисекции (бинарным поиском можно найти место, где что-то сломалось) в случае, если ошибка смогла пробраться в релиз

- Ошибки обнаруживаются сразу же после их появления
    - Постоянный запуск тестов гарантирует практически моментальное обнаружение ошибки
    - Малый размер тестов позволяет быстро найти причину ошибки
- Сильно упрощается рефакторинг кода
    - Программист уверен в том, что его изменения ничего не ломают
    - Облегчается раздельное владение кодом (из-за этого TDD любят в экстремальном программировании)
- Проще переключиться из разрушительного состояния при написании тестов в созидательное при написании кода

### Минусы TDD

- Синдром «розовых очков»
    - Большое количество тестов создает иллюзию бесконечной надежности системы тестирования
    - При создании теста разработчик может сделать те же допущения, что и при разработке самого компонента (переключение из разрушения в созидание не всегда хорошо)
- **Поддержание тестов в актуальном состоянии (тестов слишком много)**
    - При внесении изменений в интерфейсы компонентов системы необходимо соответствующим образом изменить и все тесты
    - Большое количество тестов приводит к значительным затратам на рефакторинг тестов (даже Google от TDD иногда отказывался из-за этого)
    - Именно из-за этого минуса TDD не используется в проектах или на определённых стадиях проекта.
    - На начальных итерациях происходит некий конфликт между этим минусом и плюсами TDD, так как с одной стороны придётся переписывать много тестов при неправильной реализации, а с другой стороны эти тесты помогают нам понять, что мы неправильно реализовали.
- Невозможность тестирования сложного взаимодействия нескольких компонентов
    - Каждый тест проверяет ограниченный фрагмент функциональности системы
    - Взаимодействие компонентов затрагивает множество аспектов системы сразу
    - Для более обширных видов тестирования (не модульных) TDD подходит не очень хорошо

### Стоит ли использовать TDD?

- Чёткого ответа на этот вопрос нет, потому что всё зависит от того, какой у нас проект, какие у нас есть временные ресурсы, ресурсы разработчиков и сколько мы готовы потратить денег, чтобы что-то протестировать. Также зависит от того, хорошо ли встраивается TDD в наш проект.
- Для 80% проектов TDD подходит хорошо, однако с другой стороны для тех же 80% подходит практически любая методология тестирования.

## 6. Интеграционное тестирование

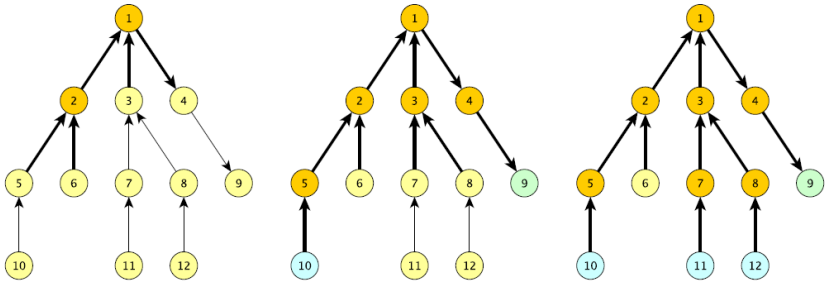
- Интеграционное тестирование – это тип тестирования, при котором программные модули объединяются логически и тестируются как группа.

## Проблема «Большого Взрыва»

- При замене заглушек на реализацию поведение этой реализации может (и скорее всего будет) отличаться от поведения заглушки.
- Если заглушек было много, то у нас всё разваливается, потому что кто-то что-то понял неправильно при реализации, и данный просчёт проявляется при взаимодействии между несколькими системами и вылезает вообще в другом месте программы.
- Каскадное распространение сбоев
- Сложность локализации ошибок (трудно понять где и что пошло не так)
- Большая стоимость исправления ошибок (не все ошибки всплывают при использовании заглушек, а при их реализации всплывает ещё больше)
- Что мы можем сделать, чтобы решить эту проблему?
    - Протестировать заглушку относительно реальной реализации, что нам даст информацию о том, что и где пошло не так (решается проблема сложности локализации ошибок).
    - Заглушки пишет тот разработчик, что и будет реализовывать функционал.
    - Ускорить процесс замены заглушек на реализацию
        - Выполнять тестирование взаимодействия постоянно, в процессе разработки
        - Заменять заглушки на реализацию **инкрементально** (инкрементальное интеграционное тестирование)

## Нисходящее интеграционное тестирование

- Тестирование начинается с верхних уровней системы (ядро системы, основные модули)
- Отсутствующие на данный момент модули заменяются “заглушками”
- По мере реализации новых модулей они подключаются к системе вместо “заглушек”

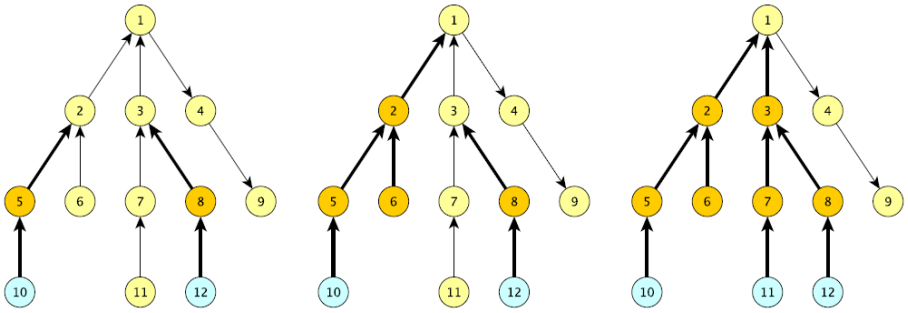



- Плюсы:
    - Возможность ранней проверки корректности высокоуровневого поведения
    - Модули могут добавляться по одному, независимо друг от друга
    - Не требуется разработка множества драйверов (заглушка специального типа, которая находится сверху и умеет что-то делать с модулями, которые находятся ниже драйвера)
    - Можно разрабатывать систему как в глубину, так и в ширину
- Минусы:
    - Отложенная проверка низкоуровневого поведения
    - Требуется разработка “заглушек” (скорее факт, чем минус)

- Крайне сложно корректно сформулировать требования ко входам/выходам частичной системы (придётся писать много частичных спецификаций между работой модулей)

### 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




- Плюсы:
  - Возможность ранней проверки корректности низкоуровневого поведения
  - Не требуется написание заглушек
  - Просто определить требования ко входам/выходам модулей
- Минусы:
  - Отложенная проверка высокоуровневого поведения
  - Требуется разработка драйверов
  - При замене драйвера на модуль высокого уровня может произойти “мини-Большой Взрыв”
- На практике используется комбинация нисходящего и восходящего интеграционного тестирования

## 7. Проблема наблюдаемости. Ассерты.

### Проблема наблюдаемости

- Необходимо обнаружить сбой и распространить его, сделав наблюдаемым снаружи (**Propagation**)
- На практике в подавляющем проценте случаев изменением входных данных мы повлиять не сможем.
- Проблему наблюдаемости можно решить путём изменения самого исходного кода. При чём код нужно изменить так, чтобы программа в какой-то момент поняла, что что-то идёт не так и сказала бы об этом, или хотя бы дала достаточной информации, чтобы снаружи, исследовав эту информацию, мы могли понять что пошло не так.

### Ассерты

- Assertion – это:
  - Формула в логике первого порядка (более сложные логики на ЯП не пишутся)
  - Проверяется на истинность во время выполнения программы
  - Также может проверяться на истинность статически (в некоторых ЯП)

    - Допускает возможность отключения проверки истинности
- Assertions позволяют проверить корректность внутреннего состояния:
    - Внутреннее состояние обычно недоступно снаружи (полностью или частично из-за инкапсуляции)
    - При изменении состояния хочется проверить, что оно остается корректным
- Неудача происходит ближе к причине ее возникновения:
    - Чем больше задержка перед обнаружением неудачи, тем сложнее найти ее исходную причину
    - Assertions позволяют найти неудачу практически в любой точке программы
- Явное документирование пред- и постусловий:
    - В общем случае программист ничего не знает о контракте используемой функции
    - Использование assertions позволяет в явном виде описать внешний контракт функции (программист может разобраться, что происходит в коде). Assertions более надёжный способ, чем комментарии в коде, так как информация в комментариях может не состыковываться с исходным кодом, а если в ассерте что-то написано, то это точно будет выполняться и проверяться.

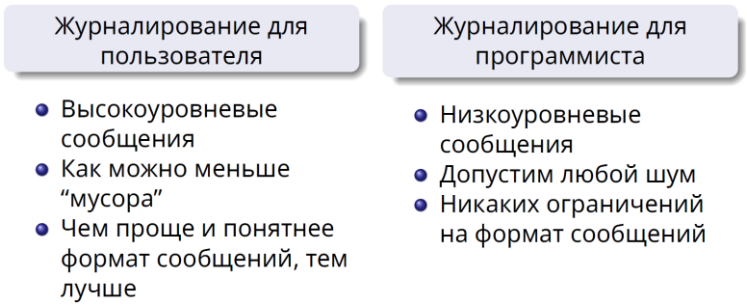
### Проблемы assertions

- Ошибки в assertions
    - Побочные эффекты в assertions. Если ассерт не только что-то проверяет, но и что-то изменяет, то это может привести к тому, что при отключении ассертов в релизной сборке что-то сломается.
    - Неправильное логическое условие срабатывания
- Влияние на производительность
    - Проверка assertions занимает время
    - Чем сложнее assertion, тем больше он замедляет работу программы
- Эффект «вышибалы»
    - Сработавший assertion превращает любую ошибку в неудачу
    - Это полностью останавливает возможность дальнейшего тестирования
    - Если у нас есть код, к которому у нас нет доступа и он с ассертами, то может возникнуть ситуация, когда какая-то структура данных детектирует в себе с помощью ассертов какой-то сбой и крашится. Это мешает пробраться по трассе выполнения программы дальше, так как нужно сначала починить баг, что выдал ассерт.
- Сложность проверки определённых условий
    - Некоторые просто формулируемые условия крайне сложно проверить на практике (ассерты не могут выражать связи)
    - Их реализация в виде assertion является крайне затруднительной

## 8. Проблема наблюдаемости. Журналирование.

### Журналирование

- Запись хода выполнения программы в том или ином виде (записываем что угодно)
- В зависимости от необходимости журнал может быть более или менее детализированным
- По журналу выполнения при необходимости возможно восстановить причину возникшей ошибки




- Как вести журнал?
  - Ручная вставка журналирующих вызовов (вплоть до println)
    - Если мы хотим вести в каком-то смысле системный журнал вручную, то количество ручным способом вызванных логов увеличивается, и более того его ещё нужно поддерживать.
  - Журналирующие аспекты/интерсепторы (что-то записывают во время работы)
  - Logging as a Service (запись логов в какой-то сервис, а не в файл)

## Основная проблема

- **Слишком много данных**
  - Чем больше мы хотим узнать о ходе выполнения программы, тем больше мы должны журналировать
  - Чем больше мы журналируем, тем сложнее разобраться в журнале
  - Чем сложнее разобраться в журнале, тем меньше мы знаем о ходе выполнения программы
- Основная идея – **необходимо ограничивать размер записываемых данных**
- Уровни журналирования
  - Сообщения пишутся в журнал с определенным уровнем (в большинстве библиотек реализовано)
  - В дальнейшем возможно фильтровать сообщения по уровням (Error / Warning / Info / Debug / Trace)
- Домены журналирования
  - Домены ортогональны уровням журналирования (подсистемы, относительно которых ведётся журнал)
  - В зависимости от типа сообщения пишутся в разные домены (Database / Network / UI / Configuration / …)
- Стохастическое журналирование
  - В случае, если какие-то события встречаются очень часто, достаточно записывать лишь их часть (записываем с какой-то вероятностью)
- Сессионное журналирование
  - Часто работа ПО разбита на набор слабо связанных сессий (например, загрузка чего-то в БД)
  - Каждая сессия может журналироваться независимо от других
  - Обычно для сессионного журналирования библиотека пишется либо под конкретный фреймворк, либо под конкретный проект
- Структурированное журналирование
  - Вместо простого текста журнал ведется в определенном формате
  - Структурированный формат облегчает поиск и анализ по журналу

```
{
  "session_id": "e6749451",
  "event": "method_call",
  "class_name": "AuthStorageBean",
  "method_name": "getAuthData"
}
```

# 9. Полнота тестирования ПО. Тестовое покрытие.

## Проблема «останова» в тестировании

- Проблема останова в классическом виде:
  - По заданному алгоритму и входным данным определить, завершится ли за конечное время его выполнение
- Заменим в этой формулировке «алгоритм» на «процесс тестирования» и «входные данные» на «тестируемая программа», тогда получим формулировку проблемы «останова» в тестировании:
  - Можно ли сказать, за какое количество тестовых итераций/шагов/запусков программа протестирована нормально?
- В подавляющем большинстве случаев процесс тестирования является бесконечным
- Мы не можем позволить себе тестировать бесконечное время
  - Слишком долго
  - Слишком дорого
  - Слишком странно
- Проблема «останова» в тестировании заключается в том, чтобы понять каким образом прекращать процесс тестирования. (в каком-то смысле вручную)
  - По заданному набору тестов и программе определить, протестировали ли мы ее **достаточно хорошо**

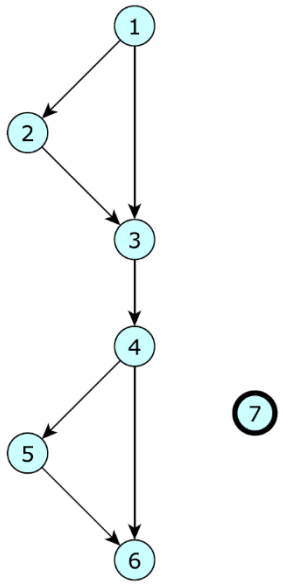
## Тестовое покрытие

- Мы протестировали программу достаточно хорошо, когда мы нашли большую часть ошибок в программе
- Чтобы найти ошибку, необходимо обеспечить выполнение трех основных свойств
- Обеспечение достижимости (**reachability**) и порчи (**corruption**) требует, чтобы мы выполнили определенный участок кода с определенными входными данными
- **Качество тестирования можно оценить через тестовое покрытие**
  - Тестовое покрытие косвенно решает проблему «останова» в тестировании
  - Все метрики тестового покрытия имеют смысл в динамике, то есть нельзя одномоментно посмотреть на тестовое покрытие и однозначно сказать, хорошо покрыто или нет.
  - На тестовое покрытие всегда имеет смысл смотреть как на метрику, которая меняется со временем (что было месяц назад и что сейчас). Тогда можно делать уже какие-то выводы.
  - Из-за того, что мы никогда не можем понять, сколько ошибок в коде было пропущено при оценке тестового покрытия, то только косвенно можем предположить, что протестировали программу хорошо или плохо.
- Выделяют два основных вида покрытия:
  - Покрытие потока управления (куда программа ходит, какие инструкции выполняет)
  - Покрытие потока данных (как программа работает с какими-то данными)
- Они работают с такими понятиями, как граф потока управления (CFG) и граф потока данных (DFG).

### Покрытие потока управления

- Покрытие потока управления пытается покрыть граф потока управления (набор линейных последовательностей инструкций, aka базовые блоки (блоки кода), из которых состоит программа, связанных переходами по каким-то условиям)

- По узлам
- По дугам
- По условиям (которые связаны с дугами)
- По путям

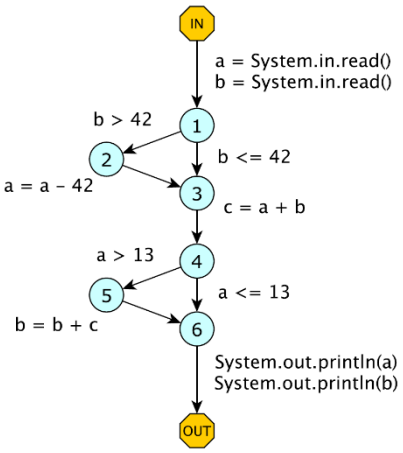



- **Покрытие операторов программы**
  - Каждый узел CFG был пройден в процессе тестирования хотя бы один раз
  - Самый слабый способ оценки тестового покрытия (лучше, чем ничего)
- **Покрытие ветвлений программы**
  - Каждая ветка программы была пройдена хотя бы один раз
  - Несколько более сильный способ оценки покрытия
- Как соотносятся между собой покрытия операторов и ветвлений?
  - Никак
  - Почему покрытие ветвлений не включает в себя покрытие операторов?
    - Потому что в программе может присутствовать “мертвый код”
    - Потому что в программе могут вообще отсутствовать ветвления
- **Покрытие условий программы**
  - Каждое ветвление может выполняться по различным причинам
  - При покрытии условий программы мы требуем, чтобы все условия ветвлений хотя бы один раз приняли значение true и false
  - try catch в таком виде покрытия обычно игнорируется, так как исключения могут вылететь практически из любого места программы
- Как соотносятся между собой покрытия ветвлений и условий?
  - Никак
  - Разные комбинации условий могут приводить к выбору одного и того же ветвления
- **Покрытие ветвлений и условий программы**
  - Комбинация соответствующих покрытий
  - Полностью их покрывает
- **Модифицированное покрытие ветвлений и условий MC/DC**
  - Является одним из обязательных условий при тестировании ПО на уровень A в рамках DO-178B
  - Каждое условие независимо повлияло на выполнение программы
  - Как это проверить?
    - Изменить **одно** условие и проверить, изменилось ли ветвление
- **Полное покрытие условий программы**
  - Полный перебор всех возможных **комбинаций** условий всех возможных ветвлений
  - Занимает слишком много времени (мало используется на практике)
- **Покрытие путей программы**
  - Мы требуем, чтобы все возможные пути программы были выполнены хотя бы один раз
  - Обычно считается самым сильным типом покрытия потока управления
  - Данный тип покрытия можно было бы использовать, если бы не циклы и рекурсия

- Для борьбы с этим используют несколько подходов, один из которых требует, чтобы тело цикла было выполнено:
  - 0
  - 1
  - k
  - max
- Почему выбраны именно эти варианты? (Доп. вопросик к экзамену)

## Покрытие потока данных

- 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
- **Определение переменной (Def):**
  - Оператор программы, в котором значение переменной v может быть изменено
  - Определение переменной – это те места в программе, где у данных в программе могут появиться новые значения
- **Использование переменной (Use):**
  - Оператор программы, в котором значение переменной v влияет на выполнение программы тем или иным способом
  - Использование переменное – это места в программе, где значение может как-то повлиять на то, что в программе происходит (влияет на значение другой переменной, влияет куда программа пошла или передаётся в какую-то функцию, где на основе этой переменной может что-то произойти)
- **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**
- **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
  - Каждое потенциальное значение, которое может оказаться у переменной, должно хотя бы 1 раз где-то повлиять на что-то в программе.
  - Самое слабое покрытие




- **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?
    - Нет
- **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?
        - Соотносится ограниченно, потому что Def-Use Chain по своей структуре из-за отсутствия переопределения в каком-то смысле естественным образом автоматически ломают все циклы
- Покрытие потока данных используется сильно реже, чем покрытие потока управления, потому что Def, Use и Def-Use Chain не отображаются простым способом в коде программы. В основном используется в нейронных сетях.
- Покрытие потока управления хорошо ложится на обычное для всех представление кода – текстовое, поэтому оно используется чаще.

## 10. Мутационное тестирование

- Основная идея мутационного тестирования:
    - Идеальный тест работает только на тестируемой программе
    - При любом изменений он перестает проходить
- 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
- Чтобы сделать мутанта, нужно что-то сделать с исходным кодом:
    - Добавить новый код
    - Удалить старый код
    - Изменить существующий код
    - Набор синтаксических трансформаций, изменяющих исходный код
- После изменения кода получатся следующие мутанты:
    - Некоторые мутанты будут синтаксически некорректны
    - Другие мутанты будут семантически некорректны (можно попытаться перевести в третью категорию, либо тоже выкинуть)
    - Оставшиеся мутанты подходят для использования в мутационном тестировании
- Недостатки:
    - Данный вид тестирования практически невозможно выполнять вручную (необходима очень серьёзная инструментальная поддержка, чтобы генерировать мутантов и проверять их на валидность)
        - Количество необходимых для оценки покрытия мутантов пропорционально объему анализируемого ПО
        - Даже для небольшой программы получение достаточного числа мутантов вручную является практически невозможным
    - Сильно возрастают затраты на проведение тестирования
        - Вместо одного запуска каждого теста требуется выполнить N запусков
        - Кроме того, дополнительное время тратится на генерацию мутантов

- Одним интересным плюсом мутационного тестирования является то, что оно лучше всего оценивает качество тестирования одномоментно

### Оценка тестового покрытия

- Проблема покрытия потока управления:
  - Чувствительность к изменениям в исходном коде

```
1 if (a && b && c) {
2     ...
3 }
```

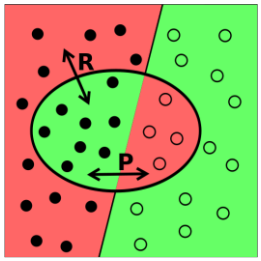
```
1 bool cond = a && b && c;
2 if (cond) {
3     ...
4 }
```

- С точки зрения MC/DC условие справа имеет одно логическое условие, а слева – 3. Такое свойство называется нестабильность тестового покрытия.
  - Чем сильнее влияют изменения в программе на тестовое покрытие, тем более нестабильным (fragile) является тестовое покрытие
  - Чем более нестабильным является тестовое покрытие, тем менее адекватно оно оценивает качество тестирования
- Для того, чтобы побороться с проблемой хрупкости тестового покрытия, можно применить мутационное тестирование для оценки тестового покрытия
  - Ограничить набор трансформаций, которые могут влиять на то, что покрытие поменяется (инвертирование условий в ветвлениях, присвоение подвыражений в переменные)
  - Посмотреть на изменение тестового покрытия для мутантов (если покрытие сильно плавает, то наше покрытие неустойчивое)
- Проблема мутационного тестирования:
  - Эквивалентные мутанты – разные экземпляры программы, но с точки зрения поведения программы одинаковые
- Эквивалентные мутанты “зашумляют” итоговую оценку качества тестирования
- Из-за шума снижается адекватность оценки
- Для того, чтобы побороться с проблемой эквивалентных мутантов, можно применить тестовое покрытие для оценки мутационного тестирования
  - Для каждого мутанта записывается трасса его выполнения
  - У эквивалентных мутантов будут похожие трассы выполнения

## 11. Тестовые оракулы

- Тестовый оракул – это какой-то компонент, который делает итоговое суждение о том, что софт не соответствует эталонной модели. (шестиугольник с вопросиком на картинке ранее)
- Проблема тестового оракула:
  - В универсальном случае его нет. Наличие или отсутствие какого-то тестового оракула, который имеет определённые характеристики, сильно зависит от того, что у него вокруг (как выглядит софт, что за эталонная модель, на какой инфраструктуре всё проводится и прочее)
  - В зависимости от того, какая комбинация факторов у нас есть, будет или не будет тестовый оракул с разным набором характеристик. Это нас сильно ограничивает, потому что мы не можем просто получить определённый тестовый оракул для определённого приложения, эталонной модели и т. д.
  - Вид тестового оракула очень сильно зависит от того, какую эталонную модель мы используем

        - ring buffer
        - strtol
        - sha1sum
        - PDF reader
- От тестового оракула в идеале мы ожидаем 100% точность и 100% полноту.
- **Точность** – способность оракула избегать ложных обнаружений
    - Ложные обнаружения при тестировании — лишние затраты на их обнаружение и игнорирование
    - Если их будет слишком много, оракул никто не будет использовать из-за зашумления результатов
    - Пример неточного оракула – flake-тесты
        - Тесты, которые иногда проходят, а иногда не проходят по каким-то стохастическим причинам (многопоточность, неявные входные данные). На самом деле тест может указывать на баг в программе, однако такой тест может быть проигнорирован.




- **Полнота** – способность оракула находить все ошибки
    - Пропущенные ошибки при тестировании — дополнительные затраты на их исправление позднее
    - Если оракул пропускает много ошибок, его необходимо усиливать другими способами

## Виды оракулов

- Варьируя используемые подходы, можно получить те или иные виды тестовых оракулов
    - Слабые
    - Средние
    - Сильные

### Слабые оракулы

- В каком-то смысле – это оракулы, которым всё равно на эталонную модель. Им достаточно просто программы. Запустив эту программу на каких-то входных данных, мы не доходим до этапа понятия корректности работы, потому что программа сразу же разваливается.
- Падение (работа с бинарями):
    - Segmentation fault
    - Core dump
        - **Работают всегда**
        - Практически ничего не говорят о причине ошибки
- Сбой при работе в обычном окружении:
    - NullPointerException
    - OutOfMemoryException
    - ClassNotFoundException
        - Работают при поддержке **стандартной среды выполнения**
        - Содержат определенную информацию о месте ошибки (поэтому часто пишут софт на ЯП с этой поддержкой)
- Сбой при работе в специальном тестовом окружении (придумали для бинарей)

    - Valgrind – фреймворк для построения средств динамического анализа программ
        - Включает встроенные реализации для
            - Memcheck
            - Cachegrind
            - Callgrind
            - Helgrind
            - DRD
            - Massif
            - DHAT
        - Предоставляют специальную среду выполнения
        - Позволяют весьма точно определить причину ошибок
- Хорошая точность, потому что падающий софт – это баг
- Плохая полнота, потому что если то, что у нас пошло не так, никак не приводит к видимой неудаче снаружи в виде падения, то мы никогда об этом не узнаем, так как слабых оракулов интересуют только фатальные краши.

### Средние оракулы

- Чаще всего используются
- Assertions
- Тесты
    - Требуют определенных усилий со стороны разработчиков
    - В зависимости от степени усилий, будут более или менее точно указывать на место возникновения ошибки и класс ошибки
- Хорошая точность
- Средняя полнота, потому что это не даёт нам универсальной схемы, по которой мы можем найти любое неправильно поведение программы. Написать все возможные тесты и ассерты практически невозможно, поэтому полнота средняя.

### Сильные оракулы

- Эталонная реализация
    - Предыдущая версия программы (смотреть на дельту поведения)
    - Формально верифицированная реализация (при разработке библиотек криптографии или высокоскоростных вычислений (верифицируем медленную версию))
    - Автоматически сгенерированная версия
- “Обратная функция”
    - Прямое/обратное преобразование Фурье
    - Архиватор/деархиватор
    - Кодер/декодер видео
- ※Хорошая точность※ (нужно учитывать различие между версиями программы, если не учитывать, то проседаем по точности)
- Хорошая полнота

## Генерация оракулов

- Можно генерировать:
    - Слабые оракулы
    - Средние оракулы

- Сильные оракулы нельзя генерировать автоматически, потому что если мы смогли сгенерировать сильного оракула в виде эталонной реализации, то нечего тестировать.

### Генерация слабых оракулов

- Всё уже есть
  - Слабый оракул предоставляется средой выполнения
  - Если что-то упало, мы всегда об этом узнаем
- В явном виде используются весьма редко
  - Случайное тестирование (уже не особо и редко)
  - …
- Сложно понять, где произошла ошибка
- Не всегда очевидно, в чем именно заключается ошибка

### Генерация средних оракулов

- Всё уже есть
  - Средние оракулы разрабатываются параллельно с разработкой ПО (параллельно пишем ассерты и тесты)
  - При возникновении проблемы мы сразу узнаем о ней
- Зачем их генерировать автоматически?
  - Мы могли что-то забыть, поленились написать какие-то ассерты, что-то не дописать в тестах
- **Генерация assertions (1 вариант)**
  - Собираем информацию о выполнении программы (какие значения принимают разные переменные в разных частях программы)
  - Выводим определенные закономерности в работе программы (какие-то две переменные никогда не равны 0 и т. д.)
  - Генерируем assertion, проверяющий не нарушение закономерностей
- Хорошо работает для FSM-подобных программ (конечный автомат)
- Плохо работает для всех остальных (уже не факт, нейронные сети могут всё)
- Минусы:
  - Обучились на неправильном состоянии, тем самым сгенерировав неправильный ассерт
  - Не сгенерировали ассерт, который никогда не видели
  - Сгенерировали слишком узкий ассерт, потому что видели мало состояний
- Если сгенерированный ассерт провалился, то мы можем уже вручную посмотреть и понять, почему он не сработал. На основе полученных данных можно улучшить генератор ассертов, чтобы он генерировался правильно, либо не генерировался вовсе в таких местах.
- Если все сгенерированные ассерты неправильные и их надо исправлять вручную, тогда мы не используем такой генератор (он не масштабируется).
- **Генерация assertions (2 вариант)**
  - Собираем информацию о выполнении программы
  - В случае падения определяем его причину (например, NullPointerException)
  - Генерируем защитный assertion
- Способ усиления слабых оракулов до средних
- Не работает, если ничего не падает
- **Генерация тестов при помощи мутационного тестирования**

    - Если тест проходит на мутанте, это плохо
    - Собираем информацию (трассу, статистику) о выполнении программы и мутанта
    - Анализируем разницу, и на основе этой разницы можно понять, как поменять тест, чтобы он начал проваливаться на изменённой программе.
    - Проблема:
        - Нужен уже какой-то набор тестов
- Самый простой способ и самый эффективный – это случайное тестирование

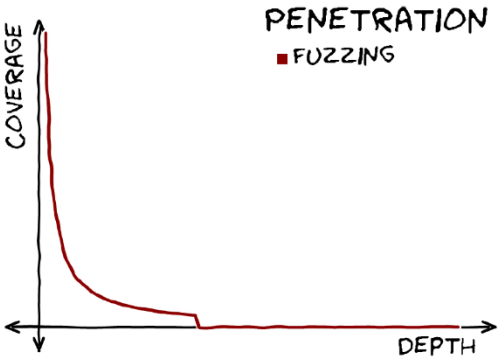
# 12. Случайное тестирование. Фаззинг.

## Генерация тестов

- Развитие идеи генерации тестовых оракулов
- Полная автоматизация процесса тестирования
- Основная идея – заставить компьютер работать вместо нас
    - Дешевле
    - Быстрее
    - Нет человеческого фактора
- Автоматическая генерация компонентов тестов
    - Входные данные (файл, аргументы) для стандартных программ
    - Последовательности вызовов API для какой-то библиотеки
    - Тестовые оракулы
- Результаты очень сильно зависят от того, что именно мы тестируем.

## Fuzzing

- Прародитель случайного тестирования (тестировались утилиты командной строки для семейств ОС UNIX)
- Полностью случайные данные
- Вариант smoke testing (минимальный набор тестов на явные ошибки)
- Используем слабые оракулы (фаззинг интересует упала программа или нет)
- При необходимости вставляем заглушки (чтобы стабилизировать то, что мы не собираемся фаззить (неявные входные данные))
- Что такое случайные данные?
    - Набор байт
    - Вызовы функций API
    - Пользовательский ввод
- Фаззинг дошёл до того состояния, что сейчас его можно спокойно использовать, просто подтянув его к своему проекту, и посмотреть что он за баги сможет найти.

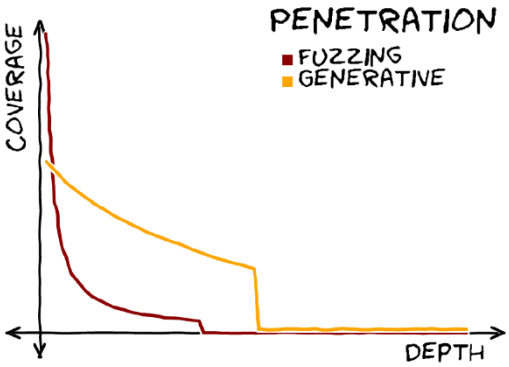



- Проблема валидности данных
    - Фаззинг работает для всех программ, однако полностью случайные данные являются невалидными входными данными для большинства программ.
        - ring buffer
        - strtol
        - sha1sum
        - PDF reader
    - Большинство программ ожидают **структурированные входные данные**.

# 13. Случайное тестирование. Генеративный и мутационный подходы.

## Generative random testing

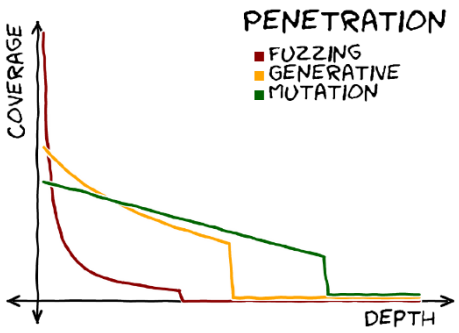
- Если мы знаем структуру, то мы можем ей воспользоваться
  - Генерируем отдельные элементы
  - Комбинируем их в соответствии с заданной структурой
  - Вносим случайные нарушения структуры, чтобы проверить входной парсер на пропуск невалидных данных
- Структура – это:
  - Набор правил генерации
  - Грамматика
  - Формальная спецификация
  - Стандарт на формат входных файлов
- Проблема сложной структуры
  - Работает для структурированных входных данных, но иногда структура входных данных является слишком сложной
    - ring buffer
    - strtol
    - sha1sum
  - Чтобы написать какой-то полноценный генератор входных данных, который покрывает очень глубоко всё то, с чем работает программа, нам придётся переимплементировать большую часть того, что программа и так должна делать. То есть для теста программы мы в каком-то смысле пишем её копию.




## Mutation random testing

- Обычно у нас есть какой-то набор тестовых входных данных (pdf-файлы из процесса разработки для PDF Reader)
  - Подвергаем тестовые данные мутации
  - При этом возможно использование знания структуры данных (используем Generative random testing)
  - Часть данных может генерироваться случайно
- Мутационные трансформации
  - Добавление нового фрагмента
  - Удаление старого фрагмента
  - Изменение фрагмента
  - Обмен двух фрагментов местами
  - Замена значений на граничные
- Проблема скелета в шкафу
  - Работает практически для всего, но добраться до самых дальних закоулков нельзя.
    - PDF reader

        - Web browser
    - Все изменения, что делает мутационное случайное тестирование, не знают о том, влияют ли они действительно на то, что внутри программы происходит. То есть мы никаким образом не знаем, есть ли у нас какая-то связь между тем, как мы поменяли данные, и добрались ли мы куда-то глубже в программе или нет.




## 14. Случайное тестирование. Направленное случайное тестирование и друзья.

### Directed (направленное) random testing

- В чём заключаются основные проблемы случайного тестирования (что мы не можем пробраться глубже по программе)?
    - Некорректные тесты (упал на входе парсера)
    - Эквивалентные тесты
    - Длинные тесты (очень долго работает)
- Что можно сделать, чтобы побороть эти проблемы?
- Некорректные тесты
    - Более строгие правила генерации/мутации
    - Явный учет некорректных тестов (если тест достиг малого покрытия программы, то ему задаётся малый приоритет, и в окрестности этих входных данных больше не ходим)
- Эквивалентные тесты
    - Обнаружение тестов, на которых программа ведет себя одинаковым образом (тестовое покрытие)
        - Статически
        - Динамически
- Длинные тесты (по сути это слабый оракул того, что что-то не так работает, потому что обычно всё работает быстрее)
    - Минимизация тестов
        - Дихотомия
        - Стохастический поиск
        - Эволюционные алгоритмы

### Concolic testing

- Чтобы сразу решить все три вышеописанные проблемы нужно, чтобы:
    - Каждый новый тест минимально отличается от имеющихся (в каком-то смысле решает проблему длинных тестов)
    - Каждый новый тест улучшает тестовое покрытие
    - Каждый новый тест должен генерироваться достаточно быстро
- Для решения этих проблем используем комбинацию статического и динамического анализов
    - Комбинируем информацию о конкретных выполнениях программы и информацию о символическом поведении программы
- Как работает concolic (*concrete* and *symbolic)* testing?

- Программа выполнилась на каких-то входных данных. Concolic запоминает, что в программе произошло, где мы в программе были, и понимает, каким образом нужно изменить входные данные, чтобы добраться в программе в новое место.
- Программа складывается в символическое представление (SMT), которое позволяет описывать логические взаимосвязи между специальными символьными переменными.
- Решатель SMT позволяет проверять и узнавать автоматически для каких переменных что-то будет выполняться в программе.
- Если решатель ошибся, то он уточняет, насколько правильно он понимает программу.

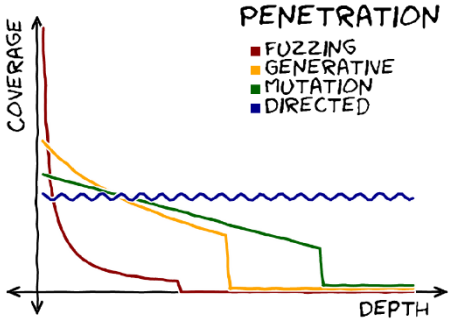
```
1  void f(int x) {
2      int y = x^2;
3
4      bool A = x > 0;
5      bool B = x < 10;
6      bool C = y > 20;
7
8      if (B && C) {
9          if (A) {
10             ...
11         }
12     }
13 }
```

SMT to the rescue

```
1 x, y = Ints('x y')
2 A, B, C = Bools('A B C')
3
4 s = Solver()
5
6 s.add(y == x * x)
7 s.add(A == (x > 0))
8 s.add(B == (x < 10))
9 s.add(C == (y > 20))
```

```
1  print s.check(B and C, A)
2  print s.model()
3  # sat
4  # [A = True, B = True, y = 25, x = 5, C = True]
5
6  print s.check(B and C, Not(A))
7  print s.model()
8  # sat
9  # [A = False, B = True, y = 25, x = -5, C = True]
10
11 print s.check(Not(B and C))
12 print s.model()
13 # sat
14 # [A = False, B = True, y = 0, x = 0, C = False]
```

- Проблемы:
  - Очень сложно
  - Инструментирование программы (необходимо детально знать, что и как происходит в программе, чтобы проверить, совпало ли то, как видит логическая модель программы, с тем, что произошло на самом деле)
  - Моделирование памяти (в зависимости от того, как хорошо мы моделируем память, нам будет проще или сложнее, во-первых, решать формулы, во-вторых, полученные результаты будут адекватнее или неадекватнее соответствовать тому, что происходит в реальной программе)
  - Взрыв пространства состояний (чем больше программа, тем больше количество состояний -> медленнее работа)
- Все эти проблемы можно побороть путём аккуратных оптимизаций, однако они очень сильно зависят от того, что мы пытаемся тестировать при помощи concolic (бинарный код, байт код и т. д.)
- Проблема мистера Икс
  - Directed random testing работает для всего, но всё равно некоторые части системы могут быть «чёрным ящиком» (какие-то внешние компоненты)
    - Visual Studio
    - Microsoft Office



# 15. Регрессионное тестирование. Выборочное регрессионное тестирование.

## Тестирование ПО в процессе разработки

- Как изменяется ПО в процессе разработки?
  - Инкрементально, небольшими независимыми шагами
    - Изменение существующего кода
    - Исправление ошибок
    - Добавление новой функциональности
    - Адаптация имеющихся компонентов к новым задачам
  - Любые (**даже самые незначительные**) изменения могут серьезно повлиять на качество ПО
- После любого изменения требуется проверить, что в программе не появилось новых ошибок
- Для этого мы выполняем все имеющиеся тесты и проверяем, что все они успешно завершаются

## Регрессионное тестирование

- Основной вид тестирования в процессе разработки ПО — это **регрессионное тестирование** (80-90% тестов только этим и занимаются)
  - Тестирование приложения после того, как сделаны небольшие изменения, чтобы убедиться, что эти изменения не внесли новых неожиданных ошибок.
- Одна итерация регрессионного тестирования:
  - Мы модифицируем программу P и получаем программу P’
  - Из всего множества тестов T мы выбираем набор тестов T’, который необходимо выполнить на P’
  - Для новой функциональности мы разрабатываем новые тесты T’’
  - Полученный набор тестов T’ + T’’ запускается на P’
  - Результаты выполнения анализируются с последующей возможной модификацией как программы, так и набора тестов

### Проблема 1. Как выбрать набор тестов T’ после изменения в программе?

- Консервативный подход
  - Выбираем все имеющиеся тесты
  - Полное регрессионное тестирование (сильно медленнее, но надёжнее)
- Случайный подход
  - Выбираем случайное подмножество всех тестов
  - Случайное выборочное регрессионное тестирование (быстрее и ненадёжнее)

## Выборочное регрессионное тестирование

- Каким свойствам должно удовлетворять выборочное регрессионное тестирование?
  - **Эффективность** — способность выполняться быстрее, чем полное регрессионное тестирование (иначе какой смысл от ВРТ)
  - **Полнота** — способность выбирать те тесты, которые могут обнаружить ошибки, связанные с изменениями в коде
  - **Точность** — способность пропускать такие тесты, которые не изменяют своего поведения на модифицированной программе
  - **Универсальность** — применимость в большинстве практических ситуаций

## Подходы к ВРТ

- Умный подход
  - Выбирать тесты, которые “затрагивают” при выполнении измененные части программы
  - Выборочное регрессионное тестирование
- Все подходы к ВРТ различаются по двум основным критериям
  - Способ идентификации измененных программных компонентов
  - Метод получения информации о покрытии элементов программы тестами
- Для реализации ВРТ нам нужно уметь отвечать на два вопроса:
  - Как именно понять, что что-то поменялось в программе?
    - Можно использовать систему контроля версий, но тогда нам нужно на одном уровне и смотреть изменения в программе, и собирать информацию о покрытии (например, в виде строк в коде)
  - Насколько подробно мы должны получать информацию о том, где наши тесты ходили?
    - Собирать максимально подробную трассу очень дорого, поэтому мы можем не удовлетворить свойству эффективности
- **Подход МакКарти**
  - Анализ изменений на уровне целого модуля
  - Связь элементов программы с тестами задается вручную разработчиком (по аналогии с системой сборки make)
  - Преимущества:
    - Это аккуратное закодирование проблемы ВРТ в задачу, которая решает система сборки
  - Недостатки:
    - Человечески фактор (забыли обновить/убрать взаимосвязь при изменении программы)
    - Хромает точность. Если модуль представляет целый класс (во многих ЯП единица компиляции, которая порождает модуль – это класс), и у этого класса поменялся один метод, то мы считаем, что всё в этом классе поменялось. Из-за этого можно много лишнего взять в следующую итерацию ВРТ.
    - Полнота зависит от того, насколько хорошо или плохо мы руками задали связи
- **Подход Ротермела и Харролд**
  - Анализ изменений на уровне узлов CFG (граф потока управления) программы
  - Связь элементов программы с тестами задается на уровне CFG на основе динамической информации о выполнении каждого теста
  - Преимущества:
    - Практически полностью убран человеческий фактор
    - Улучшена точность, зависит от того, как глубоко мы уходим в вызовы методов
  - Недостатки:
    - Чем глубже мы уходим (больше трасса, больше зафиксировано изменений), тем больше теряем эффективность

- **Подход Балла**[1]
  - С точки зрения практики ничего нового по сравнению с подходом Ротермела и Харролд не было предложено.
  - Он переформулировал задачу, что позволило проанализировать с теоретической точки зрения то, как изменения того, на каком уровне мы анализируем разницу, влияют на итоговые показатели на эффективности и точности BPT.
  - Кроме этого структурировал причины потери полноты в некоторых случаях
  - Недостатки:
    - Очень часто мы не можем построить граф потока управления программы исходя только из статической информации, потому что могут быть полиморфные вызовы, вызовы через указатели и прочее. В таком случае мы не можем построить гарантированно полную разницу, из-за чего придётся идти на жертвы полноты или точности.
- **Подход на уровне AST (абстрактное синтаксическое дерево)**
  - Анализ изменений на уровне вершин AST программы (представление стало структурным, а не текстовым)
  - Связь элементов программы с тестами задается на уровне AST на основе динамической информации о выполнении каждого теста
  - Преимущества:
    - Лучшая эффективность (основная причина, почему предпочитают этот подход)
  - Недостатки:
    - Небольшая потеря точности из-за того, что граф потока управления некоторые моменты может более детально позволить проанализировать как по разнице, так и по натягиванию трассы на граф

# 16. Регрессионное тестирование. Другие проблемы.

## Проблема 2. Как управлять набором регрессионных тестов?

- Когда надо добавлять новый регрессионный тест?
  - Когда в ПО появилась новая функциональность
  - Когда в ПО была исправлена ошибка
  - Когда мы хотим улучшить тестовое покрытие ПО
  - Когда мы можем себе позволить добавить новый неповторяющийся тест
- С течением времени число тестов увеличивается
- Чем больше тестов, тем лучше тестовое покрытие
- Проблемы начинаются, когда тестов становится слишком много (долго выполняются)
- Когда можно удалять старый тест?
  - Никогда
  - Когда тест дублирует другие тесты
  - Когда тест не улучшает тестовое покрытие
  - Когда тест ни разу не обнаружил ошибки за все время тестирования
  - Когда тест обнаруживает такие же ошибки, как и другие тесты

[1] https://dl.acm.org/doi/10.1145/271775.271802

- Почему мы можем позволить себе удалить тесты?
    - Их всегда можно восстановить через систему контроля версий

## Проблема 3. Как запускать регрессионные тесты? (приоритезация)

- Мы можем изменить **порядок**, в котором мы запускаем регрессионные тесты
    - Чем раньше мы узнаем о том, что в ПО появилась регрессионная ошибка, тем скорее мы сможем приступить к ее исправлению
    - Часто причиной непрохождения различных (напрямую не связанных друг с другом) тестов является одна и та же ошибка
    - Иногда время на тестирование является ограниченным, и необходимо найти наибольшее число ошибок с учетом всех ограничений
- Как мы можем приоритизировать регрессионные тесты?
    - При помощи интуиции
        - Подход работает, если у Вас хорошая интуиция
        - Кроме интуиции можно использовать имеющийся опыт разработки
    - На основе знаний о тестовом покрытии ПО
        - Сперва выполняются тесты, которые имеют наибольшее покрытие
        - Сперва выполняются тесты, которые покрывают более важные компоненты ПО
    - На основе истории разработки
        - Приоритет отдается тестам, которые чаще других обнаруживали регрессионные ошибки
        - Первыми выполняются тесты, проверяющие корректность работы наиболее “проблемных” компонентов ПО
    - Случайным образом
        - Подход перекликается со случайным ВРТ
        - Если мы можем случайным образом поменять порядок выполнения тестов, то почему бы это не сделать? (чтобы убедиться, что порядок не влияет на то, что тесты проходят или нет)
    - На основе характеристик тестов
        - Первыми выполняются тесты с наименьшим временем выполнения
        - Приоритет отдается тестам, которые наиболее активно работают с окружением программной системы

## Проблема 4. Что делать с результатами регрессионного тестирования?

- Если все тесты проходят — все хорошо
- Если тест не проходит — то все зависит от того, по какой причине он не проходит (модель программной ошибки)

## Регрессионное тестирование на практике

- РТ используется очень часто
    - TDD
    - Agile Development
    - RUP
- ВРТ практически не используется:
- Крайняя сложность выбора регрессионных тестов
- Опасность пропустить регрессионную ошибку при использовании небезопасного ВРТ

- Страх перед использованием “непонятной” технологии
- Простота экстенсивного пути решения проблем РТ (проще запустить на 5000 машин тесты параллельно, чем придумывать ВРТ)
- Отсутствие хорошей инструментальной поддержки (она вроде и есть, а вроде и нет)
    - Test impact analysis (Visual Studio)
    - junit4git
    - Chianti
    - Jimpa
- Недетерминизм в тестах (неспособность программного обеспечения производить одинаковый результат при тех же входных данных)
    - Если выполнение тестов может отличаться от запуска к запуску — всё плохо
- Эффект “лавины”
    - Пропущенный тест в ВРТ может дальше привести к каскадному пропуску множества тестов (не перезаписали более актуальную трассу)
        - Чтобы решить эту проблему можно раз в какое-то время перезапускать все тесты, чтобы собирать новые трассы

## 17. Дебаггинг. Дельта дебаггинг.

- Дебаггинг – процесс поиска ошибок в программе.
- Когда не нужен дебаггинг:
    - Когда решена проблема наблюдаемости (в идеальном случае)
        - Assertions
        - Журнал выполнения
- Основная проблема дебаггинга:
    - **Наблюдение за состоянием программы**
- Основные задачи отладчика
    - Пошаговое выполнение программы
    - Наблюдение за внутренним состоянием программы, чтобы посмотреть что и где пошло не так
- Дебаггер — это программные компоненты, которые имеют «2 конца». Одним смотрят на потребителя и выполняют его требования, например, поставить брейк-поинт на такой-то строке, таком-то символе. Другой же конец смотрит в низкоуровневые вещи в микропроцессоры и с помощью прерываний и различных инструкций пытается реализовать запрос пользователя.
- Дебаггеру необходима тесная интеграция со средой выполнения:
    - Понимание того, что такое инструкция
    - Возможность останавливать выполнение в произвольные моменты времени
    - Знание модели памяти
    - Способность модифицировать выполняемый код
- Алгоритм отладки:
    - Выдвигаем гипотезу о предполагаемом месте ошибки
    - Настраиваем отладчик на анализ выбранного фрагмента кода
    - Ставим метки остановки (брейк-поинты)
    - Запускаем режим отладки, ждём остановки и проверяем, подтвердилась ли наша гипотеза. Если да, то все круто, если нет повторяем алгоритм заново уже с новой гипотезой

- Такой алгоритм отладки имеет следующую проблему:
    - Гипотезы обычно раскручиваются в обратном порядке. Если мы выдвинули неправильную гипотезу, то мы делаем несколько шагов назад по выполнению программы.

### Time-traveling debugging (отладка с возможностью путешествия во времени)

- Отладка с возможностью выполнения программы в обратную сторону
    - Возможность детального исследования поведения программы
    - Нет необходимости перезапускать программу
- Как это работает?
    - Обратная функция (выполнение инструкций с конца), однако многие функции являются необратимыми (целочисленное деление)
    - Трассировка (состояний программы)
    - Запись состояния памяти (memory snapshots). Чаще всего используется на практике, потому что относительно дёшево и решает проблему обратных функций и трассировки (нужно чётко понимать, что нужно трассировать, а что нет)
- Проблемы записи состояния памяти:
    - На сколько часто нам необходимо делать snapshot'ы?
        - Можно делать частые слепки в окрестности брейк-поинта, а за его окрестностями делать раз в несколько шагов. Если понадобится попасть в место, где не был сделан снэпшот, то нужно откатиться до ближайшего и выполнить программу вперёд.
- Проблемы time-traveling debugging:
    - Неявные входные данные
        - Дата время
        - Внешнее состояние (БД, сериализованные данные)
        - Работа с аппаратурой
        - Взаимодействие с другими программами
    - Чем больше неявных данных мы хотим учитывать, тем сложнее это сделать эффективно
        - В худшем случае — полная запись всей трассы выполнения
        - Значение каждой переменной в каждый момент времени

### Time-traveling virtual machine

- Если очень хочется учитывать все неявные данные, то есть time-traveling VM
    - Всё как у time-traveling debugging, но откат на уровне виртуальной машины
    - Запись взаимодействия ПО с аппаратурой
    - Возможность отладки целого ансамбля программ
    - Никаких проблем с неявными входными данными
- Достигается это при помощи:
    - Memory snapshots
    - Запись источников недетерминизма
        - Пользовательский ввод
        - Сетевое взаимодействие
        - Прерывания

## Delta debugging

- Delta Debugging – способ автоматической минимизации теста
  - Задача: из огромного набора данных на входе получить минимальный тест-кейс, который воспроизводит баг.
- Дано: $T = \{C_i\} : P(T) \rightarrow fail$
  - Мы хотим из теста, который пришёл снаружи и состоит из каких-то компонентов, и на котором наша программа валится
- Найти: $DD = min\{C_i\} \in T : P(DD) \rightarrow fail$
  - Хотим найти такой минимальный набор подкомпонентов, чтобы наша программа на этом уменьшенном наборе всё ещё воспроизводит баг
- Что такое $C_i$
  - Строка в файле
  - Вызов метода
  - Тэг в XML
  - Один байт данных
  - Коммит в VCS
  - Такой компонент выбирается всегда индивидуально под каждую программу
- Что такое $min\{C_i\}$
  - Удаление любого компонента из минимального теста приводит к исчезновению ошибки
  - Чем меньше тест тем проще анализировать, что и где пошло не так
- Самая простая идея дельта-дебаггинга: деление на 2 (субоптимальный способ)
  - Делим данные на 2 части. Если в первой части ошибки, то перезапускаем оставляя только первую часть и т. д. пока не будет минимальный результат.
  - В таком подходе есть проблема: удаление части теста может привести к его падению

```
1:  function DD(CC)
2:      L = {CC_1, ..., CC_|CC|/2}
3:      R = {CC_|CC|/2+1, ..., CC_|CC|}
4:      if only P(L) ↦ fail then
5:          return DD(L)
6:      else if only P(R) ↦ fail then
7:          return DD(R)
8:      else
9:          return CC
10:     end if
11: end function
```

- Квази-оптимальный способ:
  - Убираем по 1 компоненту и проверяем сохраняются ли ошибки
  - Такой способ гарантированно даст нам ответ, при чём с хорошей долей вероятности даст минимальный набор входных данных
  - С точки зрения производительно – плохой, потому что полный перебор

```
1:  function DD(CC)
2:      CC' = CC \ {CC_i}
3:      if P(CC') ↦ fail then
4:          return DD(CC')
5:      else if P(CC') ↦ pass then
6:          return CC_i ∪ DD(CC')
7:      end if
8:  end function
```

- Умный перебор
  - Разбиение на N групп
  - Выкидываем группы (а не мин. компоненты)
  - В случае необходимости можем разделить на более мелкие множества для большей детальности

```
1:  function DD(CC, N)
2:      CC = CC^1 ∪ ... ∪ CC^N
3:      for all CC^i do
4:          if P(CC^i) ↦ fail then
5:              return DD(CC^i)
6:          else if P(CC \ CC^i) ↦ fail then
7:              return DD(CC \ CC^i)
8:          end if
9:      end for
10:     return DD(CC, 2 × N)
11: end function
```

- Супер-оптимальный способ
  - Умный перебор с учётом доменной области
  - Учитывает особенности тестов
    - Теряет в универсальности
    - Приобретает в производительности

- Дельта-дебаггинг для языка программирования:
    - При удалении переменной надо удалить её использования
    - Операции нельзя оставлять без аргументов
- На практике: готовых коробочных средств для delta debugging над данными нет, есть ряд библиотек
    - Зависимость $C_i$ от типа данных
    - Разные проекты могут накладывать дополнительные ограничения