Технология программирования

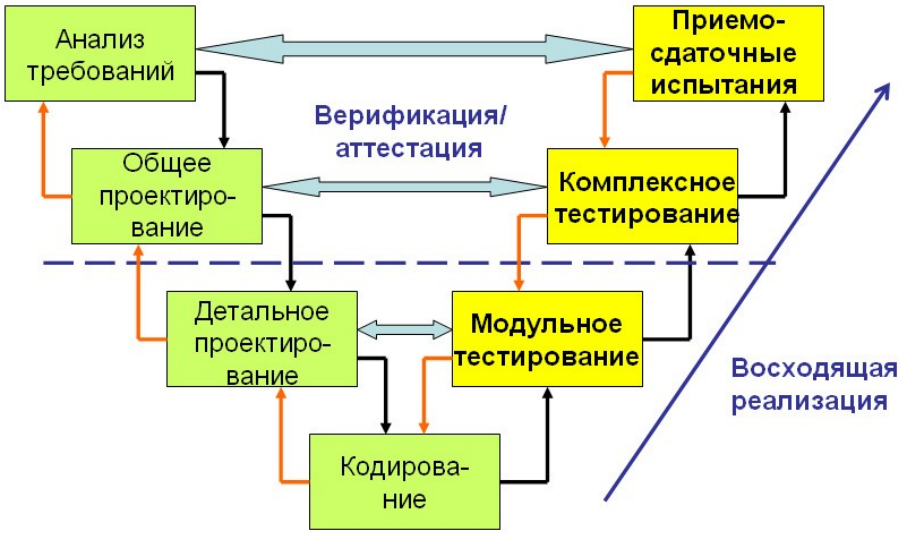
Курс лекций для гр. 3057, 4057

Лекция №8

Содержание

Место тестирования в ЖЦ ПП

Водопадная модель ЖЦ



- Тестирование занимает 30-50% трудоемкости разработки ПП
- В спиральной модели тестирование повторяется на каждом витке ЖЦ
- Чем раньше начинается тестирование, тем лучше
- Agile “Разработка через тестирование” (Test-Driven Development, TDD):
  1. Пишется тест новой функциональности
  2. Пишется код для нее и тестируется
  3. Код включается в сборку и сборка тестируется

Улучшается структура кода: меньшая связанность (coupling) компонентов

Основные понятия тестирования

Тестирование – это выполнение программы с целью *обнаружения* ошибок

- *Отладка* – сопутствующий процесс локализации и исправления ошибок
- Существует понятие «статическое тестирование»: анализ кода без его выполнения
  - Правильнее считать это не тестированием, а просмотром (review) кода
- Тестирование имеет критическую, разрушительную направленность

Тест – это набор контрольных входных данных вместе с ожидаемыми результатами

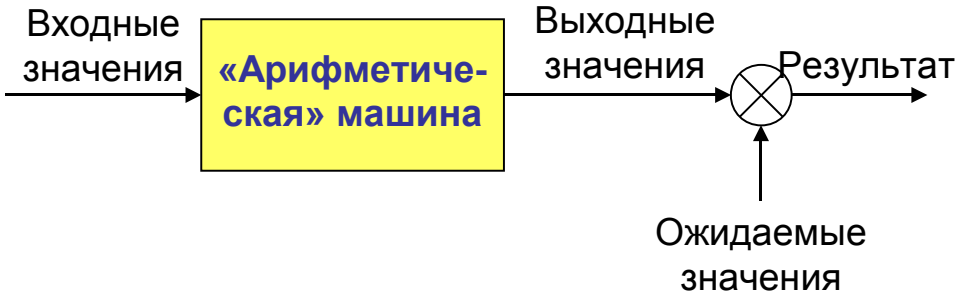
- Контрольные входные данные:
  - значения вводимых данных – для трансформационных программ
  - последовательность событий и их временные параметры – для реагирующих программ
- Ожидаемые результаты берутся из спецификаций ПП, а на этапе приемо-сдаточных испытаний это – ожидания пользователей

"Никакое тестирование не может подтвердить правильность программы: в лучшем случае оно может показать только ее ошибочность"

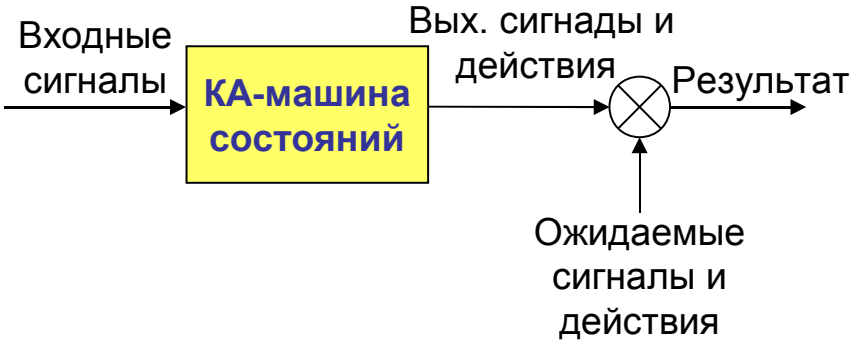
Э.Дейкстра

Тестирование двух видов программ

	Трансформационные ПП	Реагирующие системы
Примеры	Численный эксперимент	Сетевой протокол
Цель	Вычисление выходных значений как функций входных	Поведение, т.е. последовательность реакций на события
Тест	Входные значения + ожидаемые выходные	Последовательность событий + ожидаемые реакции на них



- Тест – набор чисел
- Проверяются вычисления
- Результат теста: сравнение двух значений – тест прошел / не прошел



- Тест – последовательность сигналов о событиях
- Проверяются траектории поведения
- Результат теста: сравнение двух последовательностей сигналов

Виды тестирования

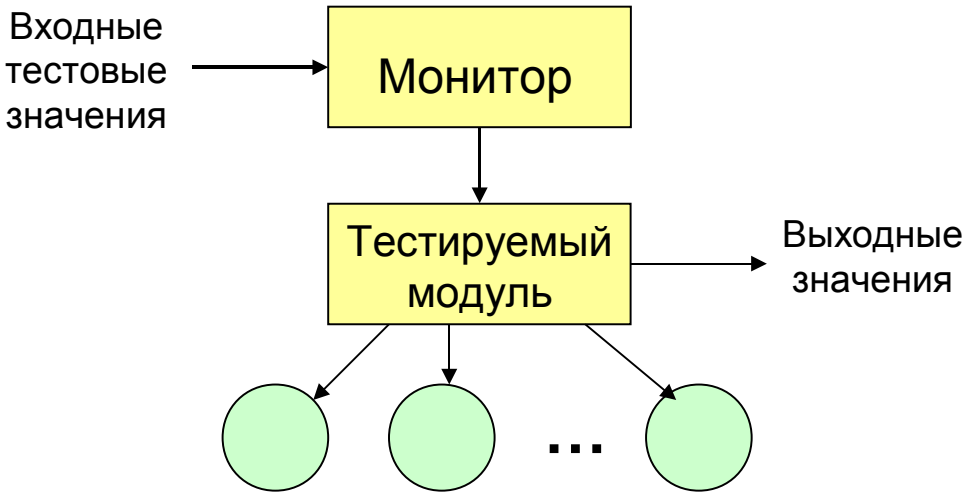
1. Модульное (unit testing) - тестирование отдельных модулей
2. Комплексное
   - 2.1 Интеграционное - тестирование части системы, состоящей из двух и более модулей
     - ✓ Выявляются ошибки, связанные с несогласованностью интерфейсов, неверной трактовкой данных, производительностью
   - 2.2 Системное - тестирование системы в целом на соответствие требованиям (так же и приемо-сдаточные испытания)
1. Нагрузочное (стрессовое) - на предельных объемах данных и интенсивности входного потока
2. Тестирование потребительских свойств: производительности, сиабильности, удобства использования (usability). совместимости с окружением, соответствия стандартам, (де)инсталляции, безопасности, и т.д

Тестирование функциональности

- В гибких технологиях модульное и комплексное тестирование совмещаются во времени; акцент делается на быстрый переход к системному тестированию

Модульное тестирование

- с помощью тестовой среды, состоящей из *драйверов* и *заглушек*, для всех интерфейсов тестируемого модуля
  - драйверы имитируют внешние вызовы функций или методов тестируемого модуля с передачей тестовых входных значений
  - заглушки заменяют вызываемые модули и возвращают некоторые стандартные значения:
    - трассировочное сообщение
    - постоянное значение
    - осуществляют упрощенную реализацию недостающей компоненты



Виды программных ошибок

Виды ошибок	Способы обнаружения
Синтаксические	Статический контроль и диагностика компилятором и компоновщиком (слайд 35: DO 5 K=1.3)
Ошибки выполнения, выявляемые автоматически: 1. Переполнение, защита памяти, … 2. Несоответствие типов 3. Зависание (зацикливание, тупик, …) 4. Нарушение утверждений, исключения	 Аппаратурой процессора Run-time системы программирования Операционной системой Run-time системы программирования
Программа не соответствует требованиям (спецификации)	Целенаправленное тестирование
Программа не соответствует ожиданиям пользователей (ошибка спецификации)	Бета-тестирование, приемо-сдаточные испытания

См. слайд 35: знаменитые примеры ошибок, не выявленных при тестировании,

Ключевой вопрос тестирования

– полнота: *какое* количество *каких* тестов гарантирует возможно более полную проверку программы ?

- Исчерпывающая проверка на всем множестве входных данных недостижима
  - Пример 1: трансформационная программа, вычисляющая Y = f (X, Z)
    - Если X, Y, Z – real, то полное число тестов $(2^{32})^2 = 2^{64} \approx 10^{31}$
    - Если на каждый тест тратить 1 мс, то 10^{31} мс ≈ 10^{20} лет
    - Отсюда видно, что ошибка FDIV Pentium'a вполне простительна
  - Пример 2: реагирующая программа: 6 взаимодействующих автоматов с 10 состояниями каждый:
    - 10^6 потенциально возможных глобальных состояний
    - число возможных траекторий поведения каждого автомата бесконечно

Следовательно:

- В любой нетривиальной программе на любой стадии ее готовности содержатся необнаруженные ошибки
- Продолжительность тестирования – технико-экономическая проблема: компромисс между временем и полнотой тестирования
- Поэтому нужно возможно меньшее количество *хороших* тестов

Критерии выбора тестов

Желательные свойства теста

- *Детективность:* тест должен с большой вероятностью обнаруживать возможные ошибки
- *Покрывающая способность:* один тест должен выявлять как можно больше ошибок
- *Воспроизводимость:* ошибка должна выявляться независимо от изменяющихся условий (например, от временных соотношений) – это трудно достижимо для время-зависимых программ, результаты которых часто невоспроизводимы

Для направленного выбора руководствуются *критериями выбора тестов.* Критерий должен показать, когда некоторое конечное множество тестов достаточно для проверки программы с некоторой полнотой

Два вида критериев:

- *Функциональные* – если тесты составляются исходя из спецификации программы (тестирование черного ящика)
  - Проверяется правильность выполнения программой всех ее заданных функций
  - Именно этим критериям в основном и следуют при независимом тестировании
- *Структурные* – если тесты составляются исходя из исходного текста программы (тестирование прозрачного ящика)
  - Проверяется правильность работы при прохождении всех участков кода
  - Эту работу сами программисты выполняют постоянно в ходе отладки.

Структурные критерии тестирования

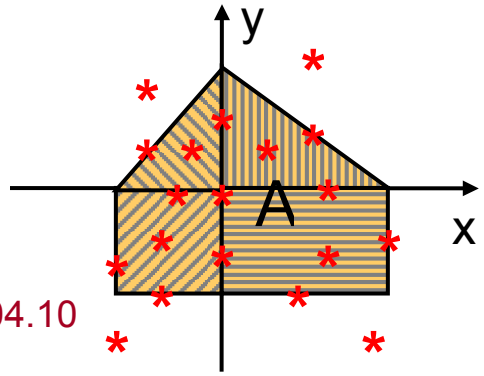
Вид критерия	Что должно обеспечивать множество тестов
1. Тестирование команд	- Каждая команда (оператор) д.б. выполнена >= 1раза
2. C1: Тестирование ветвей	- Каждая ветвь кода (дуга в графе программы) д.б. выполнена >= 1раза
3. C2: Тестирование путей	- Каждый путь в графе программы должен быть проверен >= 1раза

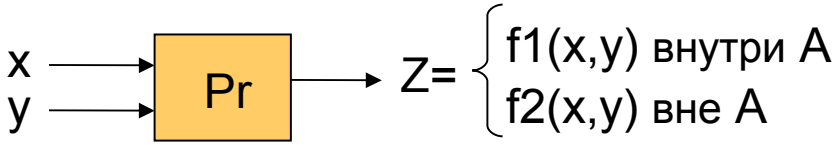
- Реально применимы только при модульном тестировании
- C1 – вполне выполнимый
  - Проблема: приходится вручную подбирать нужные входные данные
  - Профайлер (profiler) помогает проверить выполнение C1
- C2 – не выполнимый для программ реальной сложности
  - Вложенные циклы – экспоненциальный рост числа путей
  - Итерационные циклы – неопределенно большое число повторений
- Реально применим *ограниченный* C2: проверяются три пути для каждого цикла: 0, 1 и N повторений цикла

Функциональные критерии для трансформационных программ

Вид критерия	Что должно обеспечивать множество тестов
1. Тестирование классов вх. данных 2. Тестирование классов вых. данных	Содержать представителей всех вх. или вых. классов и точки на границах
3. Тестирование функций	- Каждая функция внешнего интерфейса должна быть проверена >= 1раза

- Классы назначаются исходя из внешней спецификации программы
- Предположение: программа на всем классе ведет себя так же (не)правильно, как на его одном представителе
- Для числовых данных естественные границы областей классов – оси координат и границы областей задания функций





• Заштрихованы 4 класса вх. данных в области А
• Минимальный набор - 19 тестов

План тестирования

- документ, содержащий определение классов входных и/или выходных данных

Пример: план тестирования классов *входных* данных программы, обрабатывающей вводимые целые числа от 1 до 99 и имена:

1. Ввод числа
   1.1 Допустимые варианты
      1.1.1 Числа от 1 до 99
   1.2 Недопустимые варианты
      1.2.1 0
      1.2.2 Отрицательные числа
      1.2.3 Числа >= 100
      1.2.4 Буквы и другие нечисловые символы
      1.2.4.1 Буквы
      1.2.4.2 Символы с ASCII-кодами, меньшими кода 0
      1.2.4.3 Символы с ASCII-кодами, большими кода 9
2. Ввод букв имени
   2.1 Допустимые варианты
      2.1.1 Первый символ является заглавной буквой
   <и т.д.>

Тестирование классов выходных данных

Пример: минимальный набор тестов для программы нахождения вещественных корней квадратного уравнения: ax2 + bx + c = 0

№	a	b	c	Ожидаемый результат	Что проверяется
1	2	-5	2	$x_1=2$, $x_2=0,5$	Корни - вещественные
2	3	2	5	Сообщение	Корни - комплексные
3	3	-12	0	$x_1=4$, $x_2=0$	Нулевой корень
4	0	0	10	Сообщение	Неразрешимое уравнение
5	0	0	0	Сообщение	Неразрешимое уравнение
6	0	5	17	Сообщение	Не квадратное уравнение
7	9	0	0	$x_1=x_2=0$	Корень из 0

Предлагаемый минимальный набор содержит 7 тестов

Последовательность разработки набора тестов для программного модуля

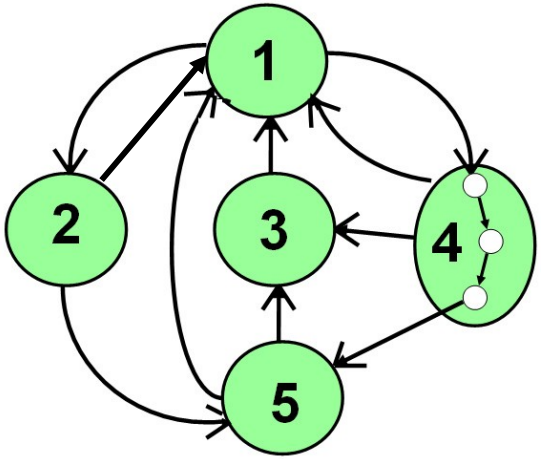
- требования Минобороны США к тестированию управляющих программ для вооружений:

1. По спецификации модуля готовятся тесты:
   - для каждого класса входных данных
   - для граничных и особых значений входных данных
2. Проверяется, все ли классы выходных данных при этом проверяются, и при необходимости добавляются нужные тесты
3. Готовятся тесты для тех функций, которые не проверяются в п. 1 и 2
4. По тексту программы проверяется, все ли условные переходы выполнены в каждом направлении (C1). При необходимости добавляются новые тесты
5. Аналогично проверяется, проходятся ли пути для каждого цикла: без выполнения тела, с однократным и максимальным числом повторений

Проблема ручного подбора значений входных данных для структурного тестирования – п. 4 и 5 !

Тестирование траекторий поведения

Реагирующая система – телефонный процесс



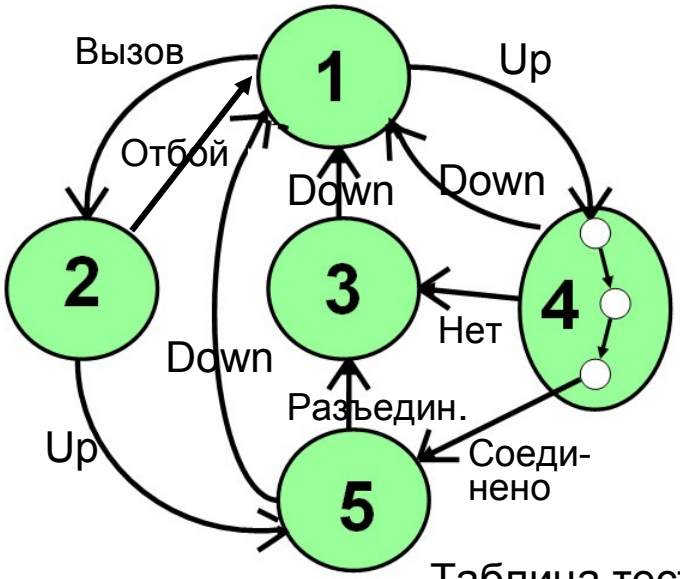
- Требуется пройти возможно большее число траекторий поведения – путей в графе диаграммы состояний с начальным состоянием 1
- Как минимум, нужно пройти все пути с однократным повторением цикла и возвратом в состояние 1
  - Таких путей семь: 1-4-1, 1-4-3-1, 1-4-5-1, 1-4-5-3-1, 1-2-1, 1-2-5-1, 1-2-5-3-1
  - Плюс 6 путей через вложенные состояния состояния 4 $\rightarrow$ всего 13 путей, однократно проходящих через вершины графа

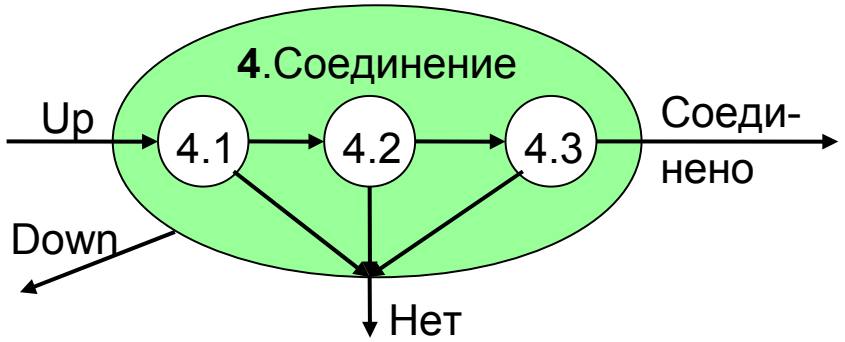
Это не исчерпывающее тестирование: не проверяются траектории с неоднократным повторением цикла

Т.е., минимальный набор тестов, проверяющих все переходы КА с разными предысториями, состоит из 13 тестов – последовательностей входных сигналов автомата

Это функциональное тестирование, т.к. проверяются специфицированные реакции на входы, а с другой стороны - структурное, т.к. прослеживаются пути в графе диаграммы состояний, однозначно соответствующие путям в графе программы

Тестирование траекторий поведения (2)





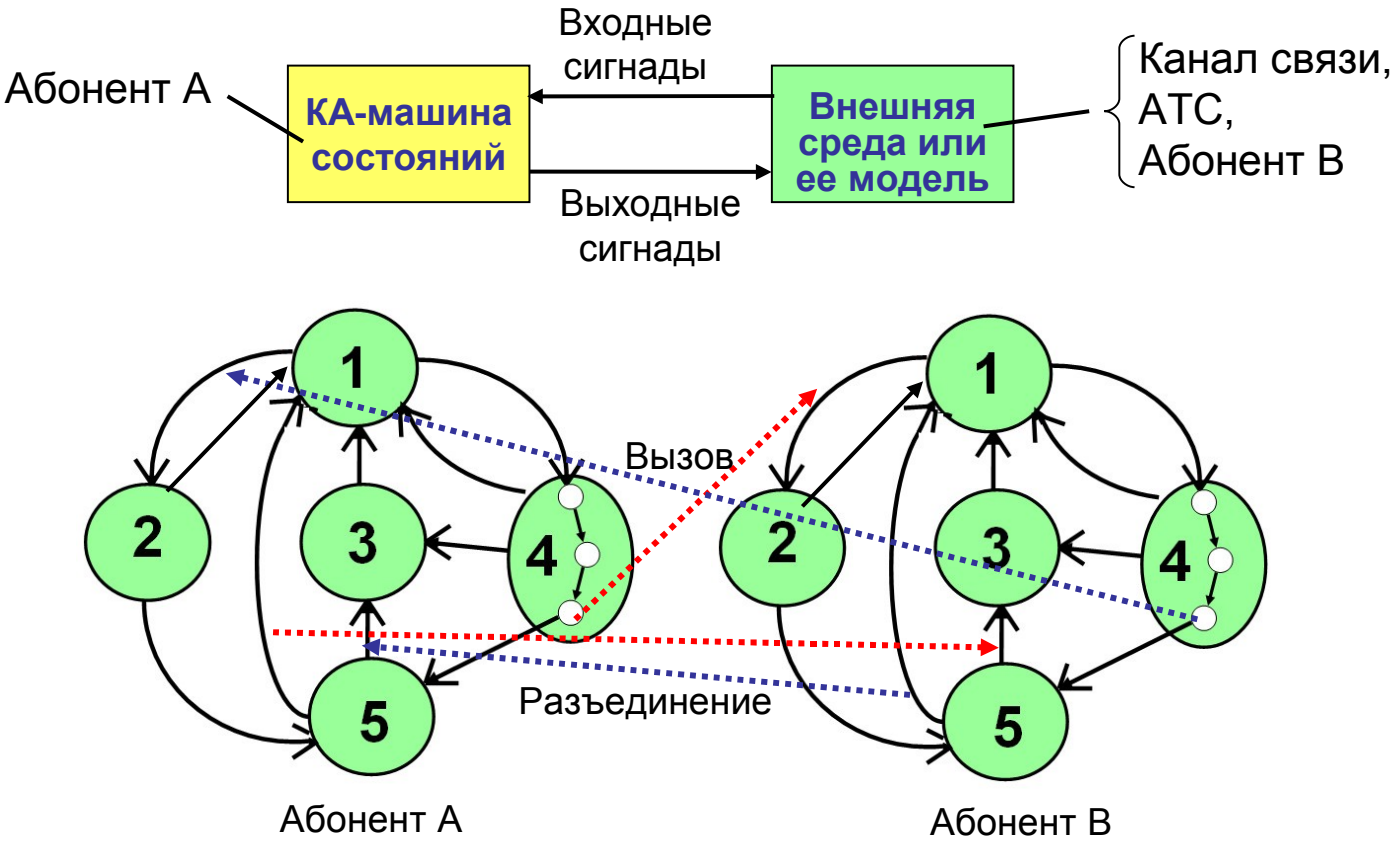
(См. слайд 20 лекции 7)

Таблица тестовых последовательностей

№ тес-та	Вход	Ожидаемая реакция	Новые состояния	Ожидаемая деятельность
1	1.1 Up	-	4.1	-
	1.2 Down	-	1	-
2	2.1 Up	Дл. гудок	4.1 → 4.2	Дл. Гудок
	2.2 Down	-	1	
3	3.1 Up	Кор. гудок	4.1 → 3	Кор. гудок
	3.2 Down	-	1	-

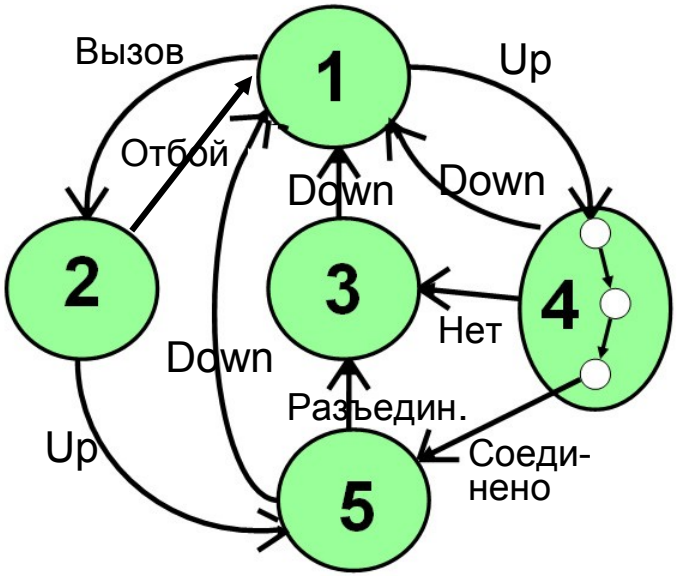
Эти тесты неполно специфицированы: не указано, как управлять переходами, зависящими от другого абонента и от АТС (4→3, 4.1→ 4.2 и пр.)

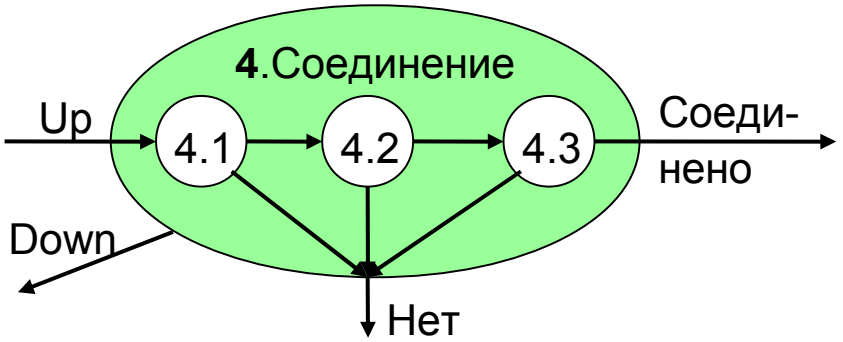
Тестирование траекторий поведения в контексте среды



- В модели внешней среды проще управлять генерацией событий, чем в реальной среде
- Однако реальная среда необходима для финального тестирования и испытаний

Тестирование траекторий поведения (3)





(См. слайд 20 лекции 7)

Таблица тестовых последовательностей

№	Вход	Новое состояние		Ожидаемый сигнал/ деятельность
		Абонент А	Абонент В	
2	2.1 Абонент А: UP	4.1	1	—
	2.2 АТС: подключено к АК	4.2	1	Длинный гудок
	2.3 Абонент А: Down	1	1	—
4	4.1 Абонент В: вызов	2	4.3	Звонок
	4.2 Абонент А: UP	5	5	Разговор
	4.3 Абонент А: Down	1	3	Сигнал разъединения для В

Аксиомы тестирования (Майерс, 1984)

1. Тест должен быть направлен на обнаружение ошибки, а не на подтверждение правильности программы
2. Автор теста – не автор программы
3. Тесты разрабатываются *до* разработки программы или одновременно с ней
4. Необходимо *предсказывать* ожидаемые результаты теста *до* его выполнения и *анализировать* причины расхождения результатов
5. После *каждого* исправления ошибки нужно повторять тест, ее обнаруживший
6. Следует повторять *полное* тестирование после *каждого* внесения исправлений и изменений в программу или после переноса ее в другую среду
7. Для тех программ, в которых обнаружено много ошибок, необходимо дополнить первоначальный набор тестов

Пункт 6 называют *регрессионным* тестированием

Регрессионное тестирование

- повторное выполнение полного набора тестов после внесения исправлений

- Исправление может не устранить ошибку, а может и породить новые ошибки. Статистика:
  - 10% неудачных исправлений – это очень хороший показатель
  - 25% - средний
  - до 80% - в сложных проектах
- Требование п. 6 – слишком сильное; на практике прогон *полного* набора тестов (или его представительного подмножества) производится не после каждого исправления, а после их серии – очередного *цикла* тестирования
  - В больших проектах проходят 10-30 и более таких циклов, синхронизированных с различными стадиями готовности продукта
- Набор регрессионных тестов прилагается к ПП и прогоняется после изменения среды, новой языковой локализации и т.п.
  - Пример: стандартный набор тестов для приемо-сдаточных испытаний трансляторов с языка Ада: 1200 коротких программ с исходными данными и ожидаемыми результатами

Автоматизация тестирования

Автоматизируемые этапы (в порядке возрастания сложности):

1. Прогон (выполнение) набора тестов - необходимо для регрессионного тестирования
2. Анализ результатов тестирования
3. Генерация тестов

1. Прогон последовательности тестов

А) Для трансформационной программы

Циклическая программа на скриптовом языке (Perl, Bash, …) с телом цикла:

1. Загрузка очередного теста из набора (вход X, ожидаемый результат Y)
2. Запуск тестируемой программы
3. Сравнение полученного результата Y* с ожидаемым Y

После выполнения всех тестов набора – вывод результатов сравнения в виде:

<№ теста> <Прошел / Не прошел>

- Несколько сложнее программа прогона модуля, скомпонованного с заглушками и драйверами
  - ✓ Для ее быстрой разработки существуют каркасы, напр. Google Mocking Framework, C++test и пр.

B) Автоматизация тестирования реагирующей программы: прогон тестов

- Программа на скриптовом языке, имитирующая входные сигналы, связанные с событиями и сравнивающая реакцию на них с ожидаемой
- Ручная разработка такой программы сложнее, чем в случае трансформационной программы, поэтому есть средства автоматической регистрации последовательности тестовых событий:

1. Для PC со стандартной диалоговой аппаратурой: клавиатурой и мышью - функциональность Record / Playback в инструментальных ПП: WinRunner, SilkTest, Rational Robot
    - При однократном ручном выполнении тестового набора последовательность нажатий клавиш и кликов мыши записывается в скрипт на внутреннем языке, который потом можно неоднократно выполнять
2. Для встроенных систем управления – подобная функциональность в инструментальных системах разработчика

2. Средства анализа результатов тестирования

Анализаторы динамики (profilers – профайлеры): помогают проверять соответствие тестовых наборов *структурным* критериям тестирования и другим требованиям

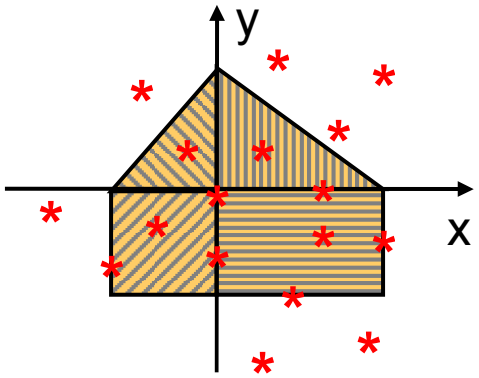
Примеры профайлеров: Intel's VTune Performance Analyser и ряд инструментов IBM Rational:

- *Rational Pure Coverage* - выявляет участки кода, пропущенные при тестировании: инструмент подсвечивает их красным цветом
  - То есть, он помогает убедиться, выполнен ли критерий C1
- *Rational Quantify* - генерирует в табличной форме список всех вызываемых в процессе работы приложения функций, указывая временн**ы**е характеристики каждой из них - показывает «узкие» места в коде, где теряется производительность
- *Rational Purify* – собирает статистику об использовании памяти приложением, написанном на VC++, и позволяет локализовать ошибки: утечки памяти, потерянные блоки, фиктивные ссылки

NB: инструменты Rational интегрируются с системами программирования MS Visual Studio, так что не нужно выходить из среды разработки

3. Генерация тестов

- Генераторы *случайных* функциональных тестов в заданных классах входных данных:



- Ненаправленное тестирование: «ковровые бомбардировки»

- Попытки автоматической генерации тестов на основе исходного текста программы («Метод символического выполнения» Кинга, 1976) закончились неудачно

Организация процесса тестирования

Служба обеспечения качества

- Quality Assurance (QA) - это отдел фирмы, специализирующийся на контроле качества ПП на разных стадиях их разработки

Его задачи:

- Разработка тестов и тестовой среды
- Независимое тестирование
- Отслеживание процесса исправления выявленных ошибок в их базе данных и накопление статистики об этом процессе
- Накопление базы данных тестов и результатов их прогонов
- Выработка мер по улучшению качества

Две специализации: тестировщик (тестер) и инженер по обеспечению качества

Документирование и отслеживание программных ошибок

- С каждой обнаруженной программной ошибкой связывается ее паспорт – отчет об ошибке
- Отчет об ошибке хранится в базе данных (Bugzilla, Jira и пр.); его содержание изменяется в процессе исправления ошибки
- Первоначальный отчет об ошибке оформляет тестер или программист, ее обнаруживший; возможный формат отчета:

№ ошибки, № версии программы, сотрудник, дата

Тип ошибки: кодирование, проектирование, расхождение с документацией, …

Степень важности: фатальная, серьезная, незначительная

Приложения (да/нет): распечатки результатов, копии экрана, тестовые программы и/или данные

Проблема: краткое описание сути проблемы

Воспроизводимость: всегда, не всегда (при каких условиях?)

Предлагаемое исправление (необязательный пункт)

Пункты отчета об ошибке, заполняемые в ходе ее исправления

Функциональная область: категория ошибки с точки зрения разработчиков

Поручено: Программист, ответственный за исправление (устан. руководителем проекта)

Комментарии: поле для записи текстов обсуждения проблемы сотрудниками

Состояние: *Открыто* (начальное состояние), *Закрыто* (устанавливается тестером)

Приоритет: срочность исправления (заполняется руководителем проекта)

Резолюция (подсостояния состояния *Открыто)*

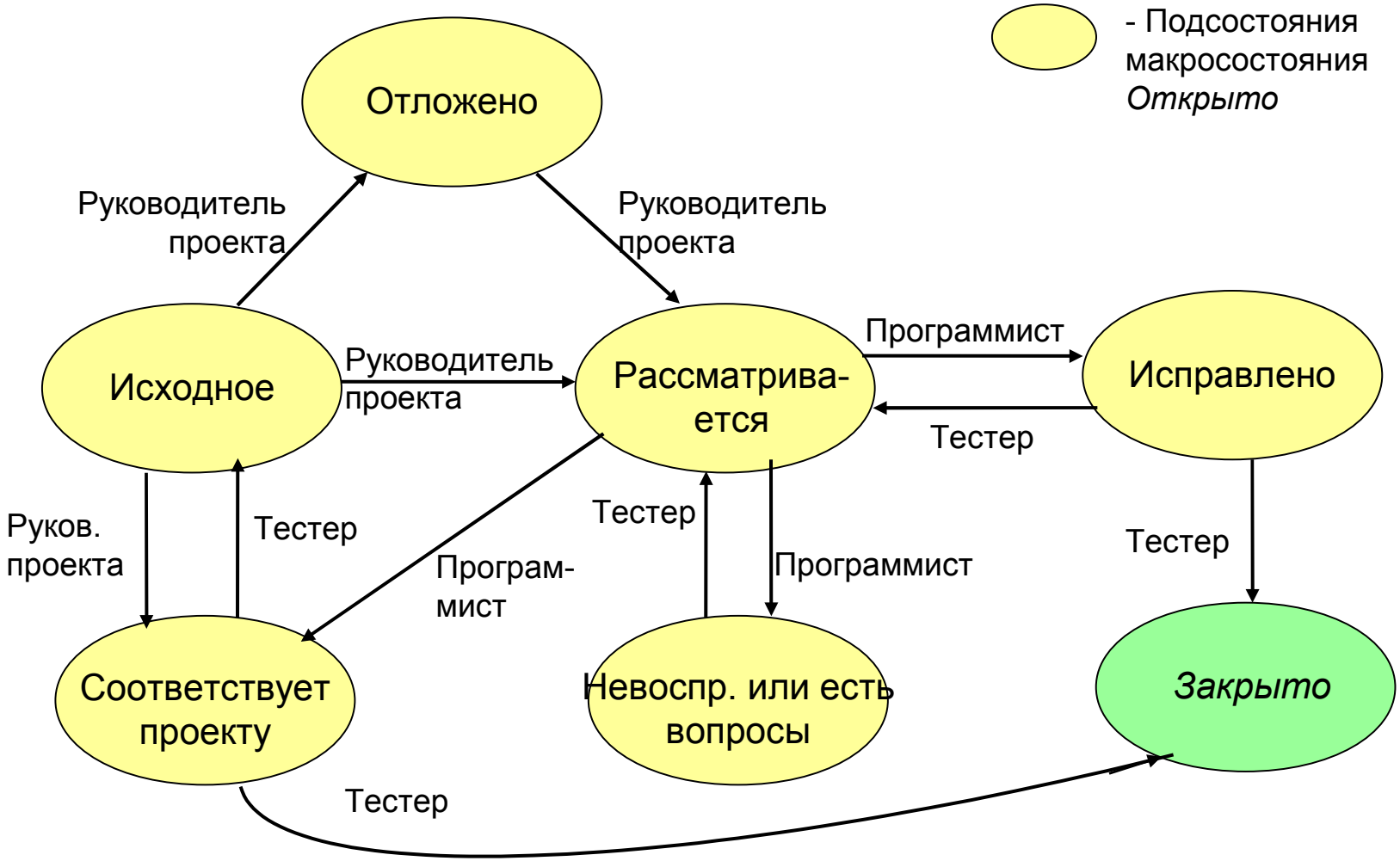
- Рассматривается (устанавливается руководителем проекта)
- Исправлено (уст. программистом)
- Не воспроизводится (уст. программистом)
- Отложено (уст. руководителем проекта)
- Соответствует проекту (уст. программистом)
- Нужна доп. информация (у программиста есть вопросы к тестеру)

Исправленная версия: № и дата версии

Рассмотрено/дата: сотрудник, решивший проблему (он уст. резолюцию *Исправлено*)

Проверено/дата: тестер, установивший состояние *Закрыто*

Диаграмма состояний отчета об ошибке



Состояния пунктов «Состояние» и «Резолюция» отчета об ошибке

Регулярные сводные отчеты

Состояние проекта
Программа Calcdog **Выпуск** 2.10
Отчет сгенерирован 08.03.01. Предыдущий отчет датирован 26.02.01.

К-во неисправленных ошибок	Сейчас	На дату предыдущего отчета
Фатальных	113	100
Серьезных	265	220
Незначительных	333	300
Всего	711	620

С момента составления предыдущего отчета:	
выявлено ошибок	182
исправлено ошибок	85
отложено исправление ошибок	7
Всего отложено исправление ошибок	118

В современных инструментах (напр., Jira) такую статистику можно получить в любой момент, причем в форме диаграмм

Отладка

– это процесс локализации ошибок в коде и их последующего исправления.

- Отладка включает в себя элементы тестирования (целенаправленное экспериментирование с кодом) и разработки (изменение кода)
- В сложных проектах отладка может занимать до 50% всего времени разработки
- Для многих программистов отладка – это самая трудная часть программирования.
- Программа-отладчик — составная часть системы программирования на языке; она дает возможность прослеживать выполнение кода и изменение значений переменных по шагам

Отладка как научный эксперимент

Обобщенный научный подход

1. Сбор данных через повторяемые эксперименты
2. Создание гипотезы, отражающей максимум доступных данных
3. Разработка экспериментов для проверки гипотезы
4. Подтверждение или опровержение гипотезы
5. При необходимости - итерация предыдущих шагов

Отладка

1. Стабилизация ошибки
2. Обнаружение точного места ошибки (роль экспериментов играют вспомогательные тесты)
3. Исправление ошибки
4. Тестирование исправления
5. Поиск похожих ошибок

Нестабильные, т.е. плохо воспроизводимые ошибки обычно связаны с проблемами синхронизации параллельных процессов, инициализации или с висячими указателями

Для стабилизации ошибки требуется свести тест к минимально возможному – такому, что изменение любого аспекта этого теста изменяет и внешнее проявление ошибки.

Заключение

1. Тестирование направлено на поиск программных ошибок, а не на подтверждение правильности программы
2. Тестирование поведения реагирующих (reactive) систем существенно отличается от тестирования традиционных трансформационных программ
3. Тестовый код (тестовая среда) для ПП ответственного применения может превышать объем кода ПП в несколько раз. В него входят:
   - тестовые наборы, драйверы и заглушки
   - имитационные модели внешней среды ПП
   - программы автоматического прогона тестов
   - базы данных отчетов об ошибках
4. Инструменты разработчиков тестов:
   - регистраторы последовательности тестовых событий
   - генераторы случайных тестов
   - средства анализа результатов тестирования
5. Отслеживание процесса тестирования с использованием базы данных ошибок дает много информации о надежности ПП и о качестве технологического процесса разработки
6. Наборы приемо-сдаточных тестов – принадлежность ПП, необходимая для контроля его модификаций при сопровождении

Дополнительные вопросы

1. К слайду 8: Какова причина диагностического сообщения «Runtime Error 53: File Not Found» ОС Windows? Какие еще ошибки периода выполнения (Run-time) в этой ОС вам известны?
2. Задание: составьте план тестирования (в форме текста на слайде 13) и минимальный набор функциональных тестов (в форме таблицы на слайде 14) для программы, считывающей две даты D1 и D2 (от 0 до 2100 г.) и вычисляющей, сколько дней их разделяет.
3. В каких случаях и для каких видов программ результаты, полученные с помощью отладчика, не адекватны реальным результатам программы на одном и том же тесте?

Приложение
Знаменитые ошибки в аппаратуре и ПП

1. Аппаратная ошибка Pentium FDIV

Ранние варианты процессоров Pentium (1993) имели ошибку, которая в редких случаях приводила к уменьшению точности операции деления (максимальная абсолютная ошибка не превышала 0,00005). Этот дефект был обнаружен через год после выпуска процессора. Причина ошибки: несколько элементов в справочной таблице для алгоритма деления были пропущены, и из нее извлекалось неверное число в случае, если делитель содержал шесть последовательных бит, от 5-го до 10-го, установленных в единицу. Такие значения делителя редки (в среднем 6 на $10^{5)}$, и при тестировании процессора они не попали в случайную выборку значений операндов.
Фирма Intel потратила $475 млн на замену дефектных процессоров.

2. Ошибка в Фортран-программе бортового вычислителя ракеты к Венере (1975)

“Точка стоимостью 800 млн $” (вместо запятой) в операторе DO 5 K = 1.3 – он стал оператором присваивания переменной K значения 1.3 вместо оператора цикла DO 5 K = 1,3. (В Фортране объявления переменных не обязательны.) При тестировании тело цикла если и выполнялось, то только 1, а не 3 раза, но это не приводило к неверному результату..

Приложение (2)

3. 4 июня 1996 г. Новая ракета-носитель Ariane 5, результат многолетней работы европейских ученых, взорвалась через 40 секунд после своего первого старта. Только научное оборудование на борту ракеты стоило около $500 млн, не говоря о множестве побочных финансовых последствий. Система автоподрыва ракеты сработала после остановки обоих процессоров в результате цепочки ошибок. Началом этой цепочки послужило переполнение буфера, поскольку система навигации подала недопустимо большое значение параметра горизонтальной скорости. Дело в том, что система управления Ariane 5 переделывалась из Ariane 4, а там такого большого значения не могло быть теоретически. В целях снижения нагрузки на рабочий компьютер инженеры сняли защиту от ошибок переполнения буфера в этом программном модуле, поскольку были уверены, что такого значения горизонтальной скорости не может быть в принципе - и просчитались.