**МИНИСТЕРСТВО ЭНЕРГЕТИКИ, ПРОМЫШЛЕННОСТИ И СВЯЗИ СТАВРОПОЛЬСКОГО КРАЯ**

**Государственное бюджетное образовательное учреждение среднего профессионального образования «Ставропольский колледж связи имени Героя Советского Союза В.А. Петрова»**

Цикловая комиссия вычислительной техники

|  |  |
| --- | --- |
|  | УТВЕРЖДАЮ  Заместитель директора  по учебной работе  \_\_\_\_\_\_\_\_ / Т.В. Захарова /  «\_\_\_» \_\_\_\_\_\_\_\_\_ 2019 г. |

**КОНСПЕКТ ЛЕКЦИЙ**

**по профессиональному модулю ПМ.01 Разработка модулей программного обеспечения компьютерных систем**

**МДК 01.02 «Поддержка и тестирование программных модулей»**

для специальности 09.02.07 «Информационные системы и программирование»

|  |  |
| --- | --- |
| Согласовано  Методист  \_\_\_\_\_\_\_\_\_\_\_\_\_В. В. Петренко  «\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_ 2019 г. | Разработчик: преподаватель Артемов С.В.  Обсуждено на заседании цикловой комиссии «Вычислительная техника»  Протокол №\_\_\_  «\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_2019 г.  Председатель цикловой комиссии  \_\_\_\_\_\_\_\_\_\_\_\_ / И.В. Ерёмина/ |

Ставрополь, 2019

**Содержание**

[Лекция 1. Введение: тестирование - способ обеспечения качества программного продукта 3](#_Toc20170885)

[Лекция 2. Основные понятия тестирования 7](#_Toc20170886)

[Лекция 3. Критерии выбора тестов 21](#_Toc20170887)

[Лекция 4. Оценка оттестированности проекта: метрики и методика интегральной оценки 31](#_Toc20170888)

[Лекция 5. Модульное и интеграционное тестирование 39](#_Toc20170889)

[Лекция 6. Интеграционное тестирование и его особенности для объектно-ориентированного программирования 47](#_Toc20170890)

[Лекция 7. Разновидности тестирования: системное и регрессионное тестирование 54](#_Toc20170891)

[Лекция 8. Автоматизация тестирования 61](#_Toc20170892)

[Лекция 9. Особенности индустриального тестирования 65](#_Toc20170893)

[Лекция 10. Документирование и оценка индустриального тестирования 75](#_Toc20170894)

[Лекция 11. Регрессионное тестирование: цели и задачи, условия применения, классификация тестов и методов отбора 83](#_Toc20170895)

[Лекция 12. Регрессионное тестирование 94](#_Toc20170896)

[Лекция 13. Регрессионное тестирование: методики, не связанные с отбором тестов и методики порождения тестов 98](#_Toc20170897)

[Лекция 14. Регрессионное тестирование: алгоритм и программная система поддержки 108](#_Toc20170898)

[Лекция 15. Система поддержки регрессионного тестирования 110](#_Toc20170899)

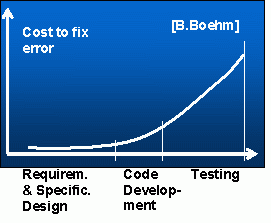
[Список источников 113](#_Toc20170900)

# Лекция 1. Введение: тестирование - способ обеспечения качества программного продукта

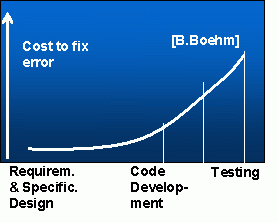
### Тестирование - способ обеспечения качества

***Качество программного продукта*** характеризуется набором свойств, определяющих, насколько продукт "хорош" с точки зрения заинтересованных сторон, таких как заказчик продукта, спонсор, конечный пользователь, разработчики и тестировщики продукта, инженеры поддержки, сотрудники отделов маркетинга, обучения и продаж. Каждый из участников может иметь различное представление о продукте и о том, насколько он хорош или плох, то есть о том, насколько высоко качество продукта. Таким образом, постановка задачи обеспечения *качества продукта* выливается в задачу определения заинтересованных лиц, их критериев качества и затем нахождения оптимального решения, удовлетворяющего этим критериям. *Тестирование* является одним из наиболее устоявшихся способов обеспечения качества разработки программного обеспечения и входит в набор эффективных средств современной системы обеспечения *качества программного продукта*.

С технической точки зрения ***тестирование*** заключается в выполнении приложения на некотором множестве исходных данных и сверке получаемых результатов с заранее известными (эталонными) с целью установить соответствие различных свойств и характеристик приложения заказанным свойствам. Как одна из основных фаз процесса разработки программного продукта (*Дизайн* приложения - Разработка кода - *Тестирование* ), *тестирование* характеризуется достаточно большим вкладом в суммарную трудоемкость разработки продукта. Широко известна оценка распределения трудоемкости между фазами создания программного продукта: 40%-20%-40% ([Рис. 1 1](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1424?page=1#image.1.1)), из чего следует, что наибольший эффект в снижении трудоемкости может быть получен прежде всего на фазах Design и *Testing*. Поэтому основные вложения в автоматизацию или генерацию кода следует осуществлять, прежде всего, на этих фазах. Хотя в современном индустриальном программировании *автоматизация* *тестирования* является широко распространенной практикой, в то же время технология верификации требований и спецификаций пока делает только свои первые шаги. Задачей ближайшего будущего является движение в сторону такого распределения трудоемкости (60%-20%-20% ([Рис. 1 2](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1424?page=1#image.1.2))), чтобы суммарная цена обнаружения большинства дефектов стремилась к минимуму за счет обнаружения преимущественного числа на наиболее ранних фазах разработки программного продукта.



**Рис. 1.1.**Оценка трудоемкости обнаружения и исправления ошибок при создании программного продукта



**Рис. 1.2.**Аналогичная оценка при автоматизации тестирования

Настоящий курс посвящен обсуждению способов решения задачи контроля качества разработки программного обеспечения с позиций *тестирования*. В этой области наряду с решением научных и технических проблем немаловажная роль принадлежит проблеме подготовки кадров, способных решать задачи *тестирования* и автоматизации *тестирования* в условиях производства программного продукта. Задачей курса, реализующейся через лекционный материал и практикум, является подготовка тестировщиков программного проекта. Это тем более важно, что в существующих вузовских программах подготовки профессиональных программистов не предусмотрен достаточный для решения данной задачи объем лекционного материала и практикумов. Поэтому предлагаемое пособие следует рассматривать как дополнительный учебник для будущих тестировщиков программных проектов.

Предлагаемый вниманию читателей курс обобщает *опыт* многолетней работы учебного центра "Политехник - Моторола" в Санкт-Петербургском государственном политехническом университете. Естественно, наш учебник не единственный.

Среди учебников, посвященных подготовке тестировщиков, мы рекомендуем обратить внимание на книги [[ 1 ]](https://www.intuit.ru/studies/professional_retraining/941/courses/48/literature#literature.1),[[ 2 ]](https://www.intuit.ru/studies/professional_retraining/941/courses/48/literature#literature.2),[[ 3 ]](https://www.intuit.ru/studies/professional_retraining/941/courses/48/literature#literature.3),[[ 4 ]](https://www.intuit.ru/studies/professional_retraining/941/courses/48/literature#literature.4),[[ 5 ]](https://www.intuit.ru/studies/professional_retraining/941/courses/48/literature#literature.5),[[ 6 ]](https://www.intuit.ru/studies/professional_retraining/941/courses/48/literature#literature.6), также посвященные передаче опыта промышленного *тестирования* студентам и аспирантам, выбравшим своей специальностью профессиональное *программирование*.

#### Требования к курсу

*Курс* соответствует требованиям специальности 220400 "Программное обеспечение вычислительной техники и автоматизированных систем", ориентированной на подготовку профессиональных программистов, в частности покрывает разделы курса "Технология программирования", посвященные *тестированию*.

Разделы курса соответствуют следующим разделам Computing Curricula 2001: Computer Science [[ 7 ]](https://www.intuit.ru/studies/professional_retraining/941/courses/48/literature#literature.7):

* Раздел SE4 Процессы разработки ПО
* Раздел SE5 Спецификации и требования к ПО
* Раздел SE6 Проверка соответствия ПО

#### Основные темы лекционного курса

* Основные понятия *тестирования*: терминология *тестирования*, различия *тестирования* и отладки, фазы и технология *тестирования*, проблемы *тестирования*
* Критерии выбора тестов: структурные, функциональные, стохастические, мутационный, оценки покрытия проекта
* Разновидности *тестирования*: модульное, интеграционное, системное, регрессионное, автоматизация *тестирования*, издержки *тестирования*
* Особенности процесса и технологии индустриального *тестирования*: планирование *тестирования*, подходы к разработке тестов, особенности ручной разработки и генерации тестов, автоматизация *тестового цикла*, документирование *тестирования*, обзоры и метрики
* Регрессионное *тестирование*: особенности и виды регрессионного *тестирования*, методы *отбора тестов*, оценка эффективности
* Терминологический словарь: содержит глоссарий терминологии *тестирования* в соответствии с IEEE Standard *Glossary* of Software Engineering [[ 8 ]](https://www.intuit.ru/studies/professional_retraining/941/courses/48/literature#literature.8),[[ 9 ]](https://www.intuit.ru/studies/professional_retraining/941/courses/48/literature#literature.9)

В курсе использованы примеры, разработанные на языке С#, для читателей не владеющих С# эти же примеры продублированы на С. С++ в Приложении.

#### Основные темы практикума

Для демонстрации и закрепления *теоретических знаний* разработан практикум, содержащий:

* описание практических работ (для студентов)
* методические указания по проведению практических работ (для преподавателей)
* рекомендации по подготовке компьютерной лаборатории к проведению практических работ

В рамках практикума студенты осваивают различные подходы к разработке тестов и *тестированию* и условия их применения.

Практикум представлен в форме тренинга, в котором рассмотрены следующие темы:

* Разработка документации на тестируемую систему и ее окружение: описание требований (Requirement Specification) и спецификаций разработчика (High Level Design)
* Планирование *тестирования*
* Практикум модульного *тестирования*
* Практикум интеграционного *тестирования*
* Практикум системного *тестирования*
* Ручное *тестирование* и *тестовые процедуры*
* Автоматизированное *тестирование* на основе скриптов
* Автоматизированное *тестирование* на основе MSC-диаграмм и генерация тестов
* Средства поддержки автоматизации *тестирования*

Используя модель реальной системы управления, студенты могут:

* разрабатывать различные виды тестов и тестирующих программ
* искать дефекты системы в процессе *тестирования,* участвовать в их исправлении и модернизации тестируемого приложения
* разрабатывать документацию - требования к системе, тесты и тестовые процедуры - и отслеживать взаимосвязь этих документов с разработанными тестами

#### Прогнозируемые результаты

В результате изучения курса:

1. Вырабатывается понимание условий применения Верификации, Валидации и *Тестирования*
2. Вырабатываются навыки и приемы *тестирования*, применяемые на различных фазах разработки *качественного программного продукта*
3. Оцениваются условия эффективного применения инструментальных средств в разработке качественного программного обеспечения
4. Вырабатываются навыки разработки тестовых программ и тестовых наборов в программном проекте
5. Вырабатываются навыки разработки проектной документации для этапа *тестирования*
6. Вырабатываются навыки планирования и отслеживания задач *тестирования*
7. Обеспечиваются основы обучения проектной команды, состоящей из разработчиков и тестировщиков
8. Вырабатываются навыки *тестирования* программного обеспечения проектов, разработанных на C#

**Выводы:**

Ruby:

* имеет простой синтаксис;
* поддерживает обработку исключений;
* позволяет переопределять операторы;
* является чисто объектно-ориентированным языком (complete, full, pureobjectorientedlanguage), в котором, в отличие от Java или Perl, все — объекты;
* позволяет работать с целыми числами произвольной величины;
* не требует объявления переменных;
* использует префиксы (@, $, @@) для задания области видимости (scope) переменных;
* поддерживает многопоточное программирование.

**Вопросы для самоконтроля:**

1. На каком языке программирования написан Ruby?
2. Под влиянием какого из языков был написан Ruby?
3. В каком году был создан язык программирования Ruby?
4. Кто создателем языка Руби является;
5. Каким образом осуществляется применение метода к объекту в Ruby?

# Лекция 2. Основные понятия тестирования

### Концепция тестирования

*Программа* – это аналог формулы в обычной математике.

Формула для функции f, полученной *суперпозицией* функций f1, f2, ... fn – *выражение*, описывающее эту *суперпозицию*.

f = f1\* f2\* f3\*... \* fn

Если аналог f1,f2,... fn – *операторы* языка программирования, то их формула – *программа*.

Существует два метода обоснования *истинности формул*:

1. **Формальный подход** или **доказательство** применяется, когда из исходных формул-аксиом с помощью формальных процедур (правил вывода) выводятся искомые формулы и утверждения (теоремы). Вывод осуществляется путем перехода от одних формул к другим по строгим правилам, которые позволяют свести процедуру перехода от формулы к формуле к последовательности текстовых подстановок:
2. A\*\*3 = A\*A\*A

A\*A\*A = A -> R, A\*R -> R, A\*R -> R

Преимущество формального подхода заключается в том, что с его помощью удается избегать обращений к бесконечной области значений и на каждом шаге доказательства оперировать только конечным множеством символов.

1. **Интерпретационный подход** применяется, когда осуществляется подстановка констант в формулы, а затем интерпретация формул как осмысленных утверждений в элементах множеств конкретных значений. *Истинность* интерпретируемых формул проверяется на конечных множествах возможных значений. Сложность подхода состоит в том, что на конечных множествах комбинации возможных значений для реализации исчерпывающей проверки могут оказаться достаточно велики.

Интерпретационный подход используется при экспериментальной проверке соответствия программы своей спецификации

Применение интерпретационного подхода в форме экспериментов над исполняемой программой составляет суть *отладки* и *тестирования*.

### Основная терминология

***Отладка (debug, debugging)*** – процесс поиска, локализации и исправления ошибок в программе [[ 9 ]](https://www.intuit.ru/studies/professional_retraining/941/courses/48/literature#literature.9)[IEEE Std.610-12.1990].

Термин " *отладка* " в отечественной литературе используется двояко: для обозначения активности по поиску ошибок (собственно *тестирование*), по нахождению причин их появления и исправлению, или активности по локализации и исправлению ошибок.

***Тестирование*** обеспечивает выявление (констатацию наличия) фактов расхождений с требованиями (ошибок).

Как правило, на ***фазе тестирования*** осуществляется и исправление идентифицированных ошибок, включающее локализацию ошибок, нахождение причин ошибок и соответствующую корректировку программы тестируемого приложения (Application Under Testing (AUT) или Implementation Under Testing (IUT)).

Если *программа* не содержит синтаксических ошибок (прошла трансляцию) и может быть выполнена на компьютере, она обязательно вычисляет какую-либо функцию, осуществляющую *отображение* входных данных в выходные. Это означает, что *компьютер* на своих ресурсах доопределяет частично определенную программой функцию до тотальной определенности. Следовательно, судить о правильности или неправильности результатов выполнения программы можно, только сравнивая спецификацию желаемой функции с результатами ее вычисления, что и осуществляется в процессе *тестирования*.

#### Пример поиска и исправления ошибки

***Отладка*** обеспечивает локализацию ошибок, поиск причин ошибок и соответствующую корректировку программы ([Пример 2.1](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1426?page=1#example.2.1), [Пример 2.2](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1426?page=1#example.2.1)).

// Метод вычисляет неотрицательную

// степень n числа x

static public double Power(double x, int n)

{

double z=1;

for (int i=1;n>=i;i++)

{

z = z\*x;

}

return z;

}

2.1. Исходный текст метода Power

double Power(double x,int n)

{

double z=1;

int i;

for(i=1;n>=i;i++)

{

z=z\*x;

}

return z;

}

2.1.1. Исходный текст метода Power

Если вызвать метод Power с отрицательным значением степени n Power(2,-1), то получим некорректный результат 1. Исправим метод так, чтобы ошибочное значение параметра (недопустимое по спецификации значение) идентифицировалось специальным сообщением, а возвращаемый результат был равен 1 ([Пример 2.2](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1426?page=1#example.2.2)).

// Метод вычисляет неотрицательную

// степень n числа x

static public double PowerNonNeg(double x,

int n)

{

double z=1;

if (n>0)

{

for (int i=1;n>=i;i++)

{

z = z\*x;

}

}

else Console.WriteLine(

"Ошибка ! Степень числа n" +

" должна быть больше 0.");

return z;

}

2.2. Скорректированный исходный текст

double PowerNonNeg(double x, int n)

{

double z=1;

int i;

if (n>0)

{

for (i=1;n>=i;i++)

{

z = z\*x;

}

}

else printf("Ошибка! Степень числа n должна быть больше 0.\n");

return z;

}

2.2.1. Скорректированный исходный текст

Если вызвать скорректированный метод PowerNonNeg(2,-1) с отрицательным значением параметра степени, то сообщение об ошибке будет выдано автоматически.

*Тестирование* разделяют на статическое и динамическое:

Статическое *тестирование* выявляет формальными методами анализа без выполнения тестируемой программы неверные конструкции или неверные отношения объектов программы (ошибки формального задания) с помощью специальных инструментов контроля кода – CodeChecker.

Динамическое *тестирование* (собственно *тестирование*) осуществляет выявление ошибок только на выполняющейся программе с помощью специальных инструментов автоматизации *тестирования* – *Testbed* [[ 9 ]](https://www.intuit.ru/studies/professional_retraining/941/courses/48/literature#literature.9)или Testbench.

### Организация тестирования

*Тестирование* осуществляется на заданном заранее множестве входных данных X и множестве предполагаемых результатов Y – (X,Y), которые задают *график* желаемой функции. Кроме того, зафиксирована процедура ***Оракул (oracle)***, которая определяет, соответствуют ли *выходные данные* – Yв (вычисленные по входным данным – X) желаемым результатам – Y, т.е. принадлежит ли каждая вычисленная точка (X,Yв) графику желаемой функции (X,Y).

*Оракул* дает заключение о факте появления неправильной пары (X,Yв) и ничего не говорит о том, каким образом она была вычислена или каков правильный *алгоритм* – он только сравнивает вычисленные и желаемые результаты. *Оракулом* может быть даже Заказчик или программист, производящий соответствующие вычисления в уме, поскольку *Оракулу* нужен какой-либо альтернативный способ получения функции (X,Y) для вычисления эталонных значений Y.

#### Пример сравнения словесного описания пункта спецификации с результатом выполнения фрагмента кода

Пункт спецификации: "Метод Power должен принимать входные параметры: x – целое число, возводимое в степень, и n – неотрицательный порядок степени. Метод должен возвращать вычисленное значение xn ".

Выполняем метод со следующими параметрами: Power(2,2)

Проверка результата выполнения возможна, когда результат вычисления заранее известен – 4. Если результат выполнения 22 = 4, то он соответствует спецификации.

В процессе *тестирования* *Оракул* последовательно получает элементы множества (X,Y) и соответствующие им результаты вычислений (X,Yв) для идентификации фактов несовпадений (test *incident*).

При выявлении (X,Yв)\notin (X,Y) запускается процедура исправления ошибки, которая заключается во внимательном анализе (просмотре) протокола промежуточных вычислений, приведших к (X,Yв), с помощью следующих *методов*:

1. "Выполнение программы в уме" (deskchecking).
2. Вставка операторов протоколирования (печати) промежуточных результатов (logging).

#### Пример вставки операторов протоколирования промежуточных результатов

Можно выводить промежуточные значения переменных при выполнении программы. Код, осуществляющий вывод, расположен ниже ([Пример 2.3](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1426?page=2#example.2.3)). Этот *метод* относится к наиболее популярным средствам автоматизации *отладки* программистов прошлых десятилетий. В настоящее время он известен как *метод* внедрения "агентов" в текст отлаживаемой программы.

// Метод вычисляет неотрицательную

// степень n числа x

static public double Power(double x, int n)

{

double z=1;

for (int i=1;n>=i;i++)

{

z = z\*x;

Console.WriteLine("i = {0} z = {1}",

i, z);

}

return z;

}

2.3. Исходный текст метода Power со вставкой оператора протоколирования

double Power(double x, int n)

{

double z=1;

int i;

for (i=1;n>=i;i++)

{

z = z\*x;

printf("i = %d z = %f\n",i,z);

}

return z;

}

2.3.1. Исходный текст метода Power со вставкой оператора протоколирования

1. *Пошаговое выполнение* программы (*single-step* running).

#### Пример пошагового выполнения программы

При *пошаговом выполнении* программы код выполняется строчка за строчкой. В среде Microsoft Visual Studio.NET возможны следующие команды *пошагового выполнения*:

* Step Into – если выполняемая строчка кода содержит вызов функции, процедуры или метода, то происходит вызов, и программа останавливается на первой строчке вызываемой функции, процедуры или метода.
* Step Over - если выполняемая строчка кода содержит вызов функции, процедуры или метода, то происходит вызов и выполнение всей функции и программа останавливается на первой строчке после вызываемой функции.
* Step Out – предназначена для выхода из функции в вызывающую функцию. Эта команда продолжит выполнение функции и остановит выполнение на первой строчке после вызываемой функции.

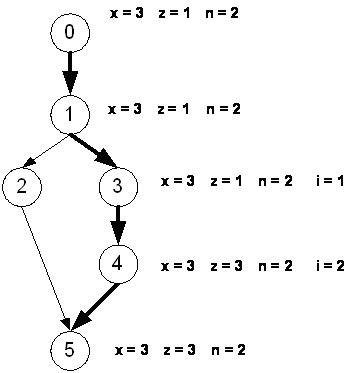
*Пошаговое выполнение* до сих пор является мощным методом автономного *тестирования* и *отладки* небольших программ.

1. Выполнение с заказанными *остановками (breakpoints)*, анализом трасс (traces) или состояний памяти - *дампов (dump)*.

#### Пример выполнения программы с заказанными контрольными точками и анализом трасс и дампов

* ***Контрольная точка (breakpoint)*** – точка программы, которая при ее достижении посылает отладчику сигнал. По этому сигналу либо временно приостанавливается выполнение отлаживаемой программы, либо запускается программа "агент", фиксирующая состояние заранее определенных переменных или областей в данный момент.
* Когда выполнение в контрольной точке приостанавливается, отлаживаемая программа переходит в режим останова (break mode). Вход в режим останова не прерывает и не заканчивает выполнение программы и позволяет анализировать состояние отдельных переменных или структур данных. Возврат из режима break mode в режим выполнения может произойти в любой момент по желанию пользователя.
* Когда в контрольной точке вызывается программа "агент", она тоже приостанавливает выполнение отлаживаемой программы, но только на время, необходимое для фиксации состояния выбранных переменных или структур данных в специальном электронном журнале - Log-файле, после чего происходит автоматический возврат в режим исполнения.
* Трасса - это "сохраненный *путь* " на *управляющем графе* программы, т.е. зафиксированные в журнале записи о состояниях переменных в заданных точках в ходе выполнения программы.

**Например**: на [Рис. 2.1](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1426?page=2#image.2.1) условно изображен *управляющий граф* некоторой программы. Трасса, проходящая через вершины 0-1-3-4-5 зафиксирована в [Табл. 2.1](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1426?page=2#table.2.1). Строки таблицы отображают вершины *управляющего графа* программы, или *breakpoints*, в которых фиксировались текущие значения заказанных пользователем переменных.



**Рис. 2.1.**Управляющий граф программы

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Таблица 2.1. Трасса, проходящая через вершины 0-1-3-4-5 | | | | |
| **№ вершины-оператора** | **Значение переменной x** | **Значение переменной z** | **Значение переменной n** | **Значение переменной i** |
| 0 | 3 | 1 | 2 | не зафиксировано |
| 1 | 3 | 1 | 2 | не зафиксировано |
| 3 | 3 | 1 | 2 | 1 |
| 4 | 3 | 3 | 2 | 2 |
| 5 | 3 | 3 | 2 | не зафиксировано |

* ***Дамп*** – область памяти, состояние которой фиксируется в контрольной точке в виде единого массива или нескольких связанных массивов. При анализе, который осуществляется после выполнения трассы в режиме off-line, состояния *дампа* структурируются, и выделенные области или поля сравниваются с состояниями, предусмотренными спецификацией. Например, при моделировании поведения управляющих программ контроллеров в виде *дампа* фиксируются области общих и специальных регистров, или целые области оперативной памяти, состояния которой определяет алгоритм управления внешней средой.

1. реверсивное (обратное) выполнение (reversible execution)

Обратное выполнение программы возможно при условии сохранения на каждом шаге программы всех значений переменных или состояний программы для соответствующей трассы. Тогда поднимаясь от конечной точки трассы к любой другой, можно по шагам произвести вычисления состояний, двигаясь от следствия к причине, от состояний на выходе преобразователя данных к состояниям на его входе. Естественно, такие возможности мы получаем в режиме off-line анализа при фиксации в Log – файле всей истории выполнения трассы.

#### Пример обратного выполнения для программы вычисления степени числа x

В программе на [Пример 2.4](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1426?page=1#example.2.1) фиксируются значения всех переменных после выполнения каждого оператора.

// Метод вычисляет неотрицательную

// степень n числа x

static public double PowerNonNeg(double x,

int n)

{

double z=1;

Console.WriteLine("x={0} z={1} n={2}",

x,z,n);

if (n>0)

{

Console.WriteLine("x={0} z={1} n={2}",

x,z,n);

for (int i=1;n>=i;i++)

{

z = z\*x;

Console.WriteLine(

"x={0} z={1} n={2}" +

" i={3}",x,z,n,i);

}

}

else Console.WriteLine(

"Ошибка ! Степень" +

" числа n должна быть больше 0.");

return z;

}

2.4. Исходный код с фиксацией результатов выполнения операторов

double PowerNonNeg(double x, int n)

{

double z=1;

int i;

printf("x=%f z=%f n=%d\n",x,z,n);

if (n>0)

{

printf("x=%f z=%f n=%d\n",x,z,n);

for (i=1;n>=i;i++)

{

z = z\*x;

printf("x=%f z=%f n=%d i=%d\n",

x,z,n,i);

}

}

else printf(

"Ошибка ! Степень "

"числа n должна быть больше 0.\n");

return z;

}

2.4.1. Исходный код с фиксацией результатов выполнения операторов

Зная структуру *управляющего графа* программы и имея значения всех переменных после выполнения каждого оператора, можно осуществить обратное выполнение (например, в уме), подставляя значения переменных в операторы и двигаясь снизу вверх, начиная с последнего.

Итак, в процессе *тестирования* сравнение промежуточных результатов с полученными независимо эталонными результатами позволяет найти причины и место ошибки, исправить текст программы, провести повторную трансляцию и настройку на выполнение и продолжить *тестирование*.

*Тестирование* заканчивается, когда выполнилось или "прошло" (pass) успешно достаточное количество тестов в соответствии с выбранным критерием *тестирования*.

***Тестирование*** – это:

* Процесс выполнения ПО системы или компонента в условиях анализа или записи получаемых результатов с целью проверки (оценки) некоторых свойств тестируемого объекта.

The process of operating a system or component under specified conditions, *observing* or recording the results, and making an evaluation of some *aspect* of the system or component [[ 9 ]](https://www.intuit.ru/studies/professional_retraining/941/courses/48/literature#literature.9).

* Процесс анализа пункта требований к ПО с целью фиксации различий между существующим состоянием ПО и требуемым (что свидетельствует о проявлении ошибки) при экспериментальной проверке соответствующего пункта требований.

The process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate features of software items [[IEEE Std.610-12.1990], [[ 9 ]](https://www.intuit.ru/studies/professional_retraining/941/courses/48/literature#literature.9).

* Контролируемое выполнение программы на конечном множестве тестовых данных и анализ результатов этого выполнения для поиска ошибок [IEEE Std 829-1983].

#### Сквозной пример тестирования

Возьмем несколько отличающуюся от [Пример 2.4](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1426?page=3#example.2.4) программу:

// Метод вычисляет степень n числа x

static public double Power(int x, int n)

{

int z=1;

for (int i=1;n>=i;i++)

{

z = z\*x;

}

return z;

}

[STAThread]

static void Main(string[] args)

{

int x;

int n;

try

{

Console.WriteLine("Enter x:");

x=Convert.ToInt32(Console.ReadLine());

if ((x>=0) & (x<=999))

{

Console.WriteLine("Enter n:");

n=Convert.ToInt32(Console.ReadLine());

if ((n>=1) & (n<=100))

{

Console.WriteLine("The power n" + " of x is {0}", Power(x,n));

Console.ReadLine();

}

else

{

Console.WriteLine("Error : n " + "must be in [1..100]");

Console.ReadLine();

}

}

else

{

Console.WriteLine("Error : x " + "must be in [0..999]");

Console.ReadLine();

}

}

catch (Exception e)

{

Console.WriteLine("Error : Please enter " + "a numeric argument.");

Console.ReadLine();

}

}

Пример 2.5. Другой пример вычисления степени числа

#include <stdio.h>

double Power(int x, int n)

{

int z=1;

int i;

for (i=1;n>=i;i++)

{

z = z\*x;

}

return z;

}

void main(void)

{

int x;

int n;

printf("Enter x:");

if(scanf("%d",&x))

{

if ((x>=0) & (x<=999))

{

printf("Enter n:");

if(scanf("%d",&n)) {

if ((n>=1) & (n<=100))

{

printf("The power n of x is %f\n", Power(x,n));

}

else

{

printf("Error : n must be in [1..100]\n");

}

}

else

{

printf("Error : Please enter a numeric argument\n");

}

}

else

{

printf("Error : x must be in [0..999]\n");

}

}

else

{

printf("Error : Please enter a numeric argument\n");

}

}

2.5.1. Другой пример вычисления степени числа

Для приведенной программы, вычисляющей степень числа ([Пример 2.5](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1426?page=3#example.2.5)), воспроизведем последовательность действий, необходимых для *тестирования*.

**Спецификация программы**

На вход программа принимает два параметра: x - число, n – степень. Результат вычисления выводится на консоль.

Значения числа и степени должны быть целыми.

Значения числа, возводимого в степень, должны лежать в диапазоне – [0..999].

Значения степени должны лежать в диапазоне – [1..100].

Если числа, подаваемые на вход, лежат за пределами указанных диапазонов, то должно выдаваться сообщение об ошибке.

**Разработка тестов**

Определим области эквивалентности входных параметров.

Для x – числа, возводимого в степень, определим классы возможных значений:

1. x < 0 (ошибочное)
2. x > 999 (ошибочное)
3. x - не число (ошибочное)
4. 0 <= x <= 999 (корректное)

Для n – степени числа:

1. n < 1 (ошибочное)
2. n > 100 (ошибочное)
3. n - не число (ошибочное)
4. 1 <= n <= 100 (корректное)

**Анализ тестовых случаев**

1. Входные значения: (x = 2, n = 3) (покрывают классы 4, 8).

Ожидаемый результат: The power n of x is 8.

1. Входные значения: {(x = -1, n = 2),(x = 1000, n = 5)} (покрывают классы 1, 2).

Ожидаемый результат: Error : x must be in [0..999].

1. Входные значения: {(x = 100, n = 0),(x = 100, n = 200)} (покрывают классы 5,6).

Ожидаемый результат: Error : n must be in [1..100].

1. Входные значения: (x = *ADS* n = ASD) (покрывают классы эквивалентности 3, 7).

Ожидаемый результат: Error : Please enter a numeric argument.

1. Проверка на граничные значения:
   1. Входные значения: (x = 999, n = 1).

Ожидаемый результат: The power n of x is 999.

* 1. Входные значения: (x = 0, n = 100).

Ожидаемый результат: The power n of x is 0.

**Выполнение тестовых случаев**

Запустим программу с заданными значениями аргументов.

**Оценка результатов выполнения программы на тестах**

В процессе *тестирования* *Оракул* последовательно получает элементы множества (X,Y) и соответствующие им результаты вычислений YВ. В процессе *тестирования* производится оценка результатов выполнения путем сравнения получаемого результата с ожидаемым.

### Три фазы тестирования

Реализация *тестирования* разделяется на три этапа:

* Создание *тестового набора (test suite)* путем ручной разработки или автоматической генерации для конкретной среды *тестирования* (*testing environment*).
* *Прогон программы на тестах*, управляемый тестовым монитором (*test monitor*, *test driver* [IEEE Std 829-1983], [[ 9 ]](https://www.intuit.ru/studies/professional_retraining/941/courses/48/literature#literature.9)) с получением протокола результатов *тестирования* (*test log*).
* Оценка результатов выполнения программы на *наборе тестов* с целью принятия решения о продолжении или остановке *тестирования*.

Основная *проблема тестирования* - *определение* достаточности *множества* тестов для истинности вывода о правильности реализации программы, а также нахождения *множества* тестов, обладающего этим свойством.

### Простой пример

Рассмотрим вопросы *тестирования* на примере простой программы ([Пример 2.6](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1426?page=4#example.2.6)) на языке С#. Текст этой программы и некоторых других несколько видоизменен с целью сделать иллюстрацию описываемых фактов более прозрачной.

/\* Функция вычисляет неотрицательную

степень n числа x \*/

1 double Power(double x, int n){

2 double z=1; int i;

3 for (i=1;

4 n>=i;

5 i++)

6 {z = z\*x;} /\* Возврат в п.4 \*/

7 return z;}

2.6. Пример простой программы на языке С#

/\* Функция вычисляет неотрицательную

степень n числа x \*/

1 double Power(double x, int n){

2 double z=1; int i;

3 for (i=1;

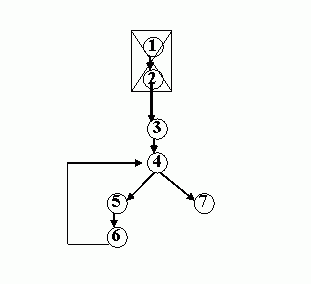
4 n>=i;

5 i++)

6 {z = z\*x;} /\* Возврат в п.4 \*/

7 return z;}

2.6.1. Пример простой программы на языке С



**Рис. 2.2.**Управляющий граф программы

*Управляющий граф программы (УГП)* на [Рис. 2.2](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1426?page=4#image.2.2) отображает *поток* управления программы. *Нумерация* узлов графа совпадает с нумерацией строк программы. Узлы 1 и 2 не включаются в *УГП*, поскольку отображают строки описаний, т.е. не содержат управляющих операторов.

### Управляющий граф программы

***Управляющий граф программы (УГП)*** – граф G(V,A), где V(V1,… Vm) – множество вершин (операторов), A(A1,… An) – множество дуг (управлений), соединяющих операторы-вершины.

***Путь*** – последовательность вершин и дуг УГП, в которой любая дуга выходит из вершины Vi и приходит в вершину Vj, например: (3,4,7), (3,4,5,6,4,5,6), (3,4), (3,4,5,6)

***Ветвь*** – *путь* (V1, V2, … Vk), где V1 - либо первый, либо условный оператор программы, Vk - либо условный оператор, либо оператор выхода из программы, а все остальные операторы – безусловные, например: (3,4) (4,5,6,4) (4,7). *Пути*, различающиеся хотя бы числом прохождений цикла – разные *пути*, поэтому число *путей* в программе может быть не ограничено. *Ветви* - линейные участки программы, их конечноe число.

Существуют реализуемые и нереализуемые *пути* в программе, в нереализуемые *пути* в обычных условиях попасть нельзя.

float H(float x,float y)

{

float H;

1 if (x\*x+y\*y+2<=0)

2 H = 17;

3 else H = 64;

4 return H\*H+x\*x;

}

2.7. Пример описания функции с реализуемыми и нереализуемыми путями

float H(float x,float y)

{

float H;

1 if (x\*x+y\*y+2<=0)

2 H = 17;

3 else H = 64;

4 return H\*H+x\*x;

}

2.7.1. Пример описания функции с реализуемыми и нереализуемыми путями

Например, для функции [Пример 2.7](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1426?page=4#example.2.7) *путь* (1,3,4) реализуем, *путь* (1,2,4) нереализуем в условиях нормальной работы. Но при сбоях даже нереализуемый *путь* может реализоваться.

### Основные проблемы тестирования

Рассмотрим два примера *тестирования*:

1. Пусть программа H(x:int, y:int) реализована в машине с 64 разрядными словами, тогда мощность множества тестов ||(X,Y)||=2\*\*128

Это означает, что компьютеру, работающему на частоте 1Ггц, для *прогона* этого *набора тестов* (при условии, что один тест выполняется за 100 команд) потребуется ~ 3K лет.

1. На [Рис. 2.3](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1426?page=4#image.2.3) приведен фрагмент схемы программы управления схватом робота, где интервал между моментами срабатывания схвата не определен.

Этот тривиальный пример требует *прогона* бесконечного множества последовательностей входных значений с разными интервалами срабатывания схвата ([Пример 2.8](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1426?page=4#example.2.8)).

// Прочитать значения датчика

static public bool ReadSensor(bool Sensor)

{

//...чтение значения датчика

Console.WriteLine("...reading sensor value");

return Sensor;

}

// Открыть схват

static public void OpenHand()

{

//...открываем схват

Console.WriteLine("...opening hand");

}

// Закрыть схват

static public void CloseHand()

{

//...закрываем схват

Console.WriteLine("...closing hand");

}

[STAThread]

static void Main(string[] args)

{

while (true)

{

Console.WriteLine("Enter Sensor value (true/false)");

if (ReadSensor(Convert.ToBoolean(Console.ReadLine())))

{

OpenHand();

CloseHand();

}

}

}

2.8. Фрагмент программы срабатывания схвата

#include <stdio.h>

/\* Прочитать значения датчика \*/

int ReadSensor(int Sensor)

{

/\* ...чтение значения датчика \*/

printf("...reading sensor value\n");

return Sensor;

}

/\* Открыть схват \*/

void OpenHand()

{

/\* ...открываем схват \*/

printf("...opening hand\n");

}

/\* Закрыть схват \*/

void CloseHand()

{

/\* ...закрываем схват \*/

printf("...closing hand\n");

}

void main(void)

{

int s;

while (1)

{

printf("Enter Sensor value (0/1)");

scanf("%d",&s);

if (ReadSensor(s))

{

OpenHand();

CloseHand();

}

}

}

2.8.1. Фрагмент программы срабатывания схвата



**Рис. 2.3.**Тестовая последовательность сигналов датчика схвата

Отсюда *вывод*:

* *Тестирование* программы на всех входных значениях невозможно.
* Невозможно *тестирование* и на всех *путях*.
* Следовательно, надо отбирать конечный *набор тестов*, позволяющий проверить программу **на основе наших интуитивных представлений**

**Требование к тестам** - программа на любом из них должна останавливаться, т.е. **не зацикливаться**. Можно ли заранее гарантировать *останов* на любом тесте?

* В теории алгоритмов доказано, что не существует общего метода для решения этого вопроса, а также вопроса, достигнет ли программа на данном тесте заранее фиксированного оператора.

Задача о *выборе конечного набора тестов* (X,Y) для проверки программы в общем случае неразрешима.

Поэтому для решения практических задач остается искать частные случаи решения этой задачи.

**Вывод:**

В Ruby тип данных совпадает с классом, к которому принадлежат объекты. Основными типами данных являются: числа целые и дробные, строки, символы, массивы, хэши, диапазоны.

**Вопросы для самоконтроля:**

1. К какому типу относятся числа 5 и 5.0?
2. К какому типу относятся объект [1,2,42,25]?
3. Чем отличаются строки в двойных и одинарных кавычках?
4. Что такое хэш?
5. Что такое символы?

# Лекция 3. Критерии выбора тестов

### Требования к идеальному критерию тестирования

Требования к *идеальному критерию* были выдвинуты в работе [[ 11 ]](https://www.intuit.ru/studies/professional_retraining/941/courses/48/literature#literature.11):

1. **Критерий должен быть достаточным**, т.е. показывать, когда некоторое конечное множество тестов достаточно для тестирования данной программы.
2. **Критерий должен быть полным**, т.е. в случае ошибки должен существовать тест из множества тестов, удовлетворяющих критерию, который раскрывает ошибку.
3. **Критерий должен быть надежным**, т.е. любые два множества тестов, удовлетворяющих ему, одновременно должны раскрывать или не раскрывать ошибки программы
4. **Критерий должен быть легко проверяемым**, например вычисляемым на тестах

Для нетривиальных классов программ в общем случае **не существует полного и надежного критерия**, зависящего от программ или спецификаций.

Поэтому мы стремимся к идеальному общему критерию через реальные частные.

### Классы критериев

1. *Структурные критерии* используют информацию о структуре программы (критерии так называемого "белого ящика")
2. *Функциональные критерии* формулируются в описании требований к программному изделию ( **критерии так называемого "черного ящика"** )
3. Критерии *стохастического тестирования* формулируются в терминах проверки наличия заданных свойств у тестируемого приложения, средствами проверки некоторой статистической гипотезы.
4. *Мутационные критерии* ориентированы на проверку свойств программного изделия на основе подхода Монте-Карло.

### Структурные критерии (класс I).

*Структурные критерии* используют *модель программы* в виде "белого ящика", что предполагает *знание* исходного текста программы или *спецификации программы* в виде потокового графа управления. Структурная *информация* понятна и доступна разработчикам подсистем и модулей приложения, поэтому данный *класс* критериев часто используется на этапах модульного и *интеграционного тестирования* (*Unit testing*, *Integration testing*).

*Структурные критерии* базируются на основных элементах УГП, операторах, ветвях и путях.

* Условие критерия **тестирования команд** (критерий С0) - набор тестов в совокупности должен обеспечить прохождение каждой команды не менее одного раза. Это слабый критерий, он, как правило, используется в больших программных системах, где другие критерии применить невозможно.
* Условие критерия **тестирования ветвей** (критерий С1) - набор тестов в совокупности должен обеспечить прохождение каждой ветви не менее одного раза. Это достаточно сильный и при этом экономичный критерий, поскольку множество ветвей в тестируемом приложении конечно и не так уж велико. Данный критерий часто используется в системах *автоматизации тестирования*.
* Условие критерия **тестирования путей** (критерий С2) - набор тестов в совокупности должен обеспечить прохождение каждого пути не менее 1 раза. Если программа содержит цикл (в особенности с неявно заданным числом итераций), то число итераций ограничивается константой (часто - 2, или *числом классов* выходных путей).

На [пример 3.1](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1428?page=1#example.3.1) приведен пример простой программы. Рассмотрим условия ее тестирования в соответствии со *структурными критериями*.

1 public void Method (ref int x)

{

2 if (x>17)

3 x = 17-x;

4 if (x==-13)

5 x = 0;

6 }

3.1. Пример простой программы, для тестирования по структурным критериям

1 void Method (int \*x)

{

2 if (\*x>17)

3 \*x = 17-\*x;

4 if (\*x==-13)

5 \*x = 0;

6 }

3.1.1. Пример простой программы, для тестирования по структурным критериям

*Тестовый набор* из одного теста, удовлетворяет критерию команд (C0):

(X,Y)={(xвх=30, xвых=0)} покрывает все *операторы* трассы 1-2-3-4-5-6

*Тестовый набор* из двух тестов, удовлетворяет критерию ветвей (C1):

(X,Y)={(30,0), (17,17)} добавляет 1 тест к множеству тестов для С0 и трассу 1-2-4-6. Трасса 1-2-3-4-5-6 проходит через все ветви достижимые в операторах if при условии true, а трасса 1-2-4-6 через все ветви, достижимые в операторах if при условии false.

*Тестовый набор* из четырех тестов, удовлетворяет критерию путей (*C2*):

(X,Y)={(30,0), (17,17), (-13,0), (21,-4)}

Набор условий для двух операторов if c метками 2 и 4 приведен в [таблица 3.1](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1428?page=1#table.3.1)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Таблица 3.1. Условия операторов if | | | | |
|  | (30,0) | (17,17) | (-13,0) | (21,-4) |
| 2 if (x>17) | > | \le | \le | > |
| 4 if (x==-13) | \ne | \ne | = | \ne |

Критерий путей С2 проверяет программу более тщательно, чем критерии - C1, однако даже если он удовлетворен, нет оснований утверждать, что *программа* реализована в соответствии со спецификацией.

Например, если спецификация задает условие, что |x|<=100, невыполнимость которого можно подтвердить на тесте (-177,-177). Действительно, *операторы* 3 и 4 на тесте (-177,-177) не изменят величину х=-177 и результат не будет соответствовать спецификации.

*Структурные критерии* не проверяют соответствие спецификации, если оно не отражено в структуре программы. Поэтому при успешном тестировании программы по критерию *C2* мы можем не заметить ошибку, связанную с невыполнением некоторых условий спецификации требований.

### Функциональные критерии (класс II)

*Функциональный критерий* - важнейший для программной индустрии критерий тестирования. Он обеспечивает, прежде всего, *контроль* степени выполнения требований заказчика в программном продукте. Поскольку требования формулируются к продукту в целом, они отражают взаимодействие тестируемого приложения с окружением. При *функциональном тестировании* преимущественно используется модель "черного ящика". Проблема *функционального тестирования* - это, прежде всего, трудоемкость; дело в том, что документы, фиксирующие требования к программному изделию (*Software* requirement *specification*, *Functional specification* и т.п.), как правило, достаточно объемны, тем не менее, соответствующая проверка должна быть всеобъемлющей.

Ниже приведены частные виды *функциональных критериев*.

* **Тестирование пунктов спецификации** - набор тестов в совокупности должен обеспечить проверку каждого тестируемого пункта не менее одного раза.

Спецификация требований может содержать сотни и тысячи пунктов требований к программному продукту и каждое из этих требований при тестировании должно быть проверено в соответствии с критерием не менее чем одним тестом

* **Тестирование классов входных данных** - набор тестов в совокупности должен обеспечить проверку представителя каждого класса входных данных не менее одного раза.

При создании тестов классы входных данных сопоставляются с режимами использования тестируемого компонента или подсистемы приложения, что заметно сокращает варианты перебора, учитываемые при разработке тестовых наборов. Следует заметить, что перебирая в соответствии с критерием величины входных переменных (например, различные файлы - источники входных данных), мы вынуждены применять мощные тестовые наборы. Действительно, наряду с ограничениями на величины входных данных, существуют ограничения на величины входных данных во всевозможных комбинациях, в том числе проверка реакций системы на появление ошибок в значениях или структурах входных данных. Учет этого многообразия - процесс трудоемкий, что создает сложности для применения критерия

* **Тестирование правил** - набор тестов в совокупности должен обеспечить проверку каждого правила, если входные и выходные значения описываются набором правил некоторой грамматики.

Следует заметить, что грамматика должна быть достаточно простой, чтобы трудоемкость разработки соответствующего набора тестов была реальной (вписывалась в сроки и штат специалистов, выделенных для реализации *фазы тестирования*)

* **Тестирование классов выходных данных** - набор тестов в совокупности должен обеспечить проверку представителя каждого выходного класса, при условии, что выходные результаты заранее расклассифицированы, причем отдельные классы результатов учитывают, в том числе, ограничения на ресурсы или на время (time out).

При создании тестов классы выходных данных сопоставляются с режимами использования тестируемого компонента или подсистемы, что заметно сокращает варианты перебора, учитываемые при разработке тестовых наборов.

* **Тестирование функций** - набор тестов в совокупности должен обеспечить проверку каждого действия, реализуемого тестируемым модулем, не менее одного раза.

Очень популярный на практике критерий, который, однако, не обеспечивает покрытия части функциональности тестируемого компонента, связанной со структурными и поведенческими свойствами, описание которых не сосредоточено в отдельных функциях (т.е. описание рассредоточено по компоненту).

Критерий тестирования функций объединяет отчасти особенности *структурных* и *функциональных критериев*. Он базируется на модели "полупрозрачного ящика", где явно указаны не только входы и выходы тестируемого компонента, но также состав и структура используемых методов (функций, процедур) и классов.

* **Комбинированные критерии для программ и спецификаций** - набор тестов в совокупности должен обеспечить проверку всех комбинаций непротиворечивых условий программ и спецификаций не менее одного раза.

При этом все комбинации непротиворечивых условий надо подтвердить, а условия противоречий следует обнаружить и ликвидировать.

#### Пример применения функциональных критериев тестирования для разработки набора тестов по критерию классов входных данных

Пусть для решения задачи тестирования системы "Система управления автоматизированным комплексом хранения подшипников" (см. Приложение 1, FS) был разработан следующий фрагмент спецификации требований:

1. Произвести опрос статуса склада (вызвать функцию GetStoreStat ). Добавить в журнал сообщений запись "СИСТЕМА : Запрошен статус СКЛАДА". В зависимости от полученного значения произвести следующие действия:
   * Полученный статус склада = 32. В приемную ячейку склада поступил подшипник. Система должна:
     1. Добавить в журнал сообщений запись "СКЛАД : Статус СКЛАДА = 32".
     2. Получить параметры поступившего подшипника с терминала подшипника (должна быть вызвана функция GetRollerPar ).
     3. Добавить в журнал сообщений запись "СИСТЕМА: Запрошены параметры подшипника".
     4. В зависимости от возвращенного функцией GetRollerPar значения должны быть выполнены следующие действия ([таблица 3.2](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1428?page=2#table.3.2)):

|  |  |
| --- | --- |
| Таблица 3.2. Действия по результатам функции GetRollerPar | |
| **Значение, возвращенное функцией GetRollerPar** | **Действия системы** |
| **...** | ... |
| **0** | * + Добавить на первое место команду GetR - "ПОЛУЧИТЬ ИЗ ПРИЕМНИКА В ЯЧЕЙКУ"   + Добавить в журнал сообщений запись "ТЕРМИНАЛ ПОДШИПНИКА: 0 - параметры возвращены <Номер\_группы>" |
| **1** | Добавить в журнал сообщений запись "ТЕРМИНАЛ ПОДШИПНИКА: 1 - нет данных" |
| **...** | ... |

1. Произвести опрос терминала оси (вызвать функцию получения сообщения от терминала - GetAxlePar ). В журнал сообщений должно быть добавлено сообщение "СИСТЕМА : Запрошены параметры оси". В зависимости от возвращенного функцией GetAxlePar значения должны быть выполнены следующие действия ([таблица 3.3](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1428?page=2#table.3.3)):

|  |  |
| --- | --- |
| Таблица 3.3. Действия по результатам функции GetAxlePar | |
| **Значение, возвращенное функцией GetAxlePar** | **Действия системы** |
| **...** | ... |
| **1** | Добавить в журнал сообщений запись "ТЕРМИНАЛ ОСИ: 1 - нет данных" |
| **...** | ... |

Определим классы входных данных для параметра - статус склада:

1. Статус склада = 0 (правильный).
2. Статус склада = 4 (правильный).
3. Статус склада = 16 (правильный).
4. Статус склада = 32 (правильный).
5. Статус склада = любое другое значение (ошибочный).

Теперь рассмотрим тестовые случаи:

1. Тестовый случай 1 (покрывает класс 4):

Состояние окружения (входные данные - X ):

Статус склада - 32.

...

Ожидаемая последовательность событий (выходные данные - Y ):

Система запрашивает статус склада (вызов функции GetStoreStat ) и получает 32

...

1. Тестовый случай 2 (покрывает класс 5):

Состояние окружения (входные данные - X ):

Статус склада - 12dfga.

...

Ожидаемая последовательность событий (выходные данные - Y ):

Система запрашивает статус склада (вызов функции GetStoreStat ) и согласно пункту спецификации при ошибочном значении статуса склада в журнал добавляется сообщение "СКЛАД : ОШИБКА : Неопределенный статус".

### Стохастические критерии (класс III)

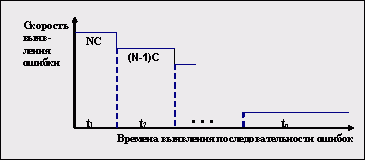
*Стохастическое тестирование* применяется при тестировании сложных программных комплексов - когда набор детерминированных тестов (X,Y) имеет громадную *мощность*. В случаях, когда подобный набор невозможно разработать и исполнить на *фазе тестирования*, можно применить следующую методику.

* Разработать программы - имитаторы случайных последовательностей входных сигналов {x}.
* Вычислить независимым способом значения {y} для соответствующих входных сигналов {x} и получить тестовый набор (X,Y).
* Протестировать приложение на тестовом наборе (X,Y), используя два способа контроля результатов:
  + Детерминированный контроль - проверка соответствия вычисленного значения yв\in \{ y\} значению y, полученному в результате *прогона теста* на наборе {x} - случайной последовательности входных сигналов, сгенерированной имитатором.
  + Стохастический контроль - проверка соответствия множества значений {yв}, полученного в результате прогона тестов на наборе входных значений {x}, заранее известному распределению результатов F(Y).

В этом случае множество Y неизвестно (его вычисление невозможно), но известен закон распределения данного множества.

Критерии *стохастического тестирования*

* **Cтатистические методы** окончания тестирования - стохастические методы принятия решений о совпадении гипотез о распределении случайных величин. К ним принадлежат широко известные: метод Стьюдента ( St ), метод Хи-квадрат ( \chi ^{2} ) и т.п.
* Метод *оценки скорости выявления ошибок* - основан на модели скорости выявления ошибок [[ 12 ]](https://www.intuit.ru/studies/professional_retraining/941/courses/48/literature#literature.12), согласно которой тестирование прекращается, если оцененный интервал времени между текущей ошибкой и следующей слишком велик для *фазы тестирования* приложения.



**Рис. 3.1.**Зависимость скорости выявления ошибок от времени выявления

При формализации модели скорости выявления ошибок ([рис. 3.1](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1428?page=3#image.3.1)) использовались следующие обозначения:

N - исходное число ошибок в программном комплексе перед тестированием,

C - константа снижения скорости выявления ошибок за счет нахождения очередной ошибки,

t1, t2,… tn - *кортеж* возрастающих интервалов обнаружения последовательности из n ошибок,

T - время выявления n ошибок.

Если допустить, что за время T выявлено n ошибок, то справедливо соотношение (1), утверждающее, что *произведение* скорости выявления i ошибки и времени выявления i ошибки есть 1 по определению:

(1) (N-i+1)\*C\*ti = 1

В этом предположении справедливо соотношение (2) для n ошибок:

(2) N*C*t_{1}+(N-1)*C*t_{2}+…+(N-n+1)*C*t_{n}=n 
\\
    N*C*(t_{1}+t_{2}+…+t_{n}) - C*\Sigma (i-1)t_{i} = n
\\
    NCT - C*\Sigma (i-1)t_{i} = n

Если из (1) определить ti и просуммировать от 1 до n, то придем к соотношению (3) для времени T выявления n ошибок


(3) \Sigma 1/(N-i+1) = TC

Если из (2) выразить C, приходим к соотношению (4):

(4) C = n/(NT - \Sigma (i-1)t_{i})

Наконец, подставляя C в (3), получаем окончательное соотношение (5), удобное для оценок:

(5) \Sigma 1/(N-i+1) = n/(N - 1/T*\Sigma (i-1)t_{i})

Если оценить величину N приблизительно, используя известные методы оценки числа ошибок в программе [[ 2 ]](https://www.intuit.ru/studies/professional_retraining/941/courses/48/literature#literature.2), [[ 13 ]](https://www.intuit.ru/studies/professional_retraining/941/courses/48/literature#literature.13)или данные о *плотности ошибок* для проектов рассматриваемого класса из исторической *базы данных* проектов, и, кроме того, использовать текущие данные об *интервалах между ошибками* t1, t2 … tn, полученные на *фазе тестирования*, то, подставляя эти данные в (5), можно получить оценку tn+1 -временного интервала необходимого для нахождения и исправления очередной ошибки (будущей ошибки).

Если tn+1>Td - допустимого времени тестирования проекта, то тестирование заканчиваем, в противном случае продолжаем *поиск* ошибок.

Наблюдая последовательность интервалов ошибок t1, t2 … tn, и время, потраченное на выявление n ошибок T=\Sigma t_{i}, можно прогнозировать *интервал* времени до следующей ошибки и уточнять в соответствии с (4) величину C.

Критерий Moranda очень практичен, так как опирается на информацию, традиционно собираемую в процессе тестирования.

### Мутационный критерий (класс IV).

Постулируется, что профессиональные программисты пишут сразу почти правильные программы, отличающиеся от правильных мелкими ошибками или описками типа - *перестановка* местами максимальных значений индексов в описании массивов, ошибки в знаках арифметических операций, занижение или завышение границы *цикла* на 1 и т.п. Предлагается подход, позволяющий на основе мелких ошибок оценить общее число ошибок, оставшихся в программе.

Подход базируется на следующих понятиях:

***Мутации*** - мелкие ошибки в программе.

***Мутанты*** - программы, отличающиеся друг от друга *мутациями* .

**Метод мутационного тестирования** - в разрабатываемую программу P вносят *мутации*, т.е. искусственно создают программы-*мутанты* P1, P2... Затем *программа* P и ее *мутанты* тестируются на одном и том же наборе тестов (X,Y).

Если на наборе (X,Y) подтверждается *правильность программы* P и, кроме того, выявляются все внесенные в программы-*мутанты* ошибки, то **набор тестов (X,Y) соответствует***мутационному критерию*, а тестируемая *программа* объявляется **правильной**.

Если некоторые *мутанты* не выявили всех *мутаций*, то надо расширять набор тестов (X,Y) и продолжать тестирование.

#### Пример применения мутационного критерия

Тестируемая программа P приведена на [пример 3.2](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1428?page=4#example.3.2). Для нее создается две программы-мутанта P1 и P2.

В P1 изменено начальное значение переменной z с 1 на 2 ([пример 3.3](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1428?page=4#example.3.3)).

В P2 изменено начальное значение переменной i с 1 на 0 и граничное значение индекса цикла с n на n-1 ([пример 3.4](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1428?page=4#example.3.4)).

При запуске тестов (X,Y) = {(x=2,n=3,y=8),(x=999,n=1,y=999), (x=0,n=100,y=0 } выявляются все ошибки в программах-мутантах и ошибка в основной программе, где в условии цикла вместо n стоит n-1:

// Метод вычисляет неотрицательную

// степень n числа x

static public double PowerNonNeg(

double x, int n)

{

double z=1;

if (n>0)

{

for (int i=1;n-1>=i;i++)

{

z = z\*x;

}

}

else Console.WriteLine(

"Ошибка ! Степень числа n должна

быть больше 0.");

return z;

}

3.2. Основная программа P

double PowerNonNeg(double x, int n)

{

double z=1;

int i;

if (n>0)

{

for (i=1;n-1>=i;i++)

{

z = z\*x;

}

}

else printf(

"Ошибка ! Степень числа n должна

быть больше 0.\n");

return z;

}

3.2.1. Основная программа P

Измененное начальное значение переменной z в *мутанте* Р1 ( z=2 ):

// Метод вычисляет неотрицательную

// степень n числа x

static public double PowerMutant1(

double x, int n)

{

double z=2;

if (n>0)

{

for (int i=1;n>=i;i++)

{

z = z\*x;

}

}

else Console.WriteLine(

"Ошибка ! Степень числа n должна

быть больше 0.");

return z;

}

3.3. Программа мутант P1.

double PowerMutant1(double x, int n)

{

double z=2;

int i;

if (n>0)

{

for (i=1;n>=i;i++)

{

z = z\*x;

}

}

else printf(

"Ошибка ! Степень числа n должна

быть больше 0.\n");

return z;

}

3.3.1. Программа мутант P1.

Измененное начальное значение переменной i и границы цикла в *мутанте* P2 ( i=0;n-1 ):

// Метод вычисляет неотрицательную

// степень n числа x

static public double PowerMutant2(

double x, int n)

{

double z=1;

if (n>0)

{

for (int i=0;n-1>=i;i++)

{

z = z\*x;

}

}

else Console.WriteLine(

"Ошибка ! Степень числа n должна

быть больше 0");

return z;

}

3.4. Программа-мутант P2.

double PowerMutant2(double x, int n)

{

double z=1;

int i;

if (n>0)

{

for (i=0;n-1>=i;i++)

{

z = z\*x;

}

}

else printf(

"Ошибка ! Степень числа n должна

быть больше 0.\n");

return z;

}

3.4.1. Программа-мутант P2.

**Вывод:**

Сама по себе переменная в Ruby — просто уникальный набор символов английского алфавита и цифр, который можно использовать в программе для действий с объектом. Этот набор символов называется «имя переменной» и вы сами придумываете имена своих переменных. Связь объекта с переменной происходит с помощью знака равенства.

Логический оператор «И» объединяет другие условия в одно большое условие, и это большое условие выполняется только когда все условия выполняются. Оператор же «ИЛИ» объединяет другие условия в одно большое условие, и это большое условие выполняется, когда хотя бы одно из условий выполняются.

**Вопросы для самоконтроля:**

1. Как работает оператор if
2. Что такое переменная?
3. Как работает оператор И?
4. Как работает оператор ИЛИ?
5. Как определить класс объекта?

# Лекция 4. Оценка оттестированности проекта: метрики и методика интегральной оценки

### Оценка Покрытия Программы и Проекта

*Тестирование программы* Р по некоторому критерию С означает покрытие *множества* компонентов программы P М = {m1...mk} по элементам или по связям

T = {t1...tn} - *кортеж* неизбыточных тестов ti.

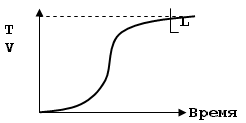
Тест ti неизбыточен, если существует покрытый им *компонент* mi из M(P,C), не покрытый ни одним из предыдущих тестов t1...ti-1. Каждому ti соответствует неизбыточный *путь* pi - последовательность вершин от входа до выхода.

V(P,C) - *сложность тестирования* Р по критерию С - измеряется *max* числом неизбыточных тестов, покрывающих все элементы *множества* M(P,C)

*DV*(P,C,Т) - остаточная *сложность тестирования* Р по критерию С - измеряется *max* числом неизбыточных тестов, покрывающих элементы *множества* M(P,C), оставшиеся непокрытыми, после прогона набора тестов Т. Величина *DV* строго и монотонно убывает от V до 0.

TV(P,C,Т) = (V-*DV*)/V - оценка *степени тестированности* Р по критерию С.

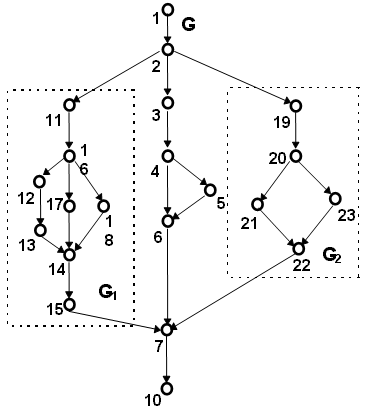
Критерий окончания тестирования TV(P,C,Т) >= L, где (0 <= L <= 1). L - уровень оттестированности, заданный в требованиях к программному продукту.



**Рис. 4.1.**Метрика оттестированности приложения

Рассмотрим две модели программного обеспечения, используемые при оценке оттестированности.

Для оценки *степени оттестированности* часто используется *УГП* - *управляющий граф программы*. *УГП* многокомпонентного объекта G ([Рис. 4.2](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1430?page=1#image.4.2), [Пример 4.4](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1430?page=2#example.4.4)), содержит внутри себя два компонента G1 и G2, *УГП* которых раскрыты.



**Рис. 4.2.**Плоская модель УГП компонента G

В результате *УГП* компонента G имеет такой вид, как если бы компоненты G1 и G2 в его структуре специально не выделялись, а *УГП* компонентов G1 и G2 были вставлены в *УГП* G. Для тестирования компонента G в соответствии с критерием путей потребуется прогнать *тестовый набор*, покрывающий следующий набор трасс *графа* G ([Пример 4.1](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1430?page=1#example.4.1)):

P1(G) = 1-2-3-4-5-6-7-10;

P2(G) = 1-2-3-4-6-7-10;

P3(G) = 1-2-11-16-18-14-15-7-10;

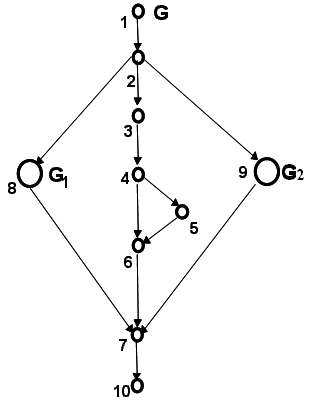
P4(G) = 1-2-11-16-17-14-15-7-10;

P5(G) = 1-2-11-16-12-13-14-15-7-10;

P6(G) = 1-2-19-20-23-22-7-10;

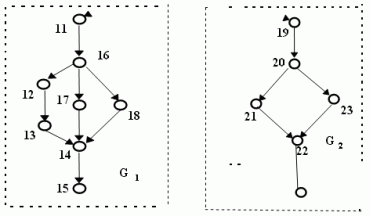
P7(G) = 1-2-19-20-21-22-7-10;

4.1. Набор трасс, необходимых для покрытия плоской модели УГП компонента G



**Рис. 4.3.**Иерархическая модель УГП компонента G

*УГП* компонента G, представленный в виде иерархической модели, приведен на [Рис. 4.3](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1430?page=1#image.4.3), [Пример 4.5](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1430?page=2#example.4.5). В иерархическом *УГП* G входящие в его состав компоненты представлены ссылками на свои *УГП* G1 и G2 ([Рис. 4.4](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1430?page=1#image.4.4), [Пример 4.5](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1430?page=2#example.4.5))



**Рис. 4.4.**Иерархическая модель: УГП компонент G1 и G2

Для исчерпывающего тестирования иерархической модели компонента G в соответствии с критерием путей требуется прогнать следующий набор трасс ([Пример 4.2](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1430?page=1#example.4.2)):

P1(G) = 1-2-3-4-5-6-7-10;

P2(G) = 1-2-3-4-6-7-10;

P3(G) = 1-2-8-7-10;

P4(G) = 1-2-9-7-10.

4.2. Набор трасс, необходимых для покрытия иерархической модели УГП компонента G

Приведенный набор трасс достаточен при условии, что компоненты G1 и G2 в свою *очередь* исчерпывающе протестированы. Чтобы обеспечить выполнение этого условия в соответствии с критерием путей, надо прогнать все трассы [Пример 4.3](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1430?page=1#example.4.3).

P11(G1)=11-16-12-13-14-15;

P12(G1)=11-16-17-14-15;

P13(G1)=19-20-23-22;

P21(G2)=19-20-21-22;

P22(G2)=11-16-18-14-15.

4.3. Набор трасс иерархической модели УГП, необходимых для покрытия УГП компонентов G1 и G2

Оценка *степени тестированности* *плоской модели* определяется долей прогнанных трасс из набора необходимых для покрытия в соответствии с критерием С.

(1) TV(G,С) = (V-DV)/V = \sum PT_{i}(G) / (\sum P_{i}(G)),

где PTi(G) - тестовый *путь* ( ti ) в *графе* G *плоской модели* равен 1, если он протестирован (прогнан), или 0, если нет.

Например, если в *УГП* ([Пример 4.1](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1430?page=1#example.4.1)) тесты t6 и t7, которым соответствуют трассы P6 и P8, не прогнаны, то в соответствии с соотношением (1) для TV(G,С) *степень тестированности* будет оценена в 0.71.

Оценка тестированности иерархической модели определяется на основе учета оценок тестированности компонентов. Если трасса некоторого теста tj*УГП* G включает узлы, представляющие компоненты Gj1,..Gjm, оценка TV *степени тестированности* которых известна, то оценка тестированности PTi(G) при реализации этой трассы определяется не 1, а минимальной из оценок TV для компонентов.

Интегральная оценка определяется соотношением (2):

(2) TV(G,C) = (V-DV)/V = (\sum PT_{i}(G) * \sum (TV(G_{ij},C))) / (\sum P_{i}(G))

где PTi(G) - тестовый *путь* ( ti ) в *графе* G равен 1, если протестирован, или 0, если нет. В *путь* PTi*графа* G может входить j узлов модулей Gij со своей *степенью тестированности* TV(Gij,С), из которых мы берем *min*, что дает худшую оценку степени тестированности пути.

// Пример плоской модели проекта

public void G()

{

int TerminalStatus=0, CommandStatus=0;

bool IsPresent=true, CommandFound=true;

1 Init();

2 switch (TerminalStatus)

{

case 11 :

11 AddCommand();

16 switch (CommandStatus)

{

case 12 :

12 GetMessage();

13 ClearQueue();

break;

case 17 :

17 ClearQueue();

break;

case 18 :

18 DumpQueue();

break;

}

14 ProcessCommand();

15 Commit();

break;

case 3 :

3 AskTerminal();

4 if (IsPresent)

{

5 Connect();

}

6 RebuildQueue();

break;

case 19 :

19 SearchValidCommand();

20 if (CommandFound)

{

21 AnalyzeCommand();

}

else

{

23 LogError();

}

22 MoveNextCommand();

break;

}

7 LogResults();

10 DisposeAll();

}

Пример 4.4. Пример программы для плоской модели (Рис. 4.2)

// Пример плоской модели проекта

void G()

{

int TerminalStatus=0, CommandStatus=0;

int IsPresent=1, CommandFound=1;

1 Init();

2 switch (TerminalStatus)

{

case 11 :

11 AddCommand();

16 switch (CommandStatus)

{

case 12 :

12 GetMessage();

13 ClearQueue();

break;

case 17 :

17 ClearQueue();

break;

case 18 :

18 DumpQueue();

break;

}

14 ProcessCommand();

15 Commit();

break;

case 3 :

3 AskTerminal();

4 if (IsPresent)

{

5 Connect();

}

6 RebuildQueue();

break;

case 19 :

19 SearchValidCommand();

20 if (CommandFound)

{

21 AnalyzeCommand();

}

else

{

23 LogError();

}

22 MoveNextCommand();

break;

}

7 LogResults();

10 DisposeAll();

}

Пример 4.4.1. Пример программы для плоской модели (Рис. 4.2)

// Пример иерархической модели проекта

public void G1()

{

int CommandStatus=0;

AddCommand();

switch (CommandStatus)

{

case 12 :

GetMessage();

ClearQueue();

break;

case 17 :

ClearQueue();

break;

case 18 :

DumpQueue();

break;

}

ProcessCommand();

Commit();

}

public void G2()

{

bool CommandFound=true;

SearchValidCommand();

if (CommandFound)

{

AnalyzeCommand();

}

else

{

LogError();

}

MoveNextCommand();

}

public void G()

{

int TerminalStatus=0;

bool IsPresent=true;

1 Init();

2 switch (TerminalStatus)

{

case 11 :

8 G1();

break;

case 3 :

3 AskTerminal();

4 if (IsPresent)

{

5 Connect();

}

6 RebuildQueue();

break;

case 19 :

// Пример иерархической модели проекта - продолжение

9 G2();

break;

}

7 LogResults();

10 DisposeAll();

}

Пример 4.5. Пример программы для иерархической модели (Рис. 4.3)

// Пример иерархической модели проекта

void G1()

{

int CommandStatus=0;

AddCommand();

switch (CommandStatus)

{

case 12 :

GetMessage();

ClearQueue();

break;

case 17 :

ClearQueue();

break;

case 18 :

DumpQueue();

break;

}

ProcessCommand();

Commit();

}

void G2()

{

int CommandFound=1;

SearchValidCommand();

if (CommandFound)

{

AnalyzeCommand();

}

else

{

LogError();

}

MoveNextCommand();

}

void G()

{

int TerminalStatus=0;

int IsPresent=1;

1 Init();

2 switch (TerminalStatus)

{

case 11 :

8 G1();

break;

case 3 :

3 AskTerminal();

4 if (IsPresent)

{

5 Connect();

}

6 RebuildQueue();

break;

case 19 :

9 G2();

break;

}

7 LogResults();

10 DisposeAll();

}

Пример 4.5.1. Пример программы для иерархической модели (Рис. 4.3)

### Методика интегральной оценки тестированности

1. Выбор критерия С и приемочной оценки тестированности программного проекта - L
2. Построение древа классов проекта и построение *УГП* для каждого модуля
3. Модульное тестирование и оценка TV на модульном уровне
4. Построение *УГП*, интегрирующего модули в единую иерархическую ( *классовую* ) модель проекта
5. Выбор тестовых путей для проведения интеграционного или системного тестирования
6. Генерация тестов, покрывающих тестовые пути шага 5
7. Интегральная оценка тестированности проекта с учетом оценок тестированности модулей-компонентов
8. Повторение шагов 5-7 до достижения заданного уровня тестированности L

**Вывод:**

При вводе данных с клавиатуры в Ruby используется метод gets. При этом получаемое значение имеет тип String. Кроме того, результат содержит символ перевода строки /n. Поэтому рекомендуется применять метод chomp, чтобы удалит данный символ из ввода данных. Для смены кодировки ввода используется метод encode.

**Вопросы для самоконтроля:**

1. Что делает команда gets?
2. Какую команду следует применить, что удалить символ перевода строки?
3. Как работает команда abort?
4. Какой тип данных получается на выходе после ввода командой gets?
5. Как использовать метода encode?

# Лекция 5. Модульное и интеграционное тестирование

### Модульное

***Модульное тестирование*** - это тестирование программы на уровне отдельно взятых модулей, функций или классов. Цель *модульного тестирования* состоит в выявлении локализованных в модуле ошибок в реализации алгоритмов, а также в определении степени готовности системы к переходу на следующий уровень разработки и тестирования. *Модульное тестирование* проводится по принципу "белого ящика", то есть основывается на знании внутренней структуры программы, и часто включает те или иные методы анализа покрытия кода.

*Модульное тестирование* обычно подразумевает создание вокруг каждого модуля определенной среды, включающей *заглушки* для всех интерфейсов тестируемого модуля. Некоторые из них могут использоваться для подачи входных значений, другие для анализа результатов, присутствие третьих может быть продиктовано требованиями, накладываемыми компилятором и сборщиком.

На уровне *модульного тестирования* проще всего обнаружить дефекты, связанные с алгоритмическими ошибками и ошибками кодирования алгоритмов, типа работы с условиями и счетчиками циклов, а также с использованием локальных переменных и ресурсов. Ошибки, связанные с неверной трактовкой данных, некорректной реализацией интерфейсов, совместимостью, производительностью и т.п. обычно пропускаются на уровне *модульного тестирования* и выявляются на более поздних стадиях тестирования.

Именно эффективность обнаружения тех или иных типов дефектов должна определять стратегию *модульного тестирования*, то есть расстановку акцентов при определении набора входных значений. У организации, занимающейся разработкой программного обеспечения, как правило, имеется историческая *база данных* ( **Repository** ) разработок, хранящая конкретные сведения о разработке предыдущих проектов: о версиях и сборках кода ( **build** ) зафиксированных в процессе разработки продукта, о принятых решениях, допущенных просчетах, ошибках, успехах и т.п. Проведя *анализ* характеристик прежних проектов, подобных заказанному организации, можно предохранить новую разработку от старых ошибок, например, определив типы дефектов, *поиск* которых наиболее эффективен на различных этапах тестирования.

В данном случае анализируется этап *модульного тестирования*. Если *анализ* не дал нужной информации, например, в случае проектов, в которых соответствующие данные не собирались, то основным правилом становится *поиск* локальных дефектов, у которых код, ресурсы и *информация*, вовлеченные в дефект, характерны именно для данного модуля. В этом случае на *модульном уровне* ошибки, связанные, например, с неверным порядком или форматом параметров модуля, могут быть пропущены, поскольку они вовлекают информацию, затрагивающую другие модули (а именно, спецификацию интерфейса), в то время как ошибки в алгоритме обработки параметров довольно легко обнаруживаются.

Являясь по способу исполнения структурным тестированием или тестированием "белого ящика", *модульное тестирование* характеризуется степенью, в которой тесты выполняют или покрывают логику программы (исходный текст). Тесты, связанные со структурным тестированием, строятся по следующим принципам:

* На основе анализа *потока управления*. В этом случае элементы, которые должны быть покрыты при прохождении тестов, определяются на основе *структурных критериев* тестирования С0, С1,С2. К ним относятся вершины, дуги, пути *управляющего графа* программы (УГП), условия, комбинации условий и т. п.
* На основе анализа *потока данных*, когда элементы, которые должны быть покрыты, определяются при помощи *потока данных*, т. е. *информационного графа* программы.

**Тестирование на основе потока управления**. Особенности использования *структурных критериев* тестирования С0,С1,С2 были рассмотрены в лекции 3. К ним следует добавить критерий покрытия условий, заключающийся в покрытии всех логических (булевских) условий в программе. Критерии покрытия решений (ветвей - С1) и условий не заменяют друг друга, поэтому на практике используется комбинированный критерий покрытия условий/решений, совмещающий требования по покрытию и решений, и условий.

К популярным критериям относятся критерий покрытия функций программы, согласно которому каждая *функция* программы должна быть вызвана хотя бы один раз, и критерий покрытия вызовов, согласно которому каждый вызов каждой функции в программе должен быть осуществлен хотя бы один раз. Критерий покрытия вызовов известен также как критерий покрытия пар вызовов (*call* *pair* *coverage*).

**Тестирование на основе потока данных**. Этот вид тестирования направлен на выявление ссылок на неинициализированные переменные и избыточные присваивания (аномалий *потока данных* ). Как основа для стратегии тестирования *поток данных* впервые был описан в [[ 14 ]](https://www.intuit.ru/studies/professional_retraining/941/courses/48/literature#literature.14). Предложенная там стратегия требовала тестирования всех взаимосвязей, включающих в себя ссылку (использование) и *определение* переменной, на которую указывает *ссылка* (т. е. требуется покрытие дуг *информационного графа* программы). Недостаток стратегии в том, что она не включает критерий С1, и не гарантирует покрытия решений.

Стратегия требуемых пар [[ 15 ]](https://www.intuit.ru/studies/professional_retraining/941/courses/48/literature#literature.15)также тестирует упомянутые взаимосвязи. Использование переменной в предикате дублируется в соответствии с числом выходов решения, и каждая из таких требуемых взаимосвязей должна быть протестирована. К популярным критериям принадлежит критерий СР, заключающийся в покрытии всех таких пар дуг v и w, что из дуги v достижима *дуга* w, поскольку именно на дуге может произойти потеря значения переменной, которая в дальнейшем уже не должна использоваться. Для "покрытия" еще одного популярного критерия Cdu достаточно тестировать пары (*вершина*, *дуга*), поскольку *определение* переменной происходит в вершине УГП, а ее использование - на дугах, исходящих из решений, или в вычислительных вершинах.

**Методы проектирования тестовых путей для достижения заданной степени тестированности в структурном тестировании**. Процесс построения набора тестов при структурном тестировании принято делить на три фазы:

* Конструирование УГП.
* Выбор тестовых путей.
* Генерация тестов, соответствующих тестовым путям.

Первая фаза соответствует статическому анализу программы, задача которого состоит в получении графа программы и зависящего от него и от критерия тестирования *множества* элементов, которые необходимо покрыть тестами.

На третьей фазе по известным путям тестирования осуществляется *поиск* подходящих тестов, реализующих прохождение этих путей.

Вторая фаза обеспечивает выбор тестовых путей. Выделяют три подхода к построению тестовых путей:

* *Статические методы*.
* *Динамические методы*.
* Методы реализуемых путей.

**Статические методы**. Самое простое и легко реализуемое решение - построение каждого пути посредством постепенного его удлинения за счет добавления дуг, пока не будет достигнута выходная *вершина* *управляющего графа* программы. Эта идея может быть усилена в так называемых адаптивных методах, которые каждый раз добавляют только один тестовый *путь* (*входной* тест), используя предыдущие пути (тесты) как руководство для выбора последующих путей в соответствии с некоторой стратегией. Чаще всего адаптивные стратегии применяются по отношению к критерию С1. Основной недостаток *статических методов* заключается в том, что не учитывается возможная нереализуемость *построенных путей* тестирования.

**Динамические методы**. Такие методы предполагают построение полной системы тестов, удовлетворяющих заданному критерию, путем одновременного решения задачи построения покрывающего множества путей и тестовых данных. При этом можно автоматически учитывать реализуемость или нереализуемость ранее рассмотренных путей или их частей. Основной идеей *динамических методов* является подсоединение к начальным реализуемым отрезкам путей дальнейших их частей так, чтобы: 1) не терять при этом реализуемости вновь полученных путей; 2) покрыть требуемые элементы структуры программы.

**Методы реализуемых путей**. Данная методика [[ 16 ]](https://www.intuit.ru/studies/professional_retraining/941/courses/48/literature#literature.16)заключается в выделении из *множества* путей подмножества всех реализуемых путей. После чего покрывающее множество путей строится из полученного подмножества реализуемых путей.

Достоинство *статических методов* состоит в сравнительно небольшом количестве необходимых ресурсов, как при использовании, так и при разработке. Однако их реализация может содержать непредсказуемый *процент* брака (нереализуемых путей). Кроме того, в этих системах переход от покрывающего *множества* путей к полной системе тестов *пользователь* должен осуществить вручную, а эта работа достаточно трудоемкая. *Динамические методы* требуют значительно больших ресурсов как при разработке, так и при эксплуатации, однако увеличение затрат происходит, в основном, за счет разработки и эксплуатации аппарата определения реализуемости пути (символический *интерпретатор*, *решатель* неравенств). Достоинство этих методов заключается в том, что их продукция имеет некоторый качественный уровень - реализуемость путей. Методы реализуемых путей дают самый лучший результат.

#### Пример модульного тестирования

Предлагается протестировать класс TCommand, который реализует команду для склада. Этот класс содержит единственный метод TCommand.GetFullName(), спецификация которого описана (Практикум, Приложение 2 HLD) следующим образом:

|  |  |
| --- | --- |
|  | ... |
|  | Операция GetFullName() возвращает полное имя команды, соответствующее ее допустимому коду, указанному в поле NameCommand. В противном случае возвращается сообщение "ОШИБКА : Неверный код команды". Операция может быть применена в любой момент. |
|  | ... |

Разработаем спецификацию тестового случая для тестирования метода GetFullName на основе приведенной спецификации класса ([Табл. 5.1](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1432?page=1#table.5.1)):

|  |  |
| --- | --- |
| Таблица 5.1. *Спецификация теста* | |
| **Название класса**: TСommand | **Название тестового случая**: TСommandTest1 |
| **Описание тестового случая**: Тест проверяет правильность работы метода GetFullName - получения полного названия команды на основе кода команды. В тесте подаются следующие значения кодов команд (входные значения): -1, 1, 2, 4, 6, 20, (причем -1 - запрещенное значение). | |
| **Начальные условия**: Нет. | |
| **Ожидаемый результат**:  Перечисленным входным значениям должны соответствовать следующие выходные:  Коду команды -1 должно соответствовать сообщение "ОШИБКА: Неверный код команды"  Коду команды 1 должно соответствовать полное название команды "ПОЛУЧИТЬ ИЗ ВХОДНОЙ ЯЧЕЙКИ"  Коду команды 2 должно соответствовать полное название команды "ОТПРАВИТЬ ИЗ ЯЧЕЙКИ В ВЫХОДНУЮ ЯЧЕЙКУ"  Коду команды 4 должно соответствовать полное название команды "ПОЛОЖИТЬ В РЕЗЕРВ"  Коду команды 6 должно соответствовать полное название команды "ПРОИЗВЕСТИ ЗАНУЛЕНИЕ"  Коду команды 20 должно соответствовать полное название команды "ЗАВЕРШЕНИЕ КОМАНД ВЫДАЧИ" | |

Для тестирования метода класса TCommand.GetFullName() был создан тестовый драйвер - класс TCommandTester. Класс TCommandTester содержит метод TCommandTest1(), в котором реализована вся функциональность теста. В данном случае для покрытия спецификации достаточно перебрать следующие значения кодов команд: -1, 1, 2, 4, 6, 20, (-1 - запрещенное значение) и получить соответствующее им полное название команды с помощью метода GetFullName() ([Пример 5.1](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1432?page=1#example.5.1) ). Пары значений (X, Yв) при исполнении теста заносятся в log-файл для последующей проверки на соответствие спецификации.

После завершения теста следует просмотреть журнал теста, чтобы сравнить полученные результаты с ожидаемыми, заданными в спецификации тестового случая TСommandTest1 ([Пример 5.2](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1432?page=1#example.5.2)).

class TCommandTester:Tester // Тестовый драйвер

{

...

TCommand OUT;

public TCommandTester()

{

OUT=new TCommand();

Run();

}

private void Run()

{

TCommandTest1();

}

private void TCommandTest1()

{

int[] commands = {-1, 1, 2, 4, 6, 20};

for(int i=0;i<=5;i++)

{

OUT.NameCommand=commands[i];

LogMessage(commands[i].ToString()+

" : "+OUT.GetFullName());

}

}

...

}

Пример 5.1. Тестовый драйвер

-1 : ОШИБКА : Неверный код команды

1 : ПОЛУЧИТЬ ИЗ ВХОДНОЙ ЯЧЕЙКИ

2 : ОТПРАВИТЬ ИЗ ЯЧЕЙКИ В ВЫХОДНУЮ ЯЧЕЙКУ

4 : ПОЛОЖИТЬ В РЕЗЕРВ

6 : ПРОИЗВЕСТИ ЗАНУЛЕНИЕ

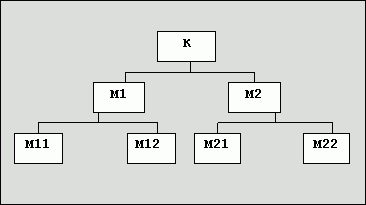
20 : ЗАВЕРШЕНИЕ КОМАНД ВЫДАЧИ

Пример 5.2. Спецификация классов тестовых случаев

### Интеграционное тестирование

***Интеграционное тестирование*** - это тестирование части системы, состоящей из двух и более модулей. Основная задача *интеграционного тестирования* - поиск дефектов, связанных с ошибками в реализации и интерпретации интерфейсного взаимодействия между модулями.

С технологической точки зрения *интеграционное тестирование* является количественным развитием *модульного*, поскольку так же, как и *модульное тестирование*, оперирует интерфейсами модулей и подсистем и требует создания тестового окружения, включая *заглушки* ( **Stub** ) на месте отсутствующих модулей. Основная разница между *модульным* и *интеграционным тестированием* состоит в целях, то есть в типах обнаруживаемых дефектов, которые, в свою *очередь*, определяют стратегию выбора входных данных и методов анализа. В частности, на уровне *интеграционного тестирования* часто применяются методы, связанные с покрытием интерфейсов, например, вызовов функций или методов, или *анализ* использования интерфейсных объектов, таких как глобальные ресурсы, средства коммуникаций, предоставляемых операционной системой.



**Рис. 5.1.**Пример структуры комплекса программ

На [Рис. 5.1](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1432?page=2#image.5.1) приведена структура комплекса программ K, состоящего из оттестированных на этапе *модульного тестирования* модулей M1, M2, M11, M12, M21, M22. Задача, решаемая методом *интеграционного тестирования*, - тестирование межмодульных связей, реализующихся при исполнении программного обеспечения комплекса K. *Интеграционное тестирование* использует модель "белого ящика" на модульном уровне. Поскольку тестировщику текст программы известен с детальностью до вызова всех модулей, входящих в тестируемый комплекс, применение структурных критериев на данном этапе возможно и оправдано.

*Интеграционное тестирование* применяется на этапе сборки модульно оттестированных модулей в единый комплекс. Известны два метода *сборки модулей*:

* **Монолитный**, характеризующийся одновременным объединением всех модулей в тестируемый комплекс
* **Инкрементальный**, характеризующийся пошаговым (помодульным) наращиванием комплекса программ с **пошаговым тестированием** собираемого комплекса. В инкрементальном методе выделяют две стратегии добавления модулей:
  + "Сверху вниз" и соответствующее ему *нисходящее тестирование*.
  + "Снизу вверх" и соответственно *восходящее тестирование*.

**Особенности монолитного тестирования** заключаются в следующем: для замены неразработанных к моменту тестирования модулей, кроме самого верхнего ( К на [Рис. 5.1](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1432?page=2#image.5.1)), необходимо дополнительно разрабатывать **драйверы** ( **test driver** ) и/или **заглушки** ( **stub** ) [[ 9 ]](https://www.intuit.ru/studies/professional_retraining/941/courses/48/literature#literature.9), замещающие отсутствующие на момент сеанса тестирования модули нижних уровней.

Сравнение *монолитного* и инкрементального подхода дает следующее:

* *Монолитное тестирование* требует больших трудозатрат, связанных с дополнительной разработкой драйверов и заглушек и со сложностью идентификации ошибок, проявляющихся в пространстве собранного кода.
* Пошаговое тестирование связано с меньшей трудоемкостью идентификации ошибок за счет постепенного наращивания объема тестируемого кода и соответственно локализации добавленной области тестируемого кода.
* *Монолитное тестирование* предоставляет большие возможности распараллеливания работ особенно на начальной *фазе тестирования*.

Особенности *нисходящего тестирования* заключаются в следующем: организация среды для исполняемой очередности вызовов оттестированными модулями тестируемых модулей, постоянная разработка и использование заглушек, организация приоритетного тестирования модулей, содержащих *операции* обмена с окружением, или модулей, критичных для тестируемого алгоритма.

Например, порядок тестирования комплекса K ([Рис. 5.1](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1432?page=2#image.5.1)) при *нисходящем тестировании* может быть таким, как показано в [примере 5.3](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1432?page=2#example.5.3), где тестовый набор, разработанный для модуля Mi, обозначен как XYi = (X, Y)i

1) K->XYK

2) M1->XY1

3) M11->XY11

4) M2->XY2

5) M22->XY22

6) M21->XY21

7) M12->XY12

Пример 5.3. Возможный порядок тестов при нисходящем тестировании

Недостатки *нисходящего тестирования*:

* Проблема разработки достаточно "интеллектуальных" заглушек, т.е. заглушек, пригодных к использованию при моделировании различных режимов работы комплекса, необходимых для тестирования
* Сложность организации и разработки среды для реализации исполнения модулей в нужной последовательности
* Параллельная разработка модулей верхних и нижних уровней приводит к не всегда эффективной реализации модулей из-за подстройки (специализации) еще не тестированных модулей нижних уровней к уже оттестированным модулям верхних уровней

**Особенности восходящего тестирования** в организации порядка *сборки* и перехода к тестированию модулей, соответствующему порядку их реализации.

Например, порядок тестирования комплекса K ([Рис. 5.1](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1432?page=2#image.5.1)) при *восходящем тестировании* может быть следующим ([пример. 5.4](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1432?page=2#example.5.4)).

1) M11->XY11

2) M12->XY12

3) M1->XY1

4) M21->XY21

5) M2(M21, Stub(M22))->XY2

6) K(M1, M2(M21, Stub(M22)) ->XYK

7) M22->XY22

8) M2->XY2

9) K->XYK

Пример 5.4. Возможный порядок тестов при восходящем тестировании

Недостатки *восходящего тестирования*:

* Запаздывание проверки концептуальных особенностей тестируемого комплекса
* Необходимость в разработке и использовании драйверов

#### Особенности интеграционного тестирования для процедурного программирования

Процесс построения набора тестов при структурном тестировании определяется принципом, на котором основывается конструирование Графа *Модели Программы* (ГМП). От этого зависит множество тестовых путей и генерация тестов, соответствующих тестовым путям.

Первым подходом к разработке программного обеспечения является *процедурное (модульное) программирование*. Традиционное *процедурное программирование* предполагает написание исходного кода в императивном (повелительном) стиле, предписывающем определенную последовательность выполнения команд, а также описание программного проекта с помощью функциональной декомпозиции. Такие языки, как Pascal и C, являются императивными. В них порядок исходных строк кода определяет порядок передачи управления, включая последовательное исполнение, выбор условий и повторное исполнение участков программы. Каждый модуль имеет несколько точек входа (при строгом написании кода - одну) и несколько точек выхода (при строгом написании кода - одну). Сложные программные проекты имеют модульно-иерархическое построение [[ 5 ]](https://www.intuit.ru/studies/professional_retraining/941/courses/48/literature#literature.5), и тестирование модулей является начальным шагом процесса тестирования ПО. Построение *графовой модели* модуля является тривиальной задачей, а тестирование практически всегда проводится по критерию покрытия ветвей C1, т.е. каждая дуга и каждая вершина графа модуля должны содержаться, по крайней мере, в одном из путей тестирования.

Таким образом, M(P,C1) = E Nij, где Е - множество дуг, а Nij - *входные вершины* ГМП.

Сложность тестирования модуля по критерию С1 выражается уточненной формулой для оценки топологической сложности МакКейба [[ 10 ]](https://www.intuit.ru/studies/professional_retraining/941/courses/48/literature#literature.10):

V(P,C1) = q + kin, где q - число бинарных выборов для условий ветвления, а kin - число входов графа.

Для *интеграционного тестирования* наиболее существенным является рассмотрение *модели программы*, построенной с использованием диаграмм *потоков управления*. Контролируются также связи через данные, подготавливаемые и используемые другими группами программ при взаимодействии с тестируемой группой. Каждая переменная межмодульного интерфейса проверяется на тождественность описаний во взаимодействующих модулях, а также на соответствие исходным программным спецификациям. Состав и структура информационных связей реализованной группы модулей проверяются на соответствие спецификации требований этой группы. Все реализованные связи должны быть установлены, упорядочены и обобщены.

При *сборке модулей* в единый программный комплекс появляется два варианта построения *графовой модели* проекта:

* Плоская или иерархическая модель проекта (например, [Рис. 4.2](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1432?page=2), [Рис. 4.3](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1432?page=2)).
* *Граф вызовов*.

Если программа P состоит из p модулей, то при интеграции модулей в комплекс фактически получается громоздкая плоская ([Рис. 4.2](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1432?page=2)) или более простая - иерархическая ([Рис. 4.3](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1432?page=2)) - модель программного проекта. В качестве критерия тестирования на интеграционном уровне обычно используется критерий покрытия ветвей C1. Введем также следующие обозначения:

|  |
| --- |
| n - число узлов в графе; |
| e - число дуг в графе; |
| q - число бинарных выборов из *условий ветвления* в графе; |
| kin - число входов в граф; |
| kout - число выходов из графов; |
| kext - число точек входа, которые могут быть вызваны извне. |

Тогда сложность *интеграционного тестирования* всей программы P по критерию C1 может быть выражена формулой [[ 17 ]](https://www.intuit.ru/studies/professional_retraining/941/courses/48/literature#literature.17):

V(P,C1) = \Sigma V(Mod_{i}, C1) - k_{in} +k_{ext} = 
\\
        e - n - k_{ext} + k_{out} = 
\\
        q + k_{ext}, (\forall Mod_{i}\in  P)

Однако при подобном подходе к построению ГМП разработчик тестового набора неизбежно сталкивается с неприемлемо высокой сложностью тестирования V(P,C) для проектов среднего и большого объема (размером в 105 - 107 строк) [[ 18 ]](https://www.intuit.ru/studies/professional_retraining/941/courses/48/literature#literature.18), что следует из роста топологической сложности *управляющего графа* по МакКейбу. Таким образом, используя плоскую или иерархическую модель, трудно дать оценку тестированности TV(P,C,T) для всего проекта и оценку зависимости тестированности проекта от тестированности отдельного модуля TV(Modi,C), включенного в этот проект.

Рассмотрим вторую модель *сборки модулей* в *процедурном программировании* - *граф вызовов*. В этой модели в случае *интеграционного тестирования* учитываются только вызовы модулей в программе. Поэтому из множества M(Modi,C) тестируемых элементов можно исключить те элементы, которые не подвержены влиянию интеграции, т. е. узлы и дуги, не соединенные с вызовами модулей: M(Mod_{i},C') = E' \cup N_{in}, где E' = \{ (n_{i}, n_{j})\in  E | n_{i} или nj содержит вызовы модулей}, т.е. E' - подмножество ребер графа модуля, а Nin - "входные" узлы графа [[ 17 ]](https://www.intuit.ru/studies/professional_retraining/941/courses/48/literature#literature.17). Эта модификация ГМП приводит к получению нового графа - *графа вызовов*, каждый узел в этом графе представляет модуль (процедуру), а каждая дуга - вызов модуля (процедуры). Для *процедурного программирования* подобный шаг упрощает графовую модель программного проекта до приемлемого уровня сложности. Таким образом, может быть определена цикломатическая сложность упрощенного графа модуля Modi как V'(Modi,C'), а громоздкая формула, выражающая сложность *интеграционного тестирования* программного проекта, принимает следующий вид [[ 19 ]](https://www.intuit.ru/studies/professional_retraining/941/courses/48/literature#literature.19):

V'(P,C1') = \Sigma  V'(Mod_{i}, C1') - k_{in} +k_{ext}

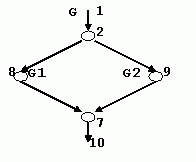
Так, для программы, ГМП которой приведена на [Рис. 4.2](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1432?page=2), для получения *графа вызовов* из иерархической модели проекта должны быть исключены все дуги, кроме:

1. Дуги 1-2, содержащей входной узел 1 графа G.
2. Дуг 2-8, 8-7, 7-10, содержащих вызов модуля G1.
3. Дуг 2-9, 9-7, 7-10, содержащих вызов модуля G2.

В результате *граф вызовов* примет вид, показанный на [Рис. 5.2](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1432?page=2#image.5.2), а сложность данного графа по критерию C1' равна:

V'(G,C1') = q + Kext =1+1=2.

V'(Modi,C') также называется в литературе сложностью модульного дизайна (complexity of module design) [[ 19 ]](https://www.intuit.ru/studies/professional_retraining/941/courses/48/literature#literature.19).



**Рис. 5.2.**Граф вызовов модулей

Сумма сложностей модульного дизайна для всех модулей по критерию С1 или сумма их аналогов для других критериев тестирования, исключая значения модулей самого нижнего уровня, дает сложность *интеграционного тестирования* для *процедурного программирования* [[ 20 ]](https://www.intuit.ru/studies/professional_retraining/941/courses/48/literature#literature.20).

**Вывод:**

В Ruby циклы явно существую в виде коменд while и forin, синтаксис которых такой же как в других языках программирования. Большей популярностью в Ruby пользуюся итераторы. Они позволяют писать компактный и понятный код.

**Вопросы для самоконтроля:**

1. Как работает цикл while?
2. Как работает цикл for?
3. Что делает команда sleep?
4. Как работает команда break?
5. Как выйти из бесконечного цикла?

# Лекция 6. Интеграционное тестирование и его особенности для объектно-ориентированного программирования

### Особенности интеграционного тестирования для объектно-ориентированного программирования

Программный проект, написанный в соответствии с объектно-ориентированным подходом, будет иметь ГМП, существенно отличающийся от ГМП традиционной "процедурной" программы. Сама разработка проекта строится по другому принципу - от определения классов, используемых в программе, построения дерева классов к реализации кода проекта. При правильном использовании классов, точно отражающих прикладную область приложения, этот метод дает более короткие, понятные и легко контролируемые программы.

Объектно-ориентированное *программное обеспечение* является событийно управляемым. Передача управления внутри программы осуществляется не только путем явного указания последовательности обращений одних функций программы к другим, но и путем генерации сообщений различным объектам, разбора сообщений соответствующим обработчиком и передача их объектам, для которых данные сообщения предназначены. Рассмотренная ГМП в данном случае становится неприменимой. Эта модель, как *минимум*, требует адаптации к требованиям, вводимым объектно-ориентированным подходом к написанию программного обеспечения. При этом происходит переход от модели, описывающей структуру программы, к модели, описывающей поведение программы, что для тестирования можно классифицировать как положительное свойство данного перехода. Отрицательным аспектом совершаемого перехода для применения рассмотренных ранее моделей является потеря заданных в явном виде связей между модулями программы.

Перед тем как приступить к описанию *графовой модели* объектно-ориентированной программы, остановимся отдельно на одном существенном аспекте разработки программного обеспечения на языке объектно-ориентированного программирования (*ООП*), например, C++ или С#. *Разработка программного обеспечения* высокого качества для MS *Windows* или любой другой операционной системы, использующей стандарт "*look* *and* *feel*", с применением только вновь созданных классов практически невозможна. Программист должен будет затратить массу времени на решение стандартных задач по созданию пользовательского интерфейса. Чтобы избежать работы над давно решенными вопросами, во всех современных компиляторах предусмотрены специальные библиотеки классов. Такие библиотеки включают в себя практически весь программный *интерфейс* операционной системы и позволяют задействовать при программировании средства более высокого уровня, чем просто вызовы функций. Базовые конструкции и классы могут быть переиспользованы при разработке нового программного проекта. За счет этого значительно сокращается время разработки приложений. В качестве примера подобной системы можно привести библиотеку Microsoft *Foundation* *Class* для компилятора MS *Visual* C++ [[ 21 ]](https://www.intuit.ru/studies/professional_retraining/941/courses/48/literature#literature.21)".

Работа по тестированию приложения не должна включать в себя проверку работоспособности элементов библиотек, ставших фактически промышленным стандартом для разработки программного обеспечения, а только проверку кода, написанного непосредственно разработчиком программного проекта. Тестирование объектно-ориентированной программы должно включать те же уровни, что и тестирование процедурной программы - модульное, интеграционное и системное. Внутри класса отдельно взятые методы имеют *императивный* характер исполнения. Все языки *ООП* возвращают *контроль* вызывающему объекту, когда сообщение обработано. Поэтому каждый метод (*функция* - член класса) должен пройти традиционное *модульное тестирование* по выбранному критерию C (как правило, С1 ). В соответствии с введенными выше обозначениями, назовем метод Modi, а сложность тестирования - V(Modi,C). Все результаты, полученные в лекции 5 для тестирования модулей, безусловно, подходят для тестирования методов классов. Каждый *класс* должен быть рассмотрен и как субъект *интеграционного тестирования*. *Интеграция* для всех методов класса проводится с использованием инкрементальной стратегии снизу вверх. При этом мы можем переиспользовать тесты для классов-родителей тестируемого класса [[ 22 ]](https://www.intuit.ru/studies/professional_retraining/941/courses/48/literature#literature.22), что следует из принципа наследования - от базовых классов, не имеющих родителей, к самым верхним уровням классов.

Графовая модель класса, как и объектно-ориентированной программы, на интеграционном уровне в качестве узлов использует методы. Дуги данной ГМП (вызовы методов) могут быть образованы двумя способами:

* Прямым вызовом одного метода из кода другого, в случае, если вызываемый метод виден (не закрыт для доступа средствами языка программирования) из класса, содержащего вызывающий метод, присвоим такой конструкции название ***Р-путь (P-path, Procedure path, процедурный путь)*** .
* Обработкой сообщения, когда явного вызова метода нет, но в результате работы "вызывающего" метода порождается сообщение, которое должно быть обработано "вызываемым" методом.

Для второго случая "вызываемый" метод может породить другое сообщение, что приводит к возникновению цепочки исполнения последовательности методов, связанных сообщениями. Подобная цепочка носит название *ММ-путь (MM-path, Metod/Message path, путь метод/сообщение)* [[ 23 ]](https://www.intuit.ru/studies/professional_retraining/941/courses/48/literature#literature.23). *ММ-путь* заканчивается, когда достигается метод, который при отработке не вырабатывает новых сообщений (т. е. вырабатывает "сообщение покоя").

Пример *ММ-путей* приведен на [рисунке 6.1](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1434?page=1#image.6.1). Данная конструкция отражает событийно управляемую природу объектно-ориентированного программирования и может быть взята в качестве основы для построения *графовой модели* класса или объектно-ориентированной программы в целом. На [рисунке 6.1](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1434?page=1#image.6.1) можно выделить четыре *ММ-пути* (1-4) и один *P-путь* (5):

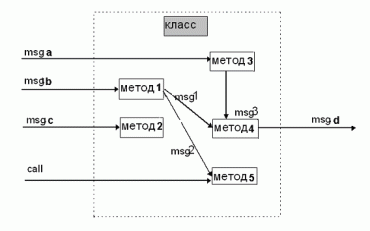
1. msg a \to метод 3 \to msg 3 \to метод 4 \to msg d
2. msg b \to метод 1 \to msg 1 \to метод 4 \to msg d
3. msg b \to метод 1 \to msg 2 \to метод 5
4. msg c \to метод 2
5. call \to метод 5

Здесь *класс* изображен как объединенное множество методов.

Введем следующие обозначения:

Kmsg - число методов класса, обрабатывающих различные сообщения;

Kem - число методов класса, которые не закрыты от прямого вызова из других классов программы.



**Рис. 6.1.**Пример MM-путей и P-путей в графовой модели класса

Если рассматривать *класс* как программу P, то можно выделить следующие отличия от программы, построенной по процедурному принципу:

* Значение Kext (число точек входа, которые могут быть вызваны извне) определяется как сумма методов - обработчиков сообщений Kmsg (например, в MS Visual C++ обозначаются зарезервированным словом afx\_msg и используются для работы с картой сообщений класса) и тех методов, которые могут быть вызваны из других классов программы Kem. Это определяется самим разработчиком путем разграничения доступа к методам класса (с помощью ключевых слов разграничения доступа public, private, protected ) при написании методов, а также назначении дружественных ( friend ) функций и дружественных классов. Таким образом, Kext = Kmsg + Kem, и имеет новый по сравнению с процедурным программированием физический смысл.
* Принцип соединения узлов в ГМП, отражающий два возможных типа вызовов методов класса (через *ММ-пути* и *Р-пути* ), что приводит к новому наполнению для множества М требуемых элементов.
* Методы (модули) непрозрачны для внешних объектов, что влечет за собой неприменимость механизма упрощения графа модуля, используемого для получения *графа вызовов* в *процедурном программировании*.

С учетом приведенных замечаний, информационные связи между модулями программного проекта получают новый физический смысл, а формула *оценки сложности интеграционного тестирования* класса Cls принимает вид: V(Cls, C) = f (Kmsg, Kem)

В ходе *интеграционного тестирования* должны быть проверены все возможные внешние вызовы методов класса, как непосредственные обращения, так и вызовы, инициированные получением сообщений

*Значение* числа *ММ-путей* зависит от схемы обработки сообщений данным классом, что должно быть определено в спецификации класса. Например, для класса, изображенного в [примере 5.4](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1434?page=2), сложность *интеграционного тестирования* V(Cls,C)=5 (множество неизбыточных тестов Т для класса составляют 4 *ММ-пути* плюс внешний вызов метода 5, т. е. *Р-путь* ).

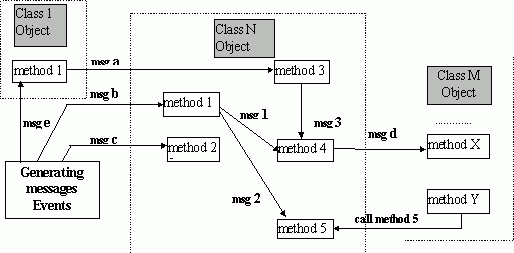
Данные - члены класса (данные, описанные в самом классе, и унаследованные от классов-родителей видимые извне данные) рассматриваются как "*глобальные переменные*", они должны быть протестированы отдельно на основе принципов тестирования потоков данных.

Когда *класс* программы P протестирован, *объект* данного класса может быть включен в общий *граф* G программного проекта, содержащий все *ММ-пути* и все вызовы методов классов и процедур, возможные в программе [рис. 6.2](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1434?page=2#image.6.2)

*Программа* P, содержащая n классов, имеет сложность *интеграционного тестирования* классов

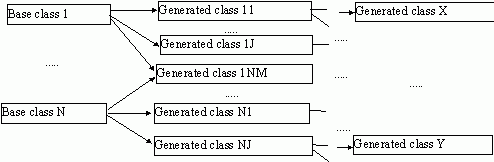
V(P, C) =\varSigma V(Cls_{i}, C)

Формальным представлением описанного выше подхода к тестированию программного проекта служит ***классовая модель программного проекта***, состоящая из дерева классов проекта [рис. 6.3](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1434?page=2#image.6.3) и модели каждого класса, входящего в программный проект [рис. 6.4](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1434?page=2#image.6.4).

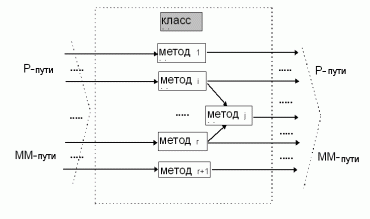


**Рис. 6.2.**Пример включения объекта в модель программного проекта, построенного с использованием MM-путей и P-путей

Таким образом и определяется *классовая модель* проекта для тестирования объектно-ориентированной программы. Как будет показано в дальнейшем, она поддерживает итерационный инкрементальный процесс разработки программного обеспечения.



**Рис. 6.3.**Дерево классов проекта



**Рис. 6.4.**Модель класса, входящего в программный проект

*Методика проведения тестирования программы*, представленной в виде *классовой модели* программного проекта, включает в себя несколько этапов, соответствующих уровням тестирования [рис. 6.5](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1434?page=2#image.6.5):

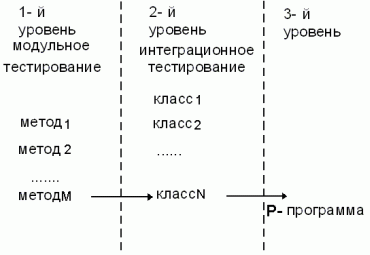
1. На первом уровне проводится тестирование методов каждого класса программы, что соответствует этапу модульного тестирования.
2. На втором уровне тестируются методы класса, которые образуют контекст *интеграционного тестирования* каждого класса.
3. На третьем уровне протестированный класс включается в общий контекст (дерево классов) программного проекта. Здесь становится возможным отслеживать реакцию программы на внешние события

Второй и третий уровни рассматриваемой модели соответствуют этапу *интеграционного тестирования*.

Для третьего уровня важным оказывается понятие атомарной системной функции (АСФ) [[ 23 ]](https://www.intuit.ru/studies/professional_retraining/941/courses/48/literature#literature.23). АСФ - это множество, состоящее из внешнего события на входе системы, реакции системы на это событие в виде одного или более *ММ-путей* и внешнего события на выходе системы. В общем случае внешнее выходное событие может быть нулевым, т. е. неаккуратно написанное *программное обеспечение* может не обеспечивать внешней реакции на действия пользователя. АСФ, состоящая из входного внешнего события, одного *ММ-пути* и выходного внешнего события, может быть взята в качестве модели для нити (*thread*). Тестирование подобной АСФ в рамках *классовой модели* ГМП реализуется довольно сложно, так как хотя динамическое взаимодействие нитей (потоков) в процессе исполнения естественно фиксируется в log-файлах, запоминающих результаты трассировки исполнения программ, оно же достаточно сложно отображается на классовой ГМП. Причина в том, что *классовая модель* ориентирована на *отображение* статических характеристик проекта, а в данном случае требуется *отображение* поведенческих характеристик. Как правило, тестирование взаимодействия нитей в ходе исполнения программного комплекса выносится на уровень *системного тестирования* и использует другие более приспособленные для описания поведения модели. Например, описание поведения программного комплекса средствами *языков спецификаций* *MSC*, SDL, *UML*.

Явный учет границ между интеграционным и системным уровнями тестирования дает преимущество при планировании *работ* на *фазе тестирования*, а возможность сочетать различные методы и критерии тестирования в ходе работы над программным проектом дает наилучшие результаты [[ 24 ]](https://www.intuit.ru/studies/professional_retraining/941/courses/48/literature#literature.24).

*Объектно-ориентированный подход*, ставший в настоящее время неявным стандартом разработки программных комплексов, позволяет широко использовать *иерархическую модель* программного проекта, приведенная на [рис. 6.5](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1434?page=2#image.6.5) схема иллюстрирует способ применения. Каждый *класс* рассматривается как *объект* модульного и *интеграционного тестирования*. Сначала каждый *метод класса* тестируется как *модуль* по выбранному критерию C. Затем *класс* становится объектом *интеграционного тестирования*. Далее осуществляется *интеграция* всех методов всех классов в единую структуру - *классовую модель* проекта, где в общую ГМП протестированные модули входят в виде узлов (интерфейсов вызова) без учета их внутренней структуры, а их детальные описания образуют *контекст* всего программного проекта.



**Рис. 6.5.**Уровни тестирования классовой модели программного проекта

Сама технология объектно-ориентированного программирования (одним из определяющих принципов которой является *инкапсуляция* с возможностью ограничения доступа к данным и методам - членам класса) позволяет применить подобную трактовку вхождения модулей в общую ГМП. При этом тесты для отдельно рассмотренных классов переиспользуются, входя в общий набор тестов для программы P.

### Пример интеграционного тестирования

Продемонстрируем тестирование взаимодействий на примере взаимодействия класса TCommandQueue и класса TСommand, а также, как и при модульном теcтировании, разработаем спецификацию тестового случая [таблица 6.2](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1434?page=3#table.6.2):

|  |  |
| --- | --- |
| Таблица 6.2. Спецификация тестового случая для *интеграционного тестирования* | |
| **Названия взаимодействующих классов**:TСommandQueue, TCommand | **Название теста**:TCommandQueueTest1 |
| **Описание теста**:тест проверяет возможность создания объекта типа TCommand и добавления его в очередь при вызове метода AddCommand | |
| **Начальные условия**:очередь команд пуста | |
| **Ожидаемый результат**:в очередь будет добавлена одна команда | |

На основе этой спецификации разработан тестовый *драйвер* [пример 6.1](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1434?page=3#example.6.1) - *класс* TCommandQueueTester, который наследуется от класса *Tester*.

*Класс* содержит:

* конструктор, в котором создаются объекты классов TStore, TTerminalBearing и объект типа TCommandQueue
* Методы, реализующие тесты. Каждый тест реализован в отдельном методе.
* Метод Run, в котором вызываются методы тестов.
* Метод dump, который сохраняет в Log-файле теста информацию обо всех командах, находящихся в очереди в формате - Номер позиции в очереди: полное название команды
* Точку входа в программу - метод Main, в котором происходит создание экземпляра класса TСommandQueueTester.

public TCommandQueueTester()

{

TB = new TTerminalBearing();

S = new TStore();

CommandQueue=new TCommandQueue(S,TB);

S.CommandQueue=CommandQueue;

...

}

6.1. Объект типа TCommandQueue

TCommandQueueTester::TCommandQueueTester()

{

TB = new TTerminalBearing();

S = new TStore();

CommandQueue=new TCommandQueue(S,TB);

S->CommandQueue=CommandQueue;

}

6.1.1. Объект типа TcommandQueue (C++)

Теперь создадим тест, который проверяет, создается ли *объект* типа TСommand, и добавляется ли *команда* в конец очереди.

private void TCommandQueueTest1()

{

LogMessage("///// TCommandQueue Test1 /////");

LogMessage("Проверяем, создается ли

объект типа TCommand");

// В очереди нет команд

dump();

// Добавляем команду

// параметр = -1 означает, что команда

// должна быть добавлена в конец очереди

CommandQueue.AddCommand(TCommand.GetR,0,0,0,

new TBearingParam(),new TAxleParam(),-1);

LogMessage("Command added");

// В очереди одна команда

dump();

}

6.2. Тест

void TCommandQueueTester::TCommandQueueTest1()

{

LogMessage("///// TCommandQueue Test1 /////");

LogMessage("Проверяем, создается ли

объект типа TCommand");

// В очереди нет команд

dump();

// Добавляем команду

// параметр = -1 означает, что команда

// должна быть добавлена в конец очереди

CommandQueue.AddCommand(GetR,0,0,0,

new TBearingParam(),

new TAxleParam(),-1);

LogMessage("Command added");

// В очереди одна команда

dump();

}

6.2.1. Тест (C++)

В *класс* включены еще два разработанных теста.

После завершения теста следует просмотреть текстовый журнал теста, чтобы сравнить полученные результаты с ожидаемыми результатами, заданными в спецификации тестового случая TCommandQueueTest1 [пример 6.3](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1434?page=3#example.6.3).

///// TCommandQueue Test1 /////

Проверяем, создается ли объект типа TCommand

0 commands in command queue

Command added

1 commands in command queue

0: ПОЛУЧИТЬ ИЗ ВХОДНОЙ ЯЧЕЙКИ

6.3. Спецификация результатов теста

**Выводы по теме:**

Массивы представляют собой хэши, в которых ключи представлены целыми числами, а значения могут быть любыми типами данных. Индексы массива начинаются от нуля. Доступ к элементам массива осуществляется с помощью указания названия массива и его индекса.

**Вопросы для самоконтроля:**

1. С какого элемента начинает нумерация в массиве?
2. К какому классу относится массив?
3. Как осуществляется доступ к элементам массива?
4. Как добавить элементы в массив?
5. Как удалить элементы из массива?

# Лекция 7. Разновидности тестирования: системное и регрессионное тестирование

### Системное тестирование

*Системное тестирование* качественно отличается от интеграционного и модульного уровней. *Системное тестирование* рассматривает тестируемую систему в целом и оперирует на уровне пользовательских интерфейсов, в отличие от последних фаз *интеграционного тестирования*, которое оперирует на уровне интерфейсов модулей. Различны и цели этих уровней тестирования. На уровне системы часто сложно и малоэффективно анализировать прохождение тестовых траекторий внутри программы или отслеживать правильность работы конкретных функций. Основная *задача системного тестирования* - в выявлении дефектов, связанных с работой системы в целом, таких как неверное использование ресурсов системы, непредусмотренные комбинации данных пользовательского уровня, несовместимость с окружением, непредусмотренные сценарии использования, отсутствующая или неверная функциональность, неудобство в применении и тому подобное.

*Системное тестирование* производится над проектом в целом с помощью метода "черного ящика". Структура программы не имеет никакого значения, для проверки доступны только входы и выходы, видимые пользователю. Тестированию подлежат коды и пользовательская документация.

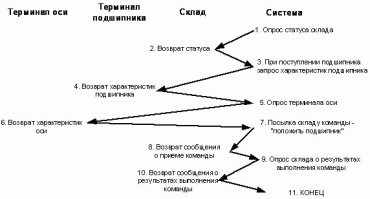
*Категории тестов* *системного тестирования*:

1. Полнота решения функциональных задач.
2. *Стрессовое тестирование* - на предельных объемах нагрузки входного потока.
3. Корректность использования ресурсов (утечка памяти, возврат ресурсов).
4. Оценка производительности.
5. Эффективность защиты от искажения данных и некорректных действий.
6. Проверка инсталляции и конфигурации на разных платформах.
7. Корректность документации

Поскольку *системное тестирование* проводится на пользовательских интерфейсах, создается иллюзия того, что построение специальной системы *автоматизации тестирования* не всегда необходимо. Однако объемы данных на этом уровне таковы, что обычно более эффективным подходом является полная или частичная *автоматизация тестирования*, что приводит к созданию тестовой системы гораздо более сложной, чем *система тестирования*, применяемая на уровне тестирования модулей или их комбинаций.

#### Пример системного тестирования приложения "Поступление подшипника на склад"

В спецификации тестового случая задано состояние окружения (входные данные) и ожидаемая последовательность событий в системе (ожидаемый результат). После прогона тестового случая мы получаем реальную последовательность событий в системе ([пример 7.1](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1436?page=1#example.7.1),[пример 7.3](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1436?page=1#example.7.3)) при заданном состоянии окружения. Сравнивая фактический результат с ожидаемым, можно сделать вывод о том, прошла или не прошла тестируемая система испытание на заданном тестовом случае. В качестве ожидаемого результата будем использовать спецификацию тестового случая, поскольку она определяет, как, для заданного состояния окружения, система должна функционировать.



**Рис. 4-15.**Краткое описание тестируемой системы 'Поступление подшипника на склад'

Спецификация тестового случая №1:

Состояние окружения (входные данные - X ):

Статус склада - 32. Пришел подшипник.

Статус обмена с терминалом подшипника (0 - есть подшипник) и его параметры - "Статус=0 Диаметр=12".

Статус обмена с терминалом оси (1 - нет оси) и ее параметры - "Статус=1 Диаметр=12".

"Статус=1 Диаметр=12".

Статус команды - 0. Команда успешно принята.

Сообщение от склада - 1. Команда успешно выполнена.

Ожидаемая последовательность событий (выходные данные – Y):

Система запрашивает статус склада (вызов функции GetStoreStat ) и получает 32

Система запрашивает параметры подшипника (вызов функции GetRollerPar ) и получает Статус = 0 Диаметр=12

Система запрашивает параметры оси (вызов функции GetAxlePar ) и получает Статус = 1 Диаметр=0

Система добавляет в очередь команд склада на последнее место команду SendR (получить из приемника в ячейку) (вызов функции SendStoreCom ) и получает сообщение о том, что команда успешно принята – статус = 0

Система запрашивает склад о результатах выполнения команды (вызов функции GetStoreMessage ) и получает сообщение о том, что команда успешно выполнена - статус = 1

Выходные данные (результаты выполнения Yв) – зафиксированы в журнале теста ([пример 7.1](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1436?page=1#example.7.1))

ВЫЗОВ: GetStoreStat

РЕЗУЛЬТАТ: 32

ВЫЗОВ: GetRollerPar

РЕЗУЛЬТАТ: Статус = 0 Диаметр = 12

ВЫЗОВ: GetAxlePar

РЕЗУЛЬТАТ: Статус = 1 Диаметр = 0

ВЫЗОВ: SendStoreCom

РЕЗУЛЬТАТ: 0

ВЫЗОВ: GetStoreMessage

РЕЗУЛЬТАТ: 1

7.1. Журнал теста

Приведенный на [примере 7.2](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1436?page=1#example.7.2)тест был разработан в соответствии со спецификацией тестового случая №1. Детальная спецификация приведена в FS (Практикум, Приложение 1), результаты прогона показаны на [примере 7.3](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1436?page=1#example.7.3).

class Test1:Test {

override public void start()

{

// Задаем состояние окружения

// (входные данные)

StoreStat="32"; //Поступил подшипник

RollerPar="0 NewUser Depot1 123456 1 12 1 1";

// статус обмена с терминалом подшипника

// (0 - есть подшипник) и его параметры

AxlePar="1 NewUser Depot1 123456 1 0 12 12";

// статус обмена с терминалом оси

// (1 - нет оси) и ее параметры

CommandStatus="0";

// команда успешно принята

StoreMessage="1"; // успешно выполнена

// Получаем информацию о функционировании

// системы

wait("GetStoreStat");

//опрос статуса склада

wait("GetRollerPar");

// Получение информации о подшипнике

// с терминала подшипника

wait("GetAxlePar");

// Получение информации об оси

// с терминала оси

wait("SendStoreCom");

// добавление в очередь команд склада

// на первое место команды GetR

// (получить из приемника в ячейку)

wait("GetStoreMessage");

// Получение сообщения от склада о

// результатах выполнения команды

// В результате первый подшипник

// должен быть принят

}

}

7.2. Тест для системного тестирования

class Test1 : public Test {

public:

void start() {

// Задаем состояние окружения

// (входные данные)

// Поступил подшипник

strcpy(StoreStat,"32");

// статус обмена с терминалом подшипника

// (0 - есть подшипник) и его параметры

strcpy(RollerPar, "0 NewUser Depot1 123456 1 12 1 1");

// статус обмена с терминалом оси

// (1 - нет оси) и ее параметры

strcpy(AxlePar, "1 NewUser Depot1 123456 1 0 12 12");

strcpy(CommandStatus,"0");

//команда успешно принята

strcpy(StoreMessage,"1");

//успешно выполнена

// Получаем информацию о

// функционировании системы

wait("GetStoreStat");

//опрос статуса склада

wait("GetRollerPar");

// Получение информации о подшипнике

// с терминала подшипника

wait("GetAxlePar");

// Получение информации об оси

// с терминала оси

wait("SendStoreCom");

// добавление в очередь команд склада

// на первое место команды GetR

// (получить из приемника в ячейку)

wait("GetStoreMessage");

// Получение сообщения от склада о

// результатах выполнения команды

// В результате первый подшипник

// должен быть принят

}

}

7.2.1. Тест для системного тестирования (C++)

После завершения теста следует просмотреть текстовый журнал теста, чтобы выяснить, какая последовательность событий в системе была реально зафиксирована (выходные данные) и сравнить их с ожидаемыми результатами, заданными в спецификации тестового случая1. Пример журнала теста ([пример 7.1](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1436?page=1#example.7.1)):

Test started

CALL:GetStoreStat 0

RETURN:32

CALL:GetRollerPar

RETURN:0 NewUser Depot1 123456 1 12 1 1

CALL:GetAxlePar

RETURN:1 NewUser Depot1 123456 1 0 12 12

CALL:SendStoreCom 1 0 0 1 0 0 0

RETURN:0

CALL:GetStoreMessage

RETURN:1

7.3. Тестовый журнал для случая прогона системного теста

### Регрессионное тестирование

*Регрессионное тестирование* - цикл тестирования, который производится при внесении изменений на фазе *системного тестирования* или сопровождения продукта. Главная проблема *регрессионного тестирования* - выбор между полным и частичным *перетестированием* и пополнением тестовых наборов. При частичном *перетестировании* контролируются только те части проекта, которые связаны с измененными компонентами. На ГМП это пути, содержащие измененные узлы, и, как правило, это методы и классы, лежащие выше модифицированных по уровню, но содержащие их в своем контексте

Пропуск огромного объема тестов, характерного для этапа *системного тестирования*, удается осуществить без потери качественных показателей продукта только с помощью регрессионного подхода.

#### Пример регрессионного тестирования

Получив отчет об ошибке, программист анализирует исходный код, находит ошибку, исправляет ее и модульно или интеграционно тестирует результат.

В свою очередь тестировщик, проверяя внесенные программистом изменения, должен:

* Проверить и утвердить исправление ошибки. Для этого необходимо выполнить указанный в отчете тест, с помощью которого была найдена ошибка.
* Попробовать воспроизвести ошибку каким-нибудь другим способом.
* Протестировать последствия исправлений. Возможно, что внесенные исправления привнесли ошибку (наведенную ошибку) в код, который до этого исправно работал.

Например, при тестировании класса TСommandQueue запускаем тесты ([пример 7.2](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1436?page=1#example.7.2)):

// Тест проверяет, создается ли объект

// типа TCommand и добавляется ли он

// в конец очереди.

private void TCommandQueueTest1()

// Тест проверяет добавление команд

// в очередь на указанную позицию.

// Также проверяется правильность

// удаления команд из очереди.

private void TCommandQueueTest2()

7.4. Набор тестов класса TСommandQueue

// Тест проверяет, создается ли объект

// типа TCommand и добавляется ли он

// в конец очереди.

void TCommandQueueTest1()

// Тест проверяет добавление команд

// в очередь на указанную позицию.

// Также проверяется правильность

// удаления команд из очереди.

void TCommandQueueTest2()

7.4.1. Набор тестов класса TСommandQueue (C++)

При этом первый тест выполняется успешно, а второй нет, т.е. команда добавляется в конец очереди команд успешно, а на указанную позицию - нет. Разработчик анализирует код, который реализует тестируемую функциональность:

...

if ((Position<-1)&&

(Position<=this.Items.Count))

{

this.Items.Insert(Position, Command);

}

else

{

if (Position==-1)

{

this.Items.Add(Command);

}

}

7.5. Фрагмент кода с зафиксированным при тестировании дефектом

if ((Position <-1)&&(Position<=this.Items.Count))

{

this.Items.Insert(Position, Command);

}

else

{

if (Position==-1)

{

this.Items.Add(Command);

}

}

7.5.1. Фрагмент кода с зафиксированным при тестировании дефектом

Анализ показывает, что ошибка заключается в использовании неверного знака сравнения в первой строке фрагмента. Далее программист исправляет ошибку, например следующим образом:

...

if ((Position>=-1)&&

(Position<=this.Items.Count))

{

this.Items.Insert(Position, Command);

}

else

{

if (Position==-1)

{

this.Items.Add(Command);

}

}

...

7.6. Исправленный фрагмент кода

if ((Position>=-1)&&

(Position<=this.Items.Count))

{

this.Items.Insert(Position, Command);

}

else

{

if (Position==-1)

{

this.Items.Add(Command);

}

}

7.6.1. Исправленный фрагмент кода

Для проверки скорректированного кода хочется пропустить только тест TCommandQueueTest2. Можно убедиться, что тест TCommandQueueTest2 будет выполняться успешно. Однако одной этой проверки недостаточно. Если мы повторим пропуск двух тестов, то при запуске первого теста, TCommandQueueTest1, будет обнаружен новый дефект. Повторный анализ кода показывает, что ветка else не выполняется. Таким образом, исправление в одном месте привело к ошибке в другом, что демонстрирует необходимость проведения полного *перетестирования*. Однако повторное *перетестирование* требует значительных усилий и времени. Возникает задача – отобрать сокращенный набор тестов из исходного набора (может быть, пополнив его рядом дополнительных - вновь разработанных - тестов), которого, тем не менее, будет достаточно для исчерпывающей проверки функциональности в соответствии с выбранным критерием. Организация повторного тестирования в условиях сокращения ресурсов, необходимых для обеспечения заданного уровня качества продукта, обеспечивается *регрессионным тестированием*.

### Комбинирование уровней тестирования

В каждом конкретном проекте должны быть определены задачи, ресурсы и технологии для каждого уровня тестирования таким образом, чтобы каждый из типов дефектов, ожидаемых в системе, был "адресован", то есть в общем наборе тестов должны иметься тесты, направленные на выявление дефектов подобного типа. [Табл. 4.3](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1436?page=3#table.4.3) суммирует характеристики свойств модульного, интеграционного и системного уровней тестирования. Задача, которая стоит перед тестировщиками и менеджерами, заключается в оптимальном распределении ресурсов между всеми тремя типами тестирования. Например, перенесение усилий на *поиск* фиксированного типа дефектов из области системного в область *модульного тестирования* может существенно снизить сложность и *стоимость* всего процесса тестирования.

|  |  |  |  |
| --- | --- | --- | --- |
| Таблица 4.3. Характеристики модульного, интеграционного и *системного тестирования* | | | |
|  | **Модульное** | **Интеграционное** | **Системное** |
| **Типы дефектов** | Локальные дефекты, такие как опечатки в реализации алгоритма, неверные операции, логические и математические выражения, циклы, ошибки в использовании локальных ресурсов, рекурсия и т.п. | Интерфейсные дефекты, такие как неверная трактовка параметров и их формат, неверное использование системных ресурсов и средств коммуникации, и т.п. | Отсутствующая или некорректная функциональность, неудобство использования, непредусмотренные данные и их комбинации, непредусмотренные или неподдерживаемые сценарии работы, ошибки совместимости, ошибки пользовательской документации, ошибки переносимости продукта на различные платформы, проблемы производительности, инсталляции и т.п. |
| **Необходимость в системе тестирования** | Да | Да | Нет (\*) |
| **Цена разработки системы тестирования** | Низкая | Низкая до умеренной | Умеренная до высокой или неприемлемой |
| **Цена процесса тестирования, то есть разработки, прогона и анализа тестов** | Низкая | Низкая | Высокая |

(\*) *прямой* необходимости в системе тестирования нет, но цена процесса *системного тестирования* часто настолько высока, что требует использования систем автоматизации, несмотря на возможно высокую их *стоимость*.

**Выводы по теме:**

Массивы представляют собой хэши, в которых ключи представлены целыми числами, а значения могут быть любыми типами данных. Индексы массива начинаются от нуля. Доступ к элементам массива осуществляется с помощью указания названия массива и его индекса.

**Вопросы для самоконтроля:**

1. С какого элемента начинает нумерация в массиве?
2. К какому классу относится массив?
3. Как осуществляется доступ к элементам массива?
4. Как добавить элементы в массив?
5. Как удалить элементы из массива?

# Лекция 8. Автоматизация тестирования

### Автоматизация тестирования

Использование различных подходов к тестированию определяется их эффективностью применительно к условиям, определяемым промышленным проектом. В реальных случаях работа группы тестирования планируется так, чтобы разработка тестов начиналась с момента согласования требований к программному продукту (выпуск Requirement *Book*, содержащей высокоуровневые требования к продукту) и продолжалась параллельно с разработкой дизайна и кода продукта. В результате, к началу системного тестирования создаются тестовые наборы, содержащие тысячи тестов. Большой набор тестов обеспечивает всестороннюю проверку функциональности продукта и гарантирует качество продукта, но пропуск такого количества тестов на этапе системного тестирования представляет проблему. Ее решение лежит в области *автоматизации тестирования*, т.е. в автоматизации разработки.

Структура программы P теста

Загрузка теста (X,Y\*)

Запуск тестируемого модуля

Cравнение полученных результатов Y c эталонными Y\*

Структура тестируемого комплекса

ModF <- МоdF1

МоdF2

МоdF3 <- МоdF31

МоdF32

Структура тестирующего модуля

Mod TestModF:

Mod TestМодF1

Mоd TestМодF2

Mоd TestМодF3

P TestМодF

Mоd TestModF1:

P TestМодF1

Mоd TestModF2:

P TestМодF2

Mоd TestModF3:

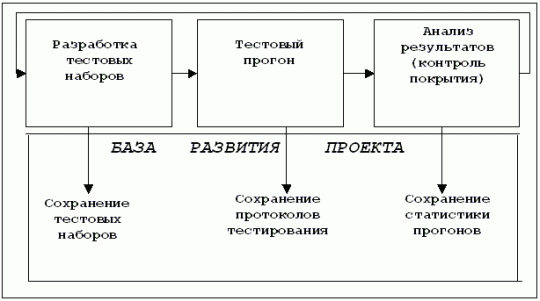
Mod TestМодF31

Mоd TestМодF32

P TestМодF3

В этом примере приведены структура теста, структура тестируемого комплекса и структура тестирующего модуля. Особенностью структуры каждого из тестирующих модулей Mi является *запуск* тестирующей программы Pi после того как каждый из модулей Mij, входящих в *контекст* модуля Mi, оттестирован. В этом случае *запуск* тестирующего модуля обеспечивает рекурсивный спуск к программам тестирования модулей нижнего уровня, а затем исполняет тестирование вышележащих уровней в условиях оттестированности нижележащих. Тестовые наборы подобной структуры ориентированы на автоматическое управление пропуском тестового набора в тестовом цикле. Важным преимуществом подобной организации является возможность регулирования нижнего уровня, до которого следует доходить в цикле тестирования. В этом случае *контекст* редуцированных в конкретном тестовом цикле модулей помечается как базовый, не подлежащий тестированию. Например, если *контекст* модуля ModF3: (ModF31, ModF32) – помечен как базовый, то в результате рекурсивный спуск затронет лишь модули ModF1, ModF2, ModF3 и вышележащий *модуль* ModF. Описанный способ организации тестовых наборов с успехом применяется в системах *автоматизации тестирования*.

Собственно использование эффективной системы *автоматизации тестирования* сокращает до минимума (например, до одной ночи) время пропуска тестов, без которого невозможно подтвердить факт роста качества (уменьшения числа оставшихся ошибок) продукта. *Системное тестирование* осуществляется в рамках циклов тестирования (периодов пропуска разработанного тестового набора над *build* разрабатываемого приложения). Перед каждым циклом фиксируется разработанный или исправленный *build*, на который заносятся обнаруженные в результате тестового прогона ошибки. Затем ошибки исправляются, и на очередной цикл тестирования предъявляется новый *build*. Окончание тестирования совпадает с экспериментально подтвержденным заключением о достигнутом уровне качества относительно выбранного критерия тестирования или о снижении плотности не обнаруженных ошибок до некоторой заранее оговоренной величины. Возможность ограничить цикл тестирования пределом в одни сутки или несколько часов поддерживается исключительно за счет средств *автоматизации тестирования*.



**Рис. 8.1.**Структура инструментальной системы автоматизации тестирования

На [рис. 8.1](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1438?page=1#image.8.1) представлена обобщенная *структура системы* *автоматизации тестирования*, в которой создается и сохраняется следующая *информация*:

* Набор тестов, достаточный для покрытия тестируемого приложения в соответствии с выбранным критерием тестирования – как результат ручной или автоматической разработки (генерации) тестовых наборов и драйвер/монитор пропуска тестового набора.
* Результаты прогона тестового набора, зафиксированные в *Log-файле*. *Log-файл* содержит трассы ("протоколы"), представляющие собой реализованные при тестовом прогоне последовательности некоторых событий (значений отдельных переменных или их совокупностей) и точки реализации этих событий на графе программы. В составе трасс могут присутствовать последовательности явно и неявно заданных меток, задающих пути реализации трасс на управляющем графе программы, совокупности значений переменных на этих метках, величины промежуточных результатов, достигнутых на некоторых метках и т.п.
* Статистика тестового цикла, содержащая: 1) результаты пропуска каждого теста из тестового набора и их сравнения с эталонными величинами; 2) факты, послужившие основанием для принятия решения о продолжении или окончании тестирования; 3) критерий покрытия и степень его удовлетворения, достигнутая в цикле тестирования.

Результатом анализа каждого прогона является *список* проблем, в виде ошибок и дефектов, который заносится в базу развития проекта. Далее происходит работа над ошибками, где каждая поднятая проблема идентифицируется, относится к соответствующему модулю и разработчику, приоритезируется и отслеживается, что обеспечивает гарантию ее решения (исправления или отнесения к списку известных проблем, решение которых по тем или иным причинам откладывается) в последующих *build*. Исправленный и собранный для тестирования *build* поступает на следующий цикл тестирования, и цикл повторяется, пока нужное качество программного комплекса не будет достигнуто. В этом итерационном процессе *средства автоматизации* тестирования обеспечивают быстрый *контроль* результатов исправления ошибок и проверку уровня качества, достигнутого в продукте. Некачественный продукт зрелая организация не производит.

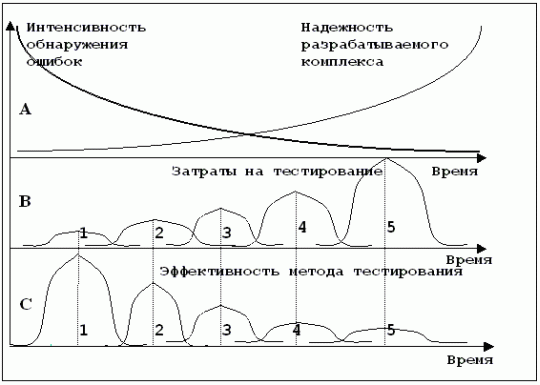
### Издержки тестирования

Интенсивность обнаружения ошибок на единицу затрат и *надежность* тесно связаны со временем тестирования и, соответственно, с гарантией качества продукта ([рис. 8.2A](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1438?page=2#image.8.2) ). Чем больше трудозатрат вкладывается в процесс тестирования, тем меньше ошибок в продукте остается незамеченными. Однако совершенство в индустриальном программировании имеет пределы, которые прежде всего связаны с затратами на получение программного продукта, а также с избытком качества, которое не востребовано заказчиком приложения. Нахождение оптимума – очень ответственная задача тестировщика и менеджера проекта.

Движение к уменьшению числа оставшихся ошибок или к качеству продукта приводит к применению различных методов отладки и тестирования в процессе создания продукта. На [рис.8.2В](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1438?page=2#image.8.2) приведен затратный *компонент* тестирования в зависимости от совершенствования применяемого инструментария и методов тестирования. На практике популярны следующие методы тестирования и отладки, упорядоченные по связанным с их применением затратам:

* Статические методы тестирования
* Модульное тестирование
* Интеграционное тестирование
* Системное тестирование
* Тестирование реального окружения и реального времени.

Зависимость эффективности применения перечисленных методов или их способности к обнаружению соответствующих классов ошибок (С) сопоставлена на [рис. 8.2](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1438?page=2#image.8.2) с затратами (B). *График* показывает, что со временем, по мере обнаружения более сложных ошибок и дефектов, эффективность низкозатратных методов падает вместе с количеством обнаруживаемых ошибок.



**Рис. 8.2.**Издержки тестирования

Отсюда следует, что все методы тестирования не только имеют право на существование, но и имеют свою нишу, где они хорошо обнаруживают ошибки, тогда как вне ниши их эффективность падает. Поэтому необходимо совмещать различные методы и стратегии отладки и тестирования с целью обеспечения запланированного качества программного продукта при ограниченных затратах, что достижимо при использовании процесса управления качеством программного продукта.

**Выводы:**

Для инкапсуляции кода в Ruby существуют методы. Это специальные конструкции def end, с помощью которых можно упростить код. По умолчанию метод возвращает последнее значение в теле. Чтобы возвратить конкретный объект следует использовать команду return

**Вопросы для самоконтроля:**

1. Что такое метод?
2. Что такое формальные переменные?
3. Что такое фактические перемнные?
4. Как можно возвратить значения из метода?
5. Могут ли методы вкладываться друг в друга?

# Лекция 9. Особенности индустриального тестирования

### Качество программного продукта и тестирование

*Качество программного продукта* можно оценить некоторым набором характеристик, определяющих, насколько продукт "хорош" с точки зрения всех потенциально заинтересованных в нем сторон. Такими сторонами являются:

* заказчик продукта
* спонсор
* конечный пользователь
* разработчики продукта
* тестировщики продукта
* инженеры поддержки
* отдел обучения
* отдел продаж и т.п.

Каждый из участников может иметь различное *представление* о продукте и по-разному судить о том, насколько он хорош или плох, то есть насколько высоко качество продукта. С точки зрения разработчика, продукт может быть настолько хорош, насколько хороши заложенные в нем алгоритмы и технологии. Пользователю продукта, скорее всего, безразличны детали внутренней реализации, его в первую *очередь* волнуют вопросы функциональности и надежности. Спонсора интересует цена и совместимость с будущими технологиями. Таким образом, задача обеспечения качества продукта выливается в задачу определения заинтересованных лиц, согласования их критериев качества и нахождения оптимального решения, удовлетворяющего этим критериям.

В рамках подобной задачи *группа* тестирования рассматривается не просто как еще одна заинтересованная сторона, но и как сторона, способная оценить удовлетворение выбранных критериев и сделать *вывод* о качестве продукта с точки зрения других участников. К сожалению, далеко не все критерии могут быть оценены группой тестирования. Поэтому ее внимание в основном сосредоточено на критериях, определяющих *качество программного продукта* с точки зрения конечного пользователя.

**Тестирование как способ обеспечения качества**. Тестирование, с технической точки зрения, есть процесс выполнения приложения на некоторых входных данных и проверка получаемых результатов с целью подтвердить их *корректность* по отношению к результату.

Тестирование не позиционируется в качестве единственного способа обеспечения качества. Оно является частью общей системы обеспечения качества продукта, элементы которой выбираются по критерию наибольшей эффективности применения в конкретном проекте.

Рассмотрим пример. В качестве приложения возьмем программу для работы с сетью (*browser*), критерии качества которой приведены в [Табл.9.1](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1440?page=1#table.9.1).

|  |  |  |  |
| --- | --- | --- | --- |
| Таблица 9.1. Критерии качества программы browser | | | |
|  | **Пользователь** | **Заказчик** | **Инженер поддержки** |
| ***Функциональная полнота*** | + | - | - |
| **Цена разработки** | - | + | - |
| **Отсутствие дефектов** | + | Косвенно | + |
| **Удобство использования** | + | - | - |
| **Возможность внесения изменений в будущем** | - | Косвенно | + |
| **Легкость исправления дефектов** | - | - | + |
| **Документация на реализацию, в том числе комментарии** | - | - | + |
| **Своевременность исполнения проекта** | - | + | - |

*Матрица* критериев качества заинтересованных в них участников для рассматриваемого проекта приведена в [таблице 9.2](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1440?page=1#table.9.2) . Допустим, что вид матрицы критериев качества и проверяющих элементов системы обеспечения качества для данного проекта будет следующим:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Таблица 9.2. Матрица критериев качества и элементов системы обеспечения качества | | | | | |
|  | **Тестирование** | **Анализ рынка и специальные лаборатории1** | **Обзоры кода** | **Анализ дизайна** | **Аудиты процесса разработки** |
| **Полнота функциональности** | +, не всегда эффективно | + | - | - | - |
| **Стоимость разработки** | - | - | - | - | + |
| **Отсутствие дефектов** | + | - | + | - | - |
| **Удобство использования** | +, не всегда эффективно | + | - | - | - |
| **Возможность внесения изменений в будущем** | - | - | +- | + | - |
| **Легкость исправления дефектов** | - | - | + | + | - |
| **Документация на реализацию, в том числе комментарии** | - | - | + | - | + |
| **Своевременность исполнения проекта** | - | - | - | - | + |

Данные ([Табл. 9.1](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1440?page=1#table.9.1), [Табл. 9.2](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1440?page=1#table.9.2)) показывают, что из восьми элементов общего качества продукта тестирование способно оценить и контролировать только три (1, 3, 4), причем наиболее эффективно тестирование контролирует отсутствие дефектов (3).

В каждом конкретном проекте элементы системы должны быть выбраны так, чтобы обеспечить приемлемое качество, исходя из приоритетов и имеющихся ресурсов. Выбирая элементы для системы обеспечения качества конкретного продукта, можно применить комбинированное тестирование, обзоры кода, *аудит*. При подобном выборе некоторые качества, например легкость модификации и исправления дефектов, не будут оценены и, возможно, выполнены. Задачей тестирования в рассматриваемом случае будет обнаружение дефектов и оценка удобства использования продукта, включая полноту функциональности. Исходя из задач, поставленных перед группой тестирования в конкретном проекте, выбирается соответствующая стратегия тестирования. Так, в данном примере, ввиду необходимости оценить *удобство использования* и полноту функциональности, преимущественный подход к разработке тестов следует планировать на основе использования сценариев.

Итак, основная последовательность действий при выборе и оценке критериев *качества программного продукта* включает:

1. Определение всех лиц, так или иначе заинтересованных в исполнении и результатах данного проекта.
2. Определение критериев, формирующих представление о качестве для каждого из участников.
3. Приоритезацию критериев, с учетом важности конкретного участника для компании, выполняющей проект, и важности каждого из критериев для данного участника.
4. Определение набора критериев, которые будут отслежены и выполнены в рамках проекта, исходя из приоритетов и возможностей проектной команды. Постановка целей по каждому из критериев.
5. Определение способов и механизмов достижения каждого критерия.
6. Определение стратегии тестирования исходя из набора критериев, попадающих под ответственность группы тестирования, выбранных приоритетов и целей

### Процесс тестирования

Как отмечалось в подразделе 2.4, в тестировании выделяются три основных уровня, или три фазы:

1. *Модульное тестирование*.
2. *Интеграционное тестирование*.
3. *Системное тестирование*.

Задача планирования активности тестирования состоит в оптимальном распределении ресурсов между всеми типами тестирования. В дальнейшем изложении мы сконцентрируемся на системной *фазе тестирования*, как на наиболее важной и критичной активности для разработки качественного программного продукта.

#### Фазы процесса тестирования

В процессе тестирования выделяют следующие фазы:

1. **Определение целей** (требований к тестированию), включающее следующую конкретизацию: какие части системы будут тестироваться, какие аспекты их работы будут выбраны для проверки, каково желаемое качество и т.п.
2. **Планирование**: создание графика (расписания) разработки тестов для каждой тестируемой подсистемы; оценка необходимых человеческих, программных и аппаратных ресурсов; разработка расписания *тестовых циклов*. Важно отметить, что расписание тестирования обязательно должно быть согласовано с расписанием разработки создаваемой системы, поскольку наличие исполняемой версии разрабатываемой системы ( **Implementation Under Testing (IUT)** или **Application Under Testing (AUT)** – часто употребляемые обозначения для тестируемой системы) является одним из необходимых условий тестирования, что создает взаимозависимость в работе команд тестировщиков и разработчиков.
3. **Разработка тестов,** то есть тестового кода для тестируемой системы, если необходимо - кода системы *автоматизации тестирования* и *тестовых процедур* (выполняемых вручную).
4. **Выполнение тестов**: реализация *тестовых циклов*.
5. **Анализ результатов**.

После анализа результатов возможно повторение процесса тестирования, начиная с пунктов 3, 2 или даже 1.

#### Тестовый цикл

***Тестовый цикл*** – это цикл исполнения тестов, включающий фазы 4 и 5 тестового процесса. *Тестовый цикл* заключается в прогоне разработанных тестов на некотором однозначно определяемом срезе системы (состоянии кода разрабатываемой системы). Обычно такой срез системы называют **build**. *Тестовый цикл* включает следующую последовательность действий:

1. Проверка готовности системы и тестов к проведению *тестового цикла* включающая:
   * Проверку того, что все тесты, запланированные для исполнения на данном цикле, разработаны и помещены в систему версионного контроля.
   * Проверку того, что все подсистемы, запланированные для тестирования на данном цикле, разработаны и помещены в систему версионного контроля.
   * Проверку того, что разработана и задокументирована процедура определения и создания среза системы, или build.
   * Проверки некоторых дополнительных критериев.
2. **Подготовка тестовой машины** в соответствии с требованиями, определенными на этапе планирования (например, полная очистка и переустановка системного программного обеспечения). Конфигурация тестовой машины, так же, как и срез системы, должны быть однозначно воспроизводимыми.
3. **Воспроизведение среза системы**.
4. **Прогон тестов** в соответствии с задокументированными процедурами.
5. **Сохранение тестовых протоколов (test log)**. *Test log* может содержать вывод системы в STDOUT, список результатов сравнения полученных при исполнении данных с эталонными или любые другие выходные данные тестов, с помощью которых можно проверить правильность работы системы.
6. **Анализ протоколов тестирования** и принятие решения о том прошел или не прошел каждый из тестов ( **Pass/Fail** ).
7. **Анализ и документирование** результатов цикла.

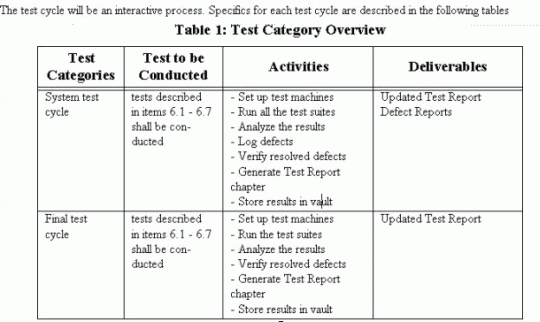
Последний перед выпуском продукта *тестовый цикл* не должен включать изменений кода build или кода продукта тестируемой системы. Этот цикл называется " финальным ". Таким образом обеспечивается ситуация, когда финальный цикл полностью повторяем, а выпускаемый продукт полностью совпадает с продуктом, который прошел тестирование. Финальный цикл необходим для гарантии достоверности результатов тестирования.

### Планирование тестирования

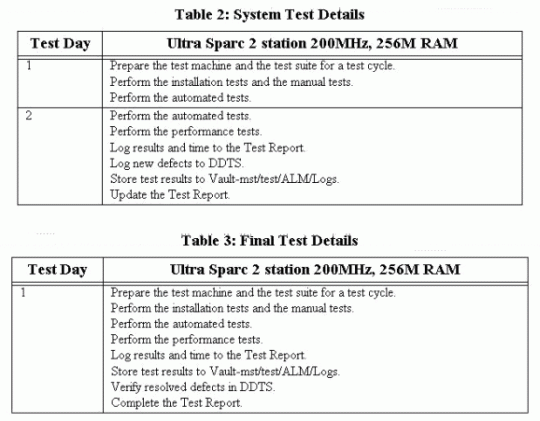
#### Тестовый план

***Тестовый план*** - это документ, или набор документов, содержащий следующую информацию:

1. Тестовые ресурсы.
2. Перечень функций и подсистем, подлежащих тестированию.
3. *Тестовую стратегию*, включающую:
   * Анализ функций и подсистем с целью определения наиболее слабых мест, то есть областей функциональности тестируемой системы, где появление дефектов наиболее вероятно.
   * Определение стратегии выбора входных данных для тестирования. Так как множество возможных входных данных программного продукта, как правило, практически бесконечно, выбор конечного подмножества, достаточного для проведения исчерпывающего тестирования, является сложной задачей. Для ее решения могут быть применены такие методы, как покрытие классов входных и выходных данных, анализ крайних значений, покрытие модели использования, анализ временной линии и тому подобные. Выбранную стратегию необходимо обосновать и задокументировать.
   * Определение потребности в *автоматизированной системе тестирования* и дизайн такой системы
4. Расписание *тестовых циклов* (пример приведен на [Рис. 9.1](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1440?page=3#image.9.1)).
5. Фиксацию тестовой конфигурации: состава и конкретных параметров аппаратуры и программного окружения (пример приведен на [Рис. 9.2](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1440?page=3#image.9.2)).
6. Определение списка тестовых метрик, которые на *тестовом цикле* необходимо собрать и проанализировать. Например, метрик, оценивающих степень покрытия тестами набора требований, степень покрытия кода тестируемой системы, количество и *уровень серьезности* дефектов, объем тестового кода и другие характеристики.



**Рис. 9.1.**Пример расписания двух последних тестовых циклов



**Рис. 9.2.**Пример детализации условий проведения системных циклов

#### Типы тестирования

В *тестовом плане* определяются и документируются различные типы тестов. Типы тестов могут быть классифицированы по двум категориям: по тому, что подвергается тестированию (по виду подсистемы) и по способу выбора входных данных.

Типы тестирования по виду подсистемы или продукта:

1. **Тестирование основной функциональности**, когда тестированию подвергается собственно система, являющаяся основным выпускаемым продуктом
2. **Тестирование инсталляции** включает *тестирование сценариев* первичной инсталляции системы, сценариев повторной инсталляции (поверх уже существующей копии), тестирование деинсталляции, тестирование инсталляции в условиях наличия ошибок в инсталлируемом пакете, в окружении или в сценарии и т.п.
3. **Тестирование пользовательской документации** включает проверку полноты и понятности описания правил и особенностей использования продукта, наличие описания всех сценариев и функциональности, синтаксис и грамматику языка, работоспособность примеров и т.п.

Типы тестирования по способу выбора входных значений:

1. **Функциональное тестирование**, при котором проверяется:
   * Покрытие функциональных требований.
   * Покрытие сценариев использования.
2. **Стрессовое тестирование**, при котором проверяются экстремальные режимы использования продукта.
3. **Тестирование граничных значений**.
4. **Тестирование производительности**.
5. **Тестирование на соответствие стандартам**.
6. **Тестирование совместимости** с другими программно-аппаратными комплексами.
7. **Тестирование работы с окружением**.
8. **Тестирование работы на конкретной платформе**

В реальных разработках используются и комбинируются различные типы тестов для обеспечения спланированного качества продукта.

### Подходы к разработке тестов

Рассмотрим разные подходы к разработке тестов, два к выбору тестовых данных и два к реализации тестового кода.

#### Тестирование спецификации

При разработке тестов, основанных на *функциональной спецификации* продукта, требования к продукту являются основным источником, определяющим, какие тесты будут разработаны. Для каждого требования пишется один или более тестов, которые в совокупности должны проверить выполнение данного требования в продукте.

##### Пример использования спецификации требований для разработки тестов.

Пусть задан следующий фрагмент набора требований для модели обмена транзакциями:

1. Функция DoTransaction должна принимать адрес и данные в соответствии с параметрами, создавать в очереди новый элемент, заполнять его адресную часть и часть полей данных переданной информацией и инициировать транзакцию
2. Функция DoAddressTenure должна принимать адрес в соответствии с параметрами, создавать в очереди новый элемент и заполнять его адресную часть
3. Функция DoDataTenure должна принимать данные в соответствии с параметрами, находить в очереди первый элемент с частично незаполненными полями данных, дополнять его переданной информацией и инициировать транзакцию

Концептуальное описание набора тестов, проверяющего спецификацию, может выглядеть следующим образом:

1. Вызвать DoTransaction с адресом и данными. Проверить появление в очереди еще одного элемента. Проверить появление на шине транзакции с правильными адресом и данными.
2. Вызвать DoAddressTenure с адресом. Проверить появление в очереди еще одного элемента. Проверить отсутствие новой транзакции на шине.
3. Вызвать DoDataTenure с данными. Проверить заполнение полей данных. Проверить появление на шине транзакции с правильными адресом и данными

#### Тестирование сценариев

Разработка тестов, основанных на использовании сценариев, осуществляется по следующей методике:

1. Определяется модель использования, включающая операционное окружение продукта и "актеров". Актером может быть пользователь, другой продукт, аппаратная часть и тому подобное, то есть все, с чем продукт обменивается информацией. Разделение на окружение и актеров условно и служит для описания оптимальных способов использования продукта.
2. Разрабатываются сценарии использования продукта. Описание сценария в зависимости от продукта и выбранного подхода может быть строго определенным, параметризованным или разрешать некоторую степень неопределенности. Например, описание сценария на языке MSC допускает задание параметризованных сценариев с возможностью переупорядочивания событий.
3. Разрабатывается набор тестов, покрывающих заданные сценарии. С учетом степени неопределенности, заложенной в сценарии, каждый тест может покрывать один сценарий, несколько сценариев, или, наоборот, часть сценария.

Использование сценариев не требует наличия полной *формальной спецификации* требований, но зато может потребовать больше времени на разработку и анализ.

Еще одна особенность *тестирования сценариев* заключается в том, что этот метод направляет тестирование на проверку конкретных режимов использования продукта, что позволяет находить дефекты, которые метод тестирования по требованиям может пропустить.

##### Пример использования спецификации требований для разработки тестов.

Так, для рассмотренного выше примера возможно создание следующего сценария и тестов.

1. Сценарий: пользователь имеет две независимые нити управления, одна из которых отвечает за генерацию полных транзакций посредством DoTransaction, а другая – за сбор транзакций из адресной части и части данных, когда эта информация приходит из разных источников. Таким образом, вторая нитка использует вызовы к DoAddressTenure и DoDataTenure.
2. Описание тестов: Вызвать DoAddressTenure c адресом А1, вызвать DoTransaction с адресом А2 и данными D2, вызвать DoDataTenure с данными D1. Проверить последовательное появление на шине двух транзакций: {А1, D1} и {А2, D2}

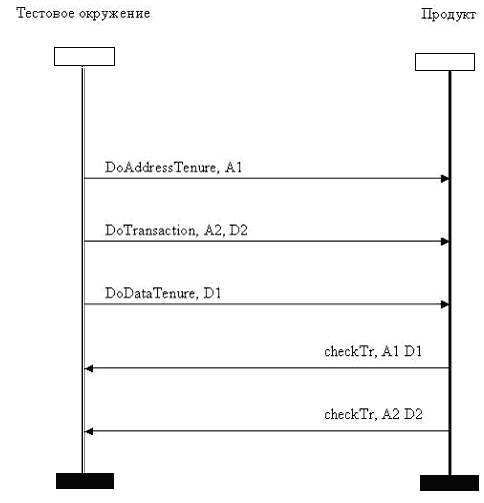
При выполнении этого теста было, в частности, обнаружено, что функция DoTransaction была реализована через вызовы к DoAddressTenure и DoDataTenure, что приводило к появлению на шине транзакций вида {А1, D2} и {А2, D1}. Подобный дефект может быть обнаружен с большим трудом, если разрабатывать тесты, основываясь только на спецификации требований.

#### Ручная разработка тестов

Наиболее распространенным способом разработки тестов является создание тестового кода вручную. Это наиболее гибкий способ разработки тестов, однако характерная для него производительность труда инженеров-тестировщиков в создании тестового кода не намного выше скорости создания кода продукта, а объемы тестового кода на практике зачастую превышают объем кода продукта в 10 раз. Учитывая этот факт, в современной индустрии все больше склоняются к более интеллектуальным способам получения тестового кода, таким как использование специальных тестовых *языков (скриптов*) и генерации тестов.

#### Генерация тестов

В настоящее время некоторые языки спецификаций, используемые для описания алгоритмов тестирования, могут быть использованы для генерации тестового кода. Рассмотрим генерацию кода из языка MSC. Тест, описанный выше, формализован на языке MSC ([Рис. 9.3](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1440?page=4#image.9.3)). Здесь каждая стрелка с пометкой DoTransaction, DoAddressTenure или DoDataTenure представлет собой вызов соответствующей функции продукта с передачей параметров. Стрелка checkTr соответствует проверке прохождения по шине транзакции с соответствующими параметрами. Каждая из стрелок диаграммы генератором тестов преобразуется в исполнимый код, при этом стрелкам, представляющим собой вызовы функций может соответствовать достаточно простой и маленький участок кода, вызывающий соответствующую функцию и проверяющий ее выходное значение на наличие ошибок.

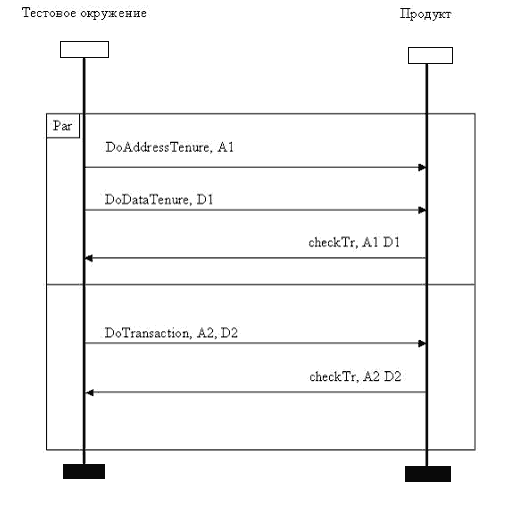


**Рис. 9.3.**Формальная запись сценарного теста на MSC

Следует отметить, что стрелки, соответствующие проверке транзакций, могут после генерации преобразоваться в достаточно сложный код, который будет выполнять ожидание появления транзакции на шине в течение заданного при генерации времени - тайм-аута, проверять фазы транзакции и сверять вычисленные значения параметров с заданными эталонными значениями.

В результате в рассматриваемом примере выигрыш от применения генерационного подхода достигается в основном за счет использования наглядного визуального представления тестов, что может быть нивелировано затратами на создание генерационного сценария на MSC.

Возможна куда более эффективная формализация MSC сценария для генерации тестов. [Рис. 9.4](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1440?page=4#image.9.4) представляет другой способ формализации для теста, выполняющего те же самые проверки.



**Рис. 9.4.**Формальная запись сценарного теста на MSC с использованием параллелизма.

В MSC на [Рис. 9.4](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1440?page=4#image.9.4) проверки транзакций сгруппированы с порождающими их вызовами в отдельные фрагменты, а параллелизм, используемый при исполнении фрагментов, задан через Par – формальную конструкцию, применяемую для изображения параллелизма в языке MSC. При генерации тестов по диаграмме [Рис. 9.4](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1440?page=4#image.9.4) тестовый генератор перебирает все возможные и неповторяющиеся варианты вызова тестируемых функций, сохраняя при этом корректность порядка проверок, что в данном примере дает три сгенерированных теста. Несложно видеть, что затраты на создание диаграммы [Рис. 9.4](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1440?page=4#image.9.4) не сильно отличаются от затрат на диаграмму [Рис. 9.3](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1440?page=4#image.9.3), в то время как количество тестов увеличивается в три раза.

Таким образом, использование методики генерации тестового кода по формализованным MSC диаграммам позволяет значительно поднять производительность тестирования, а также преобразовать формализацию (кодировку) сценариев в достаточно интеллектуальную деятельность.

**Вывод:**

В Ruby типы данных совпадают с классами, к которым относятся объекты. Кроме основных классов таких как String, Float, Integer, Hash, Array возможно создавать свои пользовательские классы и объекты. Ruby является объекто-ориентированным яхыком.

**Вопросы для самоконтроля:**

1. Как создать свой класс;
2. Что такое конструктор класса;
3. Как создать экземпляр класса;
4. Как применить метод к объекту класса;
5. Как узнать к какому классу относится объект.

# Лекция 10. Документирование и оценка индустриального тестирования

### Выполнение тестов

Рассмотрим два основных подхода к выполнению тестов: подход *ручного тестирования* и подход автоматического исполнения (*прогон) тестов*. Подходы рассмотрены на примере тестирования продукта, поддерживающего *интерфейс командной строки*. Тесты описывают вызов продукта с параметрами и проверку возвращаемого значения в виде фиксируемых при прогоне – текста из STDOUT и состояния некоторых файлов, зависящего от входных параметров.

#### Ручное тестирование

*Ручное тестирование* заключается в выполнении задокументированной процедуры, где описана методика выполнения тестов, задающая порядок тестов и для каждого теста - список значений параметров, который подается на вход, и список результатов, ожидаемых на выходе. Поскольку процедура предназначена для выполнения человеком, в ее описании для краткости могут использоваться некоторые значения по умолчанию, ориентированные на здравый смысл, или ссылки на информацию, хранящуюся в другом документе.

##### Пример фрагмента процедуры

1. Подать на вход три разных целых числа.
2. Запустить тестовое исполнение.
3. Проверить, соответствует ли полученный результат таблице [ссылка на документ1] с учетом поправок [ссылка на документ2].
4. Убедиться в понятности и корректности выдаваемой сопроводительной информации.

В приведенной процедуре тестировщик использует два дополнительных документа, а также собственное понимание того, какую сопроводительную информацию считать "понятной и корректной". Успех от использования процедурного подхода достигается в случае однозначного понимания тестировщиком всех пунктов процедуры. Например, в п.1 приведенной процедуры не уточняется, из какого диапазона должны быть заданы три целых числа, и не описывается дополнительно, какие числа считаются "разными".

#### Автоматизированное тестирование

Попытка автоматизировать приведенный выше тест приводит к созданию скрипта, задающего тестируемому продукту три конкретных числа и перенаправляющего вывод продукта в файл с целью его анализа, а также содержащего конкретное значение желаемого результата, с которым сверяется получаемое при прогоне теста значение. Таким образом, вся необходимая информация должна быть явно помещена в текст (скрипт) теста, что требует дополнительных по сравнению с ручным подходом усилий. Также дополнительных усилий и времени требует создание разборщика вывода (программы согласования форматов представления эталонных значений из теста и вычисляемых при прогоне результатов) и, возможно, создание базы хранения состояний эталонных данных.

##### Пример скрипта

Приведем пример последовательности действий, закладываемых в скрипт:

1. Выдать на консоль имя или номер теста и время его начала.
2. Вызвать продукт с фиксированными параметрами.
3. Перенаправить вывод продукта в файл.
4. Проверить возвращенное продуктом значение. Оно должно быть равно ожидаемому (эталонному) результату, зафиксированному в тесте.
5. Проверить вывод продукта, сохраненный в файле (п.3), на равенство заранее приготовленному эталону.
6. Выдать на консоль результаты теста в виде вердикта PASS/FAIL и в случае FAIL - краткого пояснения, какая именно проверка не прошла.
7. Выдать на консоль время окончания теста.

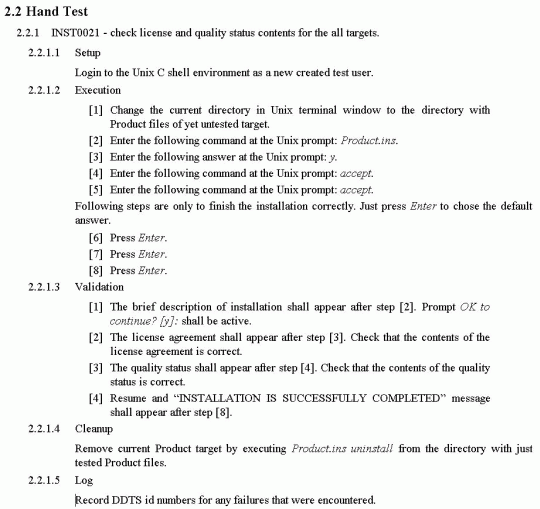
#### Сравнение ручного и автоматизированного тестирования

Результаты сравнения приведены в [Табл. 10.1](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1442?page=1#table.10.1). Сравнение показывает тенденцию современного тестирования, ориентирующую на максимальную автоматизацию процесса тестирования и генерацию тестового кода, что позволяет справляться с большими объемами данных и тестов, необходимых для обеспечения качества при производстве программных продуктов.

|  |  |  |
| --- | --- | --- |
| Таблица 10.1. Сравнение ручного и автоматизированного подхода | | |
|  | **Ручное** | **Автоматизированное** |
| **Задание входных значений** | Гибкость в задании данных. Позволяет использовать разные значения на разных циклах прогона тестов, расширяя покрытие | Входные значения строго заданы |
| **Проверка результата** | Гибкая, позволяет тестировщику оценивать нечетко сформулированные критерии | Строгая. Нечетко сформулированные критерии могут быть проверены только путем сравнения с эталоном |
| **Повторяемость** | Низкая. Человеческий фактор и нечеткое определение данных приводят к неповторяемости тестирования | Высокая |
| **Надежность** | Низкая. Длительные *тестовые циклы* приводят к снижению внимания тестировщика | Высокая, не зависит от длины *тестового цикла* |
| **Чувствительность к незначительным изменениям в продукте** | Зависит от детальности описания процедуры. Обычно тестировщик в состоянии выполнить тест, если внешний вид продукта и текст сообщений несколько изменились | Высокая. Незначительные изменения в интерфейсе часто ведут к коррекции эталонов |
| **Скорость выполнения *тестового набора*** | Низкая | Высокая |
| **Возможность генерации тестов** | Отсутствует. Низкая скорость выполнения обычно не позволяет исполнить сгенерированный набор тестов |  |

#### Тестовые процедуры

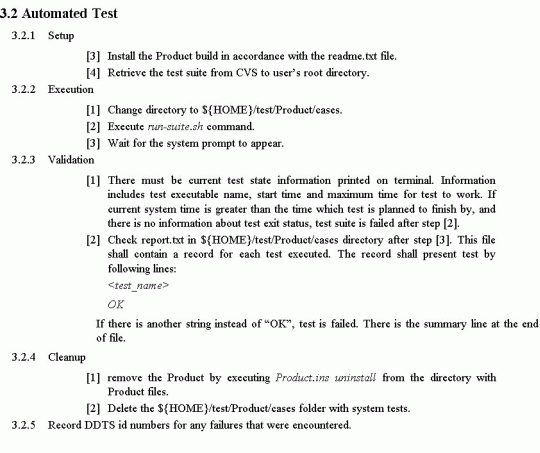
***Тестовые процедуры*** - это формальный документ, содержащий описание необходимых шагов для выполнения *тестового набора* . В случае ручных тестов *тестовые процедуры* содержат полное описание всех шагов и проверок, позволяющих протестировать продукт и вынести вердикт PASS/FAIL.



**Рис. 10.1.**Пример фрагмента тестовой процедуры для ручного тестирования

Процедуры должны быть составлены таким образом, чтобы любой инженер, не связанный с данным проектом, был способен адекватно провести цикл тестирования, обладая только самыми базовыми знаниями о применяющемся инструментарии. Пример фрагмента *тестовой процедуры* для *ручного тестирования* приведен на [Рис. 10.1](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1442?page=2#image.10.1)

В случае описания автоматизированных тестов *тестовые процедуры* должны содержать достаточную информацию для запуска тестов и анализа результатов. Пример фрагмента такой процедуры приведен на [Рис. 10.2](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1442?page=2#image.10.2)



**Рис. 10.2.**Пример фрагмента автоматизированной тестовой процедуры

#### Описание тестов

Описание тестов разрабатывается для облегчения анализа и поддержки *тестового набора*. Описание может быть реализовано в произвольной форме, но при этом должны выполнять следующие задачи:

1. Анализировать степень покрытия продукта тестами на основании описания *тестового набора*.
2. Для любой функции тестируемого продукта найти тесты, в которых функция используется.
3. Для любого теста определить все функции и их сочетания, которые данный тест использует (затрагивает).
4. Понять структуру и взаимосвязи тестовых файлов.
5. Понять принцип построения системы *автоматизации тестирования*.

#### Документирование и жизненный цикл дефекта

Каждый дефект, обнаруженный в процессе тестирования, должен быть задокументирован и отслежен. При обнаружении нового дефекта его заносят в базу дефектов. Для этого лучше всего использовать специализированные базы, поддерживающие хранение и отслеживание дефектов - типа DDTS . При занесении нового дефекта рекомендуется указывать, как минимум, следующую информацию:

1. Наименование подсистемы, в которой обнаружен дефект.
2. Версия продукта (номер build ), на котором дефект был найден.
3. Описание дефекта.
4. Описание процедуры (шагов, необходимых для воспроизведения дефекта).
5. Номер теста, на котором дефект был обнаружен.
6. Уровень дефекта, то есть степень его серьезности с точки зрения критериев качества продукта или заказчика.

Занесенный в базу дефектов новый дефект находится в состоянии " **New** " . После того, как команда разработчиков проанализирует дефект, он переводится в состояние " **Open** " с указанием конкретного разработчика, ответственного за исправление дефекта. После исправления дефект переводится разработчиком в состояние " **Resolved** ". При этом разработчик должен указать следующую информацию:

1. Причину возникновения дефекта.
2. Место исправления, как минимум, с точностью до исправленного файла.
3. Краткое описание того, что было исправлено.
4. Время, затраченное на исправление.

После этого тестировщик проверяет, действительно ли дефект был исправлен и если это так, переводит его в состояние " **Verified** ". Если тестировщик не подтвердит факт исправления дефекта, то состояние дефекта изменяется снова на " **Open** ".

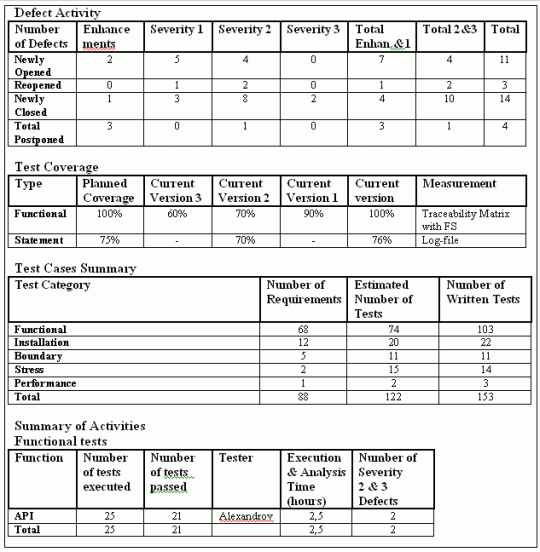
Если проектная команда принимает решение о том, что некоторый дефект исправляться не будет, то такой дефект переводится в состояние " **Postponed** " с указанием лиц, ответственных за это решение, и причин его принятия.

#### Тестовый отчет

*Тестовый отчет* обновляется после каждого цикла тестирования и должен содержать следующую информацию для каждого цикла:

1. Перечень функциональности в соответствии с пунктами требований, запланированный для тестирования на данном цикле, и реальные данные по нему.
2. Количество выполненных тестов – запланированное и реально исполненное.
3. Время, затраченное на тестирование каждой функции, и общее время тестирования.
4. Количество найденных дефектов.
5. Количество повторно открытых дефектов.
6. Отклонения от запланированной последовательности действий, если таковые имели место.
7. Выводы о необходимых корректировках в системе тестов, которые должны быть сделаны до следующего *тестового цикла*.

Пример фрагмента из *тестового отчета* представлен на [Рис. 10.3](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1442?page=2#image.10.3). Приведенный фрагмент отчета содержит примерные данные для четырех циклов тестирования и иллюстрирует структуру отчета. Такой вид отчет имеет после тестирования, перед началом цикла тестирования поля не заполнены, заполнение осуществляется по окончании соответствующего цикла.



**Рис. 10.3.**Фрагмент тестового отчета

### Оценка качества тестов

Тесты нуждаются в контроле качества так же, как и тестируемый продукт. Поскольку тесты для продукта являются своего рода эталоном его структурных и поведенческих характеристик, закономерен вопрос о том, насколько адекватен эталон. Для оценки качества тестов используются различные методы, наиболее популярные из которых кратко рассмотрены ниже.

#### Тестовые метрики

Существует устоявшийся набор тестовых метрик, который помогает определить эффективность тестирования и текущее состояние продукта. К таким метрикам относятся следующие:

1. Покрытие функциональных требований.
2. Покрытие кода продукта. Наиболее применимо для модульного уровня тестирования.
3. Покрытие множества сценариев.
4. Количество или плотность найденных дефектов. Текущее количество дефектов сравнивается со средним для данного типа продуктов с целью установить, находится ли оно в пределах допустимого статистического отклонения. При этом обнаруженные отклонения как в большую, так и в меньшую сторону приводят к анализу причин их появления и, если необходимо, к выработке корректирующих действий.
5. Соотношение количества найденных дефектов с количеством тестов на данную функцию продукта. Сильное расхождение этих двух величин говорит либо о неэффективности тестов (когда большое количество тестов находит мало дефектов) либо о плохом *качестве данного* участка кода (когда найдено большое количество дефектов на не очень большом количестве тестов).
6. Количество найденных дефектов, соотнесенное по времени, или скорость поиска дефектов. Если производная такой функции близка к нулю, то продукт обладает качеством, достаточным для окончания тестирования и поставки заказчику.

#### Обзоры тестов и стратегии

Тестовый код и стратегия тестирования, зафиксированные в виде документов, заметно улучшаются, если подвергаются коллективному обсуждению. Такие обсуждения называются обзорами ( **review** ). Существует принятая в организации процедура проведения и оценки результатов обзора. Обзоры наряду с тестированием образуют мощный набор методов борьбы с ошибками с целью повышения качества продукта. Цели *обзоров тестовой стратегии* и тестового кода различны.

Цели *обзора тестовой стратегии*:

1. Установить достаточность проверок, обеспечиваемых тестированием.
2. Проанализировать оптимальность покрытия или адекватность распределения количества планируемых тестов по функциональности продукта.
3. Проанализировать оптимальность подхода к разработке кода, генерации кода, *автоматизации тестирования*.

Цели обзора тестового кода:

1. Установить соответствие *тестового набора* тестовой стратегии.
2. Проверить правильность кодирования тестов.
3. Оценить достигнутую степень качества кода, исходя из требований по стандартам, простоте поддержки, наличию комментариев и т.п.
4. Если необходимо, проанализировать оптимальность тестового кода с целью удовлетворения требований к быстродействию и объему.

**Вывод:**

В этом уроке вы научились работать с файлами в Ruby, открывать файлы, читать их содержимое и работать с ним в удобной форме. Мы узнали о классе File и о том, как работают его методы new, dirname, exist? и методы его экземпляров read, readlines и close.

Мы научились читать из файлов данные целиком и построчно, как выводить на экран произвольную строчку файла и как, а главное, для чего необходимо закрывать файлы.

**Вопросы для самоконтроля:**

1. Как работать с файлами Ruby?
2. Какой атрибут отвечает за чтение файлов?
3. Как можно читать файлы построчно?
4. Как указать путь к файлу?
5. Как указать полный путь к файлу?

# Лекция 11. Регрессионное тестирование: цели и задачи, условия применения, классификация тестов и методов отбора

### Цели и задачи регрессионного тестирования

При корректировках программы необходимо гарантировать сохранение качества. Для этого используется *регрессионное тестирование* - дорогостоящая, но необходимая *деятельность* в рамках этапа сопровождения, направленная на перепроверку корректности измененной программы. В соответствии со стандартным определением, ***регрессионное тестирование*** - это выборочное тестирование, позволяющее убедиться, что изменения не вызвали нежелательных побочных эффектов, или что измененная система по-прежнему соответствует требованиям.

Главной задачей этапа сопровождения является реализация систематического процесса обработки изменений в коде. После каждой модификации программы необходимо удостовериться, что на функциональность программы не оказал влияния *модифицированный код*. Если такое влияние обнаружено, говорят о *регрессионном дефекте*. Для *регрессионного тестирования* функциональных возможностей, изменение которых не планировалось, используются ранее разработанные тесты. Одна из *целей регрессионного тестирования* состоит в том, чтобы, в соответствии с используемым критерием покрытия кода (например, критерием покрытия потока операторов или потока данных), гарантировать тот же уровень покрытия, что и при полном повторном тестировании программы. Для этого необходимо запускать тесты, относящиеся к измененным областям кода или функциональным возможностям.

Пусть T = {t1, t2, ..., tN} - множество из N тестов, используемое при первичной разработке программы P, а T'\subseteq T - *подмножество* регрессионных тестов для тестирования новой версии программы P'. *Информация* о покрытии кода, обеспечиваемом T', позволяет указать блоки P', требующие дополнительного тестирования, для чего может потребоваться повторный *запуск* некоторых тестов из *множества* T\supseteq T', или даже создание T'' - набора *новых тестов* для P' - и обновление T.

Другая *цель регрессионного тестирования* состоит в том, чтобы удостовериться, что *программа* функционирует в соответствии со своей спецификацией, и что изменения не привели к внесению новых ошибок в ранее протестированный код. Эта цель всегда может быть достигнута повторным выполнением всех тестов регрессионного набора, но более перспективно отсеивать тесты, на которых *выходные данные* модифицированной и старой программы не могут различаться. Результаты сравнения выборочных методов и метода повторного прогона всех тестов приведены в [таблица 11.1](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1444?page=1#table.11.1).

|  |  |
| --- | --- |
| Таблица 11.1. Выборочное регрессионное тестирование и повторный прогон всех тестов. | |
| **Повторный прогон всех тестов** | ***Выборочное регрессионное тестирование*** |
| Прост в реализации | Требует дополнительных расходов при внедрении |
| Дорогостоящий и неэффективный | Способно уменьшать расходы за счет исключения лишних тестов |
| Обнаруживает все ошибки, которые были бы найдены при исходном тестировании | Может приводить к пропуску ошибок |

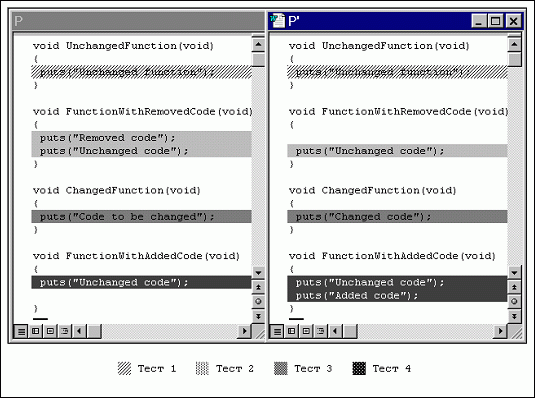
Задача *отбора тестов* из набора T для заданной программы P и измененной версии этой программы P' состоит в выборе подмножества T'_{идеальное}\subseteq T для повторного запуска на измененной программе P', где T'_{идеальное} = \{ t\in T | P'(t) \ne  P(t)\}. Так как *выходные данные* P и P' для тестов из *множества* T\supseteq T'_{идеальное} заведомо одинаковы, нет необходимости выполнять ни один из этих тестов на P'. В общем случае, в отсутствие динамической информации о выполнении P и P' не существует методики вычисления *множества* T'идеальное для произвольных множеств P, P' и T. Это следует из отсутствия общего решения проблемы останова, состоящей в невозможности создания в общем случае алгоритма, дающего ответ на вопрос, завершается ли когда-либо произвольная *программа* P для заданных значений входных данных. На практике создание T'идеальное возможно только путем выполнения на инструментированной версии P' каждого регрессионного теста, чего и хочется избежать.

Реалистичный вариант решения задачи *выборочного регрессионного тестирования* состоит в получении полезной информации по результатам выполнения P и объединения этой информации с данными статического анализа для получения *множества* T'реальное в виде аппроксимации T'идеальное. Этот подход применяется во всех известных выборочных методах *регрессионного тестирования*, основанных на анализе кода. Множество T'реальное должно включать все тесты из T, активирующие измененный код, и не включать никаких других тестов, то есть тест t\in T входит в T'реальное тогда и только тогда, когда t задействует код P в точке, где в P' код был удален или изменен, или где был добавлен новый код.

Если некоторый тест t задействует в P тот же код, что и в P', *выходные данные* P и P' для t различаться не будут. Из этого следует, что если P(t)\ne P'(t), t должен задействовать некоторый код, измененный в P' по отношению к P, то есть должно выполняться *отношение* t\in T'_{реальное}. С другой стороны, поскольку не каждое выполнение измененного кода отражается на выходных значениях теста, могут существовать некоторые такие t\in T'_{реальное}, что P(t) = P'(t). Таким образом, T'реальное содержит T'идеальное целиком и может использоваться в качестве его альтернативы без ущерба для качества тестируемого программного продукта.

Важной *задачей регрессионного тестирования* является также уменьшение стоимости и сокращение времени выполнения тестов.

Рассмотрим *отбор тестов* на примере [рис. 11.1](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1444?page=1#image.11.1). Код, покрываемый тестами, выделен цветом и штриховкой. Легко заметить, что код, покрываемый тестом 1, не изменился с предыдущей версии, следовательно, повторное выполнение теста 1 не требуется. Напротив, код, покрываемый тестами 2, 3 и 4, изменился; следовательно, требуется их повторный *запуск*.



**Рис. 11.1.**Отбор тестов для множества T

### Виды регрессионного тестирования

Поскольку *регрессионное тестирование* представляет собой повторное проведение *цикла* обычного тестирования, *виды регрессионного тестирования* совпадают с видами обычного тестирования. Можно говорить, например, о модульном *регрессионном тестировании* или о функциональном *регрессионном тестировании*.

Другой способ классификации *видов регрессионного тестирования* связывает их с типами сопровождения, которые, в свою *очередь*, определяются типами модификаций. Выделяют три типа сопровождения:

* ***Корректирующее сопровождение***, называемое обычно исправлением ошибок, выполняется в ответ на обнаружение ошибки, не требующей изменения спецификации требований. При *корректирующем сопровождении* производится диагностика и корректировка дефектов в программном обеспечении с целью поддержания системы в работоспособном состоянии.
* ***Адаптивное сопровождение*** осуществляется в ответ на требования изменения данных или среды исполнения. Оно применяется, когда существующая система улучшается или расширяется, а спецификация требований изменяется с целью реализации новых функций.
* **Усовершенствующее (прогрессивное) сопровождение** включает любую обработку с целью повышения эффективности работы системы или эффективности ее сопровождения.

В процессе *адаптивного* или усовершенствующего сопровождения обычно вводятся новые модули. Чтобы отобразить то или иное усовершенствование или адаптацию, изменяется спецификация системы. При *корректирующем сопровождении*, как правило, спецификация не изменяется, и новые модули не вводятся. Модификация программы на фазе разработки подобна модификации при *корректирующем сопровождении*, так как из-за обнаружения ошибки вряд ли требуется менять *спецификацию программы*. За исключением редких моментов крупных изменений, на фазе сопровождения изменения системы обычно невелики и производятся с целью устранения проблем или постепенного расширения функциональных возможностей.

Соответственно, определяют два типа *регрессионного тестирования*: прогрессивное и корректирующее.

* Прогрессивное *регрессионное тестирование* предполагает модификацию технического задания. В большинстве случаев при этом к системе программного обеспечения добавляются новые модули.
* При корректирующем *регрессионном тестировании* техническое задание не изменяется. Модифицируются только некоторые операторы программы и, возможно, конструкторские решения.

Прогрессивное *регрессионное тестирование* обычно выполняется после *адаптивного* или усовершенствующего сопровождения, тогда как корректирующее *регрессионное тестирование* выполняется во *время тестирования* в цикле разработки и после *корректирующего сопровождения*, то есть после того, как над программным обеспечением были выполнены некоторые корректирующие действия. Вообще говоря, корректирующее *регрессионное тестирование* должно быть более простым, чем прогрессивное *регрессионное тестирование*, поскольку допускает повторное использование большего количества тестов.

Подход к отбору регрессионных тестов может быть активным или консервативным. *Активный* подход во главу угла ставит уменьшение объема *регрессионного тестирования* и пренебрегает риском пропустить дефекты. *Активный* подход применяется для тестирования систем с высокой исходной надежностью, а также в случаях, когда эффект изменений невелик. Консервативный подход требует отбора всех тестов, которые с ненулевой вероятностью могут обнаруживать дефекты. Этот подход позволяет обнаруживать большее количество ошибок, но приводит к созданию более обширных наборов регрессионных тестов.

### Управляемое регрессионное тестирование

В течение жизненного *цикла* программы период сопровождения длится долго. Когда измененная *программа* тестируется набором тестов T, мы сохраняем без изменений по отношению к тестированию исходной программы P все факторы, которые могли бы воздействовать на *вывод* программы. Поэтому атрибуты конфигурации, в которой *программа* тестировалась последний раз (например, план тестирования, тесты tj и покрываемые элементы MT(P,C,tj)), подлежат управлению конфигурацией. Практика тестирования измененной версии программы P' в тех же условиях, в которых тестировалась исходная *программа* P, называется *управляемым регрессионным тестированием*. При неуправляемом *регрессионном тестировании* некоторые свойства методов *регрессионного тестирования* могут изменяться, например, *безопасный метод* *отбора тестов* может перестать быть безопасным. В свою *очередь*, для обеспечения *управляемости регрессионного тестирования* необходимо выполнение ряда условий:

* Как при модульном, так и при интеграционном *регрессионном тестировании* в качестве модулей, вызываемых тестируемым модулем непосредственно или косвенно, должны использоваться реальные модули системы. Это легко осуществить, поскольку на этапе *регрессионного тестирования* все модули доступны в завершенном виде.
* Информация об изменениях корректна. Информация об изменениях указывает на измененные модули и разделы спецификации требований, не подразумевая при этом корректность самих изменений. Кроме того, при изменении спецификации требований необходимо усиленное *регрессионное тестирование* изменившихся функций этой спецификации, а также всех функций, которые могли быть затронуты по неосторожности. Единственным случаем когда мы вынуждены положиться на правильность измененного технического задания, является изменение технического задания для всей системы или для модуля верхнего (в *графе вызовов*) уровня, при условии, что кроме технического задания, не существует никакой дополнительной документации и/или какой-либо другой информации, по которой можно было бы судить об ошибке в техническом задании.
* В программе нет ошибок, кроме тех, которые могли возникнуть из-за ее изменения.
* Тесты, применявшиеся для тестирования предыдущих версий программного продукта, доступны, при этом протокол *прогона тестов* состоит из входных данных, выходных данных и траектории. Траектория представляет собой путь в *управляющем графе* программы, прохождение которого вызывается использованием некоторого набора входных данных. Ее можно применять для оценки структурного покрытия, обеспечиваемого набором тестов.
* Для проведения *регрессионного тестирования* с использованием существующего набора тестов необходимо хранить информацию о результатах выполнения тестов на предыдущих этапах тестирования.

Предположим, что никакие *операторы* программы, кроме тех, чье поведение зависит от изменений, не могут неблагоприятно воздействовать на программу. Даже при таком условии существуют некоторые ситуации, требующие особого внимания, например проблема утечки памяти и ей подобные. Ситуации такого рода в разных системах программирования обрабатываются по-разному. Например, язык *Java* сам по себе включает систему управления памятью. Если же система не контролирует *распределение памяти* автоматически, мы должны считать, что все *операторы* работы с памятью также обладают поведением, зависящим от изменений.

Проблема языков типа C и C++, которые допускают произвольные арифметические *операции* над указателями, состоит в том, что указатели могут нарушать границы областей памяти, на которые они указывают. Это означает, что переменные могут обрабатываться способами, которые не поддаются анализу на уровне исходного кода. Чтобы учесть такие нарушения границ памяти, выдвигаются следующие гипотезы:

* Гипотеза 1 (четко определенная память). Каждый сегмент памяти, к которому обращается система программного обеспечения, соответствует некоторой символически определенной переменной.
* Гипотеза 2 (строго ограниченный указатель). Каждая переменная или выражение, используемое как указатель, должно ссылаться на некоторую базовую переменную и ограничиваться использованием сегмента памяти, определяемого этой переменной.

Чтобы гарантировать покрытие всех зависящих от изменений компонентов, для которых можно показать, что они затрагиваются существующими тестами, достаточно одного теста для каждого из таких компонентов. Множество тестов достаточно большого размера (как правило сценарных), может способствовать обнаружению ошибок, вызванных нарушениями условий *управляемого регрессионного тестирования*.

Существуют и организационные условия проведения *регрессионного тестирования*. Это *ресурс* (время), необходимый тестовому аналитику для ознакомления со спецификацией требований системы, ее архитектурой и, возможно, самим кодом.

### Обоснование корректности метода отбора тестов

Перечислим некоторые особенности реализации *регрессионного тестирования*.

* Некоторые участки кода программы не получают управление при выполнении некоторых тестов.
* Если участок кода реализует требование, но измененный фрагмент кода не получает управления при выполнении теста, то он и не может воздействовать на значения выходных данных программы при выполнении данного теста.
* Даже если участок кода, реализующий требование, получает управление при выполнении теста, это далеко не всегда отражается на выходных данных программы при выполнении данного теста. Действительно, если изменяется первый блок программы, например, путем добавления инициализации переменной, все пути в программе также изменяются, и, как следствие, требуют повторного тестирования. Однако может так случиться, что только на небольшом подмножестве путей действительно используется эта инициализированная переменная.
* Не каждый тест t_{k}\in T, проверяющий код, находящийся на одном пути с измененным кодом, обязательно покрывает этот измененный код.
* Код, находящийся на одном пути с измененным кодом, может не воздействовать на значения выходных данных измененных модулей программы.
* Не всегда каждый оператор программы воздействует на каждый элемент ее выходных данных.

Предположим, что изменения в программе ограничиваются одним оператором. Если при выполнении какого-либо теста на исходной программе этот оператор никогда не получает управление, можно с уверенностью сказать, что он не получит управление и в ходе выполнения теста на новой программе, а результаты тестирования новой и старой программ будут совпадать. Следовательно, нет необходимости выполнять этот тест на новой программе. Указанный метод легко можно обобщить для случая нескольких изменений: если тест не задействует ни одного измененного оператора, и его *входные данные* не изменились, код, выполняемый им в измененной программе, будет в точности таким же, как в первоначальной версии. Такой тест не выявляет различий между двумя версиями системы; следовательно, нет необходимости прогонять его повторно. Если тест не затрагивает ни одного оператора вывода, поведение которого зависит от измененных операторов, это означает, что, несмотря на изменения в программе, все *операторы*, которые получают управление при выполнении этого теста, не изменят *вывод* системы по отношению к предыдущей версии. Таким образом, нет необходимости повторно прогонять и тесты такого рода.

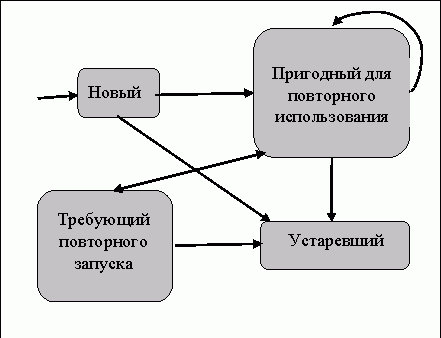
Следовательно, необходимо ориентироваться на выбор только тех тестов, которые покрывают измененный код, воздействующий, в свою *очередь*, на *вывод* программы. Такой подход гарантирует, что будут выбраны только тесты, обнаруживающие изменения, и метод будет, как говорят, точным.

### Классификация тестов при отборе

Создание наборов регрессионных тестов рекомендуется начинать с *множества* исходных тестов. При заданном критерии *регрессионного тестирования* все исходные тесты t ( t\in T ) подразделяются на четыре подмножества:

1. Множество тестов, пригодных для повторного использования. Это тесты, которые уже запускались и пригодны к использованию, но затрагивают только покрываемые элементы программы, не претерпевшие изменений. При повторном выполнении выходные данные таких тестов совпадут с выходными данными, полученными на исходной программе. Следовательно, такие тесты не требуют перезапуска.
2. Множество тестов, требующих повторного запуска. К ним относятся тесты, которые уже запускались, но требуют перезапуска, поскольку затрагивают, по крайней мере, один измененный покрываемый элемент, подлежащий повторному тестированию. При повторном выполнении такие тесты могут давать результат, отличный от результата, показанного на исходной программе. Множество тестов, требующих повторного запуска, обеспечивает хорошее покрытие структурных элементов даже при наличии новых функциональных возможностей.
3. Множество *устаревших тестов*. Это тесты, более не применимые к измененной программе и непригодные для дальнейшего тестирования, поскольку они затрагивают только покрываемые элементы, которые были удалены при изменении программы. Их можно удалить из набора регрессионных тестов.
4. *Новые тесты*, которые еще не запускались и могут быть использованы для тестирования.

[Рис. 11.2](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1444?page=3#image.11.2) дает *представление* о жизненном цикле теста. Сразу после создания тест вводится в структуру *базы данных* как новый. После исполнения *новый тест* переходит в категорию тестов, пригодных для *повторного использования* либо *устаревших*. Если выполнение теста способствовало увеличению текущей степени покрытия кода, тест помечается как пригодный для *повторного использования*. В противном случае он помечается как устаревший и отбрасывается. Существующие тесты, повторно запущенные после внесения изменения в код, также классифицируются заново как пригодные для *повторного использования* или устаревшие в зависимости от тестовых траекторий и используемого критерия тестирования.



**Рис. 11.2.**Жизненный цикл теста

Классификация тестов по отношению к изменениям в коде требует анализа последствий изменений. Тесты, активирующие код, затронутый изменениями, могут требовать повторного запуска или оказаться устаревшими. Чтобы тест был включен в *класс* тестов, требующих повторного запуска, он должен быть затронут изменениями в коде, а также должен способствовать увеличению степени покрытия измененного кода по используемому критерию. Затронутым элементом теста может быть *траектория*, выходные значения, или и то, и другое. Чтобы тест был включен в *класс* тестов, пригодных для *повторного использования*, он должен вносить вклад в увеличение степени покрытия кода и не требовать повторного запуска.

Степень покрытия кода определяется для тестов, пригодных для *повторного использования*, поскольку к этому классу относятся тесты, не требующие повторного запуска и способствующие увеличению степени покрытия до желаемой величины. Если имеется *компонент* программы, не задействованный пригодными для *повторного использования тестами*, то вместо них выбираются и выполняются с целью увеличения степени покрытия тесты, требующие повторного запуска. После запуска такой тест становится пригодным для *повторного использования* или *устаревшим*. Если тестов, требующих повторного запуска, больше не осталось, а необходимая степень покрытия кода еще не достигнута, порождаются дополнительные тесты и тестирование повторяется.

Окончательный набор тестов собирается из тестов, пригодных для *повторного использования*, тестов, требующих повторного запуска, и *новых тестов*. Наконец, *устаревшие* и избыточные тесты удаляются из набора тестов, поскольку избыточные тесты не проверяют новые функциональные возможности и не увеличивают покрытие.

### Возможности повторного использования тестов

К изменению существующих тестов могут привести три следующих вида деятельности программистов:

* Создание *новых тестов*.
* Выполнение тестов.
* Изменение кода.

Поскольку каждый тест содержит *входные данные*, *выходные данные* и траекторию, эти компоненты могут подвергнуться изменению в любой комбинации. При изменении входных данных существующего теста будем считать, что старый тест прекращает существование, и создается *новый тест*. Таким образом, к числу разрешенных изменений теста относятся всевозможные пертурбации выходных данных или траекторий. Изменение выходных данных без изменения траектории и/или входных данных невозможно. Следовательно, существует только два возможных варианта изменения теста: изменение траектории или изменение траектории и выходных данных.

В соответствии с приведенными выше рассуждениями можно выделить четыре уровня *повторного использования теста*:

* Уровень 1. Тест не допускает *повторного использования*. Требуется создание нового набора тестов (например, путем удаления или изменения этого теста).
* Уровень 2. *Повторное использование* возможно только входных данных теста. Во многих случаях цель тестирования состоит в активизации некоторых покрываемых элементов программы. Если из траектории существующего теста видно, что элементы программы, подлежащие покрытию, задействуются до измененных команд, входные данные теста могут быть использованы повторно для покрытия этих элементов. В результате изменений в программе и/или техническом задании новая траектория и выходные данные теста могут отличаться от результатов предыдущего выполнения. Таким образом, тесты первого уровня должны быть запущены повторно для получения новых выходных данных и траекторий.
* Уровень 3. Возможно *повторное использование* как входных, так и выходных данных теста. Очевидно, что на этом уровне обычно располагаются функциональные тесты. Если модуль подвергся только изменению кода с сохранением функциональности, возможно *повторное использование* существующих функциональных тестов для проверки правильности реализации. Поскольку траектория может измениться, а выходные данные - подвергнуться воздействию со стороны изменений кода, такие тесты должны быть запущены повторно, но ожидается получение идентичных результатов.
* Уровень 4. Наивысший уровень *повторного использования теста*, предусматривающий повторное использование входных данных, выходных данных и траектории теста. В этом случае на траектории теста не изменяется ни один оператор. Следовательно, в повторном запуске этих тестов необходимости нет, так как выходные данные и траектория останутся неизменными.

#### Пример регрессионного тестирования функции решения квадратного уравнения.

Код этой функции приведен на [пример 11.1](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1444?page=4#example.11.1). Входными параметрами являются коэффициенты квадратного уравнения A, B и C, а также флаг Print, ненулевое значение которого указывает, что полученное решение необходимо вывести на экран. К выходным параметрам относятся X1 и X2, предназначенные для хранения корней уравнения, и возвращаемое значение функции - дискриминант уравнения. В исходном виде функция содержит дефект, в результате чего уравнения с отрицательным дискриминантом порождают ошибку времени выполнения. В новой версии функции дефект должен быть исправлен; кроме того, необходимо реализовать запрос пользователя на изменение *формата вывода* решения. Код новой версии функции Equation приводится на [пример 11.2](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1444?page=4#example.11.2).

Существующие тесты для функции Equation приведены в [таблица 11.2](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1444?page=4#table.11.2). Входные данные тестов представляют собой совокупность значений Print, A, B и C, подаваемых на вход функции. Выходными данными для теста являются значения X1 и X2, возвращаемое значение функции, а также строка, выводимая на экран; в [таблица 11.2](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1444?page=4#table.11.2) приведены *ожидаемые значения* выходных данных. Кроме того, для каждого теста вычисляется траектория его прохождения по коду.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Таблица 11.2. Входные и выходные данные тестов | | | | | | | | |
| **Тест** | **Входные данные** | | | | **Ожидаемые выходные данные** | | | |
| **A** | **B** | **C** | **Print** | **X1** | **X2** | **Возвращаемое значение** | **Выводимая строка** |
| **1** | 1 | 1 | -6 | 1 | 2 | -3 | 25 | Solution: X1 = 2, X2 = -3 |
| **2** | 2 | -3 | 5 | 1 | 0.75 | 5.567764 | -31 | Solution: X1 = 0.75+5.567764i, X2 = 0.75-5.567764I |
| **3** | 1 | 2 | 0 | 0 | 0 | -2 | 4 |  |
| **4** | 1 | 2 | 1 | 0 | -1 | -1 | 0 |  |
| **5** | 1 | 2 | 2 | 0 | -1 | 2 | -4 |  |

double Equation(int Print, float A, float B, float C,

float& X1, float& X2)

{

float D = B \* B - 4.0 \* A \* C;

if (D >= 0)

{

X1 = (-B + sqrt(D)) / 2.0 / A;

X2 = (-B - sqrt(D)) / 2.0 / A;

}

else

{

X1 = -B / 2.0 / A;

X2 = sqrt(D) / 2.0 / A;

}

if (Print)

printf("Solution: %f, %f\n", X1, X2);

return D;

}

11.1. Функция Equation - исходная версия.

double Equation(int Print, float A, float B, float C,

float& X1, float& X2)

{

float D = B \* B - 4.0 \* A \* C;

if (D >= 0)

{

X1 = (-B + sqrt(D)) / 2.0 / A;

X2 = (-B - sqrt(D)) / 2.0 / A;

}

else

{

X1 = -B / 2.0 / A;

X2 = sqrt(D);

}

if (Print)

printf("Solution: %f, %f\n", X1, X2);

return D;

}

11.1.1. Функция Equation - исходная версия.

double Equation(int Print, float A, float B,

float C, float& X1, float& X2)

{

float D = B \* B - 4.0 \* A \* C;

if (D >= 0)

{

X1 = (-B + sqrt(D)) / 2.0 / A;

X2 = (-B - sqrt(D)) / 2.0 / A;

}

else

{

X1 = -B / 2.0 / A;

X2 = sqrt(-D);

}

if (Print)

{

if (D >= 0)

printf("Solution: X1 = %f, X2 = %f\n", X1, X2);

else

printf("Solution: X1 = %f+%fi, X2 = %f-%fi\n",

X1, X2, X1, X2);

}

return D;

}

11.2. Функция Equation - измененная версия.

double Equation(int Print, float A, float B,

float C, float& X1, float& X2)

{

float D = B \* B - 4.0 \* A \* C;

if (D >= 0)

{

X1 = (-B + sqrt(D)) / 2.0 / A;

X2 = (-B - sqrt(D)) / 2.0 / A;

}

else

{

X1 = -B / 2.0 / A;

X2 = sqrt(-D);

}

if (Print)

{

if (D >= 0)

printf("Solution: X1 = %f, X2 = %f\n", X1, X2);

else

printf("Solution: X1 = %f+%fi, X2 = %f-%fi\n",

X1, X2, X1, X2);

}

return D;

}

11.2.1. Функция Equation - измененная версия.

При изменении функции Equation от [пример 11.1](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1444?page=4#example.11.1) к [пример 11.2](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1444?page=4#example.11.2) меняется формат выводимых на экран данных, так что тесты 1 и 2, проверяющие вывод на экран, могут быть повторно использованы только на уровне 2. Тесты 3, 4 и 5 могут быть использованы на уровне 3 или 4 в зависимости от результатов анализа их траектории.

### Классификация выборочных методов

Для проверки корректности различных подходов к *регрессионному тестированию* используется модель оценки методов *регрессионного тестирования*. Основными объектами рассмотрения стали *полнота*, *точность*, эффективность и *универсальность*.

**Полнота** отражает меру *отбора тестов* из T, на которых результат выполнения измененной программы отличен от результата выполнения исходной программы, вследствие чего могут быть обнаружены ошибки в P'. Метод, полный на 100%, называется безопасным.

**Точность** - *мера* способности метода избегать выбора тестов из T, на которых результат выполнения измененной программы не будет отличаться от результата ее первоначальной версии, то есть тестов, неспособных обнаруживать ошибки в P'. Предположим, что набор T содержит r регрессионных тестов. Из них для n тестов ( n<=r ) поведение и результаты выполнения старой программы P отличаются от поведения и результатов выполнения новой программы P'. Набор тестов T'\subseteq T содержит m ( m\ne 0 ) тестов, полученных с использованием метода отбора регрессионных тестов M. Из этих m тестов для l тестов поведение P' и P различается. *Точность* T' относительно P, P', T и M, выраженная в процентах, определяется выражением 100 \* (l / m), тогда как соответствующий *процент* выбранных тестов определяется выражением 100 \* (l / n), если n \ne  0 или равен 100%, если n = 0.

Исходя из приведенного определения, *точность* *множества* тестов - это *отношение* числа тестов данного *множества*, на которых результаты выполнения новой и старой программ различаются, к общему числу тестов *множества*. *Точность* является важным атрибутом метода *регрессионного тестирования*. Неточный метод имеет тенденцию отбирать тесты, которые не должны были быть выбраны. Чем менее точен метод, тем ближе объем выбранного набора тестов к объему исходного набора тестов.

**Эффективность** - оценка вычислительной стоимости стратегии *выборочного регрессионного тестирования*, то есть стоимости реализации ее требований по времени и памяти, а также возможности автоматизации. Относительной эффективностью называется эффективность метода тестирования при условии наличия не более одной ошибки в тестируемой программе. Абсолютной эффективностью называется эффективность метода в реальных условиях, когда оценка количества ошибок в программе не ограничена.

**Универсальность** отражает меру способности метода к применению в достаточно широком диапазоне ситуаций, встречающихся на практике.

Для программы P, ее измененной версии P' и набора тестов T для P требуется, чтобы методика выборочного повторного тестирования удовлетворяла следующим критериям оценки:

* Критерий 1. **Безопасность**. Методика выборочного повторного тестирования должна быть безопасной, то есть должна выбирать все тесты из T, которые потенциально могут обнаруживать ошибки (все тесты, чье поведение на P' и P может быть различным). Безопасная методика должна рассматривать последствия добавления, удаления и изменения кода. При добавлении нового кода в P в T могут уже содержаться тесты, покрывающие этот новый код. Такие тесты необходимо обнаруживать и учитывать при отборе.
* Критерий 2. **Точность**. Стратегия повторного прогона всех тестов является безопасной, но неточной. В дополнение к выбору всех тестов, потенциально способных обнаруживать ошибки, она также выбирает тесты, которые ни в коем случае не могут демонстрировать измененное поведение. В идеале, методика выборочного повторного тестирования должна быть точной, то есть должна выбирать только тесты с изменившимся поведением. Однако для произвольно взятого теста, не запуская его, невозможно определить, изменится ли его поведение. Следовательно, в лучшем случае мы можем рассчитывать лишь на некоторое увеличение точности.

Всевозможные существующие выборочные методы *регрессионного тестирования* различаются не в последнюю очередь выбором объекта или объектов, для которых выполняется анализ покрытия и анализ изменений. Например, при анализе на уровне функции при изменении любого оператора функции вся функция считается измененной; при анализе на уровне отдельных операторов мы можем исключить часть тестов, содержащих вызов функции, но не активирующих измененный оператор. Выбор объектов для анализа покрытия отражается на уровне подробности анализа, а значит, и на его точности и эффективности. Абсолютные величины точности и количества выбранных тестов для заданных набора тестов и множества изменений должны рассматриваться только вместе с уменьшением размера набора тестов. Небольшой процент выбранных тестов может быть приемлемым, только если уровень точности остается достаточно высоким.

* Критерий 3. **Эффективность**. Методика выборочного повторного тестирования должна быть эффективной, то есть должна допускать автоматизацию и выполняться достаточно быстро для практического применения в условиях ограниченного времени *регрессионного тестирования*. Методика должна также предусматривать хранение информации о ходе исполнения тестов в минимально возможном объеме.
* Критерий 4. **Универсальность**. Методика выборочного повторного тестирования должна быть универсальной, то есть применимой ко всем языкам и языковым конструкциям, эффективной для реальных программ и способной к обработке сколь угодно сложных изменений кода.

В общем случае существует некоторый компромисс между безопасностью, точностью и эффективностью. При *отборе тестов* анализ необходимо провести за время, меньшее, чем требуется для выполнения и проверки результатов тестов из T, не вошедших в T'. С учетом этого ограничения решением *задачи регрессионного тестирования* будет *безопасный метод* с хорошим балансом дешевизны и высокой точности.

**Вывод:**

Регулярные выражения — спасение от всех бед для одних и ночной кошмар для других разработчиков, а если говорить объективно, то это мощнейший инструмент, требующий, однако, большой осторожности при применении. Регулярные выражения (регексы, регекспы, регулярки) в языке Ruby основаны на синтаксисе Perl 5 и потому в основных чертах знакомы всем, кто использовал Perl, Python или PHP. Но Ruby тем и хорош, что каждый компонент языка реализован со своим собственным подходом, упрощающим использование данного инструмента и увеличивающим его мощность. В предлагаемой мной небольшой статье рассматриваются особенности регулярок в Ruby и их применение в различных операторах.

**Вопросы для самоконтроля:**

1. К какому классу относятся регулярные выражения?
2. Как работает метод last\_match?
3. Как использовать регулярные выражения в операторе case?
4. Как использовать регулярные выражения в методе split?
5. Как использовать регулярные выражения в методе scan?

# Лекция 12. Регрессионное тестирование

### Случайные методы

Когда из-за ограничений по времени использование метода повторного прогона всех тестов невозможно, а *программные средства* *отбора тестов* недоступны, инженеры, ответственные за тестирование, могут выбирать тесты случайным образом или на основании "догадок", то есть предположительного соотнесения тестов с функциональными возможностями на основании предшествующих знаний или опыта. Например, если известно, что некоторые тесты задействуют особенно важные функциональные возможности или обнаруживали ошибки ранее, их было бы неплохо использовать также и для тестирования измененной программы. Один простой метод такого рода предусматривает случайный *отбор* предопределенного процента тестов из T. Подобные *случайные методы* принято обозначать random(x), где x - *процент* выбираемых тестов.

*Случайные методы* оказываются на удивление дешевыми и эффективными. Случайно выбранные *входные данные* могут давать больший разброс по покрытию кода, чем *входные данные*, которые используются в наборах тестов, основанных на покрытии, в одних случаях дублируя покрытие, а в других не обеспечивая его. При небольших интервалах тестирования их эффективность может быть как очень высокой, так и очень низкой. Это приводит и к большему разбросу статистики *отбора тестов* для таких наборов. Однако при увеличении интервала тестирования этот разброс становится значительно меньше, и средняя эффективность *случайных методов* приближается к эффективности метода повторного прогона всех тестов с небольшими отклонениями для разных попыток. Таким образом, в последнем случае *пользователь* *случайных методов* может быть более уверен в их эффективности. Вообще, детерминированные методы эффективнее *случайных методов*, но намного дороже, поскольку выборочные стратегии требуют большого количества времени и ресурсов при *отборе тестов*.

Если изменения в новой версии затрагивают код, выполняемый относительно часто, при случайных входных данных измененный код может в среднем активироваться даже чаще, чем при выполнении тестов, основанных на покрытии кода. Это приведет к увеличению метрики количества отобранных тестов для случайных наборов. Наоборот, относительно редко выполняемый измененный код активируется случайными тестами реже, и соответствующая *метрика* снижается. При уменьшении мощности *множества* отобранных тестов падает эффективность обнаружения ошибок.

Когда выбранное *подмножество*, хотя и совершенное с точки зрения полноты и точности, все еще слишком дорого для регрессионного тестирования, особенно важна гибкость при *отборе тестов*. Какие дополнительные процедуры можно применить для дальнейшего уменьшения числа выбранных тестов? Одно из возможных решений - случайное *исключение* тестов. Однако, поскольку такое решение допускает *произвольное* удаление тестов, активирующих изменения в коде, существует высокий риск исключения всех тестов, обнаруживающих ошибку в этом коде. Тем не менее, если *стоимость* пропуска ошибок незначительна, а *интервал* тестирования велик, целесообразным будет использование *случайного метода* с небольшим процентом выбираемых тестов ( **25-30%** ), например, random(25).

Вернемся к примеру регрессионного тестирования функции решения квадратного уравнения. *Случайный метод*, такой, как random(40), может отобрать для повторного выполнения любые 2 теста из 5. Например, если будут выбраны тесты 4 и 5, изменения *формата вывода* на экран не будут протестированы вовсе, что вряд ли может устроить разработчика.

При использовании другого *случайного метода* - *метода экспертных оценок* - в данном случае наиболее вероятен выбор всех тестов, так как *затраты* на прогон невелики. Однако при регрессионном тестировании больших программных систем, когда повторный прогон всех тестов неприемлем, эксперт вынужден отсеивать некоторые тесты, что также может приводить к тому, что часть изменений не будет протестирована полностью.

### Безопасные методы

Метод *выборочного регрессионного тестирования* называется ***безопасным***, если при некоторых четко определенных условиях он не исключает тестов (из доступного набора тестов), которые обнаружили бы ошибки в измененной программе, то есть обеспечивает выбор всех тестов, обнаруживающих изменения. Тест называется обнаруживающим изменения, если его *выходные данные* при прогоне на P' отличаются от выходных данных при прогоне на P: P(t) \ne  P'(t). Тесты, активизирующие измененный код, называются выполняющими изменение.

Выбор всех выполняющих изменение тестов является *безопасным*, но при этом отбираются некоторые тесты, не обнаруживающие изменений. *Безопасный метод* может включать в T' *подмножество* тестов, *выходные данные* которых для P и P' ни при каких условиях не отличаются. Поскольку не существует методики, кроме собственно выполнения теста, позволяющей для любой P' определить, будут ли *выходные данные* теста различаться для P и P', ни один метод не может быть *безопасным* и абсолютно точным одновременно. T' является безопасным подмножеством T тогда и только тогда, когда:

P(t) \ne  P'(t) \Rightarrow  t\in T'

Если P и P' выполняются в идентичных условиях и T' является безопасным подмножеством T, *исполнение* T' на P' всегда обнаруживает любые связанные с изменениями ошибки в P, которые могут быть найдены путем исполнения T. Если существует тест, обнаруживающий ошибку, *безопасный метод* всегда находит ее. Таким образом, ни один *случайный метод* не обладает такой же эффективностью обнаружения ошибок, как *безопасный метод*.

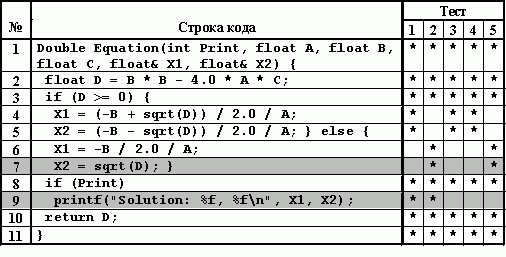
При некоторых условиях *безопасные методы* в силу определения "безопасности" гарантируют, что все "обнаруживаемые" ошибки будут найдены. Поэтому относительная эффективность всех *безопасных методов* равна эффективности метода повторного прогона всех тестов и составляет 100%. Однако их абсолютная эффективность падает с увеличением интервала тестирования. Отметим, что *безопасный метод* действительно безопасен только в предположении корректности исходного *множества* тестов T, то есть когда при выполнении всех t\in T исходная *программа* P завершилась с корректными значениями выходных данных, а все устаревшие тесты были из T удалены.

Существуют программы, измененные версии и наборы тестов, для которых применение безопасного *отбора* не дает большого выигрыша в размере набора тестов. Характеристики исходной программы, измененной версии и набора тестов могут совместно или независимо воздействовать на результаты *отбора тестов*. Например, при усложнении структуры программы *вероятность* активации произвольным тестом произвольного изменения в программе уменьшается. *Безопасный метод* предпочтительнее выполнения всех тестов набора тогда и только тогда, когда *стоимость* анализа меньше, чем *стоимость* выполнения невыбранных тестов. Для некоторых систем, критичных с точки зрения безопасности, *стоимость* пропуска ошибки может быть настолько высока, что небезопасные методы *выборочного регрессионного тестирования* использовать нельзя.

Примером *безопасного метода* может служить метод, который выбирает из T каждый тест, выполняющий, по крайней мере, один оператор, добавленный или измененный в P' или удаленный из P. Применение этого метода для регрессионного тестирования функции решения квадратного уравнения потребует построения матрицы покрытия, пример которой приведен в таблице на [Рис. 12.1](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1446?page=2#image.12.1). Следует отметить, что *матрица* покрытия соответствует исходной версии программы, поскольку аналогичная *информация* для новой версии программы пока не собрана. Звездочка в ячейке таблицы означает, что соответствующий тест покрывает определенную строку кода; если тест не покрывает строку кода, *ячейка* оставлена пустой. Строки, измененные по отношению к исходной версии, выделены цветом. Легко заметить, что в соответствии с требованиями предложенного *безопасного метода* для повторного выполнения должны быть отобраны тесты 1, 2 и 5.

### Методы минимизации

*Процедура минимизации* набора тестов ставит целью *отбор* минимального (в терминах количества тестов) подмножества T, необходимого для покрытия каждого элемента программы, зависящего от изменений. Для проверки *корректности программы* используются только тесты из минимального подмножества.



**Рис. 12.1.**Матрица покрытия тестируемого кода

Обоснование применения *методов минимизации* состоит в следующем:

* Корреляция между эффективностью обнаружения ошибок и покрытием кода выше, чем между эффективностью обнаружения ошибок и размером множества тестов. Неэффективное тестирование, например многочасовое выполнение тестов, не увеличивающих покрытие кода, может привести к ошибочному заключению о *корректности программы*.
* Независимо от способа порождения исходного набора тестов, его минимальные подмножества имеют преимущество в размере и эффективности, так как состоят из меньшего количества тестов, не ослабляя при этом способности к обнаружению ошибок или снижая ее незначительно.
* Вообще говоря, сокращенный набор тестов, отобранный при *минимизации*, может обнаруживать ошибки, не обнаруживаемые сокращенным набором того же размера, выбранным случайным или каким-либо другим способом. Такое преимущество *минимизации* перед *случайными методами* в эффективности является закономерным. Однако из всех *детерминированных методов* *минимизация* приводит к созданию наименее эффективных наборов тестов, хотя и самых маленьких. В частности, *безопасные методы* эффективнее *методов минимизации*, хотя и намного дороже.

*Минимизация* набора тестов требует определенных затрат на *анализ*. Если *стоимость* этого анализа больше затрат на выполнение некоторого порогового числа тестов, существует более дешевый *случайный метод*, обеспечивающий такую же эффективность обнаружения ошибок.

Хотя минимальные наборы тестов могут обеспечивать структурное покрытие измененного кода, зачастую они не являются безопасными, поскольку очевидно, что некоторые тесты, потенциально способные обнаруживать ошибки, могут остаться за чертой *отбора*. Набор функциональных тестов обычно не обладает избыточностью в том смысле, что никакие два теста не покрывают одни и те же *функциональные требования*. Если тесты исходно создавались по критерию структурного покрытия, *минимизация* приносит плоды, но когда мы имеем дело с функциональными тестами, предпочтительнее не отбрасывать тесты, потенциально способные обнаруживать ошибки. В существующей практике тестирования инженеры предпочитают не заниматься *минимизацией* набора тестов.

Многие критерии покрытия кода фактически не требуют выбора минимального *множества* тестов. В некотором смысле, о безопасных стратегиях и стратегиях минимизации можно думать как о находящихся на двух полюсах *множества* стратегий. На практике, использование "почти минимальных" наборов тестов может быть удовлетворительным. Стремление к сокращению объема набора тестов основано на интуитивном предположении, что неоднократное повторное выполнение кода в ходе *модульного тестирования* "расточительно". Однако усилия, требуемые для *минимизации* набора тестов, могут быть существенны, и, следовательно, могут не оправдывать затрат. Отметим, что большинство стратегий *выборочного регрессионного тестирования*, описанных в литературе, в общем-то, не зависит от критерия покрытия, возможно, использовавшегося при создании исходного набора тестов. Инженеры, занимающиеся регрессионным тестированием, часто не имеют информации о том, как разрабатывался исходный набор тестов.

Обнаружение ошибок важно для приложений, где *стоимость* выполнения тестов очень высока, в то время как *стоимость* пропуска ошибок считается незначительной. В этих условиях использование *методов минимизации* целесообразно, поскольку они связаны с *отбором* небольшого количества тестов. Примером применения *методов минимизации* служит метод, выбирающий из T не менее одного теста для каждого оператора программы, добавленного или измененного при создании P'. В таблице на [Рис. 12.1](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1446?page=2#image.12.1) для случая регрессионного тестирования функции Equation данный метод ограничится *отбором* одного теста - теста 2, так как этот тест покрывает обе измененные строки.

### Методы, основанные на покрытии кода

*Значение* *методов, основанных на покрытии кода*, состоит в том, что они гарантируют сохранение выбранным набором тестов требуемой степени покрытия элементов P' относительно некоторого критерия структурного покрытия C, использовавшегося при создании первоначального набора тестов. Это не означает, что если *атрибут* программы, определенный C, покрывается первоначальным множеством тестов, он будет также покрыт и выбранным множеством; гарантируется только сохранение процента покрываемого кода. *Методы, основанные на покрытии*, уменьшают разброс по покрытию, требуя *отбора тестов*, активирующих труднодоступный код, и исключения тестов, которые только дублируют покрытие. Поскольку на практике критерии покрытия кода обычно применяются для *отбора* единственного теста для каждого покрываемого элемента, *подходы, основанные на покрытии кода*, можно рассматривать как специфический вид *методов минимизации*.

Разновидностью *методов, основанных на покрытии кода*, являются *методы, которые базируются на покрытии* потока данных. Эти методы эффективнее *методов минимизации* и почти столь же эффективны, как *безопасные методы*. В то же время, они могут требовать, по крайней мере, такого же времени на *анализ*, как и наиболее эффективные *безопасные методы*, и, следовательно, могут обходиться дороже *безопасных методов* и намного дороже других *методов минимизации*. Они имеют тенденцию к включению избыточных тестов в набор регрессионных тестов для покрытия зависящих от изменений пар определения-использования, что, в некоторых случаях, ведет к большому числу отобранных тестов. Этот факт зафиксирован экспериментально.

Методы, основанные на использовании потока данных, могут быть полезны и для других *задач регрессионного тестирования*, кроме *отбора тестов*, например, для нахождения элементов P, недостаточно тестируемых T'.

*Метод стопроцентного покрытия* измененного кода аналогичен *методу минимизации*. Так, для примера таблицы c [Рис. 12.1](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1446?page=2#image.12.1) существует 4 способа отобрать 2 теста в соответствии с этим критерием. Одного теста недостаточно. Результаты сравнения методов *выборочного регрессионного тестирования* приведены в [Табл. 12.1](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1446?page=2#table.12.1).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Таблица 12.1. Сравнение методов *выборочного регрессионного тестирования* | | | | |
| **Класс методов** | **Случайные** | **Безопасные** | **Минимизации** | **Покрытия** |
| **Полнота** | От 0% до 100% | 100% | < 100% | < 100% |
| **Размер набора тестов** | Настраивается | Большой | Небольшой | Зависит от параметров метода |
| **Время выполнения метода** | Пренебрежимо мало | Значительное | Значительное | Значительное |
| **Перспективные свойства методов регрессионного тестирования** | Отсутствие средства поддержки регрессионного тестирования | Высокие требования по качеству | Стоимость пропуска ошибки невелика | Набор исходных тестов создается по критерию покрытия |

**Вывод:**

Чтобы работать в Ruby со временем следует использовать класс Time. Для того, чтобы записать что-то в файл, нам этот файл нужно сперва открыть. C той лишь разницей, что мы используем другой ключ: вместо "r:UTF-8" мы напишем "a:UTF-8", потому что файл нам надо открыть для «добавления» (по-англ. append) строк. Если такой файл есть, то всё, что мы запишем в файл, будет дописано в конец файла, если же такого файла ещё нет, то будет создан новый файл и всё, что мы запишем в него будет сохранено в этом новом файле.

**Вопросы для самоконтроля:**

1. Как производить запись в файлы?
2. Как работать с классом Time?
3. Какой атрибут следует ставить в опциях метода при записи в файл?

# Лекция 13. Регрессионное тестирование: методики, не связанные с отбором тестов и методики порождения тестов

### Интеграционное регрессионное тестирование

С появлением новых направлений в разработке программного обеспечения (например, объектно-ориентированного программирования), поощряющих использование большого количества маленьких процедур, повышается важность обработки межмодульного влияния изменений кода для методик уменьшения стоимости *регрессионного тестирования*. Для решения этой задачи необходимо рассматривать зависимости *по* глобальным переменным, когда переменной в одной или нескольких процедурах присваивается *значение*, которое затем используется во многих других процедурах. Такую зависимость можно рассматривать как зависимость между процедурами *по* потоку данных. Также возможны межмодульные зависимости *по* ресурсам, например *по* памяти, когда *ресурс* разделяется между несколькими процедурами. Отметим, что при системном регрессионном тестировании зависимости такого рода можно игнорировать.

Если изменение спецификации требований затрагивает глобальную переменную, могут потребоваться новые модульные тесты. В противном случае, повторному выполнению подлежат только модульные тесты, затрагивающие как измененный код, так и *операторы*, содержащие ссылку на глобальную переменную.

*Брандмауэр* можно определить как *подмножество* *графа вызовов*, содержащее измененные и зависящие от изменений процедуры и интерфейсы. Методы *отбора тестов*, использующие *брандмауэр*, требуют повторного *интеграционного тестирования* только тех процедур и интерфейсов, которые непосредственно вызывают или вызываются из измененных процедур.

### Регрессионное тестирование объектно-ориентированных программ

*Объектно-ориентированный подход* стимулирует новые приложения методик выборочного повторного тестирования. Действительно, при изменении класса необходимо обнаружить в наборе тестов класса только тесты, требующие повторного выполнения. Точно так же при порождении нового класса из существующего необходимо определить тесты из *множества* тестов базового класса, требующие повторного выполнения на классе-потомке. Хотя благодаря инкапсуляции *вероятность* ошибочного взаимодействия объектно-ориентированных модулей кода уменьшается, тем не менее, возможно, что тестирование прикладных программ выявит ошибки в методах, не найденные при *модульном тестировании* методов. В этом случае необходимо рассмотреть все прикладные программы, использующие измененный *класс*, чтобы продемонстрировать, что все существующие *тесты, способные* обнаруживать ошибки в измененных классах, были запущены повторно и было выбрано безопасное множество тестов. При повторном тестировании прикладных программ, классов или их наследников применение методик выборочного повторного тестирования к существующим наборам тестов может принести немалую пользу.

В объектно-ориентированной программе *вызов метода* во *время выполнения* может быть сопоставлен любому из ряда методов. Для заданного вызова мы не всегда можем статически определить метод, с которым он будет связан. Выборочные методы повторного тестирования, которые полагаются на статический *анализ*, должны обеспечивать *механизмы* для разрешения этой неопределенности.

### Уменьшение объема тестируемой программы

Еще один *путь* сокращения затрат на *регрессионное тестирование* состоит в том, чтобы вместо повторного тестирования (большой) измененной программы с использованием соответственно большого числа тестов доказать, что измененная *программа* адекватно тестируется с помощью выполнения некоторого (меньшего) числа тестов на остаточной программе. Остаточная *программа* создается путем использования графа зависимости системы вместо графа потока управления, что позволяет исключить ненужные зависимости между компонентами в пределах одного пути графа потока управления. Так, корректировка какого-либо оператора в идеале должна приводить к необходимости тестировать остаточную программу, состоящую из всех операторов исходной программы, способных повлиять на этот оператор или оказаться в сфере его влияния. Для получения остаточной программы необходимо знать *место* корректировки (в терминах операторов), а также информационные и *управляющие* связи в программе. Этот подход работает лучше всего для малых и средних изменений больших программ, где высокая *стоимость* *регрессионного тестирования* может заставить вообще отказаться от его проведения. Наличие дешевого метода остаточных программ, обеспечивающего такую же степень покрытия кода, делает *регрессионное тестирование* успешным даже в таких случаях.

Метод остаточных программ имеет ряд ограничений. В частности, он не работает при переносе программы на машину с другим процессором или объемом памяти. Более того, он может давать неверные результаты и на той же самой машине, если поведение программы зависит от адреса ее начальной загрузки, или если для остаточной программы требуется меньше памяти, чем для измененной, и, соответственно, на остаточной программе проходит тест, который для измененной программы вызвал бы ошибку нехватки памяти. Исследования метода на программах небольшого объема показали, что выполнение меньшего количества тестов на остаточной программе не оправдывает затрат на *отбор тестов* и уменьшение объема программы. Однако для программ с большими наборами тестов это не так.

Для теста 1 [рис. 12.1](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1448) для функции Equation остаточная *программа* выглядит так, как показано в [Табл. 13.1](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1448?page=1#table.13.1). *Нумерация* строк оставлена такой же, как в исходной программе. Таким образом, можно заметить, что были удалены строки 6 и 7, которые не затрагиваются тестом 1 в ходе его выполнения, а также строки 3 и 8, содержащие *вычисление* предикатов, которые в ходе выполнения теста всегда истинны. *Запуск* теста на полной измененной программе и на остаточной программе приводит к активизации одних и тех же операторов, поэтому выигрыша во времени получить не удается, однако за счет сокращения объема программы уменьшается *время компиляции*. Для нашего примера этот выигрыш незначителен и не оправдывает затрат на *анализ*, необходимый для уменьшения объема. Таким образом, рассмотренная технология рекомендуется к применению, прежде всего, в случаях, когда *стоимость* компиляции относительно высока.

|  |  |
| --- | --- |
| Таблица 13.1. Остаточная программа | |
| **№** | **Строка кода** |
| 1 | double Equation(int Print, float A, float B, float C, float& X1, float& X2) { |
| 2 | float D = B \* B - 4.0 \* A \* C; |
| 4 | X1 = (-B + sqrt(D)) / 2.0 / A; |
| 5 | X2 = (-B - sqrt(D)) / 2.0 / A; |
| 9 | printf("Solution: %f, %f\n", X1, X2); |
| 10 | return D; |
| 11 | } |

Сведения о *методике уменьшения объема тестируемой программы* приведены в [Табл. 13.2](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1448?page=1#table.13.2).

|  |  |
| --- | --- |
| Таблица 13.2. Результаты применения методики уменьшения объема | |
| **Характеристика** | **Изменение в результате применения методики** |
| Время компиляции тестируемой программы | Уменьшается |
| Время выполнения тестируемой программы | Не изменяется |
| Время работы метода отбора | Увеличивается |
| Риск пропуска ошибок | Увеличивается |
| Результаты применения методики на практике | Отрицательные |

**[Дальше >>](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1448?page=2)**

### Методы упорядочения

***Методы упорядочения*** позволяют инженерам-тестировщикам распределить тесты так, что тесты с более высоким приоритетом исполняются раньше, чем тесты с более низким приоритетом, чтобы затем ограничиться выбором первых n тестов для повторного исполнения. Это особенно важно для случаев, когда тестировщики могут позволить себе повторное выполнение только небольшого количества регрессионных тестов.

Одной из проблем упорядочения тестов является отсутствие представлений о том, скольких тестов в конкретном проекте достаточно для отбора. В отличие от подхода минимизации, использующего все тесты минимального набора, оптимальное число тестов в упорядоченном наборе неизвестно. Возникает проблема баланса между тем, что необходимо делать в ходе *регрессионного тестирования*, и тем, что мы можем себе позволить. В итоге количество запускаемых тестов определяется ограничениями *по* времени и бюджету и порядком тестов в наборе. С учетом этих факторов следует запускать повторно как можно больше тестов, начиная с верхней строки списка упорядоченных тестов.

Возможное преимущество упорядочения тестов состоит в том, что сложность соответствующего алгоритма, O(n2) в наихудшем случае, меньше, чем сложность алгоритма минимизации, который в некоторых случаях может требовать экспоненциального времени выполнения.

Проблему упорядочения тестов можно сформулировать следующим образом:

* Дано: T - набор тестов, PT - набор перестановок T, f - функция из PT на множество вещественных чисел.
* Найти: набор T'\in PT такой, что:

(\forall T'') (T''\in PT)(T''\ne T') [f(T')\ge f(T'')]

В приведенном определении PT представляет собой множество всех возможных вариантов упорядочения - T, а f – *функция*, которая, будучи применена к любому такому упорядочению, выдает его *вес*. (Предположим, что большие значения весов предпочтительнее малых.)

Упорядочение может преследовать различные цели:

* Увеличение частоты обнаружения ошибок наборами тестов, то есть увеличение вероятности обнаружить ошибку раньше при выполнении регрессионных тестов из этих наборов.
* Ускорение процесса покрытия кода тестируемой системы и достижение требуемой степени покрытия кода на более ранних этапах процесса тестирования.
* Быстрейший рост вероятности того, что тестируемая система надежна.
* Увеличение вероятности обнаружения ошибок, связанных с конкретными изменениями кода, на ранних этапах процесса тестирования и т.п.

***Методы упорядочения*** планируют выполнение тестов в процессе *регрессионного тестирования* в порядке, увеличивающем их эффективность в терминах достижения заданной меры производительности. Обоснование использования упорядочивающего метода состоит в том, что упорядоченный набор тестов имеет большую вероятность достижения цели, чем тесты, расположенные по какому-либо другому правилу или в случайном порядке.

Различают два типа упорядочения тестов: общее и в зависимости от версии. Общее упорядочение тестов при данных программе P и наборе тестов T подразумевает нахождение порядка тестов T, который окажется полезным для тестирования нескольких последовательных измененных версий. Считается, что для этих версий итоговый упорядоченный набор тестов позволяет в среднем быстрее достигнуть цели, ради которой производилось упорядочение, чем исходный набор тестов. Однако имеется значительный объем статистических свидетельств в пользу наличия связи между частотой обнаружения ошибок и конкретной версией тестируемой программы: разные версии программы предоставляют различные возможности *по* упорядочению тестов. При регрессионном тестировании мы имеем дело с конкретной версией программного продукта и хотим упорядочивать тесты наиболее эффективно *по* отношению именно к этой версии.

Например, упорядочивать тесты можно *по* количеству покрываемых ими изменений кода. При безопасном отборе тестов из [рис. 12.1](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1448?page=2) будут выбраны тесты 1, 2 и 5, из которых наиболее приоритетным является тест 2, так как он затрагивает оба изменения, тогда как тесты 1 и 5 – только одно.

Сведения о *методике упорядочения тестов* суммированы в [Табл. 13.3](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1448?page=2#table.13.3).

|  |  |
| --- | --- |
| Таблица 13.3. Результаты применения методики упорядочения тестов | |
| **Характеристика** | **Изменение в результате применения методики** |
| Время работы метода отбора | Увеличивается незначительно |
| Частота обнаружения ошибок | Увеличивается |
| Скорость покрытия кода | Увеличивается |
| Результаты применения методики на практике | Положительные |

### Целесообразность отбора тестов

Поскольку в общем случае оптимальный *отбор тестов* (то есть выбор в точности тех тестов, которые обнаруживают ошибку) невозможен, соотношение между затратами на применение методов *выборочного регрессионного тестирования* и выигрышем от их использования является основным вопросом практического применения *выборочного регрессионного тестирования*. На основании оценки этого соотношения делается *вывод* о целесообразности *отбора тестов*.

Эффективное *регрессионное тестирование* представляет собой *компромисс* между качеством тестируемой программы и затратами на тестирование. Чем больше регрессионных тестов, тем полнее проверка *правильности программы*. Однако большее количество выполняемых тестов обычно означает увеличение финансовых затрат и времени на тестирование, что на практике не всегда приемлемо. Выполнение меньшего количества регрессионных тестов может оказаться дешевле, но не позволяет гарантировать сохранение качества.

Когда отдельные модули невелики и несложны, а связанные с ними наборы тестов также небольшие, простой повторный *запуск* всех тестов достаточно эффективен. При *интеграционном тестировании* это менее вероятно. В то время как тесты для отдельных модулей могут быть небольшими, тесты для групп модулей и подсистем достаточно велики, что создает предпосылки для уменьшения издержек тестирования. С другой стороны, с ростом размера приложений *стоимость* применения выборочной стратегии повторного тестирования может возрасти до неприемлемой величины. *Затраты* на необходимый для отбора *анализ* могут перевешивать экономию от прогона сокращенного набора тестов и анализа результатов прогона. Однако в области тестирования достаточно больших программ положительный баланс затрат и выгод вполне достижим.

Модель затрат и выгод при использовании выборочных стратегий *регрессионного тестирования* должна учитывать прямые и косвенные *затраты*. Прямые *затраты* включают отбор и выполнение тестов и *анализ* результатов. Косвенные *затраты* включают *затраты* на управление, сопровождение баз данных и разработку программных средств. Выгоды – это *затраты*, которых удалось избежать, не выполняя часть тестов. Чтобы метод *выборочного регрессионного тестирования* был эффективнее метода повторного прогона всех тестов, *стоимость* анализа при отборе подмножества тестов вкупе со стоимостью их выполнения и проверки результатов должна быть меньше, чем *стоимость* выполнения и проверки результатов исходного набора тестов.

Пусть T' – *подмножество* T, отобранное некоторой стратегией *выборочного регрессионного тестирования* М для программы P, |T'| - обозначает *мощность* T', s – средняя *стоимость* отбора одного теста в результате применения М к P для создания T', а r – средняя *стоимость* выполнения одного теста из T на P и проверки его результата. Тогда для того, чтобы выборочное *регрессионное тестирование* было целесообразным, требуется выполнение неравенства:

s|T'| < r(|T| - |T'|)

Применяя вышеупомянутую модель стоимости с целью анализа затрат, полезно условно разделять *регрессионное тестирование* на две фазы – предварительную и критическую. Предварительная фаза *регрессионного тестирования* начинается после выпуска очередной версии программного продукта; во время этой фазы разработчики расширяют функциональность программы и исправляют ошибки, готовясь к выпуску следующей версии. Одновременно тестировщики могут планировать будущее тестирование или выполнять задачи, требующие наличия только предыдущей версии программы, такие как сбор тестовых траекторий и *анализ* покрытия. Как только в программу внесены исправления, начинается критическая фаза *регрессионного тестирования*. В течение этой фазы *регрессионное тестирование* новой версии программы является доминирующим процессом, время которого обычно ограничено моментом поставки заказчику. Именно на критической фазе *регрессионного тестирования* наиболее важна *минимизация* затрат. При использовании выборочного метода регрессионного тестирования важно использовать факт наличия этих двух фаз, уделяя как можно больше внимания выполнению задач, связанных с анализом, в течение предварительной фазы, чтобы на критической фазе заниматься только прогоном тестов и уменьшить *вероятность* срыва сроков поставки. Тем не менее, важно понимать, что до внесения последнего изменения в код *анализ* может быть выполнен только частично.

Если не учитывать не очень больших затрат на *анализ* при использовании *детерминированных методов*, решение о применении конкретного метода *отбора тестов* будет зависеть от отношения стоимости выполнения большего количества тестов к цене пропуска ошибки, что зависит от *множества* факторов, специфических для каждого конкретного случая. При отсутствии ошибок сбережения пропорциональны уменьшению размера набора тестов и могут быть измерены в терминах процента выбранных тестов, |T'| / |T|.

Модели стоимости могут использоваться как при выборе наилучшей, так и для оценки пригодности конкретной стратегии. При анализе учитываются такие факторы, как размер программы (в строках кода), *мощность множества* регрессионных тестов и количество покрываемых элементов, задействованных исходным множеством тестов.

Общий метод исследования проблемы целесообразности *отбора тестов* состоит в нахождении или создании исходной и измененной версий некоторой системы и соответствующего набора тестов. В этих условиях применяется методика *отбора тестов*, и размер и эффективность выбранного набора тестов сравнивается с размером и эффективностью первоначального набора тестов. Результаты показывают, что применение методов отбора регрессионных тестов, в том числе и безопасных, не всегда целесообразно, поскольку *затраты* и выгоды от их использования изменяются в широком диапазоне в зависимости от многих факторов. На практике наборы, основанные на покрытии, обеспечивают лучшие результаты *отбора тестов*.

Разумеется, *отношение* покрытия – не единственный фактор, который может отразиться на целесообразности применения *выборочного регрессионного тестирования*. Для некоторых приложений создание условий для тестирования (в том числе *компиляция* и *загрузка* модулей и ввод данных) может обходиться намного дороже, чем вычислительные ресурсы для непосредственного исполнения тестируемой системы. Например, в телекоммуникационной промышленности *стоимость* создания тестовой лаборатории для моделирования реальной сети связи может достигать нескольких миллионов долларов.

Подсчет порога целесообразности помогает определить, может ли *отбор тестов* вообще быть целесообразен для данного программного изделия и набора тестов. Однако даже в случаях, когда *значение* порога целесообразности указывает, что *отбор тестов* может быть целесообразен, он не обязательно будет таковым; результат зависит от параметров набора тестов, таких как размер набора, характеристики покрытия кода, уровень подробности и *время выполнения* тестов, а также от местоположения изменений. Существенно повлиять на общую оценку могут *затраты* на оплату труда тестового персонала, доступность свободного машинного времени для *регрессионного тестирования*, доступность стенда, на котором развернуто *программное обеспечение* приложения и т.п. Отметим, что *стоимость* *прогона тестов* связана не столько с размером программы, сколько с ограничениями на допустимое время прогона.

В некоторых случаях, когда число тестов, отброшенных выборочным методом *регрессионного тестирования* незначительно, но его применение тем не менее заслуживает внимания. Дело в том, что любое сокращение высокозатратного времени использования тестовой лаборатории особенно важно, а для *отбора тестов* используются другие ресурсы. Подобные обстоятельства необходимо включать в оценку стоимости анализа путем учета не только стоимости эксплуатации ресурса, но и таких факторов как время суток, день недели, время, оставшееся до выпуска очередной версии продукта и т.п. В этом случае модель стоимости должна соблюдать баланс между высокой стоимостью *прогона тестов* в тестовой лаборатории и относительно небольшой стоимостью проведения анализа на незанятых компьютерах.

Для некоторых программ и наборов тестов выборочное тестирование неэффективно, так как порог целесообразности превышает число тестов в наборе. В таких случаях методы *отбора тестов*, независимо от того, насколько успешно они уменьшают число тестов, требующих повторного выполнения, не могут давать экономию. Этот результат отражает тот факт, что целесообразность отбора зависит как от стоимости анализа, так и от стоимости выполнения тестов. Возможность достижения экономии при отборе регрессионных тестов для конкретной системы программного обеспечения и конкретного набора тестов должна оцениваться комплексно с учетом всех влияющих на решение факторов.

Стоит заметить, что целесообразность применения выборочного метода *регрессионного тестирования* нельзя воспринимать как нечто само собой разумеющееся. Следует очень осторожно подходить к оценке целесообразности отбора повторно прогоняемых тестов. В ряде случаев, когда или получаемое число остаточных тестов близко к первоначальному их количеству, или накладные *расходы* на повторное тестирование незначительны, выгоднее прогонять заново все тесты, особенно если *прогон тестов* полностью автоматизирован.

### Функции предсказания целесообразности

На практике не существует способа в точности предсказать, сколько тестов будет выбрано при минимизации (или *по* результатам применения любой другой методики *отбора тестов*). Когда количество тестов, отсеянных *по* результатам отбора, незначительно, ресурсы, потраченные на *отбор тестов*, пропадают впустую. В таких случаях говорят, что выборочное *регрессионное тестирование* нецелесообразно. Чтобы выяснить, заслуживает ли внимания попытка применения выборочного метода *регрессионного тестирования*, необходимо использовать прогнозирующую функцию.

Прогнозирующие функции основаны на покрытии кода. Для их вычисления используется *информация* о доле тестов, активирующих покрываемые сущности – *операторы*, ветви или функции, – что позволяет предсказать количество тестов, которые будут выбраны при изменении этих сущностей. Существует как *минимум* одна прогнозирующая *функция*, которая может быть использована для предсказания целесообразности применения безопасной стратегии *выборочного регрессионного тестирования*.

Пусть P – тестируемая система, S – ее спецификация, T – набор регрессионных тестов для P, а |T| означает число отдельных тестов в T. Пусть М – выборочный метод *регрессионного тестирования*, используемый для отбора подмножества T при тестировании измененной версии P ; М может зависеть от P, S, T, информации об исполнении T на P и других факторов. Через E обозначим набор рассматриваемых М сущностей тестируемой системы. Предполагается, что T и E непустые, и что каждый синтаксический элемент P принадлежит, *по* крайней мере, одной сущности из E. *Отношение* coversM(t, E) определяется как *отношение* покрытия, достигаемого методом М для P. Это *отношение* определено над TxE и справедливо тогда и только тогда, когда выполнение теста t на P приводит к выполнению сущности e как *минимум* один раз. *Значение* термина "выполнение" определено для всех *типов сущностей* P. Например, если e – *функция* или *модуль* P, e выполняется при вызове этой функции или модуля. Если e – простой оператор, условный оператор, пара определения-использования или другой вид элемента пути в графе выполнения P, e выполняется при выполнении этого элемента пути. Если e – *переменная* P, e выполняется при чтении или записи этой переменной. Если e – тип P, e выполняется при выполнении любой переменной типа e. Если e – *макроопределение* P, e выполняется при выполнении расширения этого *макроопределения*. Если e – сектор P, e выполняется при выполнении всех составляющих его операторов. Соответствующие значения термина "выполнение" могут быть определены *по* аналогии для других *типов сущностей* P.

Для данной тестируемой системы P, набора регрессионных тестов T и выборочного метода *регрессионного тестирования* М можно предсказать, стоит ли задействовать М для *регрессионного тестирования* будущих версий P, используя информацию об отношении покрытия coversM, достигаемого при использовании М для T и P. Прогноз основан на метрике стоимости, соответствующей P и T. Относительно издержек принимаются некоторые упрощающие предположения.

Пусть EC обозначает множество покрытых сущностей:

E^{C} = \{ e\in E | (\exists t\in T)(covers_{М}(t, E))\} .

Обозначение |EC| используется для числа покрытых сущностей. Иногда удобно представить зависимость coversM(t, E) в виде бинарной матрицы C, строки которой представляют элементы T, а столбцы – элементы E. При этом элемент Ci,j матрицы C определяется следующим образом:

Ci, j = 1, если coversМ(i, j)

Ci, j = 0, иначе

Степень накопленного покрытия, обеспечиваемого T, то есть общее число единиц в матрице C, обозначается CC:

|T| |E|
\\
CC = \Sigma    \Sigma  C_{i,j}
\\
    i=1 \ j=1

Отметим, что если ограничиться включением в C только столбцов, соответствующих покрытым сущностям EC, накопленное покрытие CC останется неизменным. В частности, для всех непокрытых сущностей u Ci,u равно нулю для всех тестов i (так как coversM(i, u) ложно для всех таких случаев). Следовательно, ограничение на EC при вычислении суммы, определяющей CC, приводит только к исключению слагаемых, равных нулю.

Пусть TM – *подмножество* T, выбранное М для P, и пусть |TM| обозначает его *мощность*, тогда T_{M} = \{ t\in T | M выбирает \  t\}. Пусть sM – удельная *стоимость* отбора одного теста для TM при применении М к P, и пусть r – удельная *стоимость* выполнения одного теста из T на P и проверки его результата. М целесообразно использовать в качестве метода *отбора тестов* тогда и только тогда, когда:

sM|TM| < r(|T| - |TM|), то есть *стоимость* анализа, необходимого для отбора TM, должна быть меньше стоимости прогона невыбранных тестов, T\supseteq T_{M}.

Оценка ожидаемого числа тестов, требующих повторного запуска, обозначается NM и вычисляется следующим образом:

Nm = CC/|E|

Использование этой прогнозирующей функции предполагается только в случаях, когда цель выборочной стратегии *регрессионного тестирования* состоит в повторном выполнении всех тестов, затронутых изменениями, то есть используется *безопасный метод* *отбора тестов*. Несколько усовершенствованный вариант оценки NM, использующий в качестве пространства сущностей EC вместо E:

Ncm = CC/|Ec|

Прогнозирующая *функция* для доли набора тестов, требующей повторного выполнения, то есть для |TM| / |T|, обозначается \pi _{М}:

\pi _{m}= N_{m}^{c}/|T| = CC/|E^{c}||T|

Прогнозирующая *функция* \pi _{М} полагается непосредственно на информацию о покрытии. Главные предпосылки, лежащие в основе применения прогнозирующей функции, таковы:

* *Целесообразность применения выборочного метода регрессионного тестирования* и, как следствие, наша способность к предсказанию целесообразности, непосредственно зависит от доли тестового набора, выбираемой для выполнения методом *регрессионного тестирования*.
* Эта доля в свою очередь непосредственно зависит от отношения покрытия.

*Точность* прогнозирующей функции на практике может значительно меняться от версии к версии. Проблема точности может оказаться достаточно серьезной, тем не менее, поскольку прогнозирующая *функция* используется для долговременного предсказания поведения метода на протяжении нескольких версий, применение средних значений считается допустимым. *Отношение* coversM(t, e) в ходе сопровождения изменяется очень слабо. *По* этой причине *информация*, полученная в результате анализа единственной версии, может оказаться достаточной для управления *отбором тестов* на протяжении нескольких последовательных новых версий.

Существуют факторы, влияющие на целесообразность *отбора тестов*, но не учитываемые прогнозирующей функцией. Один из подходов улучшения качества прогноза состоит в использовании информации об истории изменений программы, которую зачастую можно получить из системы управления конфигурацией.

Например, в [рис. 12.1](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1448?page=4) прогнозирующая *функция* может быть подсчитана как *отношение* общего количества звездочек в таблице к количеству строк таблицы, т.е. числу покрываемых сущностей. Эта величина составляет 42/11=3.8, т.е. *безопасный метод* будет отбирать в среднем около 4 тестов. Сведения о методике предсказания суммированы в [Табл. 13.4](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1448?page=4#table.13.4).

|  |  |
| --- | --- |
| Таблица 13.4. Результаты применения методики предсказания | |
| **Характеристика** | **Изменение в результате применения методики** |
| Время работы метода отбора в случае, если выборочное тестирование целесообразно | Увеличивается незначительно |
| Время работы метода отбора в случае, если выборочное тестирование нецелесообразно | Уменьшается до пренебрежимо малой величины |
| Снижение точности предсказания от версии к версии | Зависит от объема изменений |
| Результаты применения методики на практике | Положительные (ошибка в 0.8%) |

### Порождение новых тестов

Порождение новых тестов при структурном регрессионном тестировании обычно обусловлено недостаточным уровнем покрытия. Новые тесты разрабатываются так, чтобы задействовать еще не покрытые участки исходного кода. Процесс прекращается, когда уровень покрытия кода достигает требуемой величины (например, 80%). Разработка новых тестов при функциональном регрессионном тестировании является менее тривиальной задачей и обычно связана с вводом новых требований либо с желанием проверить некоторые сценарии работы системы дополнительно.

Основой большинства программных продуктов для управляющих применений, находящихся в промышленном использовании, является цикл обработки событий. *Сценарий* работы с системой, построенной *по* такой архитектуре, состоит из последовательности транзакций, управление после обработки каждой транзакции вновь передается циклу обработки событий. Выполнение транзакции приводит к изменению состояния программы; в результате некоторых транзакций происходит *выход* из *цикла* и *завершение работы* программы. Тесты для таких программ представляют собой последовательность транзакций.

Развитие программного продукта от версии к версии влечет за собой появление новых состояний. Поскольку большинство тестов легко может быть расширено путем добавления дополнительных транзакций в *список*, новые тесты можно создавать путем *суперпозиции* уже имеющихся, с учетом информации об изменении состояния тестируемой системы в результате *прогона теста*. Этот подход позволяет указать, какого рода новые тесты с наибольшей вероятностью обнаружат ошибки.

Обозначим тестируемую программу P, а множество ее тестов T = {t1, t2, … , tn}. Будем считать, что состояние тестируемой программы s определяется совокупностью значений некоторого подмножества глобальных и локальных переменных. При создании новых тестов будем рассматривать состояния программы перед запуском теста ( s0 ) и после его окончания ( sj ). Информацию об этих состояниях необходимо собирать для каждого теста *по* результатам запуска на предыдущей версии продукта. *Методика порождения новых тестов* на основе *анализа "подозрительных" состояний* сводится к описанной ниже последовательности действий.

1. Вычисление списка глобальных и локальных переменных, определяющих состояние программы s.
2. Сбор информации (на основе анализа профиля программы, полученного на предыдущей версии продукта i-1, для каждого существующего теста tj ) о состояниях программы перед запуском теста и после его окончания (т.е. s0 и sj ). Множество таких состояний обозначается S_{i - 1}: S_{i – 1} = s_{0}\cup \{ s_{j} | \forall j\}
3. Выполнение на текущей версии продукта i новых и выбранных регрессионных тестов из множества T’. По аналогии с Si–1 вычисляется множество Si, которое сохраняется под управлением *системы контроля версий*.
4. Оценка "подозрительных" с точки зрения наличия ошибок множества новых по сравнению с предыдущими версиями состояний Ni в соответствии со следующей формулой: Ni = Si \ Si - 1
5. Анализ состояний множества Ni, в которых дальнейшая работа продукта невозможна в соответствии со спецификацией. Предмет анализа - определить создаются ли эти состояния в результате выполнения тестов, проверяющих нештатные режимы работы продукта, или каких-либо других тестов. В последнем случае фиксируется ошибка.
6. Исключение нештатных состояний из множества Ni.
7. Переход к шагу 10, если новых состояний, допускающих продолжение выполнения программы, не обнаружено, т.е. N_{i}=\varnothing.
8. Для каждого состояния множества Ni вычисление вектора отличия от исходного состояния s0, т.е. множества переменных, измененных по сравнению с s0.
9. Модификация множества измененных строк исходного кода P на основе информации об измененных переменных и использование какой-либо методики *отбора тестов* для *выборочного регрессионного тестирования*.
10. Повторное выполнение шагов 3-9 до достижения состояния N_{i}=\varnothing либо до истечения времени, отведенного на *регрессионное тестирование*. Использование методов разбиения на классы эквивалентности для досрочного принятия решения о прекращении цикла тестирования, если ни один из тестов, созданных на очередном этапе, не принадлежит к новому классу эквивалентности.

Для приведенной методики организации тестирования, когда новые тесты получаются в результате *суперпозиции* уже имеющихся, целесообразно в качестве исходных тестов, т.е. тех "кирпичиков", из которых будут строиться тесты в дальнейшем, брать тесты, заключающие всего одну элементарную проверку. Это помогает избежать избыточности при многократном слиянии тестов, когда искомые "подозрительные" ситуации возникают в ходе работы теста, но не анализируются, так как не повторяются при его завершении.

Использование описанной методики позволяет в программном комплексе находить ошибки, не обнаруживаемые исходным набором тестов. Отметим, что применение предложенного подхода невозможно для программ, понятие состояния для которых не определено.

**Вывод:**

Программные ошибки — наверное, самая обсуждаемая тема среди программистов, так как процесс написания программ связан с решением довольно творческих задач по относительно строгим правилам. Грубо говоря, программу вы придумываете сами, но пишете её по строгим законам языка и технологии, поэтому если вы нарушаете какие-то правила, ваша программа в определённых ситуациях будет ломаться и не давать нужного результата. Но как говорится, волков бояться — в лес не ходить. Вот и ошибок в программировании бояться не стоит, нужно лишь для себя понимать, как реагировать и предупреждать разные виды ошибок.

**Вопросы для самоконтроля:**

1. Как обрабатывать исключения в Ruby?
2. Для чего нужна команда rescu?
3. Для чего нужна команда ensure?

# Лекция 14. Регрессионное тестирование: алгоритм и программная система поддержки

### Методика регрессионного тестирования

***Методика*** предназначена для эффективного решения задачи выборочного повторного тестирования. Ее исходными данными являются: программа P и ее модифицированная версия P', критерий тестирования C, множество (набор) тестов T, ранее использовавшихся для тестирования P, информация о покрытии элементов P (M(P,C)) тестами из T. Необходимо реализовать эффективный способ, гарантирующий достаточную степень уверенности в правильности P', используя тесты из T .

*Методика* строится на основе *сочетания* процедур обычного и регрессионного тестирования

Рассмотрим процедуру обычного тестирования. В ней для получения информации о тестируемых объектах в ходе тестирования необходимо установить соответствие между покрываемыми элементами и тестами для их проверки. Соответственно, процедура тестирования должна включать приведенную ниже последовательность действий:

1. Определить требуемые функциональные возможности программы с использованием, например, метода разбиения на классы эквивалентности.
2. Создать тесты для требуемых функциональных возможностей.
3. Выполнить тесты.
4. В случае необходимости - создать и выполнить дополнительные тесты для покрытия оставшихся (еще не покрытых) структурных элементов (предварительно установив их соответствие функциональным требованиям).
5. Создать базу данных тестов программы.

*По* аналогии с обычным тестированием, процедура регрессионного тестирования в процессе сопровождения состоит из следующих этапов:

1. Использование функции предсказания целесообразности. Если прогнозируемое количество выбранных тестов больше, чем порог целесообразности, провести повторный прогон всех тестов. В противном случае перейти к шагу 2.
2. Идентификация изменений \Delta P в программе P' (и множества \Delta М измененных покрываемых элементов) и установление взаимно однозначного соответствия между покрываемыми элементами М(P, C) и М(P', C) в соответствии с изменениями: \Delta M = (M(P, C) \setminus  M(P', C))\cup (M(P', C) \setminus  M(P, C))
3. Выбор T'\subseteq T - подмножества исходных тестов, потенциально способных выявить связанные с изменениями ошибки в P', для повторного выполнения на P', с использованием результатов, полученных в пункте 2. Это подмножество можно упорядочить, а также указать число тестов, выполнения которых достаточно для соответствия какому-либо критерию минимизации. Для безопасных методов отбора тестов множество T' удовлетворяет следующим ограничениям: t_{i}\in T, t_{i}\notin T' \Rightarrow  P(t_{i})\equiv P'(t_{i})
4. Применение подмножества T' для регрессионного тестирования измененной программы P' с целью проверки результатов и установления факта корректности P' по отношению к T' (в соответствии с измененным техническим заданием), а также обновления информации о прохождении тестов из T' на P'.
5. В случае необходимости - создание дополнительных тестов для дополнения набора регрессионных тестов. Это могут быть новые функциональные тесты, необходимые для тестирования изменений в техническом задании или новых функциональных возможностей измененной программы; новые структурные тесты для активизации оставшихся (непокрытых) структурных элементов (предварительно установив их соответствие проверяемым функциональным требованиям).
6. Создание T'' - нового набора тестов для P', применение его для тестирования измененной программы, проверка результатов и установление факта корректности P' по отношению к T'', обновление информации о ходе исполнения теста и создание базы данных тестов измененной программы для хранения этой информации и выходных данных тестов. Удаление устаревших тестов. T'' формируется по следующему правилу: T'' = (T\cup T_{новые}) \setminus  T_{устаревшие}

**Вывод:**

Мы уже знакомы с обычными массивами. Можно сказать, что в обычных массивах «ключом» каждого элемента является его номер. То есть, чтобы вытащить нужный элемент из массива, достаточно знать его номер. Это не всегда удобно, ведь номер элемента в массиве ничего не говорит о назначении этого элемента.

Ассоциативные массивы наглядны, ими удобнее пользоваться, когда данные в массиве разнородны, каждый элемент, таким образом, приобретает свой смысл. Из кода становится понятно, для чего какие данные в составе массива используются. Также с помощью ассоциативных массивом очень удобно передавать в метод несколько значений, не заботясь об их порядке.

**Вопросы для самоконтроля:**

1. Чем отличаются массивы и хэши
2. Как найти все ключи?
3. Как найти все значения?
4. Какой итератор нужно использовать, чтобы работать с хэшами?

# Лекция 15. Система поддержки регрессионного тестирования

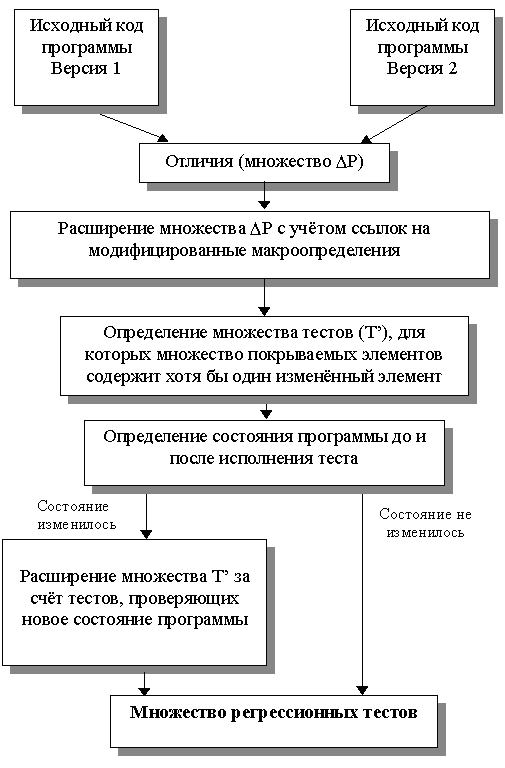
*Структура системы* поддержки регрессионного тестирования представлена на [рис. 14.1](https://www.intuit.ru/studies/professional_retraining/941/courses/48/lecture/1450?page=1#image.14.1). Исходный код обеих версий тестируемой программы хранится под управлением системы контроля версий. Для отбора тестов *по* методу покрытия точек использования неисполняемых определений средствами системы контроля версий создается *файл* различий, на основании которого вычисляется *список* добавленных, измененных и удаленных строк исходного кода, который является удобной формой представления *множества* \Delta P. Затем производится перебор этого списка. Если какая-либо строка списка представляет собой макроопределение, осуществляется *поиск* строк кода, содержащих использование этого *макроопределения* *по* всему тексту тестируемой программы; найденные строки присоединяются к множеству \Delta P. Расширенное множество \Delta P сопоставляется с результатами прогона тестов из *множества* T на предыдущей версии программы. Если в ходе выполнения какого-либо теста ti получала управление хотя бы одна строка, входящая в множество \Delta P, тест ti отбирается для повторного запуска.

При создании новых тестов *по* методу "подозрительных" состояний *функция* тестируемой программы, содержащая цикл обработки событий, дополняется операторами вывода значений глобальных и видимых локальных переменных. *Запуск* тестов из *множества* T' на профилированной версии программы позволяет получить *список* ее состояний. Этот *список* анализируется, и для каждого ранее не наблюдавшегося состояния вычисляется *список* переменных, изменившихся *по* сравнению с каким-либо известным состоянием. Множество \Delta P дополняется строками кода, где используются переменные из этого списка. Для каждого состояния указываются тесты, *запуск* которых необходим. Наконец, создается *список* рекомендованных новых тестов в форме, удобной для восприятия человеком.

*Выходные данные* каждой программы-обработчика доступны пользователю, что позволяет контролировать промежуточные результаты работы системы. К примеру, можно исключить из рассмотрения переменные, которые, хотя и изменяются в ходе выполнения программы, на ее состояние не влияют. *Архитектура* системы позволяет легко расширять функциональность; например, для поддержания какой-либо новой системы контроля версий достаточно создать один новый *модуль* объемом около 100 строк кода. Остальные модули можно использовать без изменений.

Типовой *сценарий* проведения регрессионного тестирования программ, написанных на языке C, с применением описанной выше системы состоит из следующих этапов:

1. Вычисляется множество \Delta P строк исходного кода, добавленных, удаленных или измененных по сравнению с предыдущей версией.
2. Множество \Delta P дополняется строками, непосредственно не изменявшимися, но содержащими ссылки на измененные макроопределения
3. Вычисляется упорядоченное множество регрессионных тестов T', для которых \forall i\in T' T_{i, j - 1} \cap  \Delta P \ne \varnothing, где Ti, j-1 - множество строк исходного кода продукта, получающих управление в ходе выполнения теста i на версии системы j-1. Тесты упорядочиваются по убыванию количества измененных строк в пути их исполнения.
4. Вычисляется список глобальных и локальных переменных, определяющих состояние программы s. Исходный код тестируемой программы модифицируется так, что информация о состоянии s (значения глобальных, статических и локальных переменных) выводится во внешний файл перед запуском теста и после его окончания.
5. Новые тесты и тесты из множества T' (регрессионные тесты) исполняются на текущей версии продукта j.
6. Тесты, проверяющие нештатные режимы работы продукта, т.е. создающие состояния, в которых дальнейшая работа продукта невозможна в соответствии со спецификацией требований, исключаются из рассмотрения. Если известно, что ни один тест не приводит к возникновению нештатных состояний, данный этап может быть опущен
7. Для каждого теста i вычисляется множество Tij.
8. Обрабатываются результаты выполнения тестов, и создается множество Sj, состоящее из начального состояния s0 и всех наблюдавшихся конечных состояний. Вычисляется множество Nj= Sj\Sj-1 всех новых по сравнению с предыдущими версиями состояний, которое является "подозрительным" с точки зрения наличия ошибок, и вектора их отличий от исходного состояния s0, т.е. переменные, измененные по сравнению с s0.
9. Множество измененных строк исходного кода \Delta P дополняется номерами строк, где используются заданные переменные
10. Если N_{j} =\varnothing, цикл работы завершается. Если N_{j}\ne \varnothing , следует перейти к шагу 4 или, если счетчик количества итераций работы системы превышает некоторое заданное предельное значение, известить об этом пользователя. Пользователь может принять решение о прекращении тестирования или пропуске некоторого числа циклов.



**Рис. 14.1.**Структура системы поддержки регрессионного тестирования.

Если этап 6 выполняется автоматически, а этап 7 можно опустить, возможна полная *автоматизация* регрессионного тестирования.

**Вывод:**

Наследование в Ruby позволяет использовать переменные и метода, которые определяются в родительском классе, в дочернем классе. При этом множественное наследование реализовано в языке с помозью механизма mixing. Оператор наследования представляет собой знак < .

**Вопросы для самоконтроля:**

1. Что такое наследование?
2. Как реализуется в Ruby множественное наследование?
3. С помощью какого оператора возможно отнаследоваться от определенного класса?

# Список источников

1. Оби Фернандес. Пусть Rails. Подробное руководство по созданию приложений в среде RubyonRails. – Пер. с англ. – СПб, 2009.

2. Руби С., Томас Д., Хенссон Д. Rails 4. Гибкая разработка веб-приложений. – СПб, 2014.

3. Р. С. Самарев. Создание простейших веб-приложений с помощью Ruby on Rails и AJAX. – М, 2015.

4. М. Хартл. RubyonRails для начинающих. – М, 2017.

5. www.goodprogrammer.ru

6. [www.rusrails.ru](http://www.rusrails.ru)

7. <https://www.intuit.ru/studies/professional_retraining/941/courses/48/info>