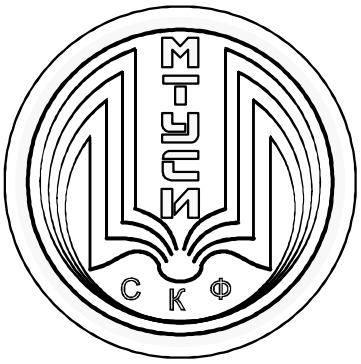
ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ
СЕВЕРО-КАВКАЗСКИЙ ФИЛИАЛ
ОРДЕНА ТРУДОВОГО КРАСНОГО ЗНАМЕНИ
ФЕДЕРАЛЬНОГО ГОСУДАРСТВЕННОГО
БЮДЖЕТНОГО ОБРАЗОВАТЕЛЬНОГО УЧРЕЖДЕНИЯ ВЫСШЕГО
ОБРАЗОВАНИЯ
«МОСКОВСКИЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ СВЯЗИ И
ИНФОРМАТИКИ»



КАФЕДРА ИНФОРМАТИКИ И ВЫЧИСЛИТЕЛЬНОЙ ТЕХНИКИ

Ткачук Е.О.

МЕТОДЫ ОТЛАДКИ И ТЕСТИРОВАНИЯ ПРОГРАММНЫХ ПРОДУКТОВ

Методическое пособие

для проведения практических занятий

Ростов-на-Дону

2019 г.

УДК 004.415.2
ББК 32.973.3
Т48
М54

Ткачук Е.О. Методы отладки и тестирования программных продуктов: Методическое пособие для проведения практических занятий. - Ростов-на-Дону: Северо-Кавказский филиал МТУСИ, 2019. – 57 с.: ил.

В пособии даются организационно-методические указания к практических занятий и порядок выполнения и оформления лабораторных работ.

Предназначено для студентов всех специальностей обеих форм обучения, изучающих дисциплину «Методы отладки и тестирования программных продуктов», а также может быть полезно всем остальным студентам, желающим самостоятельно изучать современные методы отладки и тестирования программных продуктов.

Составители:

доцент кафедры ИВТ Ткачук Е.О.

Рецензент: доц. кафедры ИВТ СКФ МТУСИ, к.т.н. доц. А.Н. Чикалов.

Издание рассмотрено и утверждено

на заседании кафедры ИВТ

26.08.2019 года (протокол № 1)

Отв. редактор _____

Оглавление

Практическое занятие 1. Разработка и предварительное тестирование “рабочего” приложения

Цель практического занятия:

Освоить работу с интегрированной средой разработки Lazarus, изучить средства создания программного обеспечения.

Задание

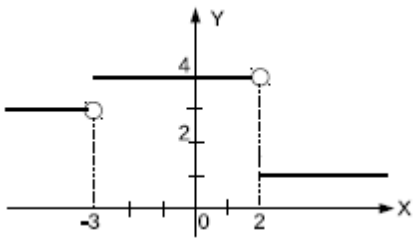
Разработать приложение с диалоговым оконным интерфейсом согласно индивидуальному варианту. Номер варианта определяется по порядковому номеру в журнале.

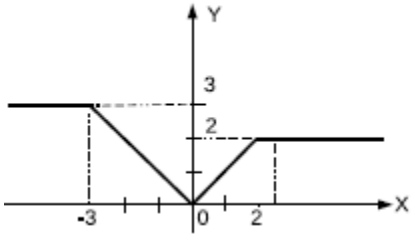
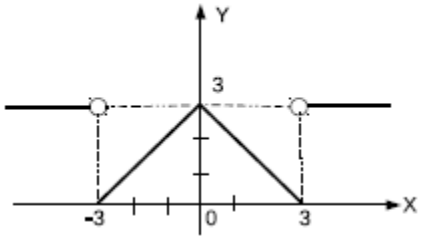
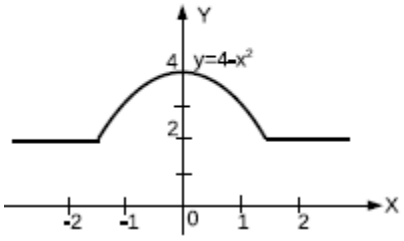
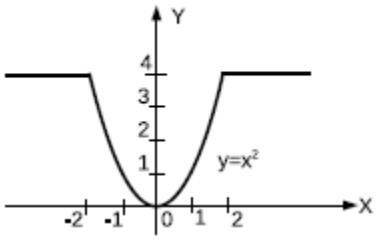
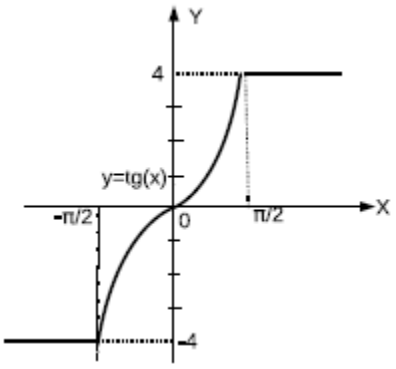
В качестве отчёта представить отчёт, содержащий описание выполненной работы, текст паскаль – модуля, скриншоты работы программы и файл архива, содержащий разработанный проект без exe – файла приложения.

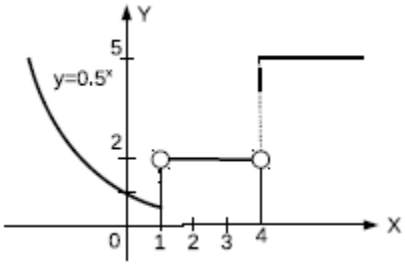
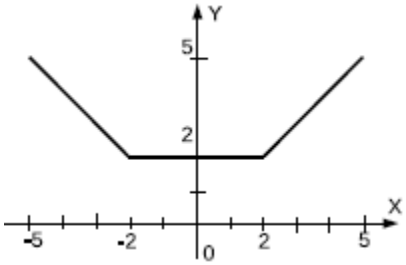
Индивидуальные варианты заданий

Часть 1, варианты 1- 8

Дано вещественное число *a.* Для функции $y = f(x)$, график которй представлен в таблице, вычислить значение $f(a)$.

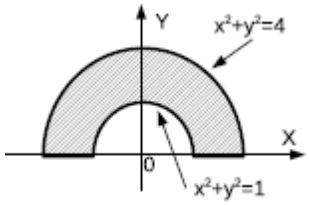
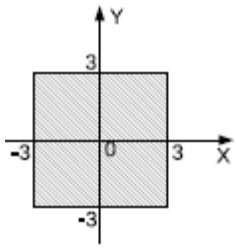
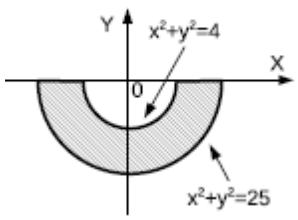
№ варианта	Задание
1	

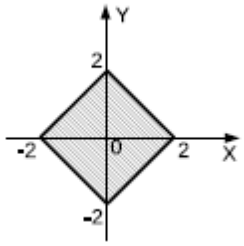
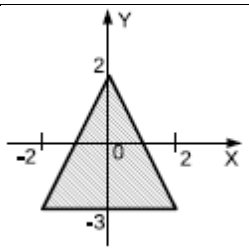
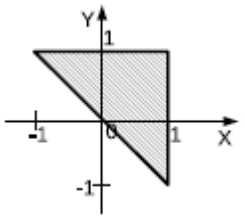
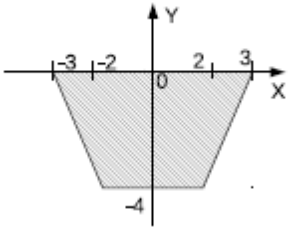
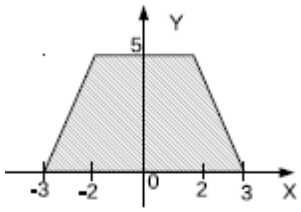
2	 Y 3 2 X -3 0 2
3	 Y 3 X -3 0 3
4	 Y 4 y=4-x² 2 X -2 -1 0 1 2
5	 Y 4 3 2 1 y=x² X -2 -1 0 1 2
6	 Y 4 y=tg(x) X -π/2 0 π/2 -4

7	
8	

Часть 2 Варианты 9 – 16

Даны вещественные числа *x* и y. Определить, принадлежит ли точка с координатами (*x*, y) заштрихованной части плоскости.

№ варианта	Задание
9	
10	
11	

12	
13	
14	
15	
16	

Часть 3. Варианты 17 – 26

17) Задан круг с центром в точке O(x0, y0) и радиусом R0 и точка A(x1, y1). Определить, находится ли точка внутри круга.
18) Определить, пересекаются ли параболы y = ax2 + bx + c и y = dx2 + mx + n. Если пересекаются, то найти точку пересечения.
19) Определить, пересекаются ли линии y = ax3 +bx2 +cx +d и y = kx3 +mx2 +
20) + nx + p. Если пересекаются, найти точку пересечения.

21) Задана окружность с центром в точке O(x0, y0) и радиусом R0. Найти точки пересечения линии с осью абсцисс.
22) Задана окружность с центром в точке $O(x_0, y_0)$ и радиусом R_0. Найти точки пересечения линии с осью ординат.
23) Определить, пересекаются ли линии $y = bx^2 + cx + d$ и $y = kx + m$. Если пересекаются, найти точки пересечения
24) Задана окружность с центром в точке $O(0,0)$ и радиусом R_0, и прямая $y = ax + b$. Определить, пересекаются ли прямая и окружность. Если пересекаются, найти точку пересечения.
25) Найти точки пересечения линии $ax^2 + bx + c$ с осью абсцисс.
26) Определить, пересекаются ли линии $y = ax^4 + bx^3 + cx^2 + dx + f$ и $y =$

$= bx^3 + mx^2 + dx + p$. Если пересекаются, найти точку пересечения.

Практическое занятие 2. Изучение классификации ошибок программирования

Классификация ошибок программирования

В настоящее время имеется достаточная статистика об ошибках в ПО, приводящих к отказам. В таблице 1 приведены основные причины отказов. Эти данные являются основой для построения математических моделей надежности программ, с целью ее оценки и прогнозирования, а также определения путей повышения безотказной работы.

Таблица 1

ПРИЧИНА ОШИБКИ	Частота %
Неполное или ошибочное задание	28
Отклонение от задания	12
Пренебрежение правилами программирования	10
Ошибочная выборка данных	10
Ошибочная логика или последовательность операций	12
Ошибочные арифметические операции	9
Нехватка времени для решения	4
Неправильная обработка прерываний	4
Неправильные const или исходные данные	3

Неточная запись	8

Ошибки программирования, более известные как «Баги» на жаргоне, бич любого разработчика программного обеспечения. Поскольку машины все чаще используются в автоматическом режиме, с бортовыми встраиваемыми системами или компьютерами, контролирующими их функционирование, программная ошибка может иметь серьезные последствия. Были случаи, когда космические челноки и самолеты, разбивались из-за ошибки в программном обеспечении во встраиваемом компьютерном оборудовании. Одна лазейка, оставленная в коде операционной системы, может обеспечить точку входа для хакеров, которые могут использовать эту уязвимость. К этим, ошибкам нужно относиться очень серьезно, так как мы все больше и больше полагаемся на компьютеры. Основные виды ошибок в программировании Компьютерное программирование это огромное поле с сотнями языков, которые используют миллионы приложений. Это программирование операционной системы, прикладное программирование, встроенное кодирование системы, веб-разработка, приложения для мобильных платформ, развитие программ, развернутых в интернете, научные вычисления. В таблице представлены основные виды ошибок.

Таблица 2

Тип ошибок программирования	**Описание**
Логическая ошибка	Это, пожалуй, наиболее серьезная из всех ошибок. Когда написанная программа на любом языке компилирует и работает правильно, но выдает неправильный вывод,

	недостаток заключается в логике основного программирования. Это ошибка, которая была унаследована от недостатка в базовом алгоритме. Сама логика, на которой базируется вся программа, является ущербной. Чтобы найти решение такой ошибки нужно фундаментальное изменение алгоритма. Вам нужно начать копать в алгоритмическом уровне, чтобы сузить область поиска такой ошибки.
Синтаксическая ошибка	Каждый компьютерный язык, такой как C, Java, Perl и Python имеет специфический синтаксис, в котором будет написан код. Когда программист не придерживаться "грамматики" спецификациями компьютерного языка, возникнет ошибка синтаксиса. Такого рода ошибки легко устраняются на этапе компиляции.
Ошибка компиляции	Компиляция это процесс, в котором программа, написанная на языке высокого уровня, преобразуется в машиночитаемую форму. Многие виды ошибок могут происходить на этом этапе, в том числе и синтаксические ошибки. Иногда, синтаксис исходного кода может быть безупречным, но ошибка компиляции все же может

	произойти. Это может быть связано с проблемами в самом компиляторе. Эти ошибки исправляются на стадии разработки.
Ошибки среды выполнения (RunTime)	Программный код успешно скомпилирован, и исполняемый файл был создан. Вы можете вздохнуть с облегчением и запустить программу, чтобы проверить ее работу. Ошибки при выполнении программы могут возникнуть в результате аварии или нехватки ресурсов носителя. Разработчик должен был предвидеть реальные условия развертывания программы. Это можно исправить, вернувшись к стадии кодирования.
Арифметическая ошибка	Многие программы используют числовые переменные, и алгоритм может включать несколько математических вычислений. Арифметические ошибки возникают, когда компьютер не может справиться с проблемами, такими как "Деление на ноль", или ведущие к бесконечному результату. Это снова логическая ошибка, которая может быть исправлена только путем изменения алгоритма.
Ошибки ресурса	Ошибка ресурса возникает, когда значение переменной переполняет

	максимально допустимое значение. Переполнение буфера, использование неинициализированной переменной, нарушение прав доступа и переполнение стека - примеры некоторых распространенных ошибок.
Ошибка взаимодействия	Они могут возникнуть в связи с несоответствием программного обеспечения с аппаратным интерфейсом или интерфейсом прикладного программирования. В случае веб-приложений, ошибка интерфейса может быть результатом неправильного использования веб-протокола.

Наиболее типичными симптомами появления ошибок в программе являются:

преждевременное окончание выполнения программы;

недопустимое увеличение времени некоторой последовательности команд одной из программ;

полная потеря или значительное искажение накопленных данных, необходимых для успешного выполнения решаемых задач;

нарушение последовательности вызова отдельных программ, в результате чего происходит пропуск необходимых программ;

искажение отдельных элементов данных (входных, выходных, промежуточных) в результате обработки искаженной исходной информации.

ЗАДАНИЕ

Для таблиц 1 и 2 составить примеры ошибочного программного кода, соответствующего данной ошибке.

Практическое занятие 3. Изучение отладочных средств

Цель практического занятия

Освоить работу с встроенным отладчиком, изучить категории ошибок, способы их обнаружения и устранения.

Тестирование и отладка программы

Чем больше опыта имеет программист, тем меньше ошибок в коде он совершает. Но, хотите верьте, хотите нет, даже самый опытный программист всё же допускает ошибки. И любая современная *среда разработки* программ должна иметь собственные инструменты для отладки приложений, а также для своевременного обнаружения и исправления возможных ошибок. Программные ошибки на программистском сленге называют *багами* (англ. *bug* - жук), а программы отладки кода - *дебаггерами* (англ. *debugger* - отладчик). Lazarus, как современная *среда разработки* приложений, имеет собственный встроенный отладчик, работу с которым мы разберем на этой лекции.

Ошибки, которые может допустить программист, условно делятся на три группы:

1. Синтаксические
2. Времени выполнения (run-time errors)
3. Алгоритмические

Синтаксические ошибки

Синтаксические ошибки легче всего обнаружить и исправить - их обнаруживает *компилятор*, не давая скомпилировать и запустить программу. Причем *компилятор* устанавливает *курсор* на ошибку, или после неё, а в окне сообщений выводит соответствующее сообщение, например, такое:

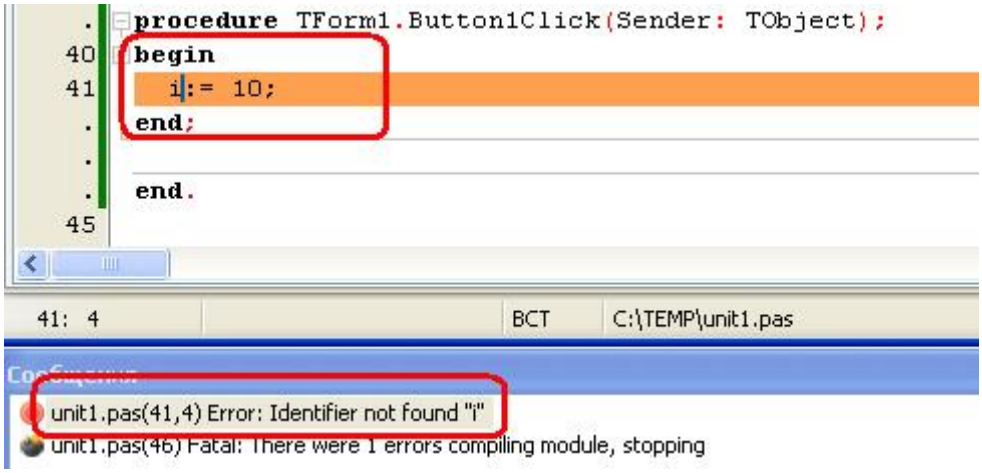


Рис. 2.1. Найденная компилятором синтаксическая ошибка - нет объявления переменной i

Подобные ошибки могут возникнуть при неправильном написании директивы или имени функции (процедуры); при попытке обратиться к переменной или константе, которую не объявляли (рис. 2.1); при попытке вызвать функцию (процедуру, переменную, константу) из модуля, который не был подключен в разделе uses; при других аналогичных недосмотрах программиста.

Как уже говорилось, *компилятор* при нахождении подобной ошибки приостанавливает процесс компиляции, выводит сообщение о найденной ошибке и устанавливает *курсор* на допущенную ошибку, или после неё. Программисту остается только внести исправления в *код программы* и выполнить повторную компиляцию.

Ошибки времени выполнения

Ошибки времени выполнения (run-time errors) тоже, как правило, легко устранимы. Они обычно проявляются уже при первых запусках программы, или во *время тестирования*. Если такую программу запустить из среды Lazarus, то она скомпилируется, но при попытке загрузки, или в момент совершения ошибки, приостановит свою работу, выведя на экран соответствующее сообщение. Например, такое:

Рис. 2.2. Сообщение Lazarus об ошибке времени выполнения

В данном случае *программа* при загрузке должна была считать в *память* отсутствующий *текстовый файл* **MyFile.txt**. Поскольку *программа* вызвала ошибку, она не запустилась, но в среде Lazarus процесс отладки продолжается, о чем свидетельствует сообщение в скобках в заголовке главного *меню*, после названия проекта. Программисту в подобных случаях нужно сбросить отладчик командой *меню* "**Запуск -> Сбросить отладчик**", после чего можно продолжить работу над проектом.

Ошибка времени выполнения может возникнуть не только при загрузке программы, но и во время её работы. Например, если бы попытка чтения несуществующего файла была сделана не при загрузке программы, а при нажатии на кнопку, то *программа* бы нормально запустилась и работала, пока *пользователь* не нажмет на эту кнопку.

Если программу запустить из самой *Windows*, при возникновении этой ошибки появится такое же сообщение. При этом если нажать "**OK**", *программа* даже может запуститься, но корректно работать все равно не будет.

Ошибки времени выполнения бывают не только явными, но и неявными, при которых *программа* продолжает свою работу, не выводя никаких сообщений, а программист даже не догадывается о наличии ошибки. Примером неявной ошибки может служить так называемая **утечка памяти**. Утечка памяти возникает в случаях, когда программист забывает

освободить выделенную под *объект память*. Например, мы объявляем переменную типа TStringList, и работаем с ней:

```
begin
  MySL:= TStringList.Create;
  MySL.Add('Новая строка');
end;
```

В данном примере программист допустил типичную для начинающих ошибку - не освободил *класс* TStringList. Это не приведет к сбою или аварийному завершению программы, но в итоге можно бесполезно израсходовать очень много памяти. Конечно, эта *память* будет освобождена после выгрузки программы (за этим следит *операционная система*), но утечка памяти во *время выполнения* программы тоже может привести к неприятным последствиям, потребляя все больше и больше ресурсов и излишне нагружая *процессор*. В подобных случаях после работы с объектом программисту нужно не забывать освобождать *память*:

```
begin
  MySL:= TStringList.Create;
  MySL.Add('Новая строка');
  ...; //работа с объектом
  MySL.Free; //освободили объект
end;
```

Однако *ошибки времени выполнения* могут случиться и во время работы с объектом. Если есть такой риск, программист должен не забывать про возможность обработки исключительных ситуаций. В данном случае вышеприведенный код правильней будет оформить таким образом:

```
begin
  try
```

```
    MySL:= TStringList.Create;
    MySL.Add('Новая строка');
    ...; //работа с объектом
  finally
    MySL.Free; //освободили объект, даже если была ошибка
  end;
end;
```

Итак, во избежание ошибок времени выполнения программист должен не забывать делать проверку на правильность ввода пользователем допустимых значений, заключать опасный код в блоки try…finally…end или try…except…end, делать проверку на существование открываемого файла функцией FileExists и вообще соблюдать предусмотрительность во всех слабых местах программы. Не полагайтесь на пользователя, ведь недаром говорят, что если в программе можно допустить ошибку, *пользователь* эту возможность непременно найдет.

Алгоритмические ошибки

Если вы не допустили ни синтаксических ошибок, ни ошибок времени выполнения, *программа* скомпилировалась, запустилась и работает нормально, то это еще не означает, что в программе нет ошибок. Убедиться в этом можно только в процессе её *тестирования*.

Тестирование - *процесс проверки работоспособности программы путем ввода в неё различных, даже намеренно ошибочных данных, и последующей контрольной проверке выводимого результата.*

Если *программа* работает правильно с одними наборами исходных данных, и неправильно с другими, то это свидетельствует о наличии алгоритмической ошибки. Алгоритмические ошибки иногда называют логическими, обычно они связаны с неверной реализацией алгоритма

программы: вместо "+" ошибочно поставили "-", вместо "/" - "*", вместо деления значения на 0,01 разделили на 0,001 и т.п. Такие ошибки обычно не обнаруживаются во *время компиляции*, *программа* нормально запускается, работает, а при анализе выводимого результата выясняется, что он неверный. При этом *компилятор* не укажет программисту на ошибку - чтобы найти и устранить её, приходится анализировать код, пошагово "прокручивать" его выполнение, следя за результатом. Такой процесс называется *отладкой*.

Отладка - *процесс поиска и устранения ошибок, чаще алгоритмических. Хотя отладчик позволяет справиться и с ошибками времени выполнения, которые не обнаруживаются явно.*

Работа с отладчиком

Давайте от теории перейдем к практике. Загрузите **Lazarus** с новым проектом, установите на форму простую кнопку и сохраните проект в папку **ЛР2**. Имена проекта, формы, модуля и кнопки изменять не нужно, оставьте имена, данные по умолчанию.

Далее, сгенерируйте событие OnClick для кнопки, в котором напишите следующий код:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i: Integer;
  st: TStringList;
begin
  //создаем список строк:
  st:= TStringList.Create;
  try
    //генерируем список:
    for i:= -3 to 3 do begin
      st.Append('10/'+IntToStr(i)+'='+FloatToStr(10/i));
```

```
    end;
    //выводим список на экран:
    ShowMessage(st.Text);
  finally
    //st будет освобождена даже в случае run-time ошибки:
    st.Free;
  end;
end;
```

Что мы тут делаем? Целочисленную переменную i используем в качестве счетчика для *цикла* for. Цикл производим от -3 до 3, то есть, 7 раз, включая ноль. В теле *цикла* мы делим 10 на *значение* i, результат оформляем в виде строки и добавляем к списку строк st. Выше говорилось, что подобные действия нужно заключать в блок обработки исключительных ситуаций try-finally-end, что мы и сделали.

Если вы внимательно изучали курс, то невооруженным глазом видите, что при четвертом проходе *цикла* произойдет ошибка времени выполнения - *деление* 10 на ноль. Такой очевидный пример больше всего подходит для знакомства с встроенным отладчиком, так как вы уже знаете, где будет ошибка, и сможете проанализировать работу отладчика. Поэтому притворимся, что не подозреваем об ошибке.

Итак, программу мы написали, сохранили, пора её компилировать. Нажмите кнопку "**Запустить**" на **Панели управления** (или **<F9>**). *Программа* нормально скомпилировалась и запустилась. Нажмем кнопку **Button1**. И тут же получаем ошибку:

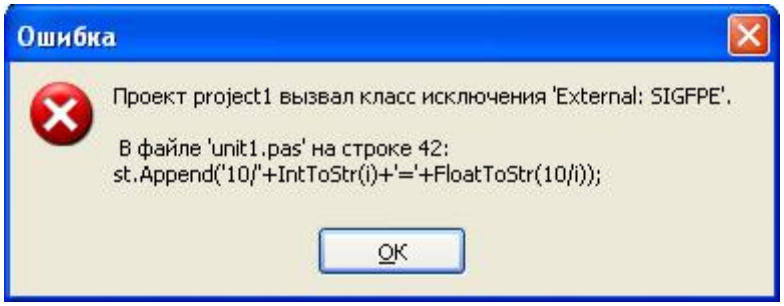


Рис. 2.3. Сообщение Lazarus об ошибке

Судя по сообщению, ошибка произошла во *время выполнения* кода 42-й строки. Ладно, нажмем "**ОК**" и командой "**Запуск -> Сбросить отладчик**" прекратим выполнение программы. Вернемся к коду и проанализируем 42-ю строку (если вы добавляли пустые строки, то у вас будет другой номер):

```
st.Append('10/'+IntToStr(i)+'='+FloatToStr(10/i));
```

Ну что, ничего криминального тут нет, почему же произошла ошибка? Код верный и должен был нормально выполняться… Когда вы заходите в подобный *тупик*, помочь вам может *здравый смысл* и встроенный отладчик. *Здравый смысл* говорит, что ошибка произошла где-то в теле *цикла* for. А чтобы воспользоваться отладчиком, нужно приостановить выполнение программы на этом цикле, чтобы потом построчно его продолжить. Для остановки работы программы служат так называемые **точки останова** (англ. *breakpoints*).

Точки останова - *это строки, перед выполнением которых отладчик приостанавливает выполнение программы, и ждет ваших дальнейших действий.*

Вы можете установить одну такую точку или несколько, в различных частях кода. Поскольку ошибка возникает в 42-й строке, разумней будет приостановить выполнение на предыдущей, 41-й строке. Переведите *курсор* на эту строку, на любое её *место*.

Установить точку останова можно разными способами:

- Командой главного меню "**Запуск -> Добавить точку останова -> Точка останова в исходном коде**". В открывшемся окне "**Параметры точки останова**" нажать "**ОК**".
- Щелкнуть по строке правой кнопкой, и в всплывающем меню выбрать "**Отладка -> Переключить точку останова**".
- Нажать "горячую клавишу" **<F5>**.
- Щелкнуть по нужной строке в левой части **Редактора кода**, где указаны номера строк.

Последние два способа наиболее удобны, но выбирать вам. В любом случае, строка с установленной точкой останова окрасится красным цветом:

Рис. 2.4. Строка с точкой останова

Снять точку останова удобней также последними двумя способами. *Точка останова* у нас есть, снова нажимаем кнопку "**Запустить**". *Программа* начинает свою работу, нажимаем кнопку "**Button1**".

Теперь *программа* не вывела ошибку, а приостановила свою работу и вывела на передний план **Редактор кодов** с выделенной серым цветом строкой, которая в данный момент готовится к выполнению:

Рис. 2.5. Строка, которая будет выполнена далее

Тут очень важно понимать, что *программа* была остановлена ДО выполнения этой строки, а не ПОСЛЕ неё. То есть, в настоящий момент переменной i еще не присвоено *значение* -3. Далее, мы можем выполнять с отладчиком различные действия, которые собраны в разделе главного *меню* "**Запуск**". Обычно требуется *пошаговое выполнение* программы. Для этого можно использовать команду "**Запуск -> Шаг в обход**" (или **<F8>**), "**Запуск -> Шаг с входом**" (или **<F7>**) или "**Запуск -> Шаг с выходом**" (или **<Shift+F8>**). "**Шаг в обход**" означает, что если в коде будет встречен вызов какой-нибудь функции или процедуры, отладчик выполнит их и остановится на следующей после вызова строке. При выборе "**Шаг со входом**", отладчик также пошагово будет выполнять и вызываемые функции-процедуры. "**Шаг с выходом**" подразумевает, что если в строке нет вызовов функций, то остановки происходят, как при "**Шаг в обход**". Если в строке есть *выражение*, то остановка происходит вначале перед строкой, затем перед вычислением каждой функции, чтобы мы имели возможность просмотреть значения параметров, передаваемых в функцию.

У нас вызовов функций нет, поэтому мы можем воспользоваться как **<F7>**, так и **<F8>** (чаще всего используют **<F8>** - Шаг в обход).

Итак, нажмем **<F8>**, и отладчик выполнит строку с точкой останова, и выделит серым следующую строку. Снова нажмем **<F8>**, и снова будет выделена эта строка - был выполнен шаг *цикла*. Нажав несколько раз **<F8>**, мы добьемся появления на экране всё той же ошибки, которая заблокирует дальнейшее выполнение программы. Становится понятно, что цикл нормально выполняется несколько проходов, после чего всё же возникает ошибка. Включаем логику: внутри *цикла* у нас изменяется только *переменная* **i**, значит, ошибка как-то связана с ней. А как узнать, как именно?

Здесь нам на помощь приходит еще один полезный инструмент отладчика - наблюдение за значениями переменных. Сбросьте программу

командой "**Запуск -> Сбросить отладчик**". Теперь снова нажмите кнопку "**Запустить**", а потом снова кнопку "**Button1**". Отладчик снова приостановил выполнение программы на строчке с циклом, однако не спешите нажимать **<F8>**. Для начала, добавим наблюдение над переменной **i**. Делается это командой "**Запуск -> Добавить наблюдение**", которая была недоступна, пока *программа* не начала выполняться. В строке "**Выражение**" укажите переменную **i**, и нажмите "**ОК**":

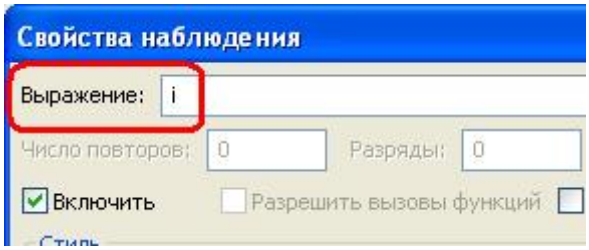


Рис. 2.6. Установка наблюдения за переменной

Теперь отладчик наблюдает за значениями переменной i, но нам от этого не легче - мы то не видим этих значений! Чтобы их увидеть, нужно вывести на экран окно **Списка наблюдений**. Делается это командой "**Вид -> Окна отладки -> Окно наблюдений**" или "горячими клавишами" **<Ctrl+Alt+W>**.

Расположите окно ниже **Редактор кода**, на *место* **Окна сообщений**. В этом окне вы сможете видеть *выражение* - нашу переменную i, и её текущее *значение*. Чтобы показать работу с отладчиком более наглядно, давайте добавим еще одно *выражение*, за которым будем наблюдать. Выберите "**Запуск -> Добавить наблюдение**" и в строке "**Выражение**" укажите не просто переменную, а *выражение*

10/i

которое у нас должно вычисляться внутри *цикла*. В окне **Списка наблюдения** вы увидите и переменную i, и *выражение*, а также их значения:

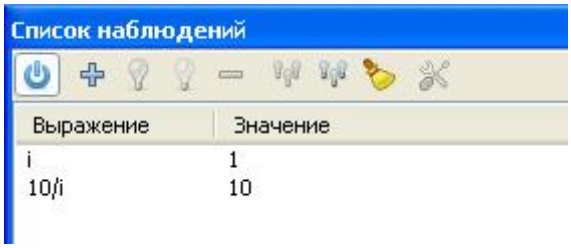


Рис. 2.7. Окно Списка наблюдений

Поскольку переменной i еще не было присвоено значения -3, то в колонке значений вы, скорее всего, увидите 1, которым по умолчанию была проинициализирована наша *переменная*. Соответственное *значение* будет и у выражения. Теперь мы готовы двигаться дальше. Нажимаем **<F8>**. В **Списке наблюдений** сразу же изменилась картина - i теперь равно -3, а *выражение* -3,3333…

Нажимаем **<F8>** ещё раз. Снова значения изменились, теперь i = -2, а *выражение* = -5. Мы понимаем, что цикл работает, и два его шага были сделаны. Нажимаем **<F8>** еще два раза. Сейчас *переменная* содержит ноль, а *значение* выражения указывает "inf". Однако строка с вычислением еще не была выполнена, не забываем об этом. Снова нажимаем **<F8>**, и снова получаем ошибку. А в значениях переменной и выражения видим *слово* "evaluating", что переводится, как "оценка". Теперь мы наглядно видим, что в строке

```
st.Append('10/'+IntToStr(i)+'='+FloatToStr(10/i));
```

возникает ошибка, когда *переменная* i равна нулю. И тут уже несложно догадаться, почему эта ошибка возникает - потому что происходит попытка деления 10 на 0.

Это можно проверить, пропустив выполнение вычисления, когда i=0. Закройте окно с ошибкой, сбросьте отладчик. Снова нажмите кнопку "**Запустить**", и кнопку "**Button1**". Снова выведите окно **Списка наблюдения**. Нажимайте **<F8>**, пока i не станет 0, а *выражение* - inf.

Теперь, в **Окне наблюдений** щелкните правой кнопкой мыши по строке с переменной i, и в всплывающем *меню* выберите команду "**Вычислить/Изменить**". В открывшемся окне вы увидите строку "**Выражение**", где будет указана *переменная* i. В *поле* "**Результат**" будет указано *значение* 0. А в строке "**Новое значение**" нам нужно указать *значение*, которое мы желаем принудительно присвоить переменной. Тут укажем 1 и нажмем **<Enter>** или кнопку "**Изменить**". В *поле* "**Результат**" *значение* должно смениться на 1. Снова нажмем **<F8>**, *значение* i изменится на 2, ошибки не будет. Нажав **<F8>** еще несколько раз, мы доберемся до конца программы и увидим сообщение, которое она и должна была вывести по нашему замыслу:

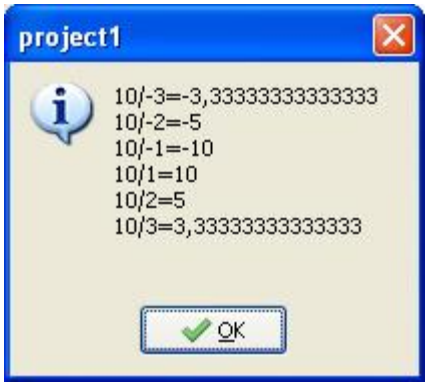


Рис. 2.8. Результирующее сообщение программы

Как видите, *вычисление*, где i равна нулю, было пропущено.

Встроенный отладчик имеет и другие инструменты, с которыми вы сами сможете со временем освоиться, экспериментируя с ними.

Контрольные вопросы и задания

1. Покажите на примерах, как в оболочке Lazarus осуществляется:

запуск и выход из оболочки,

загрузка и сохранение файла,

вызов справки, в т.ч. по ключевому слову, на которое указывает курсор,

контекстный поиск и замена текста,

компиляция и запуск программы.

2. Объясните понятия: синтаксическая ошибка, ошибка времени выполнения, логическая ошибка.
3. 4Покажите на примерах, как в оболочке Lazarus осуществляется: добавление, редактирование и удаление переменных в окне просмотра значений переменных, пошаговое выполнение программ, в т.ч. с пошаговым выполнением операторов в вызываемых подпрограммах, выполнение программы до строки, на которую указывает курсор, завершение отладки программы, создание, редактирование и удаление точек останова программы.
4. Составьте отчёт, в котором опишите свои действия во время выполнения заданий 1 – 6, проиллюстрируйте скриншотами.

Практическое занятие 4. Оценка качественных показателей программного продукта

1. Краткие теоретические сведения

Тестирование программы P по некоторому критерию C означает покрытие множества компонентов программы P M = {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 P по критерию C — измеряется max числом неизбыточных тестов, покрывающих все элементы множества M(P,C)

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

TV(P,C,T) = (V-DV)/V — оценка степени тестированности P по критерию C.

Критерий окончания тестирования TV(P,C,T) L, где (0 L 1). L — уровень оттестированности, заданный в требованиях к программному продукту.

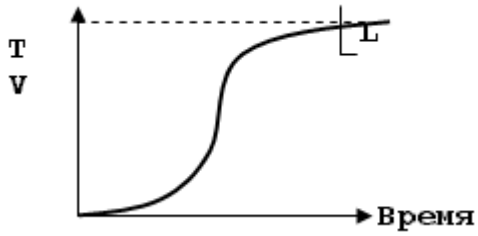


Рис. 10.1. Метрика оттестированности приложения

Рассмотрим две модели программного обеспечения, используемые при оценке оттестированности.

Для оценки степени оттестированности часто используется УГП — управляющий граф программы. УГП многокомпонентного объекта G (Рис. 10.2, Пример 10.4), содержит внутри себя два компонента G1 и G2, УГП которых раскрыты.

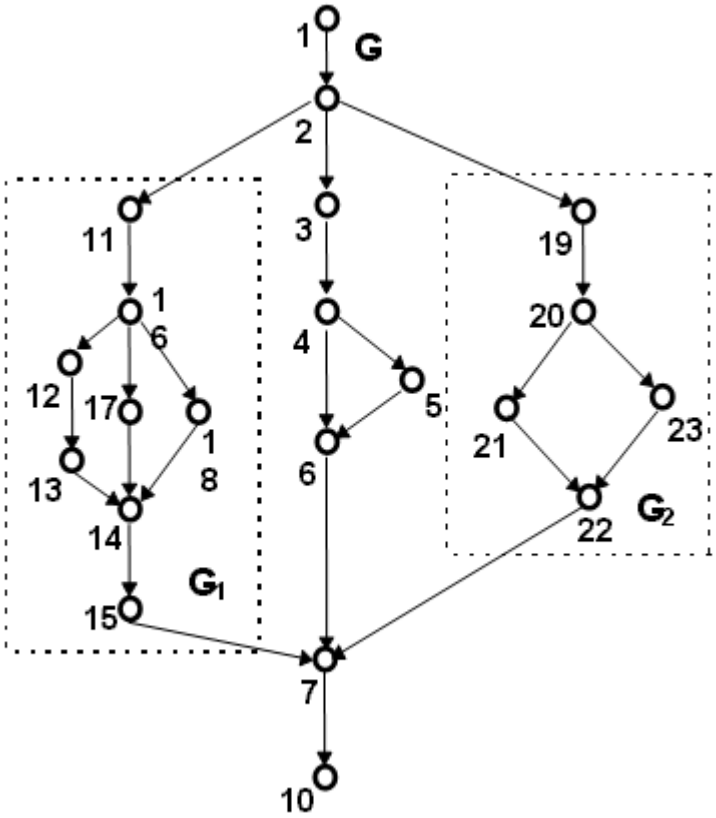


Рис. 10.2. Плоская модель УГП компонента G

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 10.1):

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;

Пример 10.1. Набор трасс, необходимых для покрытия плоской модели УГП компонента G

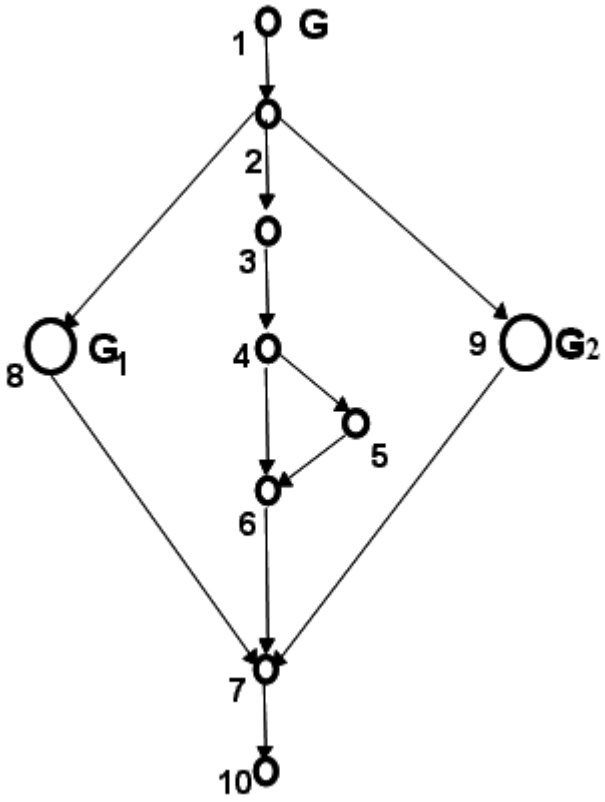


Рис. 10.3. Иерархическая модель УГП компонента G

УГП компонента G, представленный в виде иерархической модели, приведен на Рис. 10.3, Пример 10.5. В иерархическом УГП G входящие в его состав компоненты представлены ссылками на свои УГП G1 и G2 (Рис. 10.4, Пример 10.5)

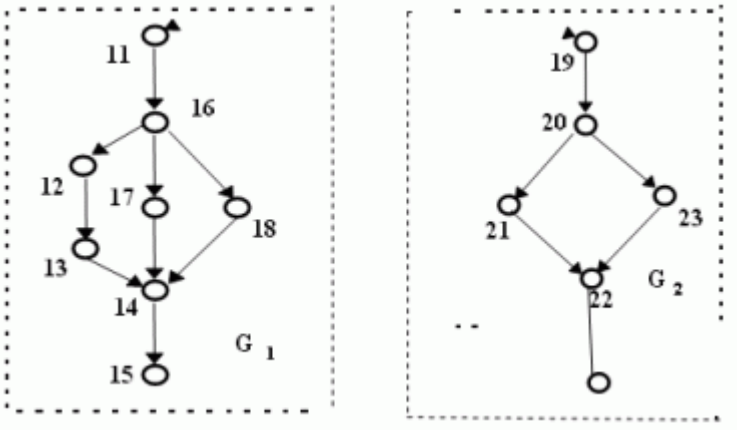


Рис. 10.10. Иерархическая модель: УГП компонент G1 и G2

Для исчерпывающего тестирования иерархической модели компонента G в соответствии с критерием путей требуется прогнать следующий набор трасс (Пример 10.2):

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.

Пример 10.2. 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 10.3.

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

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

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

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

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

Контрольные вопросы

1. Оценка степени тестируемости ПО.
2. Критерии структурного тестирования.
3. Построение управляющего графа программы.
4. Функциональное тестирование (Метод « черного ящика»).
5. Тестирование циклов.
6. Тестирование потоков данных.
7. Тестирование транзакций.
8. Характеристики хорошего теста.

Практическое занятие 5. Критерии выбора тестов

Цель. Изучить критерии выбора тестов; выполнять оценочное тестирование программного продукта.

1. Краткие теоретические сведения

Требования к идеальному критерию тестирования

Требования к идеальному критерию были выдвинуты в работе [11] :

Критерий должен быть достаточным, т.е. показывать, когда некоторое конечное множество тестов достаточно для тестирования данной программы.

Критерий должен быть полным, т.е. в случае ошибки должен существовать тест из множества тестов, удовлетворяющих критерию, который раскрывает ошибку.

Критерий должен быть надежным, т.е. любые два множества тестов, удовлетворяющих ему, одновременно должны раскрывать или не раскрывать ошибки программы

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

Структурные критерии используют информацию о структуре программы (критерии так называемого "белого ящика")

Функциональные критерии формулируются в описании требований к программному изделию (критерии так называемого "черного ящика")

Критерии стохастического тестирования формулируются в терминах проверки наличия заданных свойств у тестируемого приложения, средствами проверки некоторой статистической гипотезы.

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 C0) - 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 C1) - 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 C2) - 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).

После завершения комплексного тестирования приступают к оценочному тестированию, целью которого является тестирование программы на соответствие основным требованиям. Эта стадия тестирования особенно важна для программных продуктов, предназначенных для продажи на рынке.

Оценочное тестирование, которое также называют «тестированием системы в целом», включает следующие виды:

- *тестирование удобства использования* - последовательная проверка соответствия программного продукта и документации на него основным положениям технического задания;
- *тестирование на предельных объемах* - проверка работоспособности программы на максимально больших объемах данных, например, объемах текстов, таблиц, большом количестве файлов и т. п.;
- *тестирование на предельных нагрузках* - проверка выполнения программы на возможность обработки большого объема данных, поступивших в течение короткого времени;
- *тестирование удобства эксплуатации* - анализ психологических факторов, возникающих при работе с программным обеспечением; это тестирование позволяет определить, удобен ли интерфейс, не раздражает ли цветовое или звуковое сопровождение и т. п.;
- *тестирование защиты* - проверка защиты, например, от несанкционированного доступа к информации;

- *тестирование производительности* - определение пропускной способности при заданной конфигурации и нагрузке;
- *тестирование требований к памяти* - определение реальных потребностей в оперативной и внешней памяти;
- *тестирование конфигурации оборудования* - проверка работоспособности программного обеспечения на разном оборудовании;
- *тестирование совместимости* - проверка преемственности версий: в тех случаях, если очередная версия системы меняет форматы данных, она должна предусматривать специальные конвекторы, обеспечивающие возможность работы с файлами, созданными предыдущей версией системы;
- *тестирование удобства установки* - проверка удобства установки;
- *тестирование надежности* - проверка надежности с использованием соответствующих математических моделей;
- *тестирование восстановления* - проверка восстановления программного обеспечения, например системы, включающей базу данных, после сбоев оборудования и программы;
- *тестирование удобства обслуживания* - проверка средств обслуживания, включенных в программное обеспечение;
- *тестирование документации* - тщательная проверка документации, например, если документация содержит примеры, то их все необходимо попробовать;
- *тестирование процедуры* - проверка ручных процессов, предполагаемых в системе и др.

Естественно, целью всех этих проверок является поиск несоответствий техническому заданию. Считают, что только после

выполнения всех видов тестирования программный продукт может быть представлен пользователю или к реализации. Однако на практике обычно выполняют не все виды оценочного тестирования, так как это очень дорого и трудоемко. Как правило, для каждого типа программного обеспечения выполняют те виды тестирования, которые являются для него наиболее важными. Так базы данных обязательно тестируют на предельных объемах, а системы реального времени - на предельных нагрузках.

Системы для создания инсталляторов

Практика разработки коммерческого программного обеспечения показывает, что далеко не все пользователи умеют работать с архивами. Поэтому программы рекомендуется поставлять в виде исполняемых файлов, которые автоматически создают необходимые папки в файловой системе, копируют туда файлы программы, создают необходимые файлы настроек или ключи в реестре, а так же пункты меню запуска программы и ярлыки на рабочем столе. Для упрощения создания инсталляторов существует много специализированных программных продуктов.

Знакомство пользователя с программой чаще всего начинается с запуска инсталлятора. Внешний вид («упаковка») и функциональность продукта определяется разработчиком. Пользователю нужно иметь возможность проконтролировать процесс, выставив нужные параметры установки. Для разработчика же важно, чтобы, как минимум, его программа была установлена корректно, а инсталлятор был совместим с необходимыми платформами.

Решений для создания инсталляторов достаточно много. Чаще всего используется подсистема Windows Installer, которая уже входит в инструментарий операционной системы. Но существуют и альтернативные решения – как платные, так и бесплатные, различной функциональности.

Зачастую с их помощью можно создавать пакеты с инсталлятором, не зависящим от Windows Installer.

Основные критерии выбора системы создания инсталлятора следующие:

- среда разработки, интерфейс, поддержка сценариев;
- работа с проектом, типы создаваемых пакетов, возможности импорта проектов из других сред разработки;
- пользовательские опции инсталлятора: поддержка языков, профилей и другие опции;
- поддержка расширений.

Свободные программы для создания инсталляторов:

- NSIS (Nullsoft Scriptable Install System) – один из самых популярных инсталляторов. Обладает богатыми возможностями, которые присутствуют в большинстве коммерческих продуктов. Позволяет устанавливать различные параметры сжатия при создании дистрибутива;
- IzPack – java инсталлятор. Это универсальный инсталлятор, способен создавать дистрибутивы для Unix, Linux, FreeBSD, Mac OS X и Windows 2000, XP. Позволяет создавать как обычные пакеты инсталляции, так и Web инсталляторы, которые подгружают необходимые файлы по мере необходимости. Данная возможность позволяет свести к минимуму количество загружаемых файлов в зависимости от требуемой конфигурации установки;
- Inno Setup –довольно популярный простой инталлятор. Содержит встроенный скриптовой язык;
- WiX (Windows Installer XML) – специализированный продукт от Microsoft для создания MSI и MSM инсталляционных пакетов.

Коммерческие программы для создания инсталляторов:

- InstallShield – один из самых известных продуктов в ряду инсталляторов;
- WISE – простой в освоении с богатыми возможностями генератор инсталляторов;
- VISE - профессиональный инсталлятор для Windows, MacOS X и Macintosh;
- CreateInstall это универсальный, гибкий и мощный инсталлятор как для профессиональных разработчиков, так и для начинающих. С помощью этой программы Вы можете создать полнофункциональные инсталляционные программы для Ваших приложений, а также самораспаковывающиеся архивы с высокой степенью сжатия и многое другое;
- Advanced Installer – позволяет создавать инсталляторы для java приложений. Создает дополнительный исполняемый файл.

CreateInstall

Домашняя страница: http://www.createinstall.ru/

CreateInstall – инструментарий для создания установщиков. В его основу заложено две особенности – контроль над процессом установки и неограниченная расширяемость. Обе возможности реализованы благодаря языку программирования Gentee, применяемому для написания сценариев.

Интерфейс CreateInstall разбит на 3 вкладки – «Проект», «Скрипт установки» и «Скрипт деинсталляции». Первый раздел позволяет задать общие настройки инсталлятора: информация о продукте, поддерживаемые языки, пути, внешний вид. Дополнительно, инсталлятор можно защитить цифровой подписью и установить пароль.

«Проект» – не равноценная замена двух последующих разделов, т. е. для создания дистрибутива нужно тщательно настроить скрипты установки и деинсталляции. Соответствующие параметры отображаются в виде групп, можно отобразить их единым списком.

Дополнением для CreateInstall служит утилита Quick CreateInstall (рисунок 1). Она значительно упрощает создание инсталлятора, предоставляя только базовые настройки проекта. Из Quick CreateInstall в дальнейшем проект можно импортировать в CreateInstall.

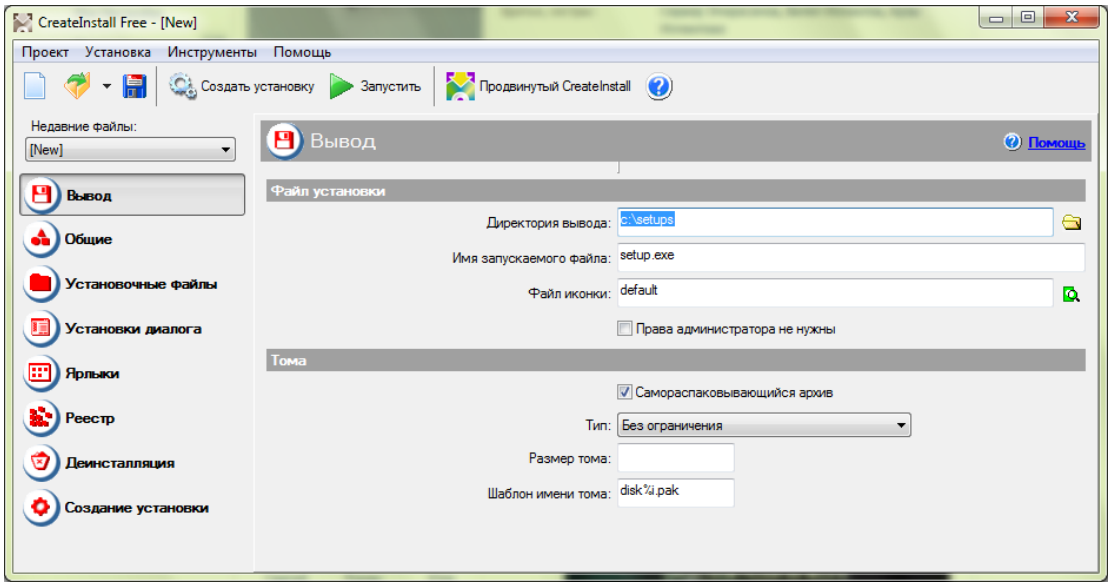


Рис. 15. 1 Окно Quick CreateInstall

Код проекта не предназначен для самостоятельного редактирования, переноса в IDE-среду, экспорта. Хотя язык Gentee имеет отличный потенциал: как минимум, это переменные и функции, условные выражения и синтаксис, базирующийся на C, C++ и Java.

Существует 3 редакции программы – полная, light (простая) и бесплатная.

Интерфейс и справка доступны на русском языке.

Advanced Installer

Advanced Installer основывается на технологии Windows Inslaller, позволяя создавать msi-, exe- и других видов дистрибутивов. Этому способствует продуманный интерфейс и работа с проектами. В Advanced Installer можно обнаружить немало возможностей, которых нет в других подобных комплексах.

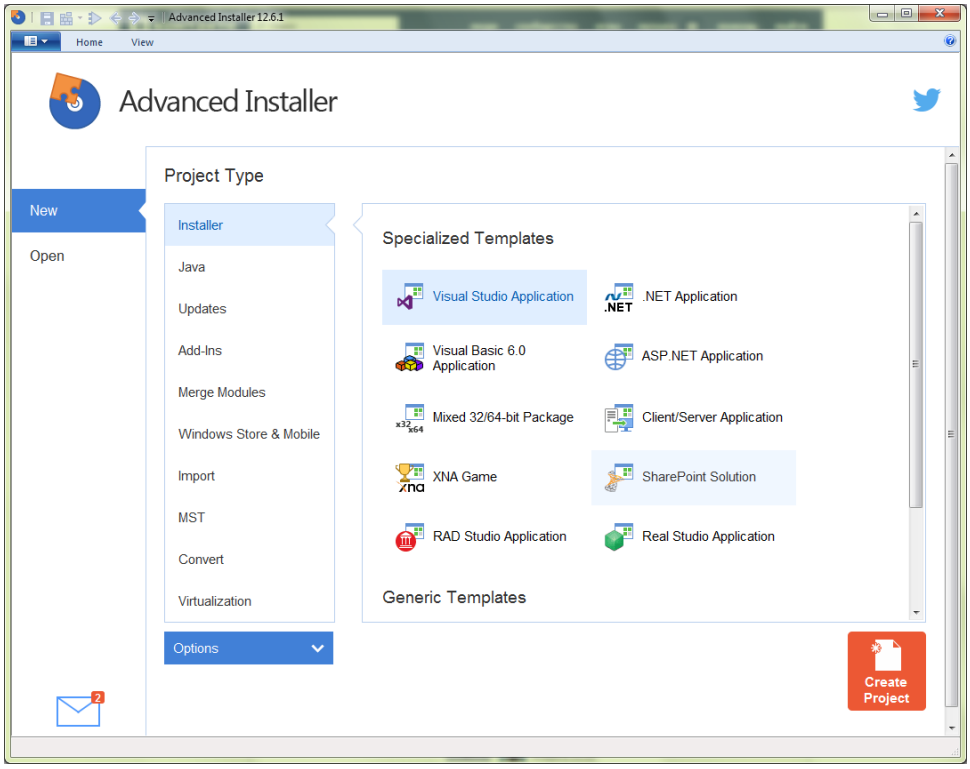


Рис. 15. 2 Окно Advanced Installer

Примечательно, прежде всего, разнообразие проектов: сюда входят инсталляторы, Java-установщики, обновления, дополнения, модули слияния и другие. В разделе меню Installer собраны команды импорта проектов из Visual Studio, RAD Studio, Real Studio, Visual Basic. Здесь раскрывается потенциал Advanced Installer во взаимодействии с IDE-средами.

Для каждого из выбранных типов проекта предусмотрен детальный мастер настройки. Есть общие шаблоны – Simple, Enterprise, Architect или Professional. Большая часть проектов доступна только для определенных типов лицензии, общедоступные проекты обозначены как None в графе License Required.

Как уже сказано, при создании проекта можно воспользоваться пошаговым мастером, где, в частности, доступен выбор способа распространения пакета, языков локализации, настройка пользовательского интерфейса, ввод текста лицензии и другие опции. Advanced Installer позволяет выбрать вариант распространения программы – оставить данные

без компрессии, разделить на CAB-архивы, сохранить в MSI и др., добавить цифровую подпись, потребовать ввод серийного номера и т. д.

Главное окно Advanced Installer (редактор проекта), в простом режиме отображения (Simple), содержит несколько секций:

- Product Information (Информация о продукте) – ввод сведений о продукте, параметры установки.
- Requirements (Требования) – указание аппаратных и системных требований, зависимостей ПО. Также имеется возможность создания пользовательских условий.
- Resources (Ресурсы) – редактор ресурсов (файлов и ключей реестра).
- Deployment (Развертывание) – выбор типа распространения продукта. Это может быть MSI, EXE или веб-инсталлятор. Для MSI, EXE ресурсы можно поместить отдельно от инсталлятора.
- System Changes – переменные среды.

При выборе ресурсов могут использоваться файлы, ключи реестра, переменные окружения, конфигурационные ini, драйверы, базы данных и переводы. С помощью модулей объединения можно добавить и другие ресурсы, такие как сервисы, разрешения, ассоциации и др.

Для выполнения более сложных задач позволяется использовать пользовательские действия, EXE, DLL или скрипты, написанные на C, C++, VBS или JS. Для создания сценариев предусмотрен удобный редактор.

Однако следует отметить, что в режиме Simple доступна лишь малая часть разделов. Работая с Advanced Installer в ознакомительном режиме, есть смысл зайти в настройки и переключиться в другой режим работы с проектом. После этих действий становятся доступны новые подразделы редактора.

Практическая часть

Задание 1. С помощью системы создания инсталляторов создайте из программы, созданной на лабораторной работе № 6, установочный файл.

Задание 2. Выполните тестирование удобства установки.

Задание 3. Выполните тестирование конфигурации оборудования.

Задание 4. Выполните тестирование восстановления.

Задание 5. Выполните тестирование удобства эксплуатации при помощи соседа.

Задание 6. Результаты выполнения практического задания запишите в отчет.

Содержание отчета

1. Тема.
2. Цель.
3. Оборудование.
4. Результат выполнения практического задания.
5. Ответы на контрольные вопросы.
6. Вывод.

Контрольные вопросы

1. Что является целью тестирования программ?
2. Какие подходы к тестированию вы знаете? В чем они заключаются?
3. Обоснуйте необходимость создания инсталляторов программ.
4. Подходы к разработке тестов.
5. Тестирование спецификации.
6. Тестирование сценариев.

Практическое занятие 6. Особенности индустриального тестирования

1. Краткие теоретические сведения

Написание тестов стабилизирует код и позволяет сократить время отладки. Тестирование системы в целом (системное тестирование) не всегда позволяет обнаружить ошибки в отдельных компонентах. Исправление ошибок на ранних стадиях разработки менее затратно.

Пример программы
Пусть есть класс, реализующий несколько

математических функций:

```
public class CustomMath {

    public static int sum(int x, int y) {
        return x + y; //возвращает результат сложения двух чисел
    }

    public static int division(int x, int y) {
        if (y == 0) {//если делитель равен нулю
            throw new IllegalArgumentException("divider is 0 ");
        } //бросается исключение
        return x / y; //возвращает результат деления
    }
}
```

Замечание
Иногда требуется снабжать программу модульными тестами.
Тесты неудобно хранить в самой программе:

7. Усложняет чтение кода.
8. Такие тесты сложно запускать.

9. Тесты не относятся к бизнес-логике приложения и должны быть исключены из конечного продукта.

Внешняя библиотека, подключенная к проекту, может существенно облегчить разработку и поддержание модульных тестов. Наиболее популярная библиотека для Java – JUnit.

Вариант модульного тестирования без библиотеки
Некоторые проверки можно поместить в сам класс.

Доработаем класс CustomMath

```
public class CustomMath {

    public static int sum(int x, int y) {...}

    public static int division(int x, int y) {...}

    public static void main(String[] args) {
        if (sum(1, 3) == 4) { //проверяем, что при сложении 1 и 3
            //нам возвращается 4
            System.out.println("Test1 passed.");
        } else {
            System.out.println("Test1 failed.");
        }
        try {
            int a = division(1, 0);
            System.out.println("Test3 failed.");
        } catch (IllegalArgumentException e) {
            //генерируется ожидаемое исключение
            System.out.println("Test3 passed.");
        }
    }
}
```

Установка JUnit

JUnit может быть использован для любого Java-приложения.

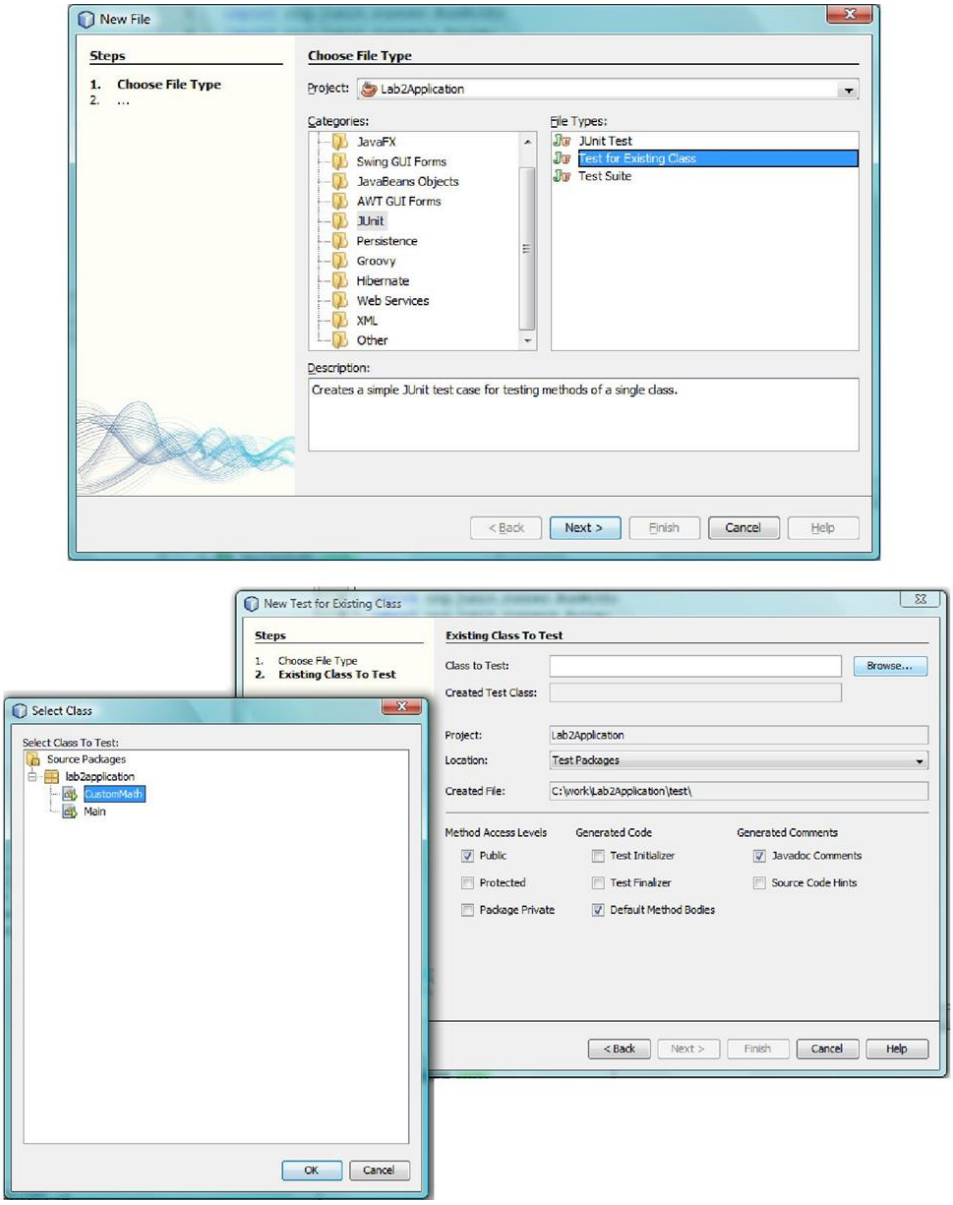
Сайт проекта www.junit.org.

Библиотека входит в состав большинства интегрированных сред разработки, в том числе NetBeans.

Создание тестового модуля

Создание тестового модуля по шаблону может быть произведено с помощью мастера.

Тесты JUnit будут располагаться в ветке Test Packages проекта. Структура папок в Test Packages в общем случае дублирует папки классов Source Packages. Выберите в меню File->New File->… в разделе JUnit пункт Тест для существующего класса («Test for Existing Class»).



В данном случае будут созданы тесты для класса CustomMath.

Настройки оставим по умолчанию: доступ к методам Public, наполнение методов по умолчанию, комментарии Javadoc.

Javadoc - форма организации комментариев в коде с использованием ключевых слов, по которым NetBeans определяет существенную информацию. Если класс оформлен

1) использованием Javadoc – по нему может быть автоматически создана документация, а также работать контекстная подсказка NetBeans (к примеру, показывать назначение функции).

Созданный по умолчанию код класса тестов:

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package lab2application;

import org.junit.AfterClass;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;

/**
 *
 * @author Alexander Sirenko
 */
public class CustomMathTest {

    public CustomMathTest() {
    }

    @BeforeClass
    public static void setUpClass() throws Exception {
    }

    @AfterClass
    public static void tearDownClass() throws Exception {
    }


    /**
     * Test of sum method, of class CustomMath.
     */
    @Test
    public void testSum() {
        System.out.println("sum");
        int x = 0;
        int y = 0;
        int expResult = 0;
        int result = CustomMath.sum(x, y);
        assertEquals(expResult, result);
        fail("The test case is a prototype.");
    }

    /**
     * Test of division method, of class CustomMath.
     */
    @Test
    public void testDivision() {
        System.out.println("division");
        int x = 0;
        int y = 0;
        int expResult = 0;
        int result = CustomMath.division(x, y);
        assertEquals(expResult, result);
        fail("The test case is a prototype.");
    }

    /**
     * Test of main method, of class CustomMath.
     */
    @Test
    public void testMain() {
        System.out.println("main");
        String[] args = null;
        CustomMath.main(args);
        fail("The test case is a prototype.");
    }

}
```

1. коде тестов можно видеть аннотации: информация о назначении методов с символом @ (@BeforeClass, @AfterClass, @Test).

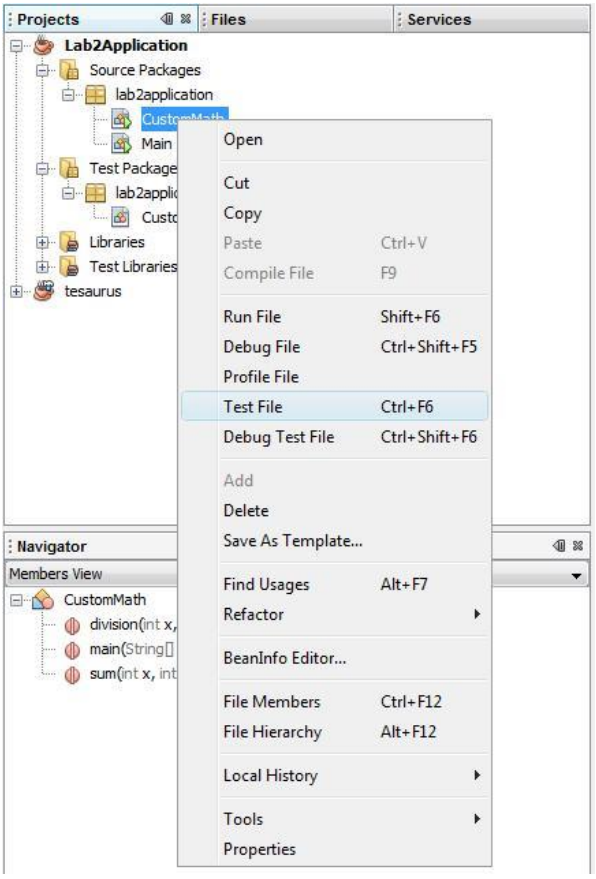
Аннотация @Test отмечает методы, автоматически запускаемые средой тестирования.

@BeforeClass и @AfterClass содержат действия, которые необходимо выполнить до запуска тестов класса (например, подключение к базе данных, или подготовку данных для

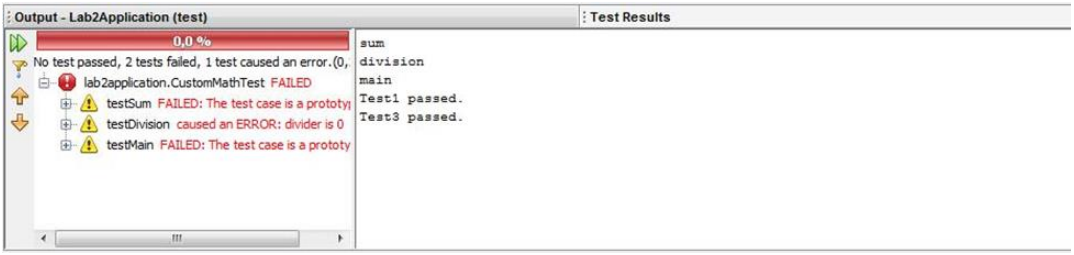
обработки) или после выполнения тестов (например, отключение от базы данных, восстановление исходного ее состояния).

Запуск созданных по умолчанию тестов

Запуск тестов выполняется через контекстное меню тестируемого класса, пунктом Test File.



Вкладка Test Results отображает выводимые на консоль сообщения (в данном случае размещенные нами в функции main проверки), а также результаты проверки методов тестируемого класса (3 неудачи).



Иерархия классов JUnit:

- java.lang.Object

- org.junit.**Assert** o org.junit.**Assume**

o java.lang.Throwable (implements java.io.Serializable) o java.lang.Error

o java.lang.AssertionError

- org.junit.ComparisonFailure o org.junit.Test.None

Annotation Type Hierarchy

- org.junit.**Test** (implements java.lang.annotation.Annotation)
- org.junit.**Ignore** (implements java.lang.annotation.Annotation)

- org.junit.**BeforeClass** (implements java.lang.annotation.Annotation) o org.junit.**Before** (implements java.lang.annotation.Annotation)

o org.junit.**AfterClass** (implements java.lang.annotation.Annotation) o org.junit.**After** (implements java.lang.annotation.Annotation)

Для проверки правильности выполнений метода в JUnit предусмотрена группа методов Assert, проверяющие условия и в случае несовпадения отмечающие тест не пройденным.

Упражнение 1.

Создайте проект с указанным выше классом CustomMath.

Уберите из метода main класса CustomMath проверку функции sum.

Уберите из метода testSum вызов метода fail. Убедитесь в прохождении теста функцией sum при текущих исходных данных.

Добавьте в отчет текст функции testSum и результат тестирования (скриншот окна test results).

Игнорирование теста

некоторых ситуациях может понадобиться отключить некоторые тесты. Например, возможно в текущей версии используемой вами библиотеки имеется ошибка, или по какой-то причине определенный тест не может быть выполнен в текущей среде. D

- JUnit 3.8.x, чтобы отключить тесты, приходилось их комментировать. В JUnit 4 для этих целей вам нужно просто промаркировать игнорируемый тест с помощью аннотации

@Ignore.
Например
public class CalculatorTest {

@Ignore("Not running ")
@Test

public void testTheWhatSoEverSpecialFunctionality() { }

}

Текст, который передается в аннотации @Ignore, поясняет причину пропуска теста

- может использоваться средой разработки. Указывать текст не обязательно, но очень полезно всегда задавать сообщение для того, чтобы позже не забыть про то, что этот тест отключен.

Упражение 2. Включите игнорирование теста testMain. Укажите в отчете описание метода testMain с аннотациями и результат тестирования ((скриншот окна test results)).

Обработка исключений

Исключения могут быть правильным поведением метода при определенных условиях (например, исключение отсутствия файла в случае, если он не доступен). Можно обрабатывать исключение в тесте с помощью блока try…catch(), либо передавать его далее с помощью ключевого слова throws в описании метода.

Изменим метод testDivision таким образом, чтобы он проверял корректное поведение при делении на 0. Корректным поведением в данном случае является генерация исключения.

Контрольные вопросы

1. Автоматизация тестирования, издержки тестирования.

2. Особенности процесса и технологии индустриального тестирования: планирование тестирования, подходы к разработке тестов.
3. Особенности ручной разработки и генерации тестов.
4. Автоматизация тестового цикла.
5. Документирование тестирования, обзоры и метрики.
6. Регрессионное тестирование: особенности и виды регрессионного тестирования.
7. Методы отбора тестов, оценка эффективности.
8. Автоматическое тестирование (что такое и когда оно нужно ввод автотестов)
9. Обзор программ для автоматического тестирования

Практическое занятие 7. Аспекты управления тестированием

Цель работы

Исследовать уровни тестирования программного обеспечения.

1. Краткие теоретические сведения

Тестирование на разных уровнях производится на протяжении всего жизненного цикла разработки и сопровождения программного обеспечения. Уровень тестирования определяет то, **над чем** производятся тесты: над отдельным модулем, группой модулей или системой, в целом. Проведение тестирования на всех уровнях системы - это залог успешной реализации и сдачи проекта.

Модульное тестирование

Модульное тестирование (unit testing) — процесс, позволяющий проверить на корректность отдельные модули исходного кода программы.

Цель модульного тестирования — изолировать отдельные части программы и показать, что по отдельности эти части работоспособны.

Идея состоит в том, чтобы писать **тесты** для каждой нетривиальной **функции** или **метода**. Это позволяет достаточно быстро проверить, не привело ли очередное изменение кода к **регрессии**, то есть к появлению ошибок в уже оттестированных местах программы, а также облегчает обнаружение и устранение таких ошибок.

Этот тип тестирования обычно выполняется **программистами**.

Модульное тестирование проверяет функциональность и ищет дефекты в частях приложения, которые доступны и могут быть протестированы по отдельности (модули программ, объекты, классы, функции и т.д.). Обычно компонентное (модульное) тестирование проводится вызывая код, который необходимо проверить и при поддержке сред разработки, таких как фреймворки (frameworks - каркасы) для модульного тестирования или инструменты для отладки. Все найденные дефекты, как правило исправляются в коде без формального их описания в системе менеджмента багов (Bug Tracking System).

Один из наиболее эффективных подходов к компонентному (модульному) тестированию - это **подготовка автоматизированных тестов** до начала основного кодирования (разработки) программного обеспечения. Это называется разработка от тестирования (**test-driven development**) или подход тестирования вначале (**test first approach**). При этом подходе создаются и интегрируются небольшие куски кода, напротив которых запускаются тесты, написанные до начала кодирования. Разработка ведется до тех пор пока все тесты не будут успешными.

Интеграционное тестирование

Интеграционное тестирование (Integration testing) – тестирование, при котором отдельные программные **модули объединяются** и **тестируются в группе**.

Обычно интеграционное тестирование проводится после модульного тестирования и предшествует системному тестированию.

Целью интеграционного тестирования является проверка соответствия проектируемых единиц функциональным, приёмным требованиям и требованиям надежности. Тестирование этих проектируемых единиц выполняется через их интерфейс, с использованием тестирования «**чёрного ящика**».

Интеграционное тестирование предназначено для проверки связи между компонентами, а также взаимодействия с различными частями системы
(операционной системой, оборудованием либо связи между различными системами).

Уровни интеграционного тестирования:
 Компонентный интеграционный

уровень

(Component
Integrationtesting)

Проверяется взаимодействие между компонентами системы после проведения компонентного тестирования.

 Системный интеграционный уровень (*System Integration Testing*)

Проверяется взаимодействие между разными системами после проведения системного тестирования.

Подходы к интеграционному тестированию:
 Снизу вверх (Bottom Up Integration)

Все низкоуровневые модули, процедуры или функции собираются воедино и затем тестируются. После чего собирается следующий уровень модулей для проведения интеграционного тестирования. Данный подход считается полезным, если все или практически все модули, разрабатываемого уровня, готовы. Также данный подход помогает определить по результатам тестирования уровень готовности приложения.

- **Сверху вниз** (Top Down Integration)

Вначале тестируются все высокоуровневые модули, и постепенно один за другим добавляются низкоуровневые. Все модули более низкого уровня симулируются заглушками с аналогичной функциональностью, затем по мере готовности они заменяются реальными активными компонентами. Таким образом мы проводим тестирование сверху вниз.

- **Большой взрыв** ("Big Bang" Integration)

Все или практически все разработанные модули собираются вместе в виде законченной системы или ее основной части, и затем проводится интеграционное тестирование. Такой подход очень хорош для сохранения времени. Однако если тест кейсы и их результаты записаны не верно, то сам процесс интеграции сильно осложнится, что станет преградой для команды тестирования при достижении основной цели интеграционного тестирования.

Системное тестирование

Системное тестирование — это тестирование, выполняемое на полной, **интегрированной системе**, с целью проверки соответствия системы **исходным требованиям**. Системное тестирование относится к методам тестирования **чёрного ящика**, и, тем самым, не требует знаний о внутреннем устройстве системы.

Основной **задачей** системного тестирования является **проверка** как **функциональных**, так и **не функциональных требований** к системе **в целом**. При этом выявляются дефекты, такие как:

- неверное использование ресурсов системы,
- непредусмотренные комбинации данных пользовательского уровня,
- несовместимость с окружением,
- непредусмотренные сценарии использования,
- отсутствующая или неверная функциональность,
- неудобство использования и т.д.

Основной задачей системного тестирования является **проверка как функциональных, так и не функциональных требований в системе в целом**. При этом выявляются дефекты, такие как неверное использование ресурсов системы, непредусмотренные комбинации данных

пользовательского уровня, несовместимость с окружением, непредусмотренные сценарии использования, отсутствующая или неверная функциональность, неудобство использования и т.д. Для минимизации рисков, связанных с особенностями поведения в системы в той или иной среде, **во время тестирования рекомендуется использовать окружение максимально приближенное к тому, на которое будет установлен продукт после выдачи**.

Можно выделить два подхода к системному тестированию:

- **на базе требований** (*requirements based*)

Для каждого требования пишутся тестовые случаи (test cases), проверяющие выполнение данного требования.

- на базе случаев использования (*use case based*)

На основе представления о способах использования продукта создаются случаи использования системы (**UseCases**). По конкретному случаю использования можно определить один или более сценариев. На проверку каждого сценария пишутся тест кейсы (test cases), которые должны быть протестированы.

Задание

Изучить уровни тестирования на примере известного приложения: модульный, интеграционный и системный уровни. Отчет должен содержать план тестирования известного приложения, краткие выводы.

Контрольные вопросы

1. Является ли требование частью функциональной спецификации?
2. Для чего предназначено интеграционное тестирование?
3. На каких уровнях разработки программного обеспечения произовдится тестирование?
4. Какое программное окружение рекомендуется использовать на уровне системного тестирования?

Литература

1. «Инженерия программного обеспечения», Соммервилл И.
2. Основы Программной Инженерии (по SWEBOK) http://swebok.sorlik.ru/
3. Майерс Г. Искусство тестирования программ.-М.:Финансы и статистика ,1982.-178с.

Контрольные вопросы

1. Назовите формальные методы проверки правильности программ.
2. Какие процессы проверки зафиксированы в стандарте?
3. Какие функции у процесса верификации программ?
4. Назовите основные задачи процесса валидации программ.
5. Сравните задачи процессов верификации и валидации программ.
6. В чем отличие верификации и валидации?
7. Определите процесс тестирования?.

Практическое занятие 8. Знакомство с программными продуктами управления тестирование

Цель работы: Освоить процесс отслеживания ошибок с использованием системы Bugzilla и Atlassian JIRA.

1. Краткие теоретические сведения

Jira — коммерческая система отслеживания ошибок, предназначена для организации взаимодействия с пользователями, хотя в некоторых случаях используется и для управления проектами. Разработана компанией Atlassian, является одним из двух её основных продуктов (наряду с вики-системой Confluence). Имеет веб-интерфейс. Система позволяет работать с несколькими проектами. Для каждого из проектов создаёт и ведёт схемы безопасности и схемы оповещения

Название системы получено путём усечения слова «Gojira» — японского имени монстра Годзилла, что, в свою очередь, является отсылкой к названию конкурирующего продукта — Bugzilla; создавалась в качестве замены Bugzilla и во многом повторяет её архитектуру..

Bugzilla (Багзилла) — свободная система отслеживания ошибок (багтрекинга) с веб-интерфейсом.

В 1998 году Bugzilla была выпущена как открытое программное обеспечение компанией Netscape. По состоянию на 2012 год разрабатывается фондом Mozilla Foundation.

Системой Bugzilla пользуются, в числе прочих, Mozilla Foundation, WebKit, Linux kernel, FreeBSD, GNOME, KDE, Apache, Red Hat, Eclipse и LibreOffice.

Ключевым понятием системы является «баг» — некоторое задание, запрос, рекламация по поводу ошибки в системе, или просто сообщение, требующее обратной связи.

Функциональность Bugzilla включает в себя следующие аспекты:

- Интегрированная система безопасности с возможностью определения безопасности на уровне продуктов
- Система зависимостей ошибок и вывод зависимостей ошибок в графическом виде
- Развитая система составления отчетов
- Стабильный, проверенный временем back-end на основе RDBMS
- Обширная система конфигурирования
- Очень понятный и хорошо продуманный протокол решения ошибок
- API для электронной почты, XML, HTTP и консоли
- Доступна интеграция с системами управления автоматического конфигурирования программного обеспечения, в том числе Perforce and CVS (через интерфейс электронной почты Bugzilla и скрипты для checkin/checkout)

Задание: Необходимо на практике продемонстрировать навыки работы с Bugzila и JIRA, перечисленные в списке вопросов.

Отчет по лабораторной работе должен содержать титульный лист, цель работы, список вопросов с кратким (2-3 предложения) описанием необходимых для выполнения вопроса действий.

Контрольные вопросы

1. Как создать дефект. Уметь рассказать о смысле заполняемых при создании полей.
2. Как посмотреть тип и приоритет дефекта.
3. Как посмотреть компоненту проекта, на которую заведен дефект.
4. Как посмотреть, добавить/удалить метку дефекта.
5. Как проголосовать за дефект.
6. Как просмотреть список пользователей, следящих за дефектом. Как удалить/добавить пользователя в список.
7. Как изменить статус дефекта для отправки его на тестирование после исправления. Уметь рассказать о смысле заполняемых при изменении полей.
8. Как задать estimated time. Как производить учет затраченного рабочего времени.
9. Как перевести дефект на другого пользователя без изменения статуса.
10. Как добавить комментарий к дефекту. Как просмотреть список комментариев. Как сделать постоянную ссылку на комментарий.