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 лабораторных работ

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 лабораторных работ. - Ростов-на-Дону: Северо-Кавказский филиал МТУСИ, 2019. – 55 с.: ил.

В пособии даются организационно-методические указания к лабораторным вопросам 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

Лабораторная работа № 1. Интегрированная среда разработки и отладки приложений LAZARUS 4
Лабораторная работа № 2 . Тестирование и отладка приложений в среде LAZARUS .. 12
Лабораторная работа № 3. Разработка и отладка приложения «Калькулятор» в среде LAZARUS 21
Лабораторная работа № 4. Наращиваемый подход к тестированию 27
Лабораторная работа № 5. Тестирование программ методами белого ящика 35
Лабораторная работа № 6. Наращиваемое тестирование приложения «Калькулятор» в среде LAZARUS 42
Лабораторная работа № 7. Задача Майерса. Развитие навыков интуитивного тестирования 45
Лабораторная работа №8. Функциональное тестирование программ 50
Рекомендуемая литература 54

Лабораторная работа № 1. Интегрированная среда разработки и отладки приложений LAZARUS

Цель лабораторной работы

Освоить работу с интегрированной средой разработки и отладки приложений LAZARUS, изучить методы создания интерактивных приложений.

1. Краткие теоретические сведения

Пользовательский интерфейс Lazarus. Интегрированная среда разработки Lazarus представляет собой многооконную систему, вид (пользовательский интерфейс) которой может различаться в зависимости от настроек. После загрузки интерфейс Lazarus представлен на рис.1.1.

К основным окнам среды программирования Lazarus относятся:

1. главное окно;
2. окно Инспектора объектов;
3. окно Формы, или Конструктора формы;
4. окно Редактора кода;
5. окно Сообщений.

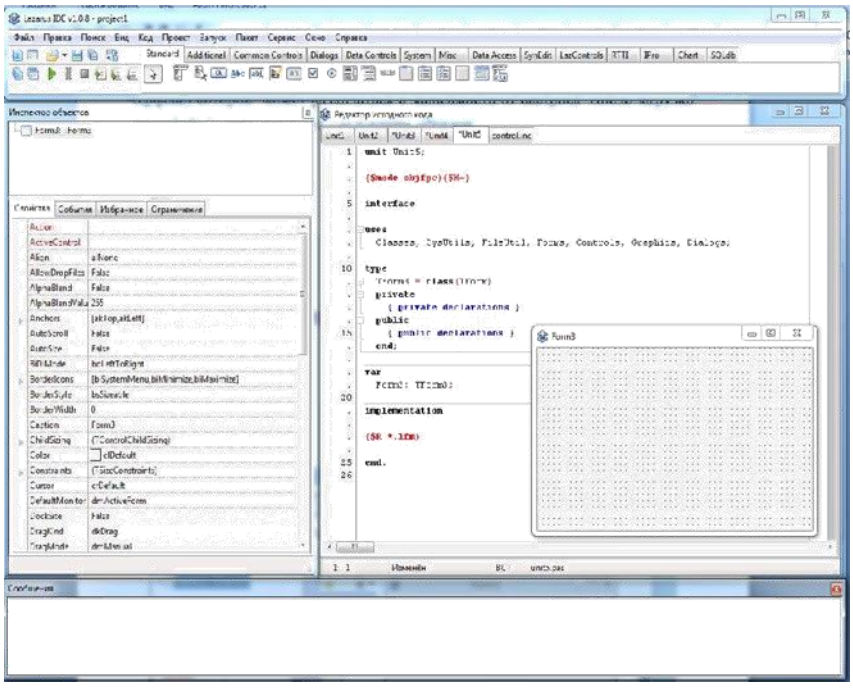


Рис. 1.1. Основные окна среды программирования Lazarus

Главное окно Lazarus содержит:

1. главное меню;
2. панели инструментов;
3. палитру компонентов.

Главное меню содержит обширный набор команд для доступа к функциям Lazarus.

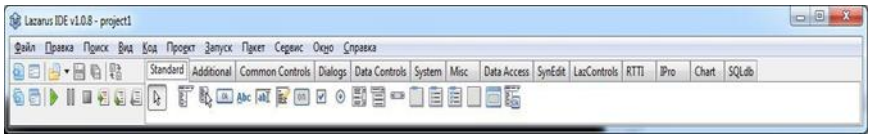


Рис. 1.2. Главное меню среды программирования Lazarus

Панели инструментов находятся под главным меню в левой части главного окна и содержат кнопки быстрого доступа к наиболее частым командам главного меню.

Палитра компонентов находится под главным меню в правой части главного окна и содержит множество компонентов, размещаемых в создаваемых формах. Компоненты являются своего рода строительными блоками, из которых конструируются формы приложения.

Окно формы (или Конструктора формы) первоначально находится в центре экрана и имеет заголовок Form1. В нем выполняется проектирование формы, путем размещения на форме различных компонентов.

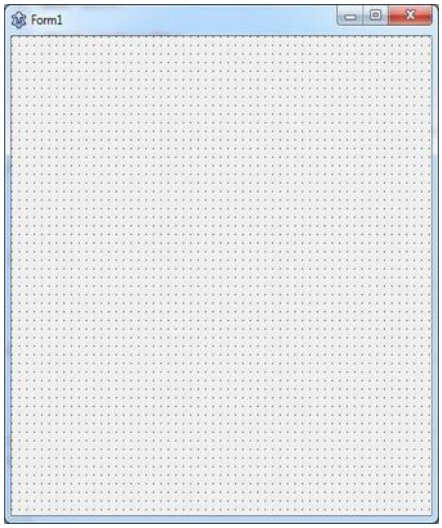


Рис. 1.3. Окно формы (или Конструктора формы)

Окно Редактора кода после запуска системы программирования находится под (или поверх) окна Формы. Редактор кода представляет собой обычный текстовый редактор, с помощью которого можно редактировать текст программы (листинг).

Окно Инспектора объектов расположено под Главным окном в левой части экрана и состоит из двух частей: в верхней части окна Инспектора объектов располагается список всех созданных объектов (форма и все компоненты, которые расположены на форме); в нижней части отображены свойства и события объектов для текущей формы или компонента. Вкладка Свойства отражает свойства выбранного компонента или формы, а вкладка События – процедуры, которые должны быть выполнены при возникновении указанного события.

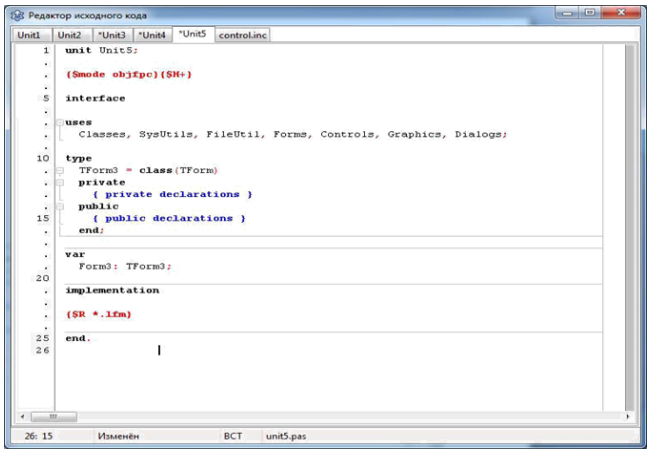


Рис. 1.4. Окно Редактора кода

Основная форма программы – содержит доступ ко всем остальным формам программы. Если закрыть подчиненное окно, то проект остается открытым. Если закрыть главную форму программы – закроется все приложение.

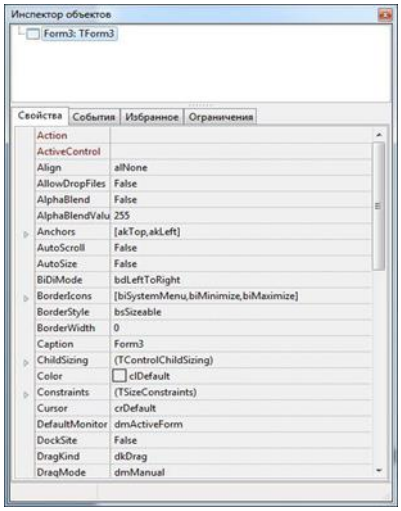


Рис. 1.5. Окно Инспектора объектов

Главное меню программы – имеет древовидную структуру и содержит доступ ко всем элементам программы от «Файл» до «Справка».

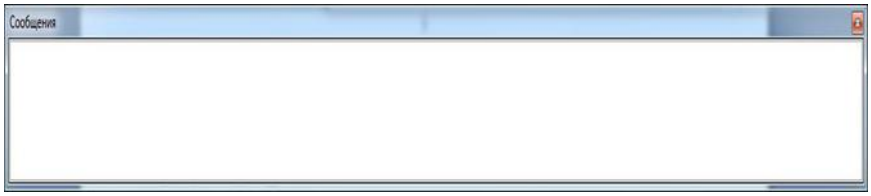


Рис. 1.6. Окно ошибок и подсказок

Управление проектом: назначение кнопок управление проек-том (рис.1.7).

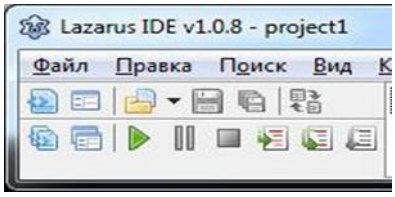


Рис. 1.7. Основные кнопки управления проектом

Верхний ряд: слева – направо:

1. Создать модуль – создает новый модуль форму.
2. Открыть – открывает проект, модуль или любой другой файл (понимаемый средой Lazarus).
3. Сохранить – сохраняет текущий модуль на диск, перед первым сохранением – запросит имя и папку, куда сохра-нить файл. Если текущий файл был сохранен и не редактировался, то кнопка серого цвета.
4. Сохранить все – Сохраняет все модули проекта, если модуль ни разу не сохранялся – предложит указать имя и папку для файлов.
5. Создать форму – создает новую форму программы и прикрепляет ее к проекту.
6. Переключить форму/модуль – осуществляет переход между окном формы и окном модуля. Можно переклю-чаться с помощью клавиши <F12>.

Примечания:

– Среда Lazarus плохо понимает русские буквы в именах модулей (это связано с синтаксисом языка Free Pascal).

– По умолчанию при создании нового проекта к нему присоединена пустая форма. Проект может содержать несколько форм, в том числе и ни одной (консольное приложение, сервис, библиотека и т.д.). Первая форма. созданная в проекте, называет-ся главной.Остальные подчиненные. При закрытии главной формы закрывается весь проект.

– Не все модули содержат форму проекта. Многие модули не имеют визуального представления, например модуль математики Math.

Нижний ряд кнопок слева – направо:

1. Показать модули – отображает список всех модулей проекта и позволяет переключится на любой из них;

2. Показать формы – отображает список всех форм проекта и позволяет переключится на любую из форм.
3. Запуск проекта – осуществляет сборку проекта и его запуск

в виде исполняемого файла под отладчиком, т. е. жизнью проекта управляет среда программирования Lazarus. (Если кнопка серая – проект запущен и находится в списке запущенных программ Windows).

4. Пауза – работу запущенного проекта можно приостановить

– поставить на «паузу».

5. Останов – принудительная остановка проекта, можно остановить проект сочетанием клавиш <Ctrl + F2>.

6. Шаг со входом – отладка проекта пошаговая с заходом во все процедуры и последовательная отладка этих процедур.

7. Шаг в обход – пошаговая отладка проекта без отладки вызываемых процедур и функций, каждая вызываемая процедура выполняется за один шаг отладки.

– Можно зайти как в процедуры своего проекта, так и библиотечные функции Lazarus, но в системные процедуры Windows (API Windows) попасть не получится.

Закладки компонентов – содержат наборы различных компо-нентов сгруппированные по различным признакам. В примерах использоваться компоненты из вкладок «Standard» и «Additional».

Компоненты – готовые настраиваемые блоки программ, которые можно устанавливать на форму и подключать к модулям. Для установки компонента на форму, необходимо один раз «кликнуть» на нем в панели компонентов, затем второй раз на форме – в том месте, где необходимо его разместить. Установленные компоненты можно выби-рать «кликнув» по нему пли с помощью рамки выделения.

– Выбранные компоненты имеют по краям черные прямоугольные маркеры, за которые можно перемешать объект по форме и изменять его размеры. Некоторые компоненты можно только перемещать.

– Свойства, как и типы данных, могут быть: целыми, вещественными, строковыми, логическими,множествами или сложными. События у компонентов создаются пустыми.

Инспектор объектов – форма позволяющая настроить свойства каждого компонента индивидуально. Инспектор объектов имеет несколько закладок.

Закладка «Свойство» отображает большинство свойств объекта, хотя и не все. Свойство объекта это имя и значение. Левый столбец – имя свойства, правый – значение.

Закладка «События» – позволяет посмотреть список боль-шинства событий, на которые может реагировать компонент, а так-же процедуры привязанные к каждому из событий.

Форма программы – форма на которой разрабатывается интерфейс программы с помощью компонентов.

Модуль программы – окно в котором содержится исходный код на языке Object Pascal.

Окно ошибок – окно в котором отображаются все ошибки и подсказки при сборке проекта.

– Среда программирования Lazarus автоматически вносит изменения в код программы при добавлении компонент на форму и создании обработчиков событий. Программист создавая тело программы вносит свои изменения в код.

Ошибки могут быть:

1. Синтаксические – когда исходный текст не понимает среда программирования Lazarus.

2. Логические – когда код с точки зрения среды программирования написан верно, но программа выполняет не те действия, которые ожидает пользователь от программы.

Первый тип ошибок может отследить среда программирования и программист, второй тип – только программист.

Проект рекомендуется сохранять в индивидуальную папку, (проект – группа связанных файлов).

Для каждого проекта рекомендуется создавать следующую структуру: Для доступа к проекту по сети разместим в папке общие документы (Мой компьютер/ Общие документы), создадим папку с номером группы (например «ДВ-31»), далее фамилия (например «Иванов»), далее «1» для первой лабораторной, для второй «2» и т. д.

2. Методика и порядок выполнения работы

Основным инструментом, которым пользуется программист в процессе разработки приложения, является Палитра Компонентов (рис.1.8).

Рис. 1.8. Палитра Компонент

Палитра Компонент, позволяет выбрать нужные объекты для размещения их на Дизайнере Форм. Для использования Палитры Компонент просто первый раз щелкните мышкой на один из объек-тов и потом второй раз – на Дизайнере Форм. Выбранный объект появится на проектируемом окне, и им можно манипулировать с помощью мыши.

Палитра Компонент использует постраничную группировку объектов. Внизу Палитры находится набор закладок – Standard, Additional, Dialogs и т.д. Если щелкнуть мышью на одну из закла-док, то можно перейти на следующую страницу Палитры Компо-нент. Принцип разбивки на страницы широко применяется в среде программирования Lazarus, и его легко можно использовать в своей программе.

Предположим, помещаете компонент TEdit на форму; можно двигать его с места на место. Также можно использовать границу, прорисованную вокруг объекта, для изменения его размеров. Боль-шинством других компонент можно манипулировать тем же обра зом. Однако, невидимые во время выполнения программы компо-ненты (типа TMenu или TDataBase) не меняют своей формы.

Каждый компонент является настоящим объектом, и можно менять его вид и поведение с помощью Инспектора Объектов, который состоит из двух страниц, каждую из которых можно использовать для определения поведения данного компонента.

Первая страница – это список свойств, вторая – список событий. Если нужно изменить что-нибудь, связанное с определенным компонентом, то это обычно делается это в Инспекторе Объектов.

К примеру, можно изменить имя и размер компонента Tlabel, изме-
няя свойства Caption, Left, Top, Height, и Width.

Можно использовать закладки внизу Инспектора Объектов для переключения между страницами свойств и событий. Страница событий связана с Редактором; если дважды щелкнуть мышкой на правую сторону какого-нибудь пункта, то соответствующий данно-му событию код автоматически запишется в Редактор, сам Редак-тор немедленно получит фокус, и сразу же имеете возможность добавить код обработчика данного события.

ЗАДАНИЕ

1. Изменить заголовок окна формы с Form1 на «Первая программа», используя свойство Caption.
2. Изменить цвет формы на clAqua, используя свойство Color.
3. Разместить в центре формы компоненту Label. Задать: надпись метки – «СВЕТОФОР», цвет метки – серый. Изменить свойство Font: шрифт – Times New Roman, начертание – жирный, размер – 14.
4. Расположить на форме компоненты Panel1, Panel2, Panel3, для которых поочередно задать свойство Caption пустым.
5. Расположить на форме три командные кнопки Button1, Button2, Button3. Задать надписи на этих кнопках «Красный», «Желтый», «Зеленый». В результате должна получиться форма, согласно рисунка.

6. Задать событие для первой командной кнопки Button1. Для этого необходимо выделить данный компонент и перейти в Ин-спекторе Объектов на страницу События. Затем на против собы-тия OnClick дважды щелкнуть левой кнопкой мыши. После выбора события автоматически открывается окно кода программы.
7. При нажатии на командную кнопку Button1 с надписью «Красный» цвет компоненты Panel1 будет меняться на красный, а цвет ком-понент Panel2 и Panel3 будет меняться на белый. Записать в процедуре следующую последовательность действий:

```
Procedure TForm1.Button1Click (Sender:TObject); Begin
Panel1.Color:=clRed;
Panel2.Color:=clWhite;
Panel3.Color:=clWhite; End;
```

8. Создать событие и реакцию на событие для командной кнопки Button2: цвет компоненты Panel2 будет меняться на желтый, а цвет компонент Panel1 и Panel3 будет меняться на белый. Если вы не знаете, как записать название цвета, посмотрите возможные цвета свойства Color в Инспекторе Объектов.
9. Создать событие и реакцию на событие для командной кнопки Button3: цвет компоненты Panel3 будет меняться на зеленый, а цвет компонент Panel1 и Panel2 будет меняться на белый.
10. На диске создать папку с номером группы (в скобках указать номер подгруппы). В папке с номером группы создать папку с именем Первая программа. Сохранить свою программу в папке Первая программа.
11. Добавить появление на компоненте Panel1 при нажатии на командную кнопку Button1 информации «СТОЙТЕ» белым цветом, жирным шрифтом, размер шрифта – 12. Для

этого необходимо в имеющуюся процедуру добавить следующие действия: Procedure TForm1.Button1Click (Sender:TObject); Begin

Panel1.Color:=clRed;

Panel2.Color:=clWhite;

Panel3.Color:=clWhite;

Panel1.Caption:='СТОЙТЕ'; {задание на панели надписи} Panel1.Font.Color:=clWhite; {задание цвета шрифта} Panel1.Font.Size:=12; {задание размера шрифта} Panel1.Font.Style:=[fsBold]; {задание начертания шрифта} End;

12. Добавить появление на компоненте Panel2 при нажатии на командную кнопку Button2 информации «ВНИМАНИЕ» белым цветом, жирным шрифтом, размер шрифта – 12.

13. Добавить появление на компоненте Panel3 при нажатии на командную кнопку Button3 информации «ИДИТЕ» белым цветом, жирным шрифтом, размер шрифта – 12.

14. Сохранить изменения в программе и запустить ее на ис-полнение.

15. На форме добавить командную кнопку Button4. Задать для нее надпись «Автор». При нажатии на кнопку должно выводится сообщение об авторе программы. Для реализации данного задания задать для нее реакцию на событие:

Procedure TForm1.Button4Click(Sender:TObject); begin

ShowMessage('Программа разработана Ивановым С.'); end;

Контрольные вопросы

1. Основные элементы IDE LAZARUS, панели, элементы управления.
2. Отладочный инструментарий LAZARUS.
3. Порядок установки точек остановки программы. Отладочный вывод информации.
4. Опишите структуру и назначение отдельных элементов головной программы приложения Lazarus.
5. Каково назначение модуля в проекте приложения Lazarus? Опишите назначение отдельных разделов модуля.
6. Как различается доступность объектов, описанных в разделах interface и implementation?
7. Как указать ссылку на свойства и методы объекта в тексте программы?
8. Какие компоненты входят в интегрированную среду разработки приложений Lazarus?
9. Перечислите основные компоненты окна среды Lazarus и укажите их назначение.
10. Понятие подпрограммы. Процедуры в Free Pascal.
11. Понятие подпрограммы. Функции в Free Pascal.
12. Составление библиотек подпрограмм. Модули.
13. Поясните понятия «Раздел объявлений», «Раздел реализации».

Лабораторная работа № 2 . Тестирование и отладка приложений в среде LAZARUS

Цель лабораторной работы

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

1. Краткие теоретические сведения

Тестирование и отладка программы

Чем больше опыта имеет программист, тем меньше ошибок в коде он совершает, но 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:

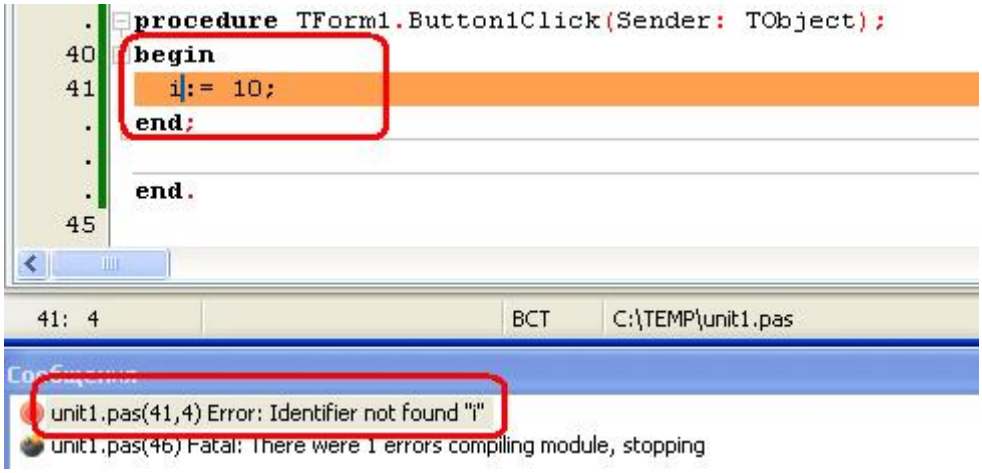


Рис. 2.1. Найденная компилятором синтаксическая ошибка - нет объявления переменной i

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 (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:

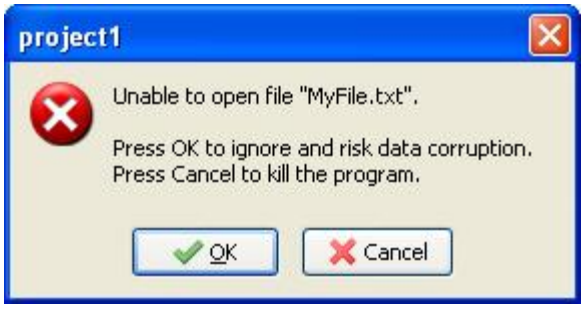


Рис. 2.2. Сообщение Lazarus об ошибке времени выполнения

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;
```

В данной программе 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:

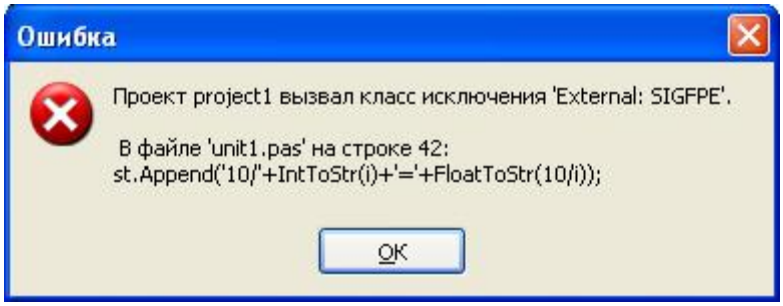


Рис. 2.3. Сообщение Lazarus об ошибке

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**":

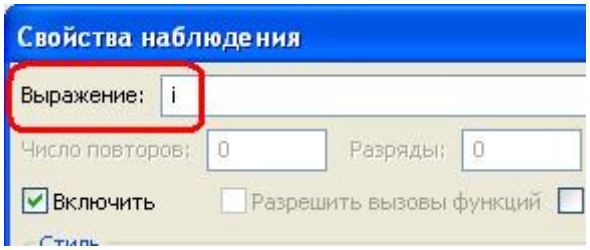


Рис. 2.6. Установка наблюдения за переменной

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

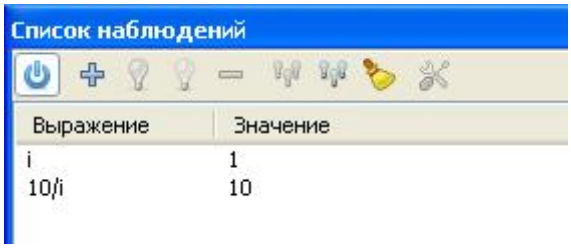


Рис. 2.7. Окно Списка наблюдений

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:

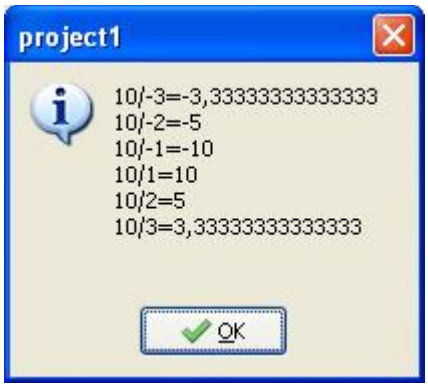


Рис. 2.8. Результирующее сообщение программы

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. Используя приемы отладки, ответьте на следующие вопросы по программе project8.lpr:
4. Каково значение переменной F при вызове подпрограммы Factorial(14), когда:

a) i равно 5
b) i равно 9
c) i равно 12 ?

5. Каково значение переменной Sum при вызове подпрограммы Harmonic(25), когда:

a) i равно 5
b) i равно 10
c) i равно 20 ?

6. 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.

Лабораторная работа № 3. Разработка и отладка приложения «Калькулятор» в среде LAZARUS

Цель лабораторной работы

Освоить работу с встроенным отладчиком инегрированной среды LAZARUS в процессе самостоятельной работы над приложением в соответствии с индивидуальным заданием.

Задание: Используя IDE LAZARUS разработать соответствующее вашему варианту приложение и предварительно протестировать его на работоспособность. В приложении учесть ситуации не корректного ввода данных. Составить отчет о выполненной работе, в которм показать на примерах применение отладочных средств IDE LAZARUS.

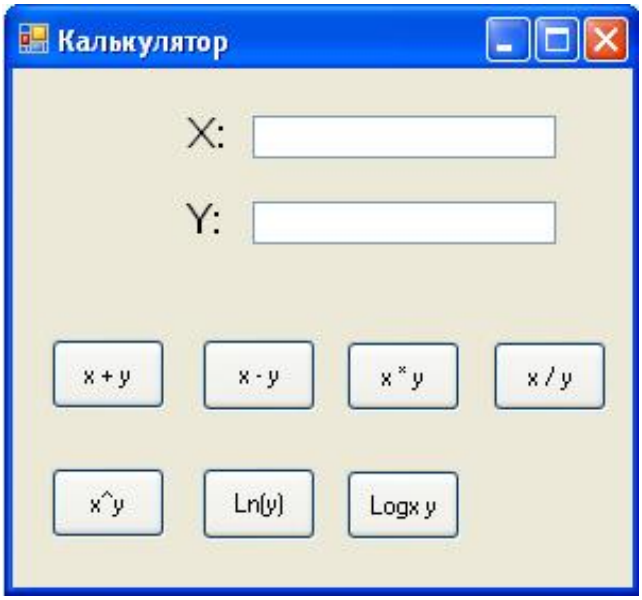
Перечень индиивдуальных заданий:

Вариант 1

Приложение – «Калькулятор»:

Приложение должно содержать два поля ввода: первое для ввода целочисленных переменных x, второе для ввода вещественных переменных y. Семь кнопок для вычисления соответствующих операций: x+y, x-y, x*y, x/y, x^y, lny, $\log_x y$. Получаемый результат должен выводиться в отдельном новом окне мессадж бокса – «Результат».

Примерный внешний вид разрабатываемого приложения:



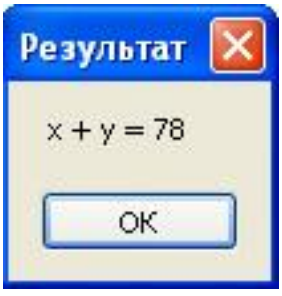


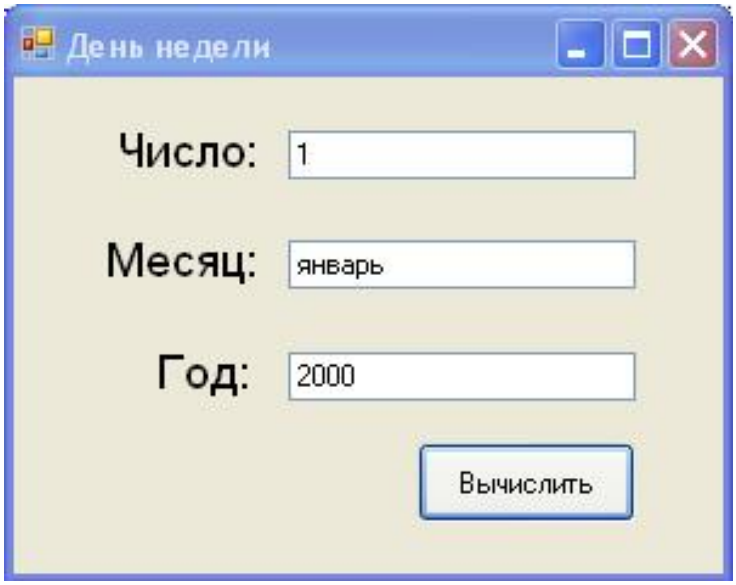
Рис. 3. 1

Вариант 2

Приложение – «День недели»:

Приложение должно содержать три поля ввода: первое для ввода целочисленной переменной Число, второе для ввода текстовой переменной Месяц (здесь возможно использовать как альтернативу содержащий список месяцев combobox), третье для ввода целочисленной переменной Год. Так же должна быть кнопка «Вычислить». Получаемый результат необходимо выводиться в отдельном новом окне messagebox – «Результат».

Примерный внешний вид разрабатываемого приложения:



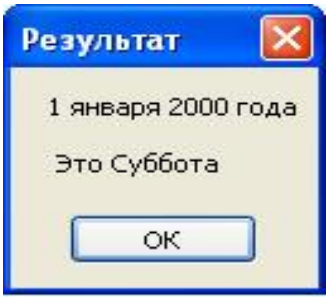


Рис. 3. 2

Вариант 3

Приложение – «Конвектор валют»:

Приложение должно содержать два combobox: первый для выбора Исходной валюты, т.е. валюты с которой будем переводить, второй для выбора Результирующей валюты, т.е. валюты в которую будем переводить. И два поля ввода: первое для ввода целочисленной переменной Исходной суммы, второе для ввода вещественного Курса по которому осуществляется перевод. Так же должна быть кнопка «Перевод». Получаемый результат необходимо выводиться в отдельном новом окне messagebox – «Результат».

Примечание1: Оба combobox должны содержать в себе, по крайней мере, следующие четыре валюты: Рубли, Доллары, Евро, Фунты.

Примечание2: После выбора обеих валют в строке для ввода Курса, должен появиться какой либо установленный по умолчанию в программе курс перевода. Но, пользователь должен иметь возможность изменить этот появившийся «курс по умолчанию», если он его не устраивает.

Примерный внешний вид разрабатываемого приложения:

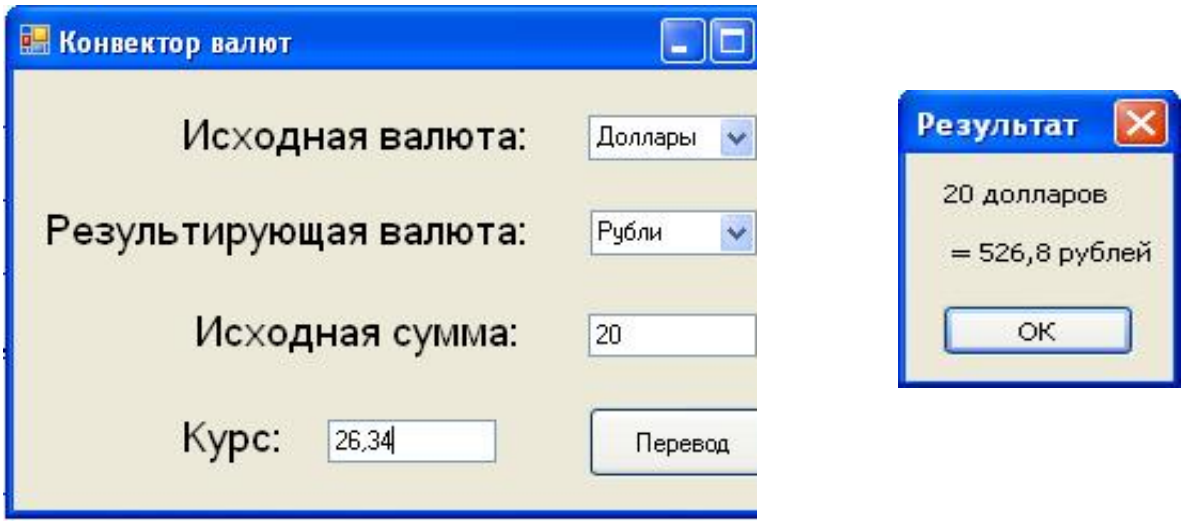


Рис. 3. 3

Вариант 4

Приложение – «Площади/Объёмы фигур»:

Приложение должно содержать один checkbox – 3D-объект, один combobox – Фигура и два поля для ввода. В зависимости от того нажат checkbox или нет, список фигур для выбора должен быть следующим:

- *При не нажатом checkbox:* Квадрат, Круг, Треугольник;
- При нажатом checkbox: Куб, Шар, Пирамида;

Поля для ввода должны отображаться (быть видимыми) в том количестве, которое необходимо для ввода необходимых данных для той или иной фигуры. Так, должны быть видимы: Сторона (для Квадрата), Радиус (для Круга), Основание и Высота (для Треугольника), Сторона (для Куба), Радиус (для Шара), Сторона и Высота (для Пирамиды).

Так же должна присутствовать кнопка «Вычислить», а получаемый результат необходимо выводиться в отдельном новом окне messagebox – «Результат».

Примечание1: Площади фигур вычисляются по следующим формулам:

- $S_{\text{квадрат}} = a^2$
- $S_{\text{круга}} = \pi \cdot r^2$
- $S_{\text{треугольника}} = \frac{1}{2} \cdot h \cdot a$
- $V_{\text{куба}} = a^3$
- $V_{\text{шара}} = \frac{4}{3} \pi \cdot r^3$
- $V_{\text{пирамиды}} = \frac{1}{3} \cdot h \cdot a^2$

Примерный внешний вид разрабатываемого приложения:

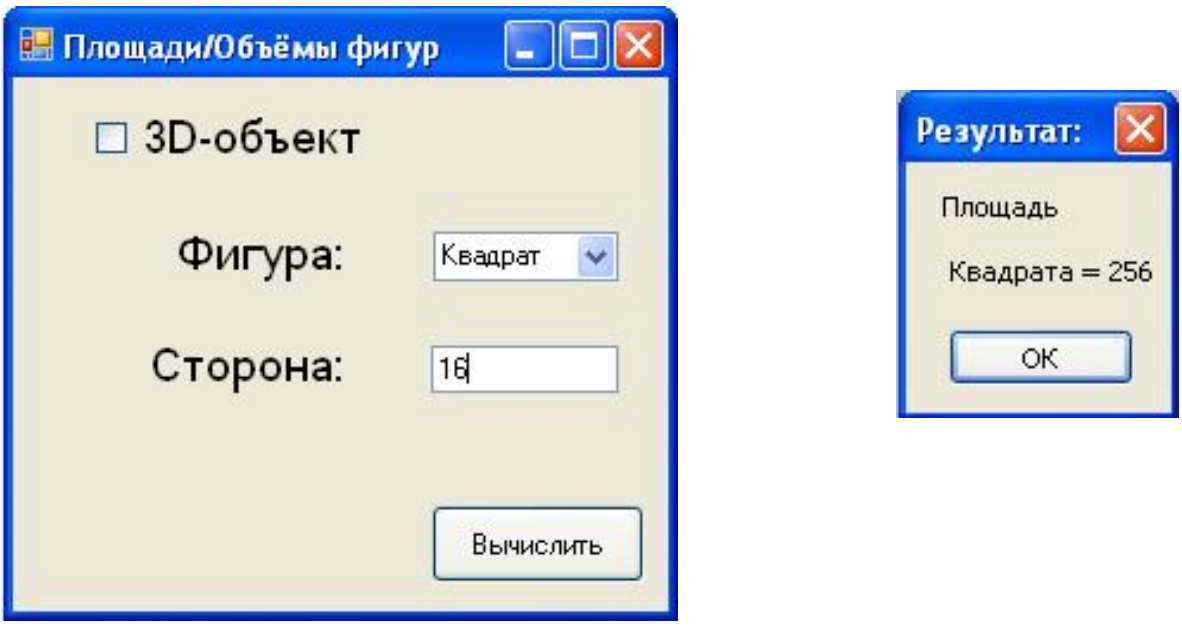


Рис. 3. 4

Вариант 5

Приложение – «Гороскопы»:

Приложение должно содержать три поля для ввода: Число (целое), Месяц (строковое или целое), Год (целое). Два checkbox-са для выбора Пола: Муж и Жен. Так же должна присутствовать кнопка «Определить», выводящая получаемый результат в отдельное окно messagebox – «Результат».

Примечание1: Для выбора пользователем месяца вместо поля для ввода можно использовать combobox содержащий, в себе все 12 месяцев года.

Примечание2: Если сложно реализовать программу, определяющую кроме зодиакального знака ещё и китайский год рождения человека, то от определения года можно в приложении отказаться.

Примерный внешний вид разрабатываемого приложения:

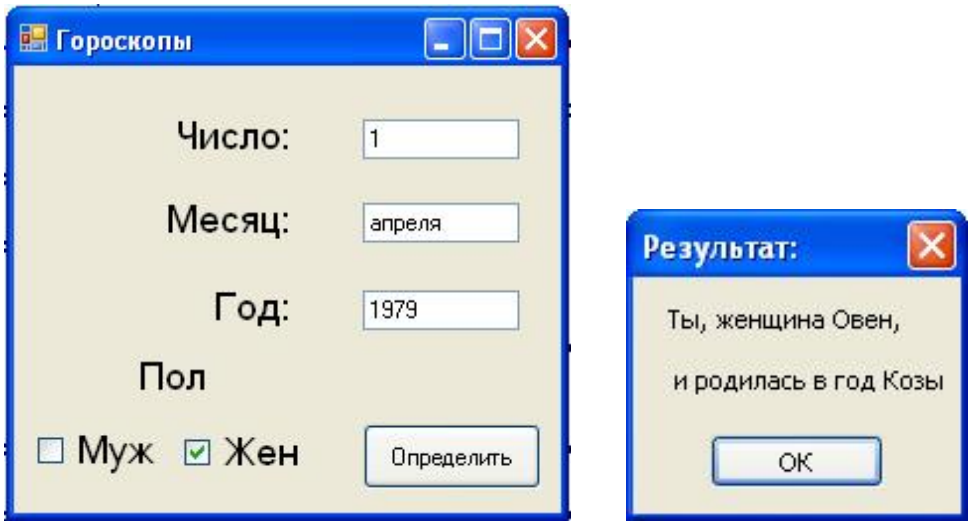


Рис. 3. 5

Контрольные вопросы

1. Каково назначение обработчиков событий? Каким образом можно инициировать создание процедуры-обработчика событий Что такое отладка программного средства?
2. Что такое тестирование программного средства?
3. Что такое автономная отладка программного средства?
4. Что такое комплексная отладка программного средства?
5. Что такое ведущий отладочный модуль?
6. Что такое отладочный имитатор программного модуля?
7. Проблемы и перспективы развития современной программной инженерии.
8. Различные подходы в программировании: «снизу-вверх», «сверху-вниз» (структурный подход), объектно-ориентированный

Лабораторная работа № 4. Наращиваемый подход к тестированию

Цель лабораторной работы

Освоить методику наращиваемого подхода к тестированию. Используя “Наращиваемый подход к тестированию” создать тестовые примеры и провести тестирование разработанной на предыдущем занятии программы.

1. Краткие теоретические сведения

“Наращиваемый подход к тестированию” содержит 8 стадий:

Стадия 1: **Изучение** (Ознакомиться с приложением);

Поработать какое-то время с тестируемым приложением, изучить его…

Например, у нас имеется приложение «Налоговый калькулятор», которое рассчитывает величину подоходного налога, взимаемого с налогоплательщика в США.

Приложение содержит много окон для ввода данных, четыре из которых показаны ниже. В выходном окне перечисляются результаты.

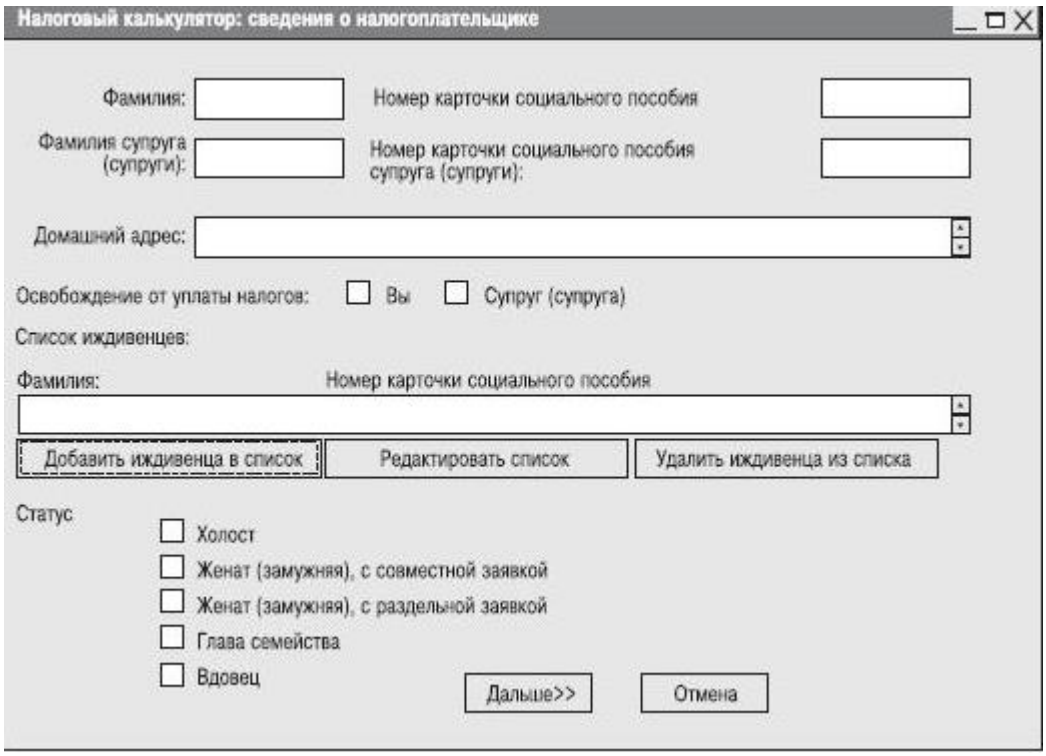


Рис. 4. 1 Налоговый калькулятор. Информация о налогоплательщике

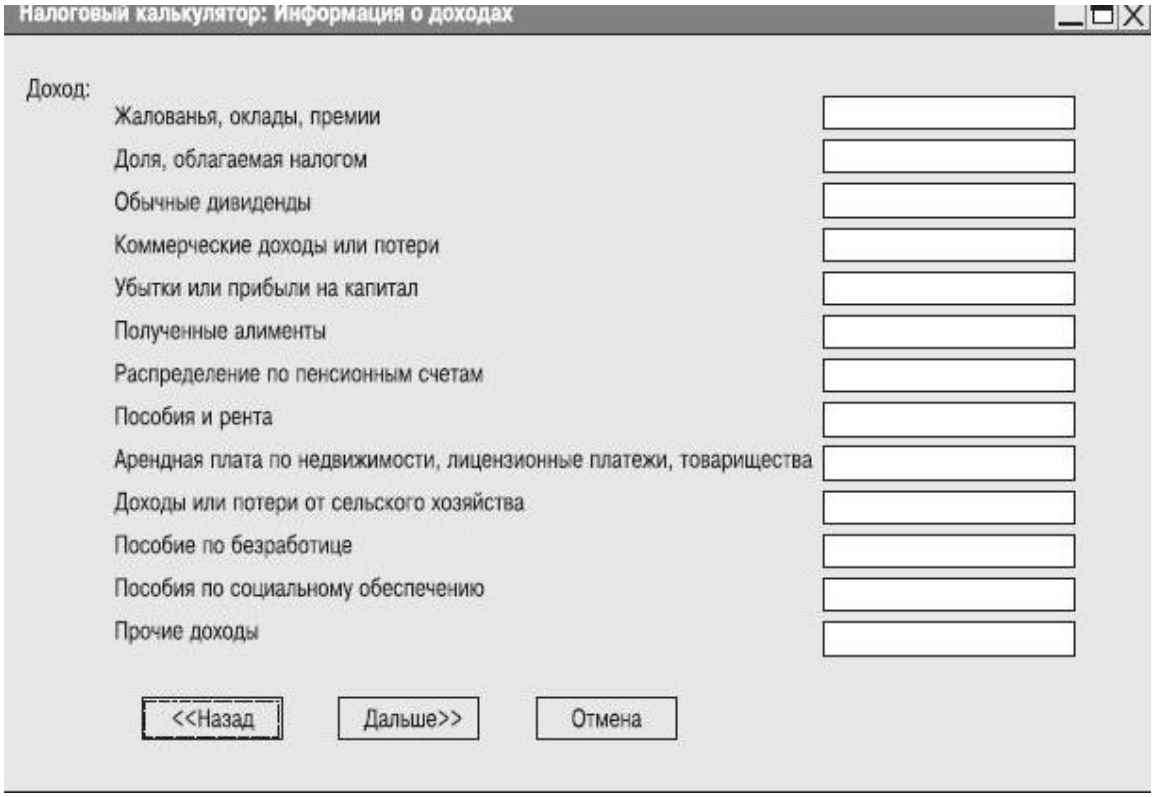


Рис. 4. 2 Налоговый калькулятор. Информация о доходах

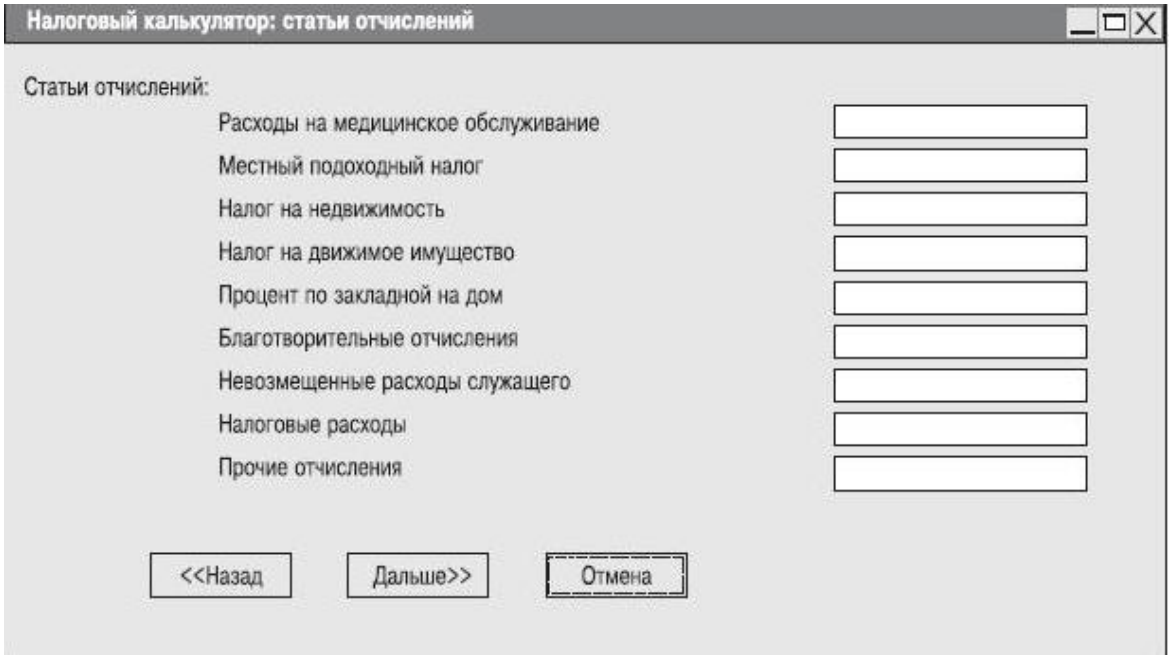


Рис. 4. 3 Налоговый калькулятор. Информация о статьях отчислений

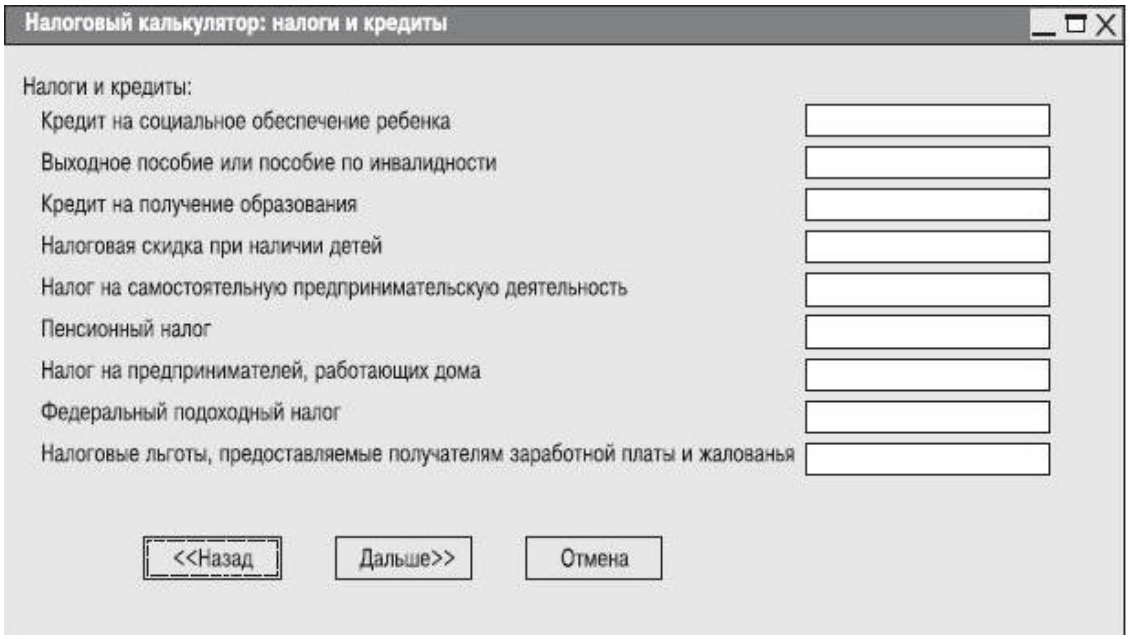


Рис. 4. 4 Налоговый калькулятор. Информация о налогах и кредитах

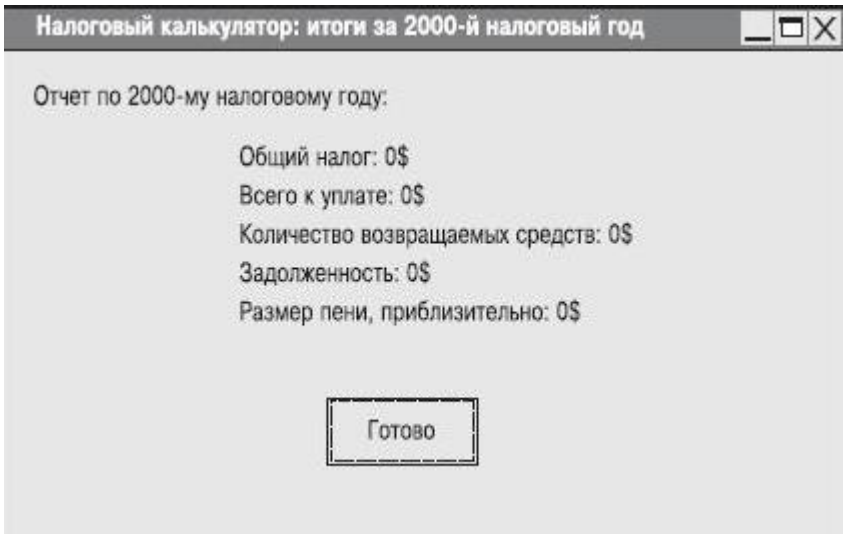


Рис. 4. 5 Налоговый калькулятор. Вычисленные результаты

Стадия 2: **Базовый тест** (Разработать и реализовать один простой тестовый пример);

Разработать при помощи MS-Excel таблицу для базового тестового примера, вбить в неё базовый тестовый пример, провести тест приложения на его основе и указать в таблице полученный при тестировании результат. Сравнить полученный результат с ожидаемым результатом, сделать вывод и отметку в таблице, о том пройден тест или нет.

Таблица 1.1. Базовый тестовый пример

ID тестового примера	Статус	Заработок, ($)	Величина отчислений, ($)	Ожидаемые результаты: причитающийся налог, ($)	Реальный результат
налог 1	одинокий (-ая)	40 000	0	5 779	

Стадия 3: **Анализ тенденций** (Определить, работает ли приложение так, как было задумано, когда ещё нельзя предварительно оценить реальные результаты);

Разработать 2-3 серии из 6-10 тестов и провести анализ тенденций при работе приложения. Результаты и выводы зафиксировать в файле электронной таблицы. (Ниже приводится таблица с одной серией тестов для анализа 1 тенденции по полю «Заработок»)

Таблица 1.2. Базовый тестовый пример

ID тестового примера	Статус	Заработок, ($)	Величина отчислений, ($)	Ожидаемые результаты: причитающийся налог, ($)	Реальный результат
налог 1	одинокий (-ая)	40 000	0	5 779	5 779
тенденция 1	одинокий (-ая)	43 000	0	тенденция к возрастанию	6 619
тенденция 2	одинокий (-ая)	46 000	0		7 459
тенденция 3	одинокий (-ая)	49 000	0		8 299
тенденция 4	одинокий (-ая)	52 000	0		9 139
тенденция 5	одинокий (-ая)	55 000	0		9 979
тенденция 6	одинокий (-ая)	58 000	0		10 819
тенденция 7	одинокий (-ая)	61 000	0		11 659
тенденция 8	одинокий (-ая)	64 000	0		12 499

Стадия 4: **Инвентаризация** (Определить различные категории данных и создать тесты для каждого элемента категории);

Провести инвентаризацию по одному полю данных вашего приложения, т.е. составить серии тестов прорабатывающую все «ветви» для этого поля приложения. (Ниже приводится таблица с серией тестов делающих *инвентаризацию* по полю «Статус» приложения «Налоговый калькулятор»)

Таблица 1.3. Тестовый пример, использующий инвентарный список *Статус*

ID тестового примера	Статус	Заработок, ($)	Величина отчислений, ($)	Ожидаемые результаты: причитающийся налог, ($)	Реальный результат
налог 1	одинокий (-ая)	40 000	44 000	5 779	
налог 2	женат (замужняя) с независимой декларацией доходов	40 000	3 675	6 537	
налог 3	женат (замужняя) с совместной декларацией доходов	40 000	7 350	4 061	
налог 4	глава семьи	40 000	6 450	4 616	
налог 5	вдовец (вдова)	40 000	7 350	4 481	

Стадия 5: **Комбинирование элементов инвентарных списков** (Скомбинировать различные входные данные по разным полям);

Разработать и затем провести тестовые примеры комбинирующих в себе различные элементы 2-х инвентарных списков из стадии 4.

Таблица 1.5. Тестовый пример с комбинацией инвентаризации

		Входные данные						Статьи отчислений						Выходные данные	
ID тестового примера	Статус	Заработок, ($)	Материально зависимые лица	Налог на медицину и стоматологию*, ($)	Государственный и местный налог, ($)	Налог на недвижимость, ($)	Налог на движимое имущество, ($)	Процент по закладной, ($)	Пожертвования благотворительным организациям, ($)	Невозмещенные затраты на содержание наемного работника*, ($)	Плата за подготовку налога, ($)	Прочие отчисления, ($)	Сумма отчислений, ($)	Ожидаемые результаты: причитающийся налог, ($)	Реальные результаты
налог 15	одинокий (-ая)	40000	0	0	2200	1890	80	2800	500	100	50	0	7470	4911	
налог 16	женат (замужняя) с независимой декларацией доходов	60000	0	0	2200	1890	80	2800 0 0 0 0 0	500	100	50	0	7470	11073	

Стадия 6: **Граничные оценки** (Оценить поведение приложения на границах данных и при переходе через них);

Разработать серию тестов, оценивающую границы различных данных. В качестве «границ» могут служить:

- минимальные и максимальные значения диапазона данных;
- минимальный и максимальный размер поля (например, минимальное и максимальное количество введённых в поле символов);
- минимальный и максимальный размер буфера (памяти);
- значение данных при переходе через которое приложение должно вести себя иначе.

Так же напомним, что общее правило тестирования границ – создать три тестовых примера, чтобы охватить следующие значения:

- граничное значение (*значение на границе*);
- граничное значение – 1 (*значение слева от границы*);
- граничное значение + 1 (*значение справа от границы*);

(В приведённом ниже первом примере тестируются две границы, при переходе через которые происходит смена налоговой категории налогоплательщика. Первая граница – это значение заработка 139.519$, при переходе через неё налоговая категория меняется с 31% на 36%. Вторая граница – это заработок в 292.749$, при его увеличении, так же происходит изменение процентов собираемого налога с 36% на 39,6%.)

Таблица 1.8. Тестовый пример, использующий список налоговых ставок

ID тестового примера	Статус	Заработок, ($)	Материально зависимые лица	Отчисления, ($)	Ожидаемые результаты: причитающийся налог, ($)	Реальные результаты	Примечание: величина облагаемой налогом прибыли, ($)	Примечание: м вычисления на
налог 17	одинокий (-ая)	107 200	0	0	25 681		100 000	налоговая кате "31%"
налог 18	одинокий (-ая)	139 519	0	0	35 787		132 599	налоговая кате "31%"
налог 19	одинокий (-ая)	139 520	0	0	35 787		132 600	налоговая кате "36%"
налог 20	одинокий (-ая)	292 749	0	0	91 857		288 349	налоговая кате "36%"
налог 21	одинокий (-ая)	292 750	0	0$	91 857		288 350	налоговая кате "39,6%"

Таблица 1.12. Тест, проверяющий нулевую границу

ID тестового примера	Статус	Заработок, ($)	Материально зависимые лица	Величина отчислений, ($)	Ожидаемые результаты причитающийся налог
налог 26	одинокий (-ая)	0	0	0	0
налог 27	одинокий (-ая)	–1	0	0	Ошибка
налог 28	одинокий (-ая)	0	0	–1	Ошибка
налог 29	одинокий (-ая)	0	–1	0	Ошибка
налог 30	одинокий (-ая)	7 200	0	0	0

Стадия 7: **Ошибочные данные** (Оценить поведение системы при вводе неправильных данных);

Разработать и провести тесты, оценивающие поведение приложения при вводе ошибочных данных.

На этой стадии обычно создаются тесты следующих категорий:

- данные не вводятся в поля вообще для того, чтобы отследить поведение приложения: «вылетит» ли оно или выведет сообщение об ошибке;
- вводятся не верные числовые данные (отрицательные значения в полях, предназначенных по умолчанию для ввода положительных чисел или буквенно-символьные комбинации символов);
- вводятся данные какого-либо формата, который для такого типа данных считается не допустимым;
- используются не обычные комбинации данных;
- проверяется использование нулевого значения, если это ещё не сделано в предыдущих тестах.

Стадия 8: Создание напряжений (Попытаться вывести систему из строя);

Создайте тест, проверяющий, как работает приложение при напряжениях в среде. А именно, проверьте и зафиксируйте, сколько требуется приложению времени на выполнение базового теста, когда в среде работает только оно одно. И сколько требуется времени на базовый тест одному приложению, когда в среде параллельно запущено 15-20 тестируемых приложений.

Контрольные вопросы

1. Перечислите основные стадии наращиваемого подхода к тестированию.
2. Опишите стадию 1 «Изучение»: Цель и описание этой стадии, Основные формы изучения

3. Опишите стадию 2 «Базовое тестирование»: Цель и описание этой стадии, Основные источники определения ожидаемых результирующих данных
4. Опишите стадию 3 «Анализ тенденций»: Цель и описание этой стадии, Условия при выполнение одного из которых она проводиться
5. Опишите стадию 4 «Инвентаризация»: Цель и описание этой стадии
6. Опишите стадию 5 «Комбинирование элементов инвентарных списков»: Цель и описание этой стадии, Два подхода к комбинированию.
7. Опишите стадию 6 «Граничные оценки»: Цель и описание этой стадии, Примеры пределов в зависимости от типов данных.
8. Опишите стадию 7 «Ошибочные данные»: Цель и описание этой стадии, Возможные категории тестов с ошибочными данными.
9. Опишите стадию 8 «Создание напряжений»: Цель и описание этой стадии, Используемые для создания напряжений тесты

Лабораторная работа № 5. Тестирование программ методами белого ящика

Цель работы: Усвоение студентами методов тестирования логики программы, формализованного описания результатов тестирования и стандартов по составлению схем программ**.**

1. Краткие теоретические сведения

Тестирование программного обеспечения охватывает целый ряд видов деятельности, аналогичных последовательности процессов разработки программного обеспечения. В него входят /1/:

а) постановка задачи для теста,

б) проектирование теста,

в) написание тестов,

г) тестирование тестов,

д) выполнение тестов,

е) изучение результатов тестирования.

Решающую роль играет проектирование тестов. Возможны разные подходы к проектированию тестов. Первый состоит в том, что тесты проектируются на основе внешних спецификаций программ и модулей, либо спецификаций сопряжения программы или модуля. Программа при этом рассматривается как черный ящик (стратегия ‘черного ящика’). Существо такого подхода - проверить соответствует ли программа внешним спецификациям. При этом логика модуля совершенно не принимается во внимание.

Второй подход основан на анализе логики программы (стратегия ‘белого ящика’). Существо подхода - в проверке каждого пути, каждой ветви алгоритма. При этом внешняя спецификация во внимание не принимается.

Ни один из этих подходов не является оптимальным. Из анализа существа первого подхода ясно, что его реализация сводится к проверке всех возможных комбинаций значений на входе программы. Тестирование любой программы для всех значений входных данных невозможно, так как их бесконечное множество, поэтому ограничиваются меньшим. При этом исходят из максимальной отдачи теста по сравнению с затратами на его создание. Она измеряется вероятностью того, что тест выявит ошибки, если они имеются в программе. Затраты измеряются временем и стоимостью подготовки, выполнения и проверки результатов теста.

Проанализируем теперь второй подход к тестированию. Даже если предположить, что выполнены тесты для всех путей программы, нельзя с полной уверенностью утверждать, что модуль не содержит ошибок.

Очевидное основание этого утверждения состоит в том, что выполнение всех путей не гарантирует соответствия программы ее спецификациям. Допустим, если требовалось написать программу для вычисления кубического корня, а программа фактически вычисляет корень квадратный, то программа будет совершенно неправильной, даже если проверить все пути. Вторая проблема - отсутствующие пути. Если программа реализует спецификации не полностью (например, отсутствует такая специализированная функция,

как проверка на отрицательное значение входных данных программы вычисления квадратного корня), никакое тестирование существующих путей не выявит такой ошибки. И, наконец, проблема зависимости результатов тестирования от входных данных. Путь может правильно выполняться для одних данных и неправильно для других. Например, если для определения равенства 3 чисел программируется выражение вида:

IF (A+B+C)/3=A,

то оно будет верным не для всех значений A, B и C (ошибка возникает в том случае, когда из двух значений B или C одно больше, а другое на столько же меньше A). Если концентрировать внимание только на тестировании путей, нет гарантии, что эта ошибка будет выявлена.

Таким образом, полное тестирование программы невозможно. Тест для любой программы будет обязательно неполным, то есть тестирование не гарантирует полное отсутствие ошибок в программе. Стратегия проектирования тестов заключается в том, чтобы попытаться уменьшить эту неполноту насколько это возможно.

2.1 Методы стратегии ‘белого ящика’

Тестирование по принципу белого ящика характеризуется степенью, какой тесты выполняют или покрывают логику (исходный текст программы).

2.1.1 Метод покрытия операторов

Целью этого метода тестирования является выполнение каждого оператора программы хотя бы один раз.

Пример:

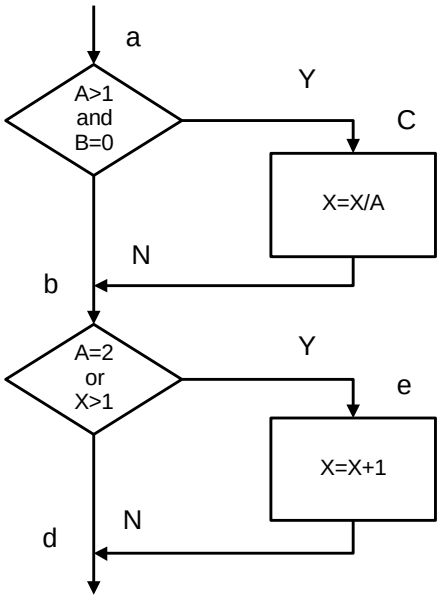


Рисунок 5.1

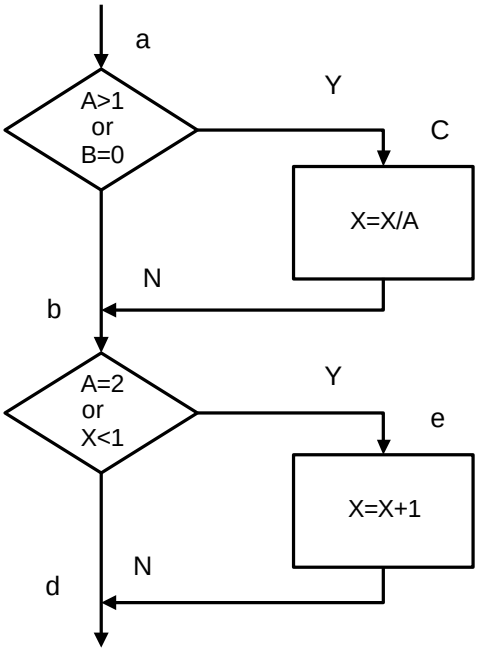


Рисунок 5.2

В этой программе можно выполнить каждый оператор, записав один единственный тест, который реализовал бы путь *ace*. Т.е., если бы на входе было: A=2, B=0, X=3,

каждый оператор выполнился бы один раз. Но этот критерий на самом деле хуже, чем он кажется на первый взгляд. Пусть в первом условии вместо “*and”→“or”* и во втором вместо *“x>1”→“x<1”* (блок-схема правильной программы приведена на рисунке 5.1, а неправильной - на рисунке 5.2). Результаты тестирования приведены в таблице 5.1. Обратите внимание: ожидаемый результат определяется по алгоритму на рисунке 5.1, а фактический - по алгоритму рисунка 5.2, поскольку определяется чувствительность метода тестирования к ошибкам программирования. Как видно из этой таблицы, ни одна из внесенных в алгоритм ошибок не будет обнаружена .

Таблица 5.1 - Результат тестирования методом покрытия операторов

Тест	Ожидаемый результат	Фактический результат	Результат тестирования
A=2, B=0, X=3	X=2,5	X=2,5	неуспешно

2.2 Метод покрытия решений (покрытия переходов)

Более сильный метод тестирования известен как покрытие решений (покрытие переходов). Согласно данному методу каждое направление перехода должно быть реализовано по крайней мере один раз.

Покрытие решений обычно удовлетворяет критерию покрытия операторов. Поскольку каждый оператор лежит на некотором пути, исходящем либо из оператора перехода, либо из точки входа программы, при выполнении каждого направления перехода каждый оператор должен быть выполнен.

Для программы приведенной на рисунке 2.2 покрытие решений может быть выполнено двумя тестами, покрывающими пути {*ace, abd*}, либо {*acd,abe*}. Пути {*acd,abe*}покроим, выбрав следующие исходные данные: {A=3, B=0, X=3} и {A=2, B=1, X=1} (результаты тестирования - в таблице 2.2).

Таблица 5.2 - Результат тестирования методом покрытия решений

Тест	Ожидаемый результат	Фактический результат	Результат тестирования
A=3, B=0, X=3	X=1	X=1	неуспешно
A=2, B=1, X=1	X=2	X=1,5	успешно

2.3 Метод покрытия условий

Лучшие результаты по сравнению с предыдущими может дать метод покрытия условий. В этом случае записывается число тестов, достаточное для того, чтобы все возможные результаты каждого условия в решении выполнялись по крайней мере один раз.

В предыдущем примере имеем четыре условия: {A>1, B=0}, {A=2, X>1}. Следовательно, требуется достаточное число тестов, такое, чтобы реализовать ситуации,

где A>1, A≤1, B=0 и B≠0 в точке *a* и A=2, A≠2, X>1 и X≤1 в точке B. Тесты, удовлетворяющие критерию покрытия условий и соответствующие им пути:

а) A=2, B=0, X=4 *ace*

б) A=1, B=1, X=0 *abd*

Таблица 5.3 - Результаты тестирования методом покрытия условий

Тест	Ожидаемый результат	Фактический результат	Результат тестирования
A=2, B=0, X=4	X=3	X=3	неуспешно
A=1, B=1, X=0	X=0	X=1	успешно

2.4 Критерий решений (условий)

Критерий покрытия решений/условий требует такого достаточного набора тестов, чтобы все возможные результаты каждого условия в решении выполнялись по крайней мере один раз, все результаты каждого решения выполнялись по крайней мере один раз и, кроме того, каждой точке входа передавалось управление по крайней мере один раз.

Два теста метода покрытия условий

а) A=2, B=0, X=4 *ace*

б) A=1, B=1, X=0 *abd*

отвечают и критерию покрытия решений/условий. Это является следствием того, что одни условия приведенных решений скрывают другие условия в этих решениях. Так, если условие A>1 будет ложным, транслятор может не проверять условия B=0, поскольку при любом результате условия B=0, результат решения ((A>1)&(B=0)) примет значение *ложь*. Следовательно, недостатком критерия покрытия решений/условий является невозможность его применения для выполнения всех результатов всех условий.

Другая реализация рассматриваемого примера приведена на рисунке 5.3. Многоусловные решения исходной программы разбиты на отдельные решения и переходы. Наиболее полное покрытие тестами в этом случае выполняется так, чтобы выполнялись все возможные результаты каждого простого решения. Для этого нужно покрыть пути HILP (тест A=2,B=0,X=4), HIMKT (тест A=3, B=1, X=0), HJKT (тест A=0, B=0, X=0), HJKR (тест A=0, B=0, X=2).

Протестировав алгоритм на рисунке 5.3, нетрудно убедиться в том, что критерии покрытия условий и критерии покрытия решений/условий недостаточно чувствительны к ошибкам в логических выражениях.

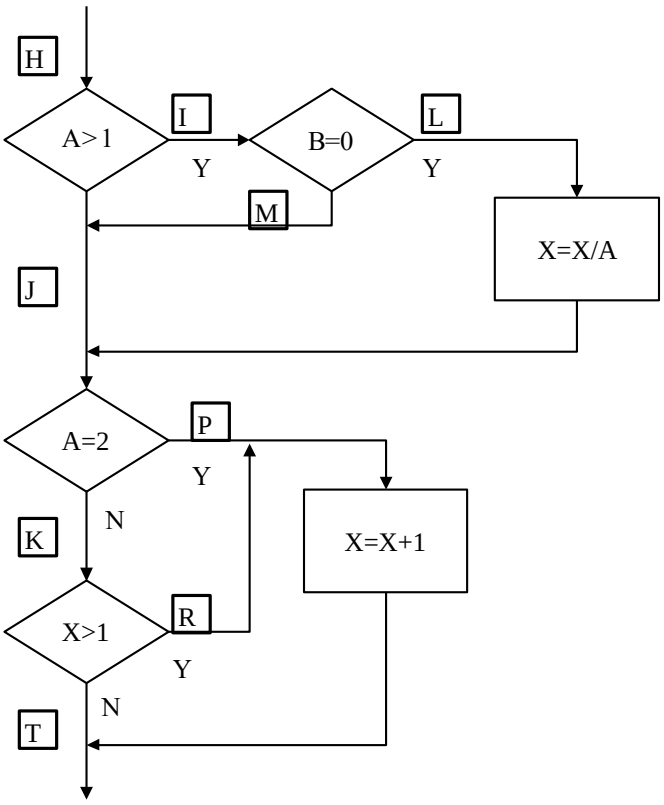


Рисунок 5.3

2.5 Метод комбинаторного покрытия условий.

Критерием, который решает эти и некоторые другие проблемы, является комбинаторное покрытие условий. Он требует создания такого числа тестов, чтобы все возможные комбинации результатов условия в каждом решении выполнялись по крайней мере один раз. Набор тестов, удовлетворяющих критерию комбинаторного покрытия условий, удовлетворяет также и критериям покрытия решений, покрытия условий и покрытия решений/условий.

По этому критерию в рассматриваемом примере должны быть покрыты тестами следующие восемь комбинаций:

а) A>1, B=0;

б)A>1, B≠0;

в) A≤1, B=0;

г) A≤1, B≠0;

д) A=2, X>1;

е) A=2, X≤1;

ж) A≠2, X>1;

з) A≠2, X≤1;

Для того чтобы протестировать эти комбинации, необязательно использовать все 8 тестов. Фактически они могут быть покрыты четырьмя тестами:

- A=2, B=0, X=4 {покрывает а, д};

- A=2, B=1, X=1 {покрывает б, е};
- A=0,5, B=0, X=2 {покрывает в, ж};
- A=1, B=0, X=1 {покрывает г, з}.

Таблица 5.4 - Результаты тестирования методом комбинаторного покрытия условий

Тест	Ожидаемый результат	Фактический результат	Результат тестирования
A=2, B=0, X=4	X=3	X=3	неуспешно
A=2, B=1, X=1	X=2	X=1,5	успешно
A=0,5 B=0, X=2	X=3	X=4	успешно
A=1, B=0, X=1	X=1	X=1	неуспешно

Методика выполнения лабораторной работы

1. Написать программу, реализующую заданный преподавателем алгоритм обработки данных.
2. Отобразить алгоритм решения задачи в виде схемы программы
3. Обозначить буквами или цифрами ветви алгоритма
4. Провести тестирование программного продукта рассмотренными методами.
5. Выписать пути алгоритма, которые должны быть проверены тестами для рассматриваемого метода тестирования.
6. Записать тесты, которые позволят пройти по путям алгоритма,.
7. Протестировать разработанную Вами программу. Результаты оформить в виде таблиц (таблицы 2.1-2.4).
8. Оформить отчет по лабораторной работе.

4 Содержание отчета

1. Цель работы.
2. Программа решения поставленной Вам задачи.
3. Схема программы (см. пп.2,3).
4. Таблицы тестирования программы (п.7).
6. Выводы по результатам тестирования (целью тестирования является обнаружение ошибок в программе).

Контрольные вопросы

1. Какие пункты содержит спецификация программного обеспечения?
2. Что такое управляющий граф программы?
3. Оценка степени тестируемости ПО.
4. Критерии структурного тестирования.
5. Построение управляющего графа программы.
6. Метод покрытия операторов
7. Метод покрытия решений (покрытия переходов)
8. Метод покрытия условий
9. Критерий решений (условий)
10. Метод комбинаторного покрытия условий

Лабораторная работа № 6. Наращиваемое тестирование приложения «Калькулятор» в среде LAZARUS

Цель лабораторной работы

Совершенствовать навыки применения методики наращиваемого подхода к тестированию. Используя “Наращиваемый подход к тестированию” создать тестовые примеры и провести тестирование разработанной на предыдущем занятии программы «Калькулятор».

1. Краткие теоретические сведения

“Наращиваемый подход к тестированию” содержит 8 стадий:

Стадия 1: **Изучение** (Ознакомиться с приложением);

Поработать какое-то время с тестируемым приложением, изучить его…

Стадия 2: **Базовый тест** (Разработать и реализовать один простой тестовый пример);

Разработать при помощи MS-Excel таблицу для базового тестового примера, вбить в неё базовый тестовый пример, провести тест приложения на его основе и указать в таблице полученный при тестировании результат. Сравнить полученный результат с ожидаемым результатом, сделать вывод и отметку в таблице, о том пройден тест или нет.

Стадия 3: **Анализ тенденций** (Определить, работает ли приложение так, как было задумано, когда ещё нельзя предварительно оценить реальные результаты);

Разработать 2-3 серии из 6-10 тестов и провести анализ тенденций при работе приложения. Результаты и выводы зафиксировать в файле электронной таблицы. (Ниже приводится таблица с одной серией тестов для анализа 1 тенденции по полю «Заработок»)

Стадия 4: **Инвентаризация** (Определить различные категории данных и создать тесты для каждого элемента категории);

Провести инвентаризацию по одному полю данных вашего приложения, т.е. составить серии тестов прорабатывающую все «ветви» для этого поля приложения.

Стадия 5: **Комбинирование элементов инвентарных списков** (Скомбинировать различные входные данные по разным полям);

Разработать и затем провести тестовые примеры комбинирующих в себе различные элементы 2-х инвентарных списков из стадии 4.

Стадия 6: **Граничные оценки** (Оценить поведение приложения на границах данных и при переходе через них);

Разработать серию тестов, оценивающую границы различных данных. В качестве «границ» могут служить:

- минимальные и максимальные значения диапазона данных;
- минимальный и максимальный размер поля (например, минимальное и максимальное количество введённых в поле символов);
- минимальный и максимальный размер буфера (памяти);
- значение данных при переходе через которое приложение должно вести себя иначе.

Так же напомним, что общее правило тестирования границ – создать три тестовых примера, чтобы охватить следующие значения:

- граничное значение (*значение на границе*);
- граничное значение – 1 (*значение слева от границы*);
- граничное значение + 1 (*значение справа от границы*);

Стадия 7: **Ошибочные данные** (Оценить поведение системы при вводе неправильных данных);

Разработать и провести тесты, оценивающие поведение приложения при вводе ошибочных данных.

На этой стадии обычно создаются тесты следующих категорий:

- данные не вводятся в поля вообще для того, чтобы отследить поведение приложения: «вылетит» ли оно или выведет сообщение об ошибке;
- вводятся не верные числовые данные (отрицательные значения в полях, предназначенных по умолчанию для ввода положительных чисел или буквенно-символьные комбинации символов);
- вводятся данные какого-либо формата, который для такого типа данных считается не допустимым;
- используются не обычные комбинации данных;
- проверяется использование нулевого значения, если это ещё не сделано в предыдущих тестах.

Стадия 8: Создание напряжений (Попытаться вывести систему из строя);

Создайте тест, проверяющий, как работает приложение при напряжениях в среде. А именно, проверьте и зафиксируйте, сколько требуется приложению времени на выполнение базового теста, когда в среде работает только оно одно. И сколько требуется времени на базовый тест одному приложению, когда в среде параллельно запущено 15-20 тестируемых приложений.

ЗАДАНИЕ:

Используя “Наращиваемый подход к тестированию” создать тестовые примеры и провести тестирование разработанной на предыдущем занятии программы.

Контрольные вопросы

1. Дополнительные методы тестирования:
2. Какие существуют ещё методы, кроме инвентаризации?
3. Что позволяет протестировать подход, основанный на теории графов ?
4. Какие ещё нужны дополнительные тесты?
5. Какие дополнительные дисциплины должно включать в себя качественное тестирование?
6. Основные преимущества проведения тестирования даже при не хватке времени на него,
7. Основная тактика тестирования при нехватке времени,
8. Как определить наиболее важные для тестирования области?
9. Классификация выборочных методов.
10. Основные понятия тестирования: концепция тестирования, подходы.
11. Различия тестирования и отладки.
12. Фазы и технология тестирования.
13. Проблемы тестирования.
14. Критерии выбора тестов: структурные, функциональные, стохастические.
15. Критерии выбора тестов: мутационные, оценки покрытия проекта.

Лабораторная работа № 7. Задача Майерса. Развитие навыков интуитивного тестирования

Цель лабораторной работы

Совершенствовать навыки применения методов тестирования на примере задачи Майерса. Изучить интуитивный подход к тестированию.

1. Краткие теоретические сведения

В повседневной жизни мы постоянно сталкиваемся с необходимостью тестирования программ, с которыми нам приходится работать. Это свежие версии текстовых и графических редакторов, почтовых клиентов и служебных утилит, новые модели мобильных телефонов и обновленный дизайн любимых сайтов. Конечно, такое персональное тестирования чаще всего проводится неформально, в процессе повседневной работы или развлечения, без оформления тест-кейсов и баг-репортов. Тем не менее, в результате практически каждый человек в сознательном возрасте приобретает определенные интуитивные навыки к тестированию программ и программно-управляемых устройств.

В своей книге «Искусство тестирования программ» Гленфорд Майерс предложил простое задание, позволяющее определить интуитивные способности человека к тестированию ПО.

Исходные данные:

Имеется гипотетическая (существующая только в воображении испытуемого) программа. Программа считывает три целочисленных значения из диалогового окна ввода данных. Эти значения интерпретируются как длины сторон треугольника. Программа выводит сообщение о том, каким является данный треугольник — неравносторонним, равнобедренным или равносторонним. Примерно вот так:

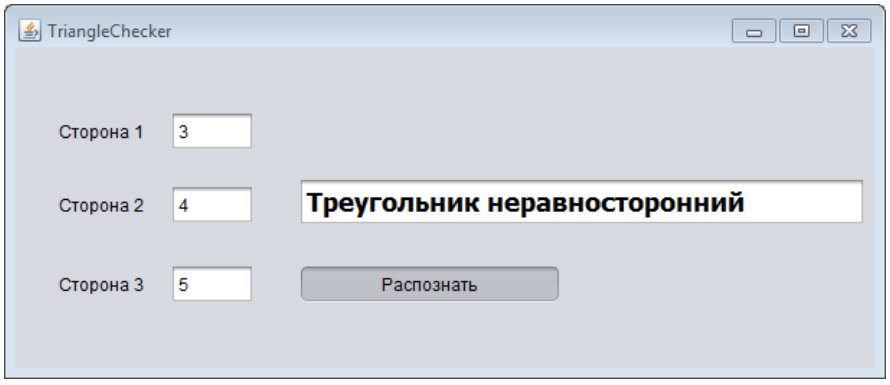


Рис. 7. 3

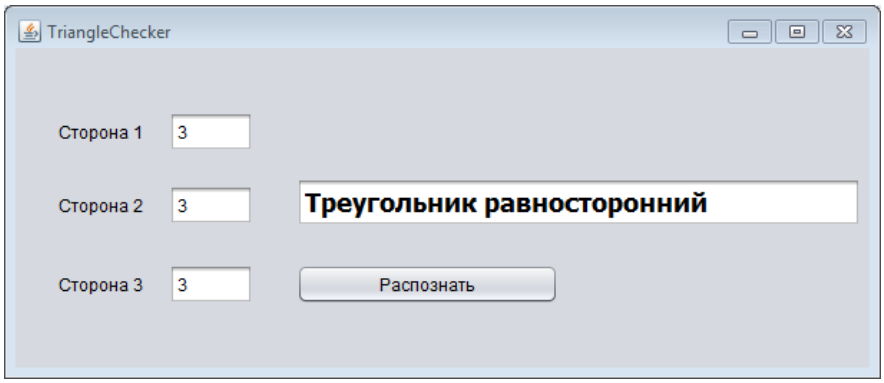


Рис. 7.1

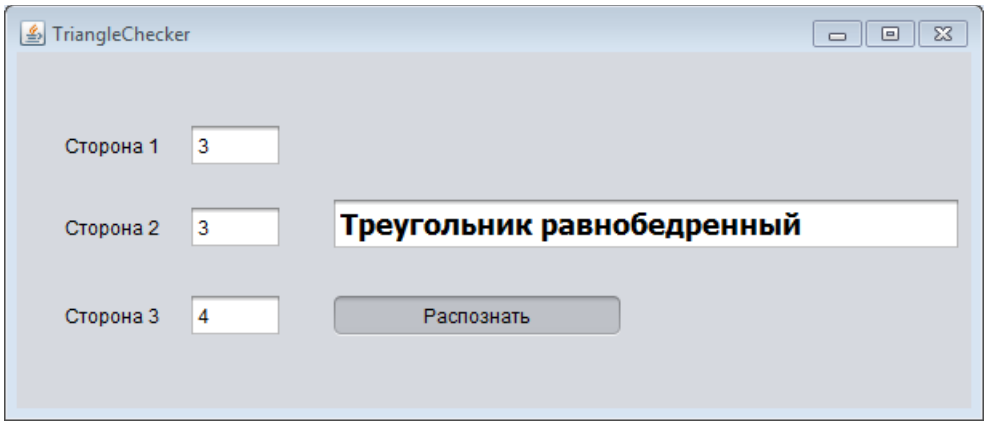


Рис. 7.2

Напомним, что треугольник называется разносторонним (неравносторонним), если все три его стороны не равны друг другу; равнобедренным — если две его стороны равны, а равносторонним — если все три его стороны равны.

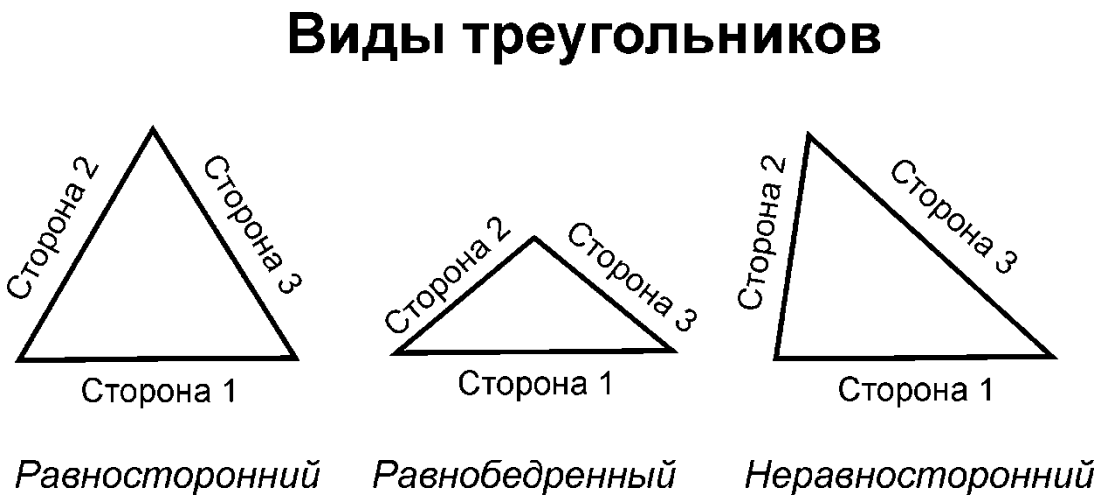


Рис. 7.4

Задание:

Необходимо составить набор тест-кейсов для тестирования описанной выше программы и занести результаты в таблицу.

Вводимые значения			
Сторона 1	Сторона 2	Сторона 3	

Лабораторная работа №1. Ключ к расчету результата:

Проставьте в правом поле ответы на вопросы (да/нет).

1. Есть ли у вас тест-кейс, представляющий действительный неравносторонний треугольник? Учтите, что наличие таких наборов, как 1, 2, 3 или 2, 5, 10, еще не дает права ответить утвердительно, поскольку треугольников с такими сторонами не существует.	
2. Есть ли у вас тест-кейс, представляющий действительный равносторонний треугольник?	
3. Есть ли у вас тест-кейс, представляющий действительный равнобедренный треугольник? Учтите, что такие наборы значений, как 2, 2, 4, не засчитываются, поскольку представляют несуществующие треугольники.	
4. Есть ли у вас хотя бы три тест-кейса, представляющих три действительных равнобедренных треугольника, которые соответствуют различным перестановкам двух равных сторон (например, 3, 3, 4; 3, 4, 3; 4, 3, 3)?	
5. Есть ли у вас тест-кейс, в котором одна из сторон имеет нулевую длину?	
6. Есть ли у вас тест-кейс, в котором длине одной из сторон соответствует отрицательное значение?	
7. Есть ли у вас тест-кейс, включающий такой набор из трех целых положительных чисел, в котором сумма двух из них равна третьему числу? Иными словами, вывод программой сообщения о том, что набор значений 1, 2, 3 представляет неравносторонний треугольник, будет означать, что в программе содержится ошибка.	
8. Есть ли у вас хотя бы три тест-кейса категории 7 для трех перестановок трех положительных чисел, представляющих треугольник, в котором длина одной из сторон равна сумме длин двух других (например, 1, 2, 3; 1, 3, 2,5, 1, 2)?	
9. Есть ли у вас тест-кейс, включающий три положительных целых числа, причем сумма двух из них меньше третьего (например: 1, 2, 4 или 12, 15, 30)?	
10. Есть ли у вас хотя бы три тест-кейса категории 9, которые охватывают три перестановки (например, 1, 2, 4; 1, 4, 2; 4, 1, 2)?	
11. Есть ли у вас тест-кейс, в котором все стороны треугольника имеют нулевую длину (т.е. 0, 0, 0)?	
12. Есть ли у вас хотя бы один тест-кейс, в котором заданы нецелочисленные значения (например, 2.5, 3.5, 5.5)?	
13. Есть ли у вас хотя бы один тест-кейс, в котором задано неверное количество значений (например, указано два целых числа вместо трех)?	
14. Были ли для каждого тест-кейса указаны не только данные на входе, но и ожидаемый результат на выходе программы? Именно для этого предназначалась последняя (никак не озаглавленная) колонка таблицы.	

Посчитайте количество ответов «да» и оцените результат:

0 - 3. Интуитивные навыки тестирования у вас развиты недостаточно. Но не отчаивайтесь, приложив немного усилий, вы можете значительно улучшить свой результат.

3 - 6. В целом неплохо. В этом диапазоне находятся результаты большинства людей, имеющих отношение к IT-технологиям.

6 - 9. Отлично! Такой результат с первого раза, как правило, демонстрируют только профессиональные разработчики ПО.

9 - 14. Супер! Вы либо тестировщик по специальности, либо выполняете задание не в первый раз.

Контрольные вопросы

1. Понятие тестирования ПО. Основные определения.
2. Цели и принципы тестирования.
3. Основная цель тестирования.
4. Уровень доверия, корректное поведение, реальное окружение.
5. Тестирование и качество.
6. Уровни восприятия тестирования в компании.
7. Участники тестирования, их роль, квалификация и обязанности

Лабораторная работа №8. Функциональное тестирование программ

Цель работы: провести функциональное тестирование разработанного программного средства в соответствии с заданным вариантом

1. Краткие теоретические сведения

Процесс тестирования состоит из трёх этапов:

1. Проектирование тестов.
2. Исполнение тестов.
3. Анализ полученных результатов.

На первом этапе решается вопрос о выборе некоторого подмножества множества тестов, которое сможет найти наибольшее количество ошибок за наименьший промежуток времени. На этапе исполнения тестов проводят, запуск тестов и отлавливают ошибки в тестируемом программном продукте.

Виды тестов

Функциональные тесты составляются на уровне спецификации, до решения задачи. Будущий алгоритм рассматривается как «черный ящик» - функция с неизвестной (или не рассматриваемой) структурой, преобразующая входы в выходы. Суть функциональных тестов: каким бы способом ни решалась задача, при заданных входных значениях должны получиться соответствующие выходные значения.

Структурные тесты составляются для проверки логики решения, или логики работы уже готового алгоритма. Логика определяется последовательностью операций, их условным выполнением или повторением (т.е. композицией базовых конструкций). Совокупность структурных тестов должна обеспечить проверку каждой из таких конструкций.

Чаще всего совокупность тщательно составленных функциональных тестов покрывает множество структурных тестов.

Приведенные понятия различаются тем, что первое рассматривает программу только с точки зрения входов и выходов, тогда как второе относится к ее структуре; но оба понятия не касаются процесса организации тестирования.

Общая последовательность разработки тестов

Наиболее рациональная процедура заключается в том, что сначала разрабатываются функциональные тесты, а затем – структурные.

Функциональное тестирование (тестирование «черного ящика»)

При функциональном тестировании выявляются следующие категории ошибок:

- некорректность или отсутствие функций;
- ошибки интерфейса;
- ошибки в структурах данных;
- ошибки машинных характеристик (нехватка памяти и др.);
- ошибки инициализации и завершения.

Техника тестирования ориентирована:

- на сокращение необходимого количества тестовых вариантов;
- на выявление классов ошибок, а не отдельных ошибок.

Способы функционального тестирования

Разбиение на классы эквивалентности

Это самый популярный способ. Его суть заключается в разделении области входных данных программы на классы эквивалентности и разработке для каждого класса одного тестового варианта.

Класс эквивалентности – набор данных с общими свойствами, в силу чего при обработке любого набора данных этого класса задействуется один и тот же набор операторов1.

Классы эквивалентности определяются по спецификации программы. Тесты строятся в соответствии с классами эквивалентности, а именно: выбирается вариант исходных данных некоторого класса и определяются соответствующие выходные данные.

Самыми общими классами эквивалентности являются классы допустимых и недопустимых (аномальных) исходных данных. Описание класса строится как комбинация условий, описывающих каждое входное данное.

Условия допустимости или недопустимости данных задают возможные значения данных и могут описывать:

- некоторое конкретное значение; определяется один допустимый и два недопустимых класса эквивалентности: заданное значение, множество значений меньше заданного, множество значений больше заданного;
- диапазон значений; определяется один допустимый и два недопустимых класса эквивалентности: множество значений в границах диапазона; множество значений, выходящих за левую границу диапазона; множество значений, выходящих за правую границу диапазона;
- множество конкретных значений; определяется один допустимый и один недопустимый класс эквивалентности: заданное множество и множество значений, в него не входящих.

Такие классы можно описать языком логики, например, языком исчисления предикатов. Описания более сложных условий и соответствующих классов (например, элементы массива должны находиться в некотором диапазоне и при этом массив не должен содержать нулевых элементов) могут быть построены на основании приведенных выше условий.

Анализ граничных значений

Этот способ построения тестов дополняет предыдущий и предполагает анализ значений, лежащих на границе допустимых и недопустимых данных. Построение таких тестов часто диктуется интуицией.

Основные правила построения тестов:

- если условие правильности данных задает диапазон, то строятся тесты для левой и правой границы диапазона; для значений чуть левее левой и чуть правее правой границы;

- если условие правильности данных задает дискретное множество значений, то строятся тесты для минимального и максимального значений; для значений чуть меньше минимума и чуть больше максимума;
- если используются структуры данных с переменными границами (массивы), то строятся тесты для минимального и максимального значения границ.

Диаграммы причин-следствий

Взаимосвязь классов эквивалентности и соответствующих им действий описывается формально в виде графа на основе автоматного подхода. Граф преобразуется в таблицу решений, столбцы которой в свою очередь преобразуются в тестовые варианты.

Порядок выполнения работы

1) По результатам лабораторных работ 1-7 разработать тестовые наборы для функционального тестирования .
2) Провести тестирование программы и представить результаты в виде таблицы. (Приложение 1).
3) Выработать рекомендации для корректировки тестируемой программы.
4) Представить отчет по лабораторной работе для защиты.

Защита отчета по лабораторной работе

Защита отчета по лабораторной работе заключается в предъявлении преподавателю полученных результатов, демонстрации полученных навыков в ответах на вопросы преподавателя.

Контрольные вопросы

1. Что такое тестирование ПС?
2. Чем тестирование отличается от отладки ПС?
3. Для чего проводится функциональное тестирование?
4. Что такое комплексное тестирование?
5. Каковы правила тестирования программы «как черного ящика»?
6. Как проводится тестирования программы по принципу «белого ящика»?
7. Что такое модульное тестирование?
8. Как осуществляется сборка программы при модульно тестировании?

Приложение 1

Шаблон таблицы для представления результатов

Тест (значения для	Ожидаемый результат (значения	Фактический результат	Результат тестирования

входных данных)	для выходных данных)	(полученные значения выходных данных)	(успешно/неуспешно)

Рекомендуемая литература

1.1. Основная литература

Код	Авторы, составители	Заглавие	Издательство, год	Кол.
Л1.1	Соловьев С. В., Цой Р. И., Гринкруг Л. С.	Технология разработки прикладного программного обеспечения /	Издательство: Академия Естествознания, 2011.- ISBN: 978-5-91327-158-7	Э1
Л1.2	Джон Роббинс	Отладка Windows-приложений.	Саратов: Профобразование, 2017.— 447 с	Э3
Л1.3	Котляров В.П.	Основы тестирования программного обеспечения	М.: Интернет-Университет Информационных Технологий (ИНТУИТ), 2016.— 334 с.	Э4

6.1.2 Дополнительная литература

Код	Авторы, составители	Заглавие	Издательство, год	Кол.
Л2.1	Синицын С.В.	Верификация программного обеспечения	Москва, Саратов: Интернет-Университет Информационных Технологий (ИНТУИТ), Вузовское образование, 2017.— 368 с	Э5
Л2.2	Сергеев С.Ф.	Методы тестирования и оптимизации интерфейсов информационных систем	СПб.: Университет ИТМО, 2013.— 117 с	Э6
Л2.3	Липаев В.В.	Сертификация программных средств	М.: СИНТЕГ, 2010.— 338 с.	Э7

6.1.3 Методическое обеспечение для самостоятельной работы обучающихся

Код	Авторы, составители	Заглавие	Издательство, год	Кол.
Л3.1	Т.Н. Ананьева, Н.Г. Новикова, Г.Н. Исаев.	Стандартизация, сертификация и управление качеством программного обеспечения : учеб. пособие	/— М. : ИНФРА-М, 2017. — 232 с	Э8
Л3.2	Монахов В.В.	Язык программирования Java и среда NetBeans: Курс лекций / -	СПб:БХВ-Петербург, 2011. - 703 с	Э9
Л1.2	Гуриков С.Р.	Программирование в среде Lazarus для школьников и студентов: Учебное пособие/ -	М.: Форум, НИЦ ИНФРА-М, 2016. - 336 с.	Э2

2 Электронные образовательные ресурсы	
Э1	https://monographies.ru/en/book/section?id=4632
Э2	http://znanium.com/bookread2.php?book=520628
Э3	http://znanium.com/bookread2.php?book=407747
Э4	http://www.intuit.ru/studies/courses/48/48/info
Э5	http://www.intuit.ru/studies/courses/1040/209/info
Э6	http://window.edu.ru/resource/441/80441/files/itmo1363.pdf
Э7	http://www.computer-museum.ru/books/lipaev/lip_sertifikacia.pdf
Э8	http://znanium.com/bookread2.php?book=792682
Э9	http://znanium.com/bookread2.php?book=355260
3 Программное обеспечение	
П.1	MS Windows
П.2	Система визуального программирования Lazarus
П.3	Пакет программ для проведения тестирования по изученным темам
П.4	Пакет презентаций MS Power Point