

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ  
РОССИЙСКОЙ ФЕДЕРАЦИИ  
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ  
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
«ОРЛОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ  
ИМЕНИ И.С. ТУРГЕНЕВА»

Кафедра программной инженерии

Работа допущена к защите

\_\_\_\_\_ Руководитель

« \_\_\_\_ » \_\_\_\_\_ 20 \_\_\_\_ г.

**КУРСОВОЙ ПРОЕКТ**

по дисциплине «Качество и тестирование программного обеспечения»

на тему: «Организация и проведение комплексного тестирования  
программного обеспечения для проведения тестирования пользователей по  
заданным темам»

Студент \_\_\_\_\_ Шорин В.Д.

Шифр 171406

Институт приборостроения, автоматизации и информационных технологий

Направление подготовки 09.03.04 «Программная инженерия»

Группа 71-ПГ

Руководитель \_\_\_\_\_ Ужаринский А.Ю.

Оценка: « \_\_\_\_\_ » Дата \_\_\_\_\_

Орел 2020

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ  
РОССИЙСКОЙ ФЕДЕРАЦИИ  
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ  
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
«ОРЛОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ  
ИМЕНИ И.С. ТУРГЕНЕВА»

Кафедра программной инженерии

УТВЕРЖДАЮ:

\_\_\_\_\_ Зав. кафедрой

«\_\_» \_\_\_\_\_ 20\_\_ г.

**ЗАДАНИЕ**  
**на курсовой проект**

по дисциплине «Качество и тестирование программного обеспечения»

Студент Шорин В.Д.

Шифр 171406

Институт приборостроения, автоматизации и информационных технологий

Направление подготовки 09.03.04 «Программная инженерия»

Группа 71-ПГ

1 Тема курсового проекта

«Организация и проведение комплексного тестирования программного обеспечения для проведения тестирования пользователей по заданным темам»

2 Срок сдачи студентом законченной работы «16» мая 2020

### 3 Исходные данные

Описание задачи, требования к разрабатываемому программному обеспечению.

### 4 Содержание курсового проекта


Проектирование и реализация программного обеспечения, тестирование программного обеспечения, оценка качества разработанного программного обеспечения.

### 5 Отчетный материал курсового проекта

Пояснительная записка курсового проекта; приложение, записанное на CD-диске

Руководитель \_\_\_\_\_ Ужаринский А.Ю.

Задание принял к исполнению: «16» марта 2020

Подпись студента \_\_\_\_\_ 

## Содержание

ВВЕДЕНИЕ .....	4
1. ПРОЕКТИРОВАНИЕ И РЕАЛИЗАЦИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ	6
1.1 Анализ требований к разрабатываемому программному обеспечению.....	6
1.2 Проектирование разрабатываемого программного обеспечения .....	6
1.2.1. Общий вид архитектуры разрабатываемого программного обеспечения....	7
1.2.2. Модуль пользовательского интерфейса .....	8
1.2.3. Модуль данных .....	9
1.2.4. Модуль логики программы.....	9
1.3 Реализация разрабатываемого программного обеспечения .....	11
1.3.1 Реализация модуля пользовательского интерфейса .....	12
1.3.2 Реализация модуля данных .....	15
1.3.3 Реализация модуля логики программы.....	16
2. ТЕСТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ .....	18
2.1 Организация процесса тестирования программного обеспечения .....	18
2.2 Тестирование элементов разработанного программного обеспечения.....	20
2.3 Тестирование интеграции модулей разработанного программного обеспечения .....	26
2.4 Тестирование правильности разработанного программного обеспечения.....	29
2.5 Системное тестирование разработанного программного обеспечения.....	33
3. ОЦЕНКА КАЧЕСТВА РАЗРАБОТАННОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.....	36
3.1 Статическая оценка качества разработанного программного обеспечения .....	37
3.2 Динамическая оценка качества разработанного программного обеспечения..	41
ЗАКЛЮЧЕНИЕ .....	45
СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ .....	46
Приложение А .....	47
(обязательное).....	47
Листинг программы .....	47

## ВВЕДЕНИЕ

Тестирование программного обеспечения — процесс исследования, испытания программного продукта, имеющий своей целью проверку соответствия между реальным поведением программы и её ожидаемым поведением на конечном наборе тестов, выбранных определённым образом.

Тестирование играет жизненно важную роль в разработке программного обеспечения и является неотъемлемой частью жизненного цикла его разработки, так как:

- повышает надежность, качество и производительность программного обеспечения;
- помогает разработчику проверить, правильно ли работает программное обеспечение, убедиться, что программное обеспечение выполняет то, для чего оно предназначено;
- помогает понять разницу между фактическим и ожидаемым результатом, что обеспечивает качество продукта.

Благодаря тому, что тестирование ПО становится частью программирования, разработчики имеют возможность исправлять ошибки уже на начальной стадии разработки. Это позволяет сократить риск появления дефектов в готовом продукте. Если ошибки найдены на начальном уровне, разработчик может создать надежное программное обеспечение. Таким образом, чем раньше начинается процесс, тем раньше обнаруживаются ошибки и тем ниже стоимость их исправления.

**Целью** курсового проекта является организация и проведение комплексного тестирования для подтверждения качества разработанного программного обеспечения.

**Задачи**, которые необходимо решить для достижения поставленной цели:

- 1) изучить задачу тестирования пользователей по заданным темам;
- 2) спроектировать программное обеспечение (ПО) для решения задачи тестирования пользователей по заданным темам;

3) реализовать ПО для решения задачи тестирования пользователей по заданным темам;

4) провести комплексное тестирование разработанного ПО;

5) оценить качество разработанного ПО.

В рамках данного курсового проекта необходимо реализовать и протестировать программу для проведения тестирования пользователей по заданным темам, в котором пользователю даются на выбор заранее подготовленные темы, знания по которым он может проверить.

## **1. ПРОЕКТИРОВАНИЕ И РЕАЛИЗАЦИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ**

### **1.1 Анализ требований к разрабатываемому программному обеспечению**

Тесты – стандартизованные задачи, выполнение которых позволяет измерить некоторые психологические, интеллектуальные характеристики, уровень знаний. Тематические тесты (тесты на заданную тему) позволяют относительно легко, быстро и объективно оценить качество знаний тестируемого в данной области науки, искусства, развлечений и т.д. Они также дают возможность за достаточно короткий промежуток времени оценить показатель успеваемости в данной сфере или осведомленности по данной проблеме большого количества людей.

В разрабатываемом программном обеспечении необходимо предоставить пользователю возможность выбрать тематику тестирования (для каждой тематики свой файл) и ввод своего имени (не обязательно реального, можно и любого вымышленного, т.н. «никнейм», что снимает ограничение на использование каких-либо символов при вводе). Вопросы теста не должны повторяться. Если пользователь не выбирает никакой вариант ответа, то ему засчитывается, что он ответил неправильно и он не получает балла. Программа должна выставлять оценку по следующему правилу: *«отлично»* – за правильные ответы на 86-100% вопросов, *«хорошо»* – если испытуемый правильно ответил на 71-85% вопросов, *«удовлетворительно»* – если правильных ответов 60-70%, *«плохо»* – если правильных ответов менее 60%.

Также, разрабатываемое в ходе курсового проекта программное обеспечение должно быть модульным. Необходимо реализовать базовый интерфейс, который позволит пользователю воспользоваться продуктом, не прикладывая больших усилий.

### **1.2 Проектирование разрабатываемого программного обеспечения**

Проектирование программного обеспечения – процесс определения архитектуры, компонентов, интерфейсов и других характеристик системы или её части. Целью проектирования является определение внутренних свойств системы

и детализации её внешних (видимых) свойств на основе требований к ПО (исходные условия задачи). Эти требования подвергаются анализу.

### **1.2.1. Общий вид архитектуры разрабатываемого программного обеспечения**

Разрабатываемое в ходе курсового проекта ПО должно быть модульным.

Модульность – это свойство системы, связанное с возможностью её декомпозиции на ряд внутренне связанных между собой модулей.

Модуль – функционально законченный фрагмент программы, отличающийся максимальной независимостью от других модулей.

Разрабатываемое ПО будет состоять из 3 модулей:

- модуль пользовательского интерфейса, отвечающий за представление работы ПО пользователю;
- модуль данных, хранящий информацию о тестах и другую информацию, необходимую в процессе работы ПО;
- модуль логики программы, включающий в себя обработку действий пользователя, обработку результата и т.д.

Также для реализации ПО необходим вспомогательный пакет подключаемых библиотек.

Общий вид архитектуры разрабатываемого программного обеспечения представлен на рисунке 1.1.



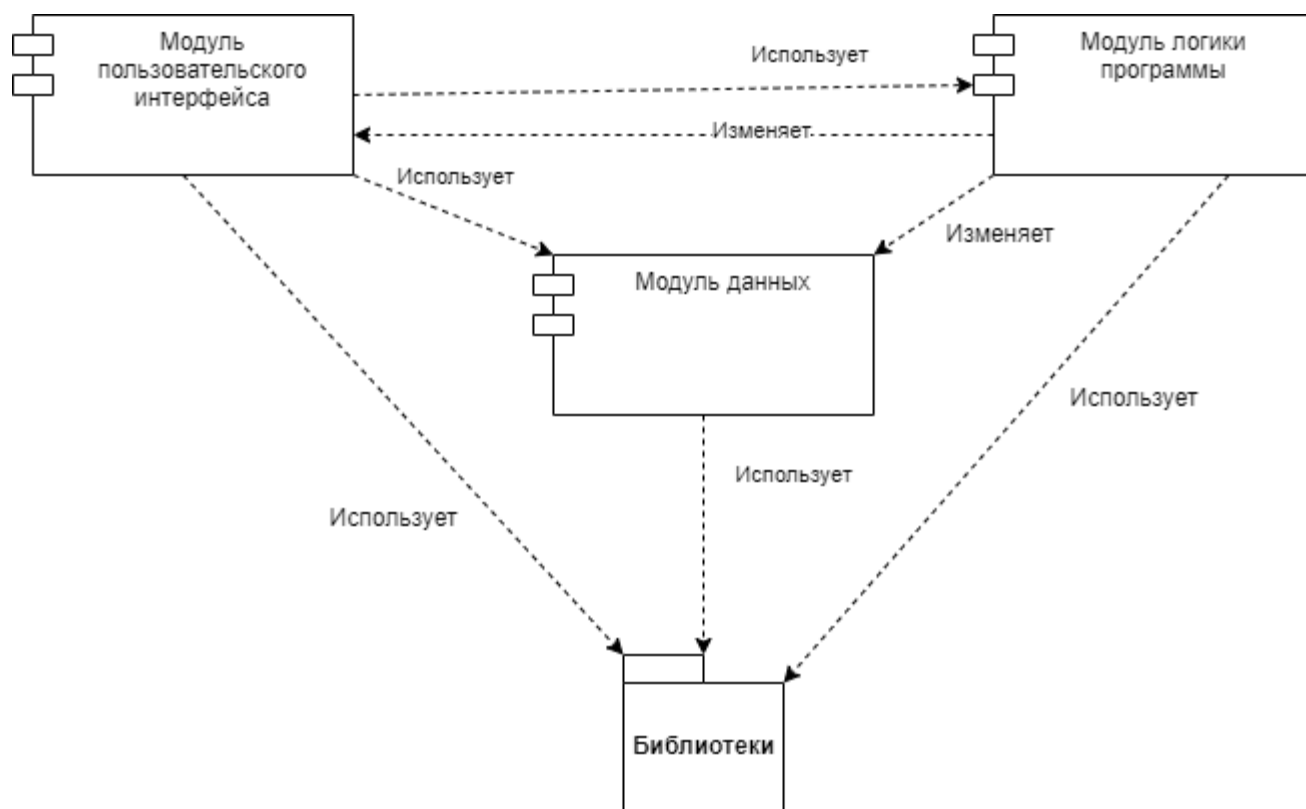


Рисунок 1.1 – Общий вид архитектуры разрабатываемого ПО

### 1.2.2. Модуль пользовательского интерфейса

При запуске ПО пользователю предоставляется возможность ввода своего имени (никнейма), перехода к выбору теста или выхода из программы. Когда пользователь переходит в меню выбора теста, ему предоставляется три темы, из которых он может выбрать любую и пройти по ней тестирование. Также, он может вернуться в главное меню. После выбора интересующей темы пользователь попадает в окно тестирования, где он отвечает на вопросы. В любой момент пользователь может вернуться в меню выбора теста и выбрать другую тему. Когда пользователь отвечает на последний вопрос, то ему выводится новое окно, где содержится результат тестирования. После ознакомления с результатами пользователь попадает в меню выбора тематики тестирования.

Результат проектирования работы модуля пользовательского интерфейса представлен на диаграмме состояний на рисунке 1.2.

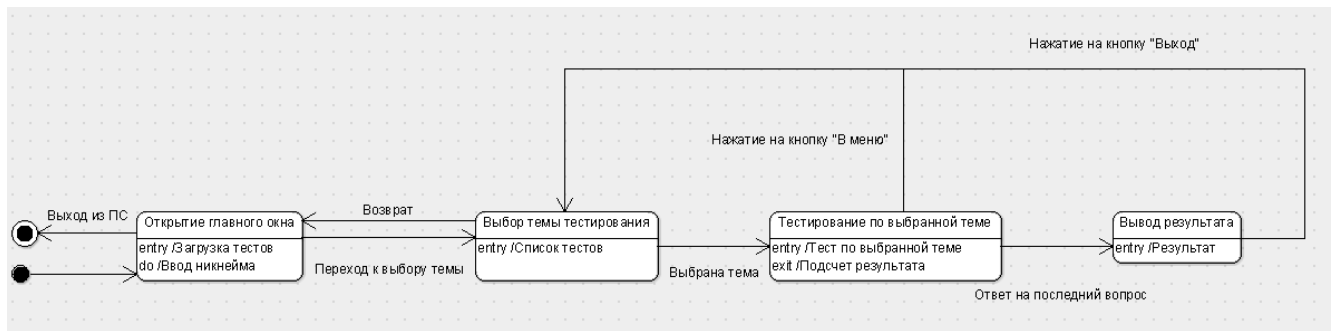


Рисунок 1.2 – Диаграмма состояний модуля пользовательского интерфейса

### 1.2.3. Модуль данных

Модуль данных представляет собой файл с кодом, в котором хранятся:

- необходимая информация о вопросе (в виде класса с его полями и методами);
- массивы с вопросами по каждому тесту;
- остальная дополнительная информация (никнейм пользователя, количество правильных ответов при прохождении теста и т.д.).

### 1.2.4. Модуль логики программы

Модуль логики программы представляет собой все обработчики событий действий пользователя в системе, а также функцию, которая обеспечивает загрузку всех тестов из файлов.

В большинстве случаев обработчики событий представляют собой тривиальные действия, как: открытие нового окна, закрытие текущего окна, запоминание значения и т.д. Потому, наибольший интерес здесь будут представлять следующие функции:

- функция загрузки теста из файла;
- функция обработки нажатия кнопки «Следующий вопрос».

Опишем алгоритм работы функции загрузки теста из файла. В общем виде данный алгоритм описывает следующая последовательность действий:

1. Открыть файл с тестом по переданному пути.
2. Считать информацию о вопросе

3. Добавить вопрос в массив вопросов.
  4. Повторять действия 2-3 пока не закончится файл.
  5. Вернуть массив вопросов.
- Общая схема алгоритма представлена на рисунке 1.3.

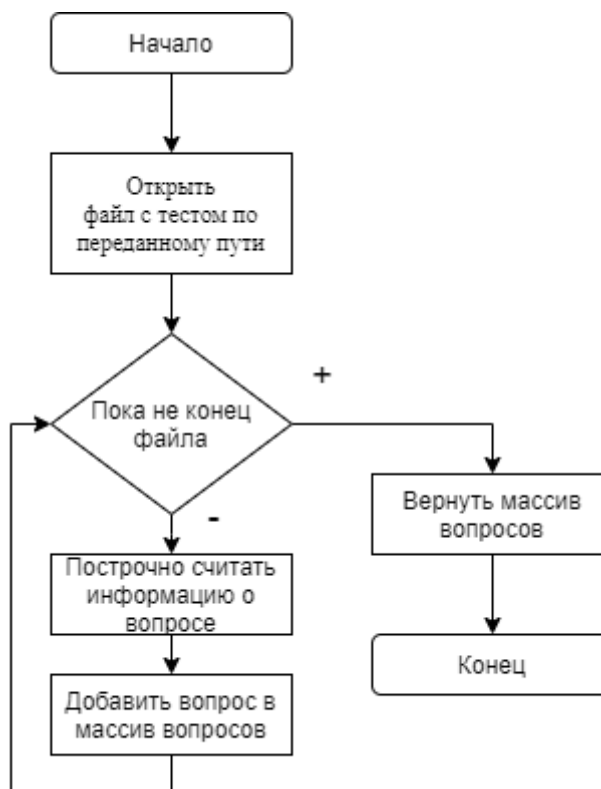


Рисунок 1.3 – Общая схема алгоритма считывания теста из файла

Опишем алгоритм работы функции обработки нажатия кнопки «Следующий вопрос». В общем виде данный алгоритм описывает следующая последовательность действий:

1. Если выбранный вариант ответа совпадает с правильным, то увеличить количество правильных ответов в соответствующей переменной
2. Удалить текущий вопрос из массива вопросов.
3. Если количество оставшихся вопросов равно одному, то заменить надпись «Следующий вопрос» на «Результат» и загрузить следующий вопрос.
4. Если количество оставшихся вопросов равно нулю, то загрузить форму с результатом.
5. В остальных случаях загрузить следующий вопрос.

Общая схема алгоритма представлена на рисунке 1.4.

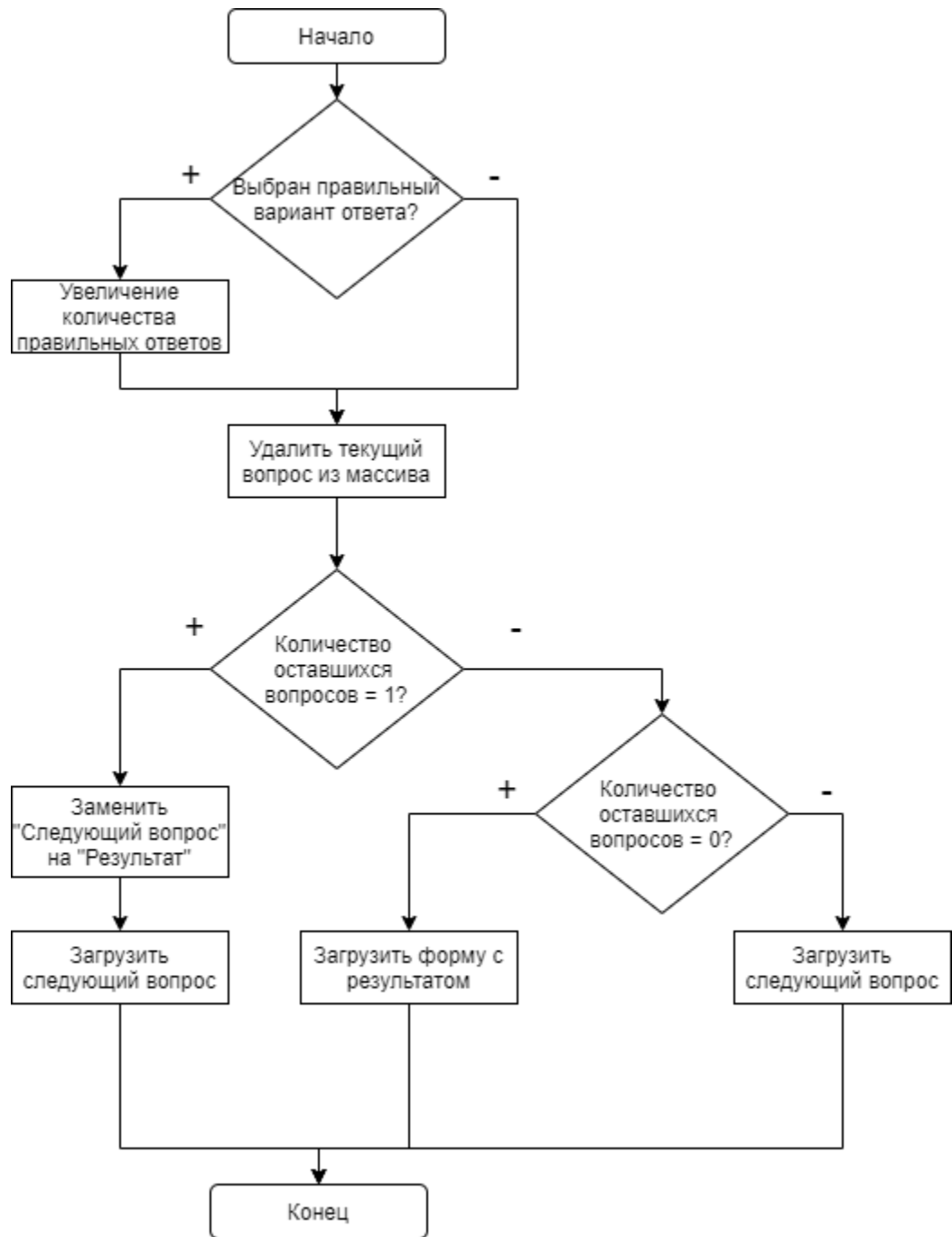


Рисунок 1.4 – Общая схема алгоритма работы функции обработки нажатия кнопки «Следующий вопрос»

### 1.3 Реализация разрабатываемого программного обеспечения

Для реализации программного обеспечения был выбран язык C#, а для реализации пользовательского интерфейса WinForms в среде Visual Studio 2019.

### 1.3.1 Реализация модуля пользовательского интерфейса

Модуль пользовательского интерфейса реализуется с помощью средств WinForms. Его суть состоит в создании форм (окон приложения) на которых располагаются необходимые элементы (такие, как кнопки, текст, поля ввода и т.д.), которые в последствии могут заполняться необходимыми данными посредством модуля логики программы при необходимости. Также, он воздействует на модуль логики программы в момент совершения пользователем какого-либо события в приложении вызывая тем самым соответствующий обработчик события [6].

Ниже представлены созданные формы с их элементами для реализуемого программного обеспечения (рисунок 1.5 – 1.8).

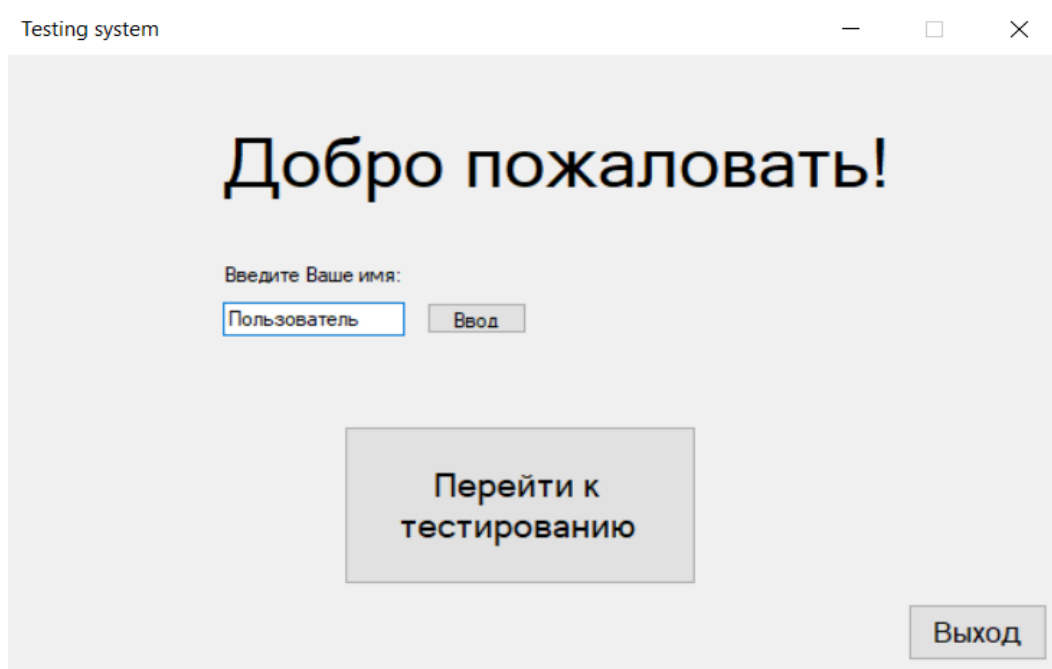


Рисунок 1.5 – Главное окно приложения

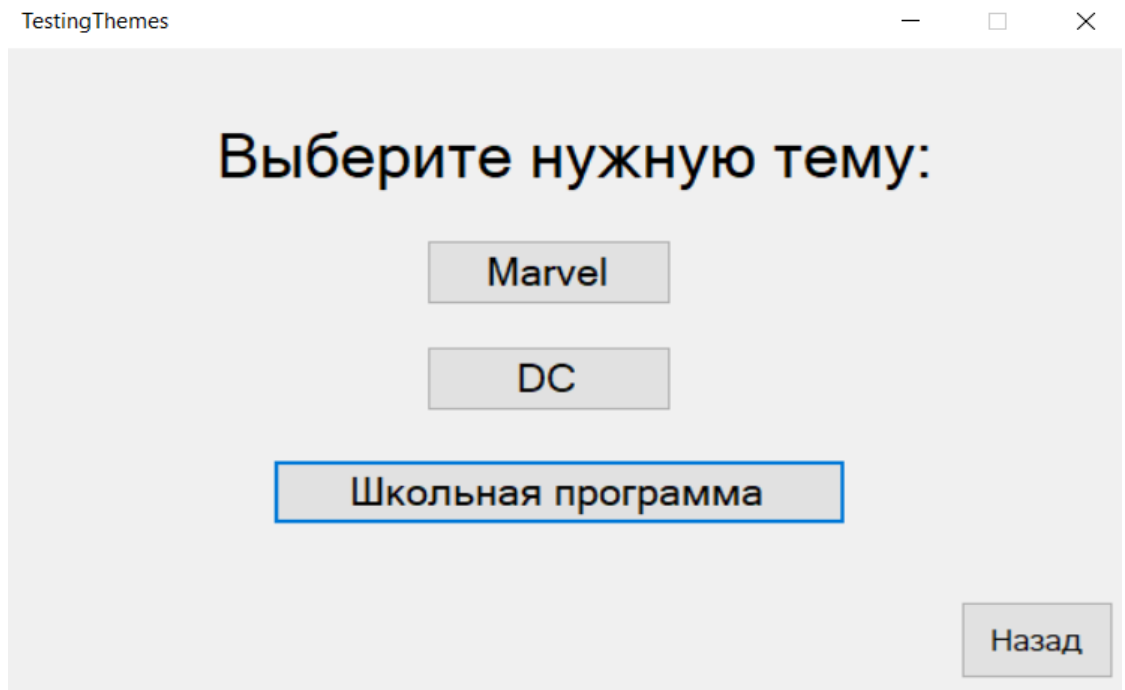


Рисунок 1.6 – Окно выбора темы тестирования

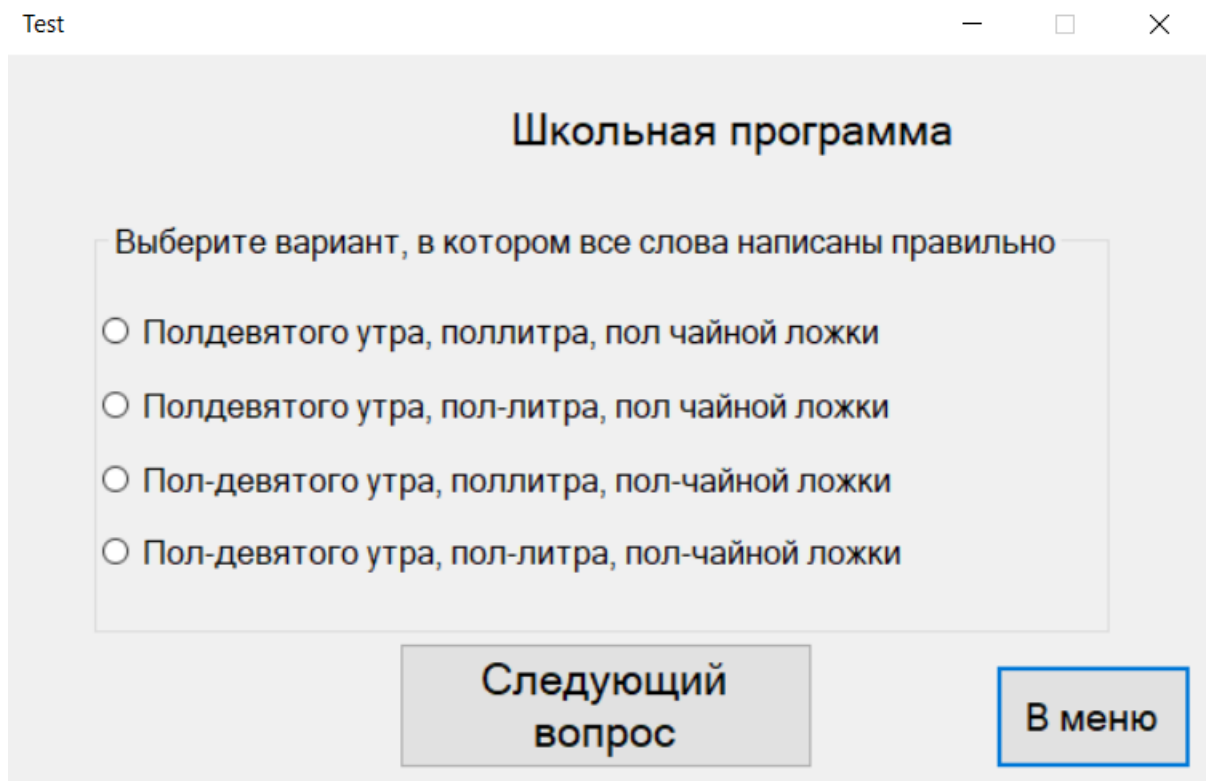


Рисунок 1.7 – Окно теста (на примере теста «Школьная программа»)

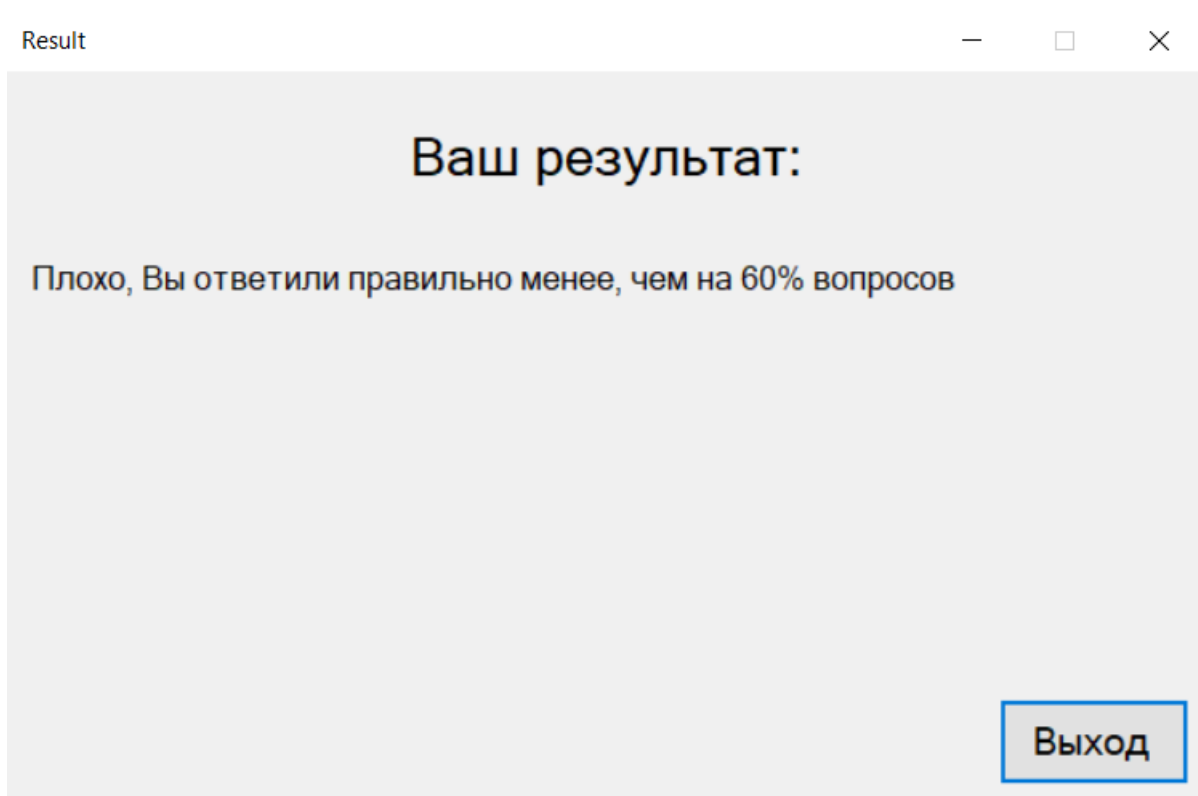


Рисунок 1.8 – Окно вывода результата (на примере получения результата «Плохо»)

Основным элементом каждого окна является кнопка. При нажатии пользователем на эту кнопку генерируется событие, которое обрабатывается по своему собственному сценарию в модуле логики программы в зависимости от того, что должно происходить (закрытие окна, запоминание имени и т.д.).

На рисунке 1.9 представлена иерархия вызова окон в процессе тестирования. Просмотрев результаты тестирования в соответствующем окне «Result», пользователь нажимает на кнопку «Выход», которая закрывает текущее окно «Result» и предыдущее окно «Test», соответствующее тесту. Пользователь попадает обратно в окно выбора темы (рисунок 1.6) «TestingThemes», откуда он опять может выбрать тему для тестирования и последовательность действий будет та же.

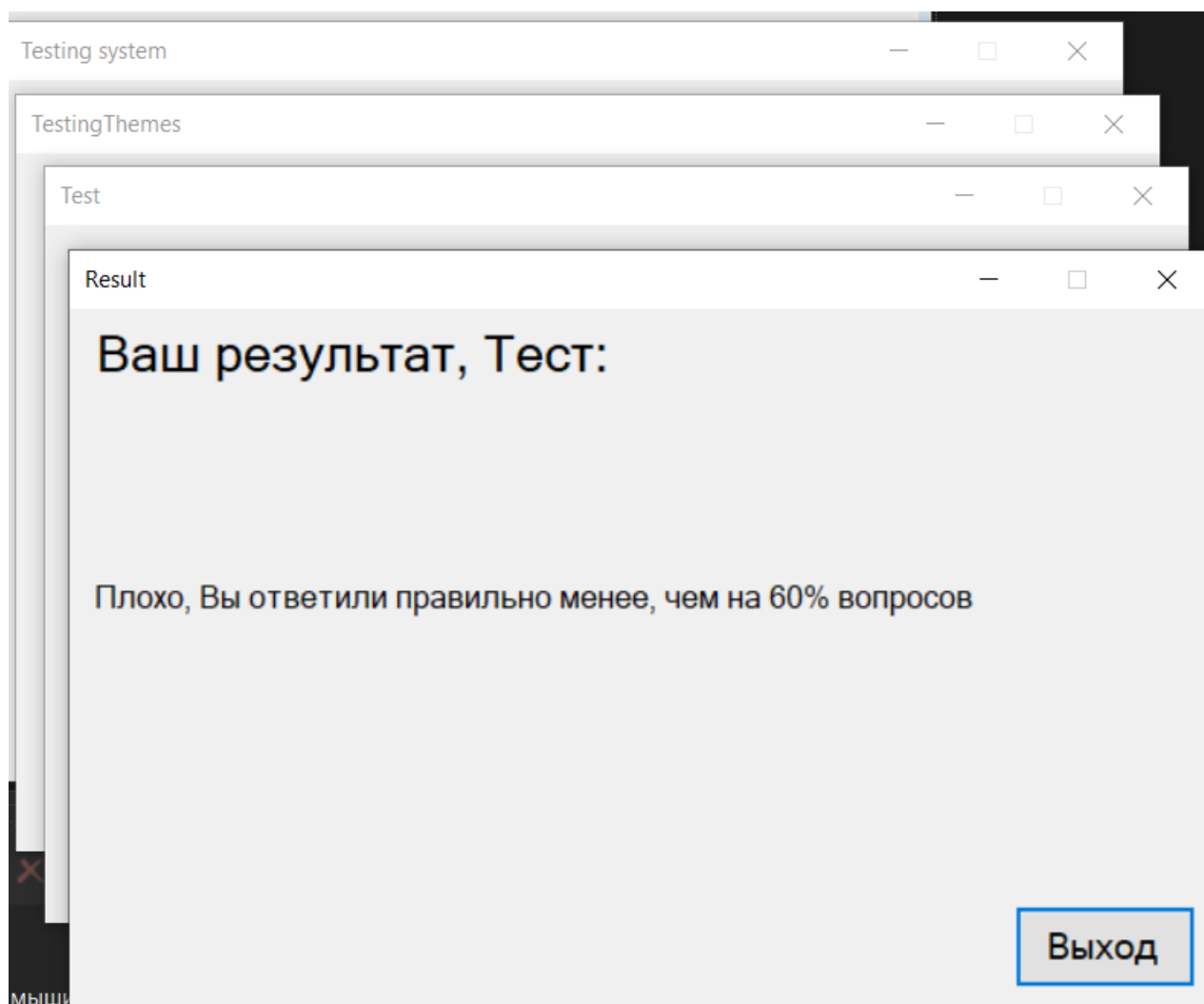


Рисунок 1.9 - Иерархия вызова окон в процессе тестирования

### 1.3.2 Реализация модуля данных

Модуль данных представляет собой файл «Model.cs» со следующим содержимым:

- класс «QuestionClass», соответствующий одному вопросу из теста. Он хранит в себе сам вопрос, все варианты ответа на данный вопрос и номер правильного варианта ответа. Также, хранит в себе необходимые в ходе работы приложения методы (например, «PrintQuestion», служащий для вывода всей информации объекта класса);
- поле для хранения никнейма пользователя «userName»;
- поле для хранения номера выбранного теста «testNumber» по которому пользователь желает пройти тестирование;
- поле с количеством правильных ответов текущего теста rightAnswersCount;



- массивы, хранящие тесты по темам: «questionsMarvel», «questionsDC» и «questionsSchool»;

Исходный текст данного модуля представлен в листинге кода в Приложении 1 (Файл «Model.cs»).

### 1.3.3 Реализация модуля логики программы

Модуль логики программы представляет собой совокупность обработчиков событий действий пользователя в модуль пользовательского интерфейса. Также, в его состав входят функции загрузки тестов из соответствующих файлов.

Большая часть обработчиков событий состоит из вызова другого окна, закрытия текущего окна, запоминания введенного значения и т.д. Поэтому, наибольший интерес может представлять обработчик нажатия на кнопку «Следующий вопрос» из окна теста.

В процессе прохождения пользователем теста, когда он нажимает на кнопку «Следующий вопрос» данная кнопка выполняет проверку выбранного пользователем варианта ответа с правильным. Если пользователь не выбрал никакого варианта ответа, то это автоматически засчитывается, как неправильный вариант.

```
foreach (Control control in groupBoxQuestion.Controls)
{
    if (control is RadioButton)
    {
        RadioButton radioButton = control as RadioButton;
        if (radioButton.Checked)
        {
            int trueAnswerNumber = testQuestions[0].TrueAnswer;
            if (radioButton.Text == testQuestions[0].Answers[trueAnswerNumber - 1])
            {
                rightAnswersCount++;
            }
        }
    }
}
```

Затем происходит удаление текущего вопроса из массива вопросов (поскольку, по условию задачи, вопросы не должны повторяться) и происходит

загрузка следующего вопроса. Если массив содержит одну запись, то название кнопки меняется на «Результат», что свидетельствует о достижении пользователем последнего вопроса [1].

```
if (testQuestions.Count == 1)
{
    LoadQuestion(0);
    buttonNextQuestion.Text = "Результат";
}
else if (testQuestions.Count == 0)
{
    Result result = new Result();
    result.ShowDialog();
    Close();
}
```

Полный код данного модуля можно просмотреть в Приложении 1.

## 2. ТЕСТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

### 2.1 Организация процесса тестирования программного обеспечения

Для успешного построения программной системы необходимо в определенную последовательность шагов объединить различные способы тестирования организовав этим процесс тестирования ПО. Любой качественно организованный процесс тестирования должен обеспечить максимальную проверку результатов, полученных как итог каждого этапа разработки. Основными этапами разработки ПО являются: кодирование, проектирование, системный анализ и анализ требований. На рисунке 2.1 изображена спиральная модель тестирования программного обеспечения, которая может служить основой представления методики тестирования программного обеспечения и схожа с известной спиральной моделью жизненного цикла программного обеспечения.

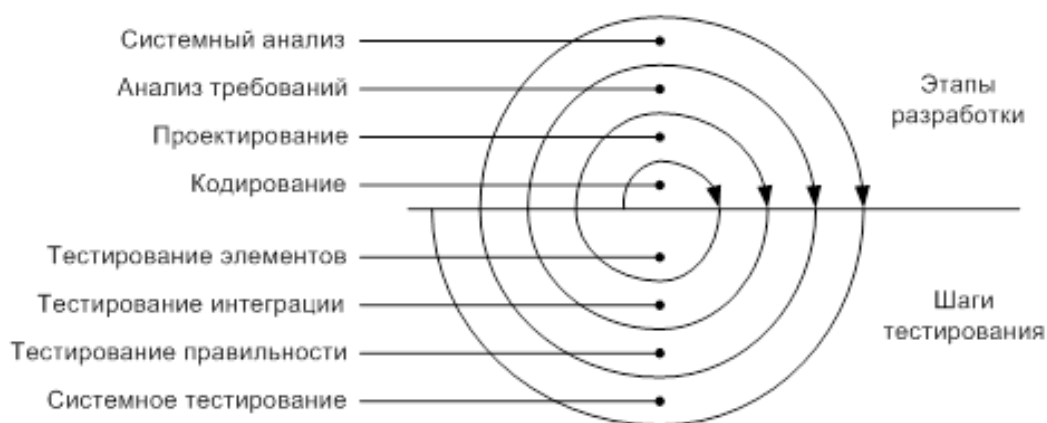


Рисунок 2.1 - Спиральная модель процесса тестирования ПО

Первым шагом является тестирование элементов (программных модулей), которое проверяет результаты этапа кодирования ПО. Целью данного шага является индивидуальная проверка каждого модуля. Одним из способов тестирования на данном шаге является способ «белого ящика».

На втором шаге в целях выявления ошибок соответствующего этапа проектирования программной системы осуществляется тестирование интеграции. Цель данного шага – тестирование сборки модулей в систему. В основном применяют способы тестирования «черного ящика». Используются методы нисходящего и

восходящего тестирования интеграций. Тесты проводятся для обнаружения ошибок интерфейса, таких, например, как:

- потеря данных при прохождении через интерфейс;
- отсутствие в модуле необходимой ссылки;
- неблагоприятное влияние одного модуля на другой;
- проблемы при работе с глобальными структурами данных и др.

На третьем шаге проверяется корректность требований к программной системе, т.е. проводится тестирование правильности. Целью данного шага является подтверждение того факта, что функции, описанные в спецификации ПО, отвечают требованиям заказчика. Здесь применяются методы «черного ящика». Важным элементом подтверждения правильности является проверка конфигурации программной системы - совокупности всех элементов информации, вырабатываемых в процессе конструирования ПО.

Выявляющее дефекты этапа системного анализа ПО, системное тестирование проводится на крайнем витке спирали. Целью данного шага является проверка правильности реализации всех системных функций, а также взаимодействия всех элементов программной системы. Могут использоваться следующие тесты:

- тестирование восстановления (проверяется время и полнота восстановления);
- тестирование безопасности (проверяется реакция всех защитных механизмов, встроенных в систему, на проникновение);
- стрессовое тестирование (проверяется работа системы при аномальных ситуациях, например, очень большом количестве запросов на ресурсы системы);
- тестирование производительности в системах реального времени.

Организация процесса тестирования в виде эволюционной разворачивающейся спирали обеспечивает максимальную эффективность поиска ошибок.

Таким образом, процесс тестирования ПО представляет собой столь же неотъемлемую часть процесса разработки, как и проектирование. Также, тестирование позволяет оценить качество разрабатываемого продукта [3].

## 2.2 Тестирование элементов разработанного программного обеспечения

Объектом тестирования элементов является наименьшая единица проектирования ПС — модуль. Для обнаружения ошибок в рамках модуля тестируются его важнейшие управляющие пути. Способом тестирования на данном этапе является метод «белого ящика».

Тестирование «белого ящика», также тестирование стеклянного ящика, структурное тестирование — тестирование, которое учитывает внутренние механизмы системы или компонента.

Обычно включает тестирование ветвей, маршрутов, операторов. При тестировании выбирают входы для выполнения разных частей кода и определяют ожидаемые результаты.

Критерии покрытия кода:

- покрытие операторов — каждая ли строка исходного кода была выполнена и протестирована;
- покрытие условий — каждая ли точка решения (вычисления истинно ли или ложно выражение) была выполнена и протестирована;
- покрытие путей — все ли возможные пути через заданную часть кода были выполнены и протестированы;
- покрытие функций — каждая ли функция программы была выполнена;
- покрытие вход/выход — все ли вызовы функций и возвраты из них были выполнены;
- покрытие значений параметров — все ли типовые и граничные значения параметров были проверены.

Входные значения выбираются, основываясь на знании кода, который будет их обрабатывать. Также, имеется и информация о том, каким должен быть результат этой обработки. Знание всех особенностей тестируемой программы и ее реализации — обязательны для этой техники. Тестирование «белого ящика» — углубление во внутренне устройство системы, за пределы ее внешних интерфейсов. На рисунке 2.2 представлена схема тестирования «белого ящика»

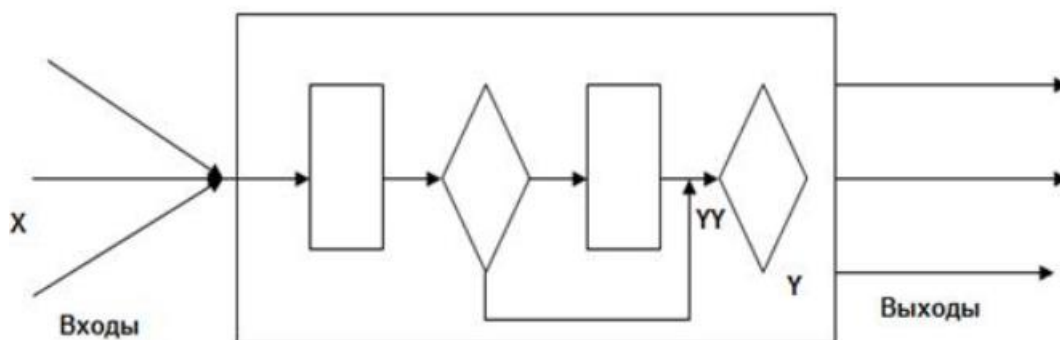


Рисунок 2.2 - Схема теста «белый ящик»

В ходе выполнения курсового проекта тестирование элементов разработанного ПО будет проводиться методом тестирования базового пути. Способ тестирования базового пути дает возможность получить оценку комплексной сложности программы и использовать эту оценку для определения необходимого количества тестовых вариантов.

Тестовые варианты разрабатываются для проверки базового множества путей (маршрутов) в программе. Они гарантируют однократное выполнение каждого оператора программы при тестировании. Метод заключается в следующей последовательности шагов:

1. На основе текста программы формируется потоковый граф.
2. Определяется цикломатическая сложность потокового графа.
3. Определяется базовое множество независимых линейных путей.
4. Подготавливаются тестовые варианты, инициирующие выполнение каждого пути.

Проведем тестирование модулей разработанного ПО.

Потоковый граф для функции загрузки теста из файла модуля логики программы представлен на рисунке 2.3

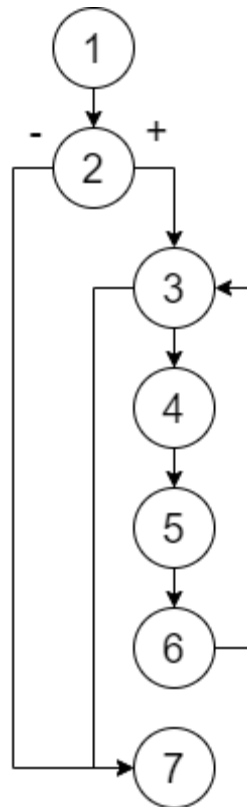


Рисунок 2.3 - Поточковый граф для функции загрузки теста из файла

На основе построенного потокового графа и с учетом особенностей программной реализации сформировано следующее базовое множество для определения тестовых вариантов:

1. 1 – 2 – 7.
2. 1 – 2 – 3 – 7.
3. 1 – 2 – 3 – 4 – 5 – 6 – 3 – 7.

Тестовые варианты для сформированного базового множества, следующие:

1. И.Д.: передан некорректный путь к файлу.

О.Р.: сообщение об ошибке, возврат пустого массива из функции.

2. И.Д.: передан корректный путь к файлу, но при его открытии он оказывается пуст.

О.Р.: возврат пустого массива из функции.

3. И.Д.: корректный путь к файлу, файл содержит информацию о тесте.

О.Р.: возврат массива вопросов.

Потоковый граф функции обработки нажатия кнопки «Следующий вопрос» представлен на рисунке 2.4.

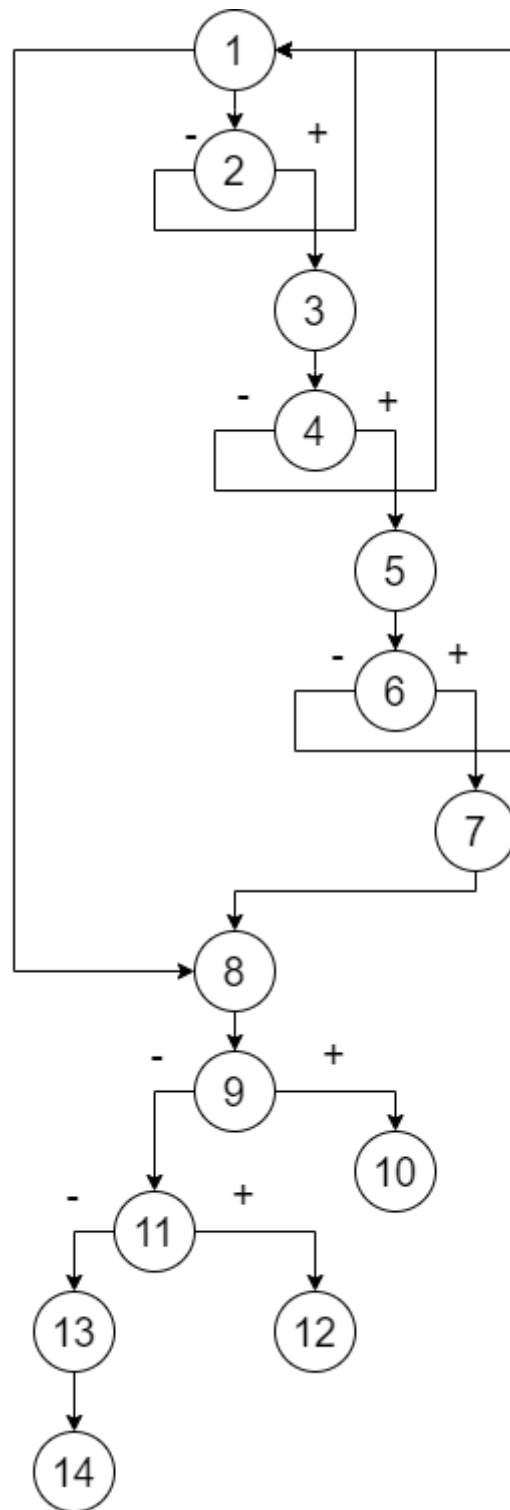


Рисунок 2.4 – Поточковый граф функции обработки нажатия кнопки  
«Следующий вопрос»

На основе построенного потокового графа и с учетом особенностей программной реализации сформировано следующее базовое множество для определения тестовых вариантов:

1. 1 – 2 – 3 – 4 – 1 – 8 – 9 – 10.



2. 1 – 2 – 3 – 4 – 1 – 8 – 9 – 11 – 12.
3. 1 – 2 – 3 – 4 – 1 – 8 – 9 – 11 – 13 – 14.
4. 1 – 2 – 3 – 4 – 5 – 6 – 1 – 8 – 9 – 10
5. 1 – 2 – 3 – 4 – 5 – 6 – 1 – 8 – 9 – 11 – 12.
6. 1 – 2 – 3 – 4 – 5 – 6 – 1 – 8 – 9 – 11 – 13 – 14.
7. 1 – 2 – 3 – 4 – 5 – 6 – 7 – 1 – 8 – 9 – 10
8. 1 – 2 – 3 – 4 – 5 – 6 – 7 – 1 – 8 – 9 – 11 – 12.
9. 1 – 2 – 3 – 4 – 5 – 6 – 7 – 1 – 8 – 9 – 11 – 13 – 14.

Тестовые варианты для сформированного базового множества, следующие:

1. И.Д.: не выбран ни один из вариантов ответа (`radioButton.Checked == false`), количество оставшихся вопросов в массиве равно единице (`testQuestions.Count == 1`).

О.Р.: загрузка последнего вопроса в форму, замена текста в кнопке «Следующий вопрос» на «Результат».

2. И.Д.: не выбран ни один из вариантов ответа (`radioButton.Checked == false`), количество оставшихся вопросов в массиве равно нулю (`testQuestions.Count == 0`).

О.Р.: открытие формы с результатом тестирования, последующее закрытие текущей формы.

3. И.Д.: не выбран ни один из вариантов ответа (`radioButton.Checked == false`), количество оставшихся вопросов в массиве больше единицы (`testQuestions.Count > 1`).

О.Р.: загрузка следующего вопроса в форму.

4. И.Д.: выбран какой-то вариант ответа (`radioButton.Checked == true`), но он не является верным (`radioButton.Text != testQuestions[0].Answers[trueAnswerNumber - 1]`), количество оставшихся вопросов в массиве равно единице (`testQuestions.Count == 1`).

О.Р.: загрузка последнего вопроса в форму, замена текста в кнопке «Следующий вопрос» на «Результат».

5. И.Д.: выбран какой-то вариант ответа (`radioButton.Checked == true`), но он не является верным (`radioButton.Text != testQuestions[0].Answers`

[trueAnswerNumber - 1]), количество оставшихся вопросов в массиве равно нулю (testQuestions.Count == 0).

О.Р.: открытие формы с результатом тестирования, последующее закрытие текущей формы.

6. И.Д.: выбран какой-то вариант ответа (radioButton.Checked == true), но он не является верным (radioButton.Text != testQuestions[0].Answers [trueAnswerNumber - 1]), количество оставшихся вопросов в массиве больше единицы (testQuestions.Count > 1).

О.Р.: загрузка следующего вопроса в форму.

7. И.Д.: выбран какой-то вариант ответа (radioButton.Checked == true) и он является верным (radioButton.Text == testQuestions[0].Answers [trueAnswerNumber - 1]), количество оставшихся вопросов в массиве равно единице (testQuestions.Count == 1).

О.Р.: увеличение значения переменной, ответственной за количество правильных ответов (rightAnswersCount++), загрузка последнего вопроса в форму, замена текста в кнопке «Следующий вопрос» на «Результат».

8. И.Д.: выбран какой-то вариант ответа (radioButton.Checked == true) и он является верным (radioButton.Text == testQuestions[0].Answers [trueAnswerNumber - 1]), количество оставшихся вопросов в массиве равно нулю (testQuestions.Count == 0).

О.Р.: увеличение значения переменной, ответственной за количество правильных ответов (rightAnswersCount++), открытие формы с результатом тестирования, последующее закрытие текущей формы.

9. И.Д.: выбран какой-то вариант ответа (radioButton.Checked == true) и он является верным (radioButton.Text == testQuestions[0].Answers [trueAnswerNumber - 1]), количество оставшихся вопросов в массиве больше единицы (testQuestions.Count > 1).

О.Р.: увеличение значения переменной, ответственной за количество правильных ответов (rightAnswersCount++), загрузка следующего вопроса в форму.

## **2.3 Тестирование интеграции модулей разработанного программного обеспечения**

**Интеграционное тестирование** – вид тестирования, при котором на соответствие требований проверяется интеграция модулей, их взаимодействие между собой, а также интеграция подсистем в одну общую систему. Для интеграционного тестирования используются компоненты, уже проверенные с помощью модульного тестирования, которые группируются в множества. Данные множества проверяются в соответствии с планом тестирования, составленным для них, а объединяются они через свои интерфейсы.

Существует несколько подходов к интеграционному тестированию:

- Снизу вверх (восходящее). Сначала собираются и тестируются модули самих нижних уровней, а затем по возрастанию к вершине иерархии. Данный подход требует готовности всех собираемых модулей на всех уровнях системы.
- Сверху вниз (нисходящее). Данный подход предусматривает движение с высокоуровневых модулей, а затем направляется вниз. При этом используются заглушки для тех модулей, которые находятся ниже по уровню, но включение которых в тест еще не произошло.
- Большой взрыв (комбинированное). Все модули всех уровней собираются воедино, а затем тестируется. Данный метод экономит время, но требует тщательной проработки тест кейсов.

При автоматизации тестирования используется Система непрерывной интеграции. Принцип ее действия заключается в следующем:

1. Система непрерывной интеграции производит мониторинг системы контроля версий.
2. При изменении исходных кодов в репозитории производится обновление локального хранилища.
3. Выполняются необходимые проверки и модульные тесты.
4. Исходные коды компилируются в готовые выполняемые модули.
5. Выполняются тесты интеграционного уровня.
6. Генерируется отчет о тестировании.

Это позволяет тестировать систему сразу после внесения изменений, что существенно сокращает время обнаружения и исправления ошибок [4].

Для тестирования интеграции разработанного программного обеспечения будет использоваться метод «сверху вниз».

Для проведения тестирования определим порядок интеграции модулей. Поскольку данное программное обеспечение использует графический пользовательский интерфейс, то модуль логики программы и пользовательского интерфейса взаимосвязаны по умолчанию. Поэтому, проведем тестирование пользовательского интерфейса и связанного с ним модуля логики программы. Вместо модуля данных будем использовать заглушку, в которой заранее будет определен список вопросов для теста и другие необходимые данные. Тестовым вариантом будет являться следующий вариант использования приложения:

- 1) запуск приложения и ввод никнейма (рисунок 2.5);

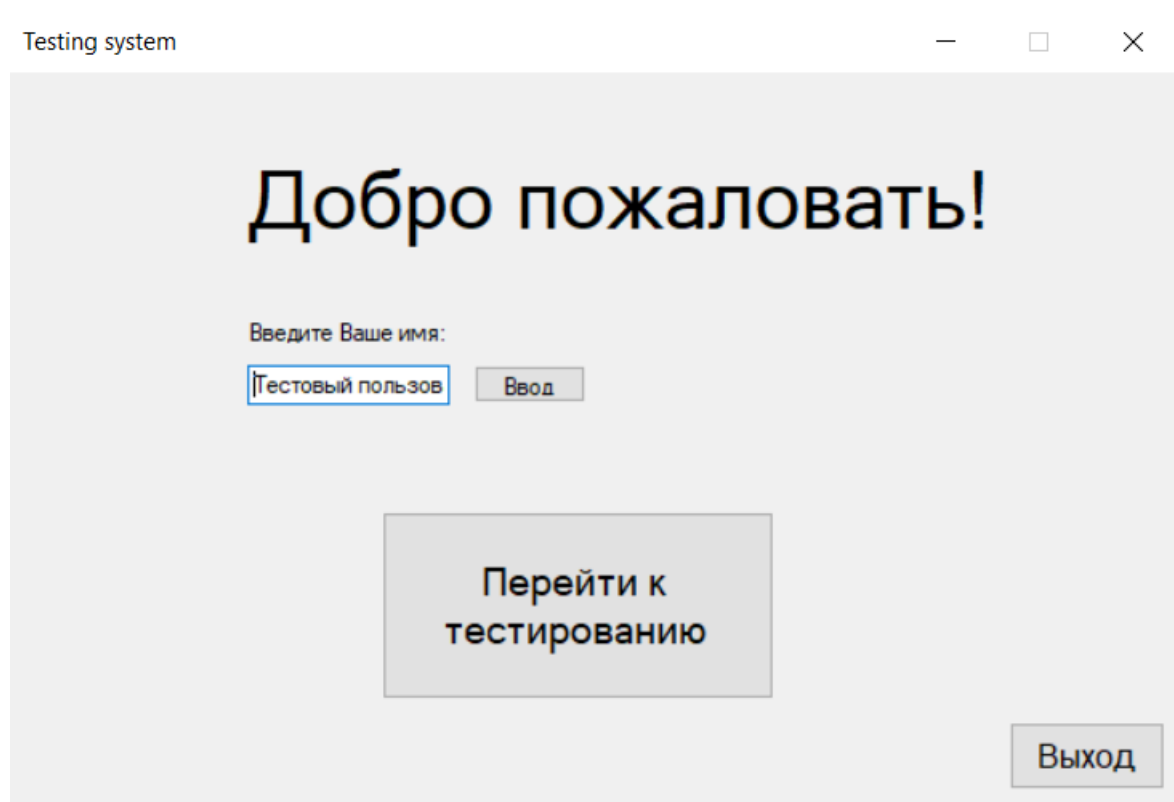


Рисунок 2.5 – Запуск приложения и ввод никнейма

- 2) выбор темы тестирования (рисунок 2.6);
- 3) прохождение теста (рисунок 2.7);
- 4) получение результата тестирования (рисунок 2.8).

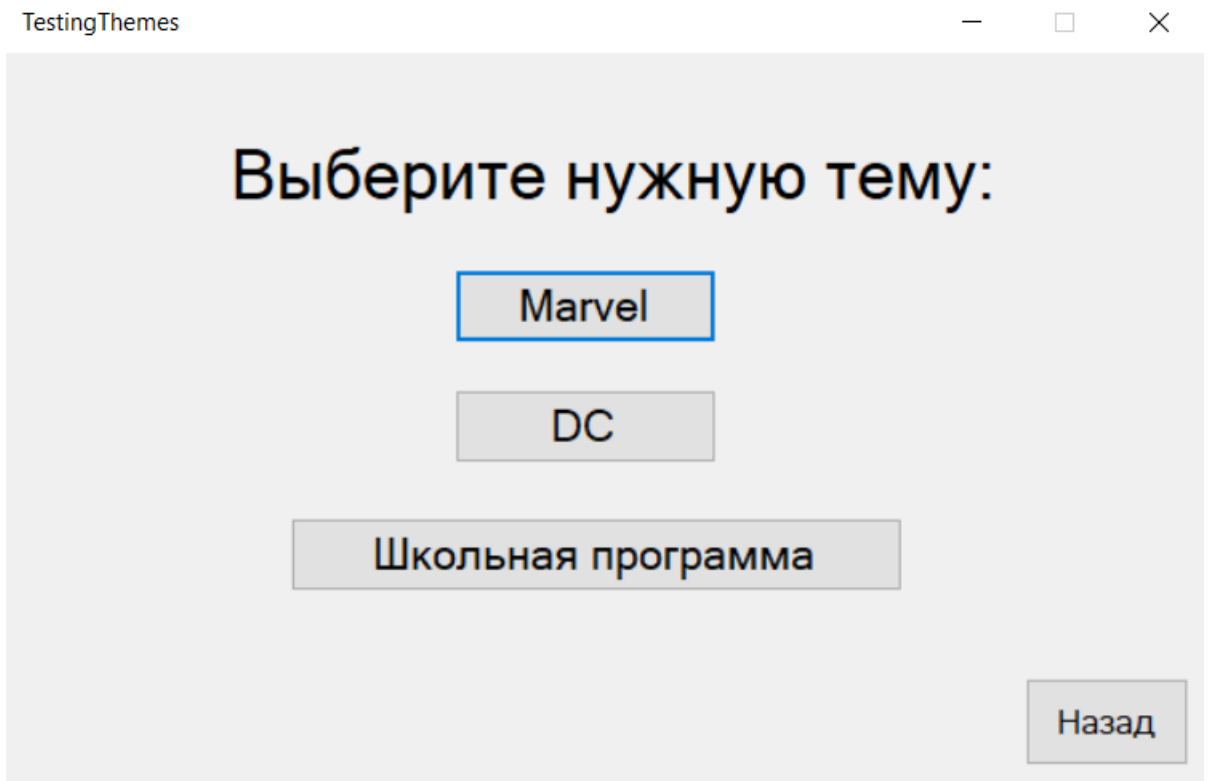


Рисунок 2.6 – Выбор темы тестирования

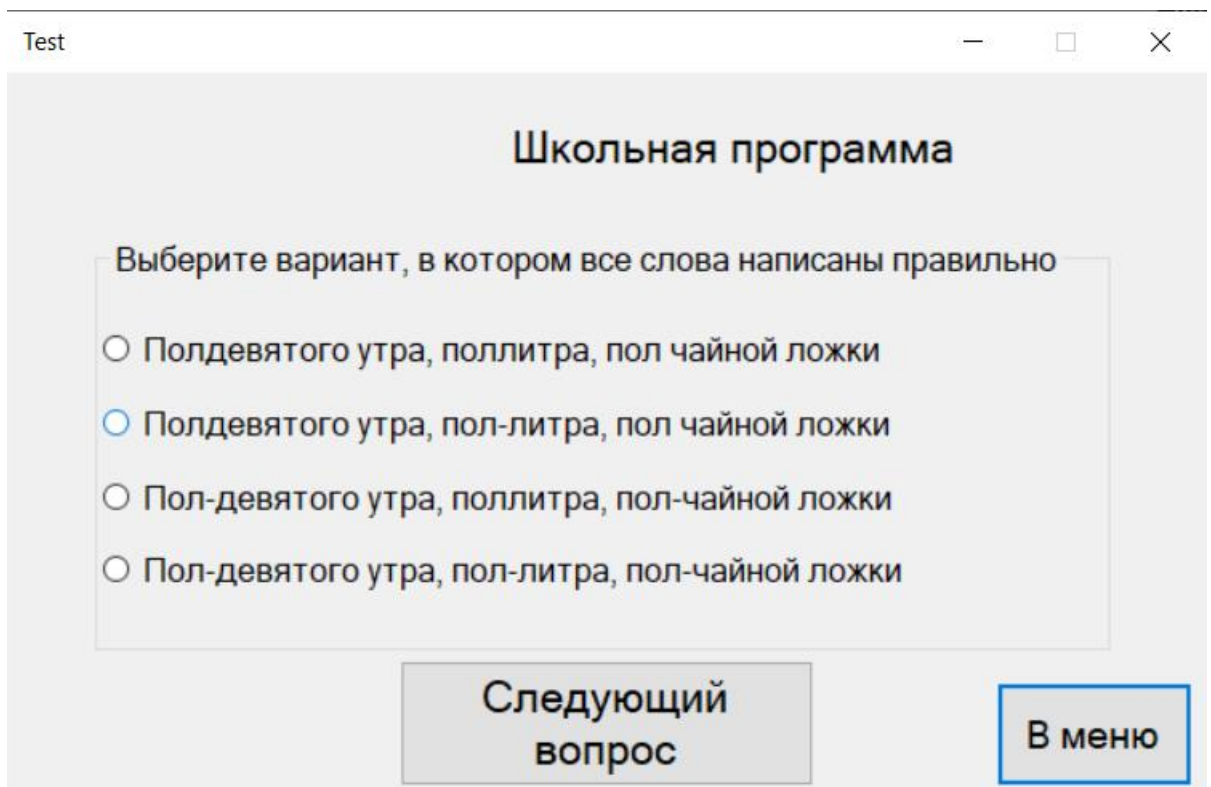


Рисунок 2.7 – Прохождение тестирования (на примере темы «Школьная программа»)

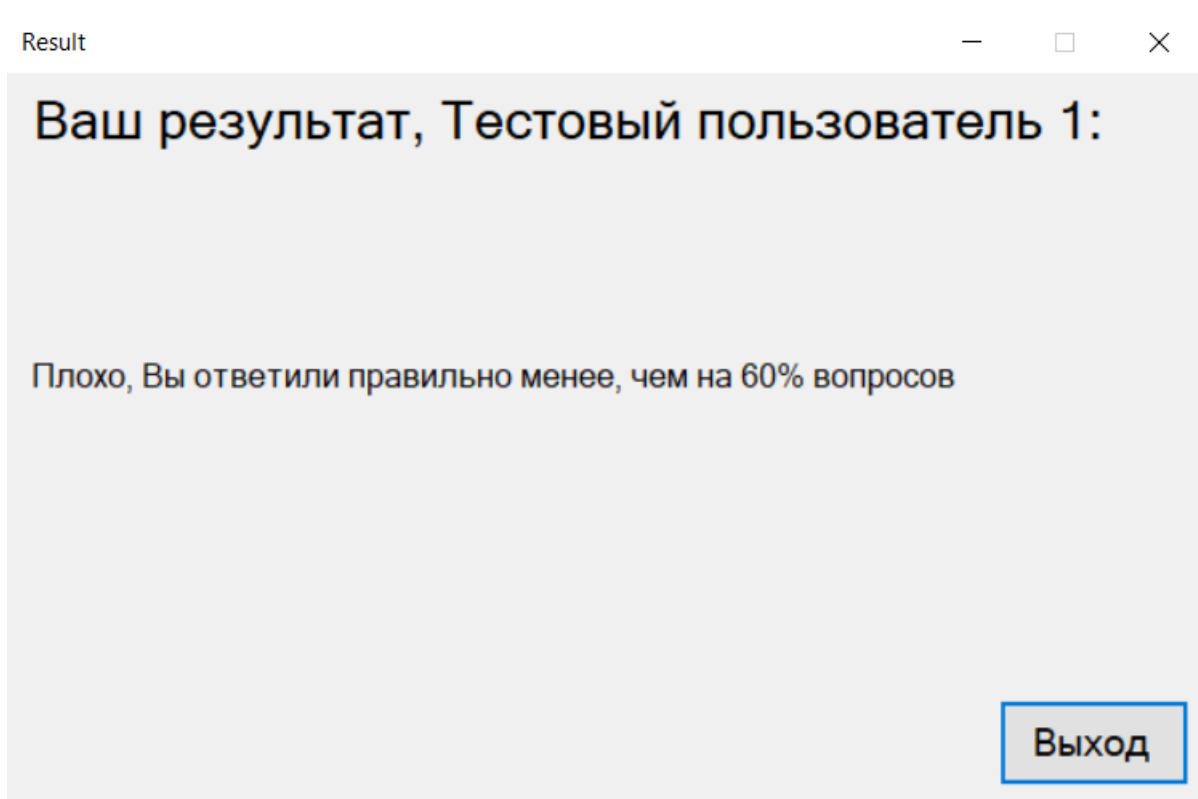


Рисунок 2.8 – Получение результата тестирования

Процесс тестирования пользовательского интерфейса в совокупности с модулем логики программы прошел успешно, получены ожидаемые от системы результаты. Поэтому можно приступить к последнему этапу тестирования интеграции – присоединению последнего модуля – модуля данных. Модуль данных заменяет заглушку, использованную на предыдущем этапе тестирования. Проходим по тому же сценарию тестирования, описанному выше, и получаем те же результаты, т.е. присоединение модуля данных прошло успешно.

Таким образом, проведя тестирование интеграции всех модулей разрабатываемого программного обеспечения в единую систему, ошибок выявлено не было, что свидетельствует об успешном этапе тестирования.

## **2.4 Тестирование правильности разработанного программного обеспечения**

После окончания тестирования интеграции программная система собрана в единый блок. Интерфейсные ошибки обнаружены и откорректированы. Цель тестирования правильности – подтвердить, что функции, описываемые в

спецификации требований в программной системе, соответствуют ожиданиям заказчика. Подтверждение правильности работы программной системы выполняется с помощью тестов «черного ящика» (рисунок 2.9), демонстрирующих соответствие требованиям [2].

На этапе тестирования правильности объектом тестирования является единое программное обеспечение.

При обнаружении отклонений от спецификации требований создается список недостатков. Как правило, отклонения и ошибки, выявленные при тестировании правильности, требуют изменения сроков разработки продукта.



Рисунок 2.9 – Схема «черного ящика»

В ходе выполнения курсового проекта тестирование правильности разработанного ПО будет проводится методом диаграмм причин-следствий.

Способ тестирования с использование диаграмм причин-следствий обеспечивает формальное выведение высокорезультативных тестовых вариантов, основанное на анализе причинно-следственных связей. Причина – это отдельное входное условие или класс эквивалентности. Следствие – выходное условие или действие системы.

Метод заключается в следующей последовательности шагов:

- 1) выделение групп причин и следствий;

- 2) присвоение идентификаторов каждой причине и каждому следствию;
- 3) построение графа причинно-следственных связей;
- 4) преобразование графа в таблицу решений;
- 5) преобразование столбцов решений в тестовые варианты.

Проведем тестирование правильности разработанного ПО.

Выделим следующие причины и присвоим им идентификаторы:

- 1 – отсутствие файла
- 2 – файл с тестом пустой
- 3 – количество правильных ответов меньше 60%
- 4 - количество правильных ответов 60 - 70%
- 5 - количество правильных ответов 71 - 85%
- 6 - количество правильных ответов 86 - 100%
- 7 – выход из программы

Аналогичным образом выделим и присвоим идентификаторы следствиям:

- 101 – корректный выход из программы
- 102 – предупреждение об ошибке в работе программы
- 103 – результат тестирования «Плохо»
- 104 – результат тестирования «Удовлетворительно»
- 105 – результат тестирования «Хорошо»
- 106 – результат тестирования «Отлично»

На рисунке 2.10 изображен граф причинно-следственных связей разработанного программного обеспечения.

Преобразуем граф в таблицу решений и приведем ее в таблице 1.

На основе таблицы решений сформированы следующие тестовые варианты:

- 1) И.Д.: отсутствует файл с темой тестирования.  
О.Р.: предупреждение об ошибке в работе программы.
- 2) И.Д.: файл с выбранным тестом пустой.  
О.Р.: предупреждение об ошибке в работе программы.
- 3) И.Д.: количество правильных ответов меньше 60%.  
О.Р.: результат тестирования «Плохо».



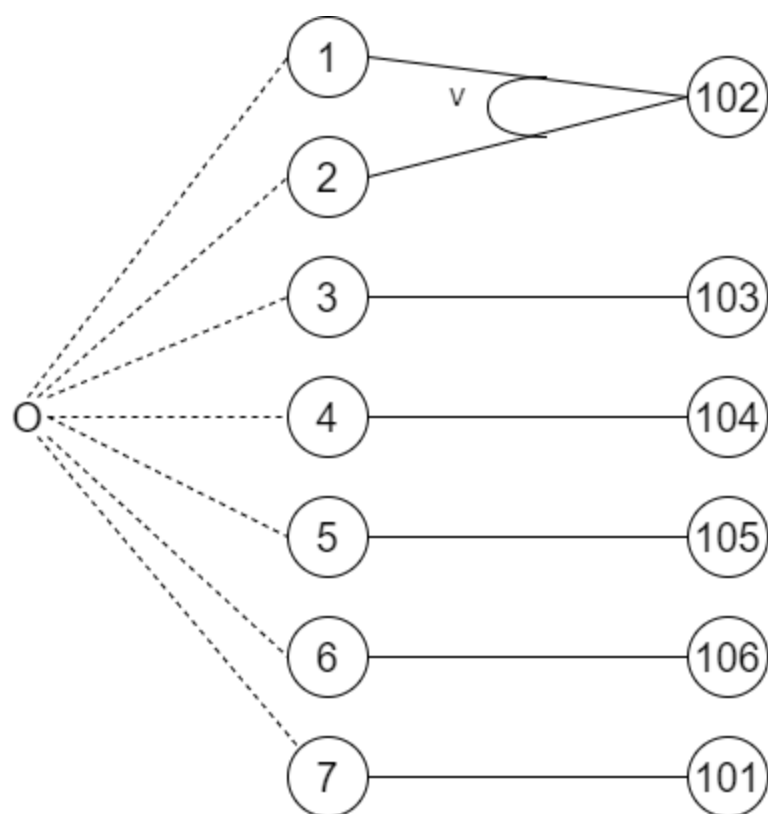


Рисунок 2.10 - Граф причинно-следственных связей

Таблица 1 – Таблица решений

Номер столбца		1	2	3	4	5	6	7
Причины	1	1	0	0	0	0	0	0
	2	0	1	0	0	0	0	0
	3	0	0	1	0	0	0	0
	4	0	0	0	1	0	0	0
	5	0	0	0	0	1	0	0
	6	0	0	0	0	0	1	0
	7	0	0	0	0	0	0	1
Следствия	101	0	0	0	0	0	0	1
	102	1	1	0	0	0	0	0
	103	0	0	1	0	0	0	0
	104	0	0	0	1	0	0	0
	105	0	0	0	0	1	0	0
	106	0	0	0	0	0	1	0

4) И.Д.: количество правильных ответов 60 - 70%.

О.Р.: результат тестирования «Удовлетворительно».

5) И.Д.: количество правильных ответов 71 - 85%.

О.Р.: результат тестирования «Хорошо»

6) И.Д.: количество правильных ответов 86 - 100%

О.Р.: результат тестирования «Отлично»

7) И.Д.: пользователем выбран выход из программы

О.Р.: корректный выход из программы

Таким образом, можно сделать вывод о том, что в процессе проведения тестирования правильности методом причинно-следственных связей программа ведет себя корректно, правильно и соответствует результатам, которые от нее ожидаются.

## **2.5 Системное тестирование разработанного программного обеспечения**

Последним этапом тестирования является системное тестирование. Объектом данного тестирования является система в целом, цель – проверка правильности объединения и взаимодействия всех элементов компьютерной системы, реализации всех системных функций, а способ тестирования – функциональное тестирование.

Системное тестирование подразумевает выход за рамки области действия программного проекта и проводится не только программным разработчиком. Классическая проблема системного тестирования — указание причины. Она возникает, когда разработчик одного системного элемента обвиняет разработчика другого элемента в причине возникновения дефекта. Для защиты от подобного обвинения разработчик программного элемента должен:

1) предусмотреть средства обработки ошибки, которые тестируют все входы информации от других элементов системы;

2) провести тесты, моделирующие неудачные данные или другие потенциальные ошибки интерфейса ПС;

3) записать результаты тестов, чтобы использовать их как доказательство невиновности в случае «указания причины»;

4) принять участие в планировании и проектировании системных тестов, чтобы гарантировать адекватное тестирование ПС.

В конечном счете системные тесты должны проверять, что все системные элементы правильно объединены и выполняют назначенные функции.

Системное тестирование включает в себя:

1. Тестирование восстановления (время восстановления, правильность повторной инициализации, восстановление данных и др.).

2. Тестирование безопасности (проверка функционирования защитных механизмов системы в случае проникновения).

3. Стрессовое тестирование (тестовые варианты направлены на оценку стабильности работы ПС при «ненормальных» нагрузках: по частоте запросу, объему данных). Частный случай – тестирование чувствительности.

4. Тестирование производительности (проверяется скорость работы ПО в компьютерной системе) [5].

Исходя из специфики разработанной программы будет проведен один из видов системного тестирования, а именно тестирование восстановления

Тестирование восстановления будет проводиться с использованием диспетчера задач. После запуска приложения вводим никнейм, выбираем первую тему для тестирования и отвечаем на несколько вопросов. Не доходя до последнего вопроса, вызываем диспетчер задач и закрываем приложение. Затем еще раз запускаем приложение, вводим тот же никнейм и выбираем ту же тему для тестирования. Доходим до конца теста, получаем ожидаемый результат (в зависимости от количества верных ответов) и убеждаемся, что программа отработала верно даже после аварийного выключения. Те же самые действия проводим и с другими темами для тестирования, а также на главном окне. Каждый раз убеждаемся в корректности работы приложения после аварийного завершения и последующего запуска.

Таким образом, проведя системное тестирование восстановления можно убедиться в том, что после каких-либо сбоев в работе программы и аварийного ее завершения, если запустить приложение повторно, то оно будет корректно работать по соответствующему сценарию.

### 3. ОЦЕНКА КАЧЕСТВА РАЗРАБОТАННОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

**Качество программного обеспечения** — способность программного продукта при заданных условиях удовлетворять установленным или предполагаемым потребностям.

Выделяют следующие характеристики качества ПО:

1. **Функциональность** — определяется способностью ПО решать задачи, которые соответствуют зафиксированным и предполагаемым потребностям пользователя, при заданных условиях использования ПО. Т.е. эта характеристика отвечает за то, что ПО работает исправно и точно, функционально совместимо, соответствует стандартам отрасли и защищено от несанкционированного доступа.

2. **Надежность** — способность ПО выполнять требуемые задачи в обозначенных условиях на протяжении заданного промежутка времени или указанное количество операций. Атрибуты данной характеристики — это завершенность и целостность всей системы, способность самостоятельно и корректно восстанавливаться после сбоев в работе, отказоустойчивость.

3. **Удобство использования** — возможность легкого понимания, изучения, использования и привлекательности ПО для пользователя.

4. **Эффективность** — способность ПО обеспечивать требуемый уровень производительности в соответствии с выделенными ресурсами, временем и другими обозначенными условиями.

5. **Удобство сопровождения** — легкость, с которой ПО может анализироваться, тестироваться, изменяться для исправления дефектов, для реализации новых требований, для облегчения дальнейшего обслуживания и адаптироваться к именуемому окружению.

6. **Портативность** — характеризует ПО с точки зрения легкости его переноса из одного окружения в другое.

На данный момент наиболее распространена и используется многоуровневая модель качества программного обеспечения. На верхнем уровне выделено шесть основных характеристик качества ПО, каждую из которых определяют набором

атрибутов, имеющих соответствующие метрики для последующей оценки (рисунок 3.1).

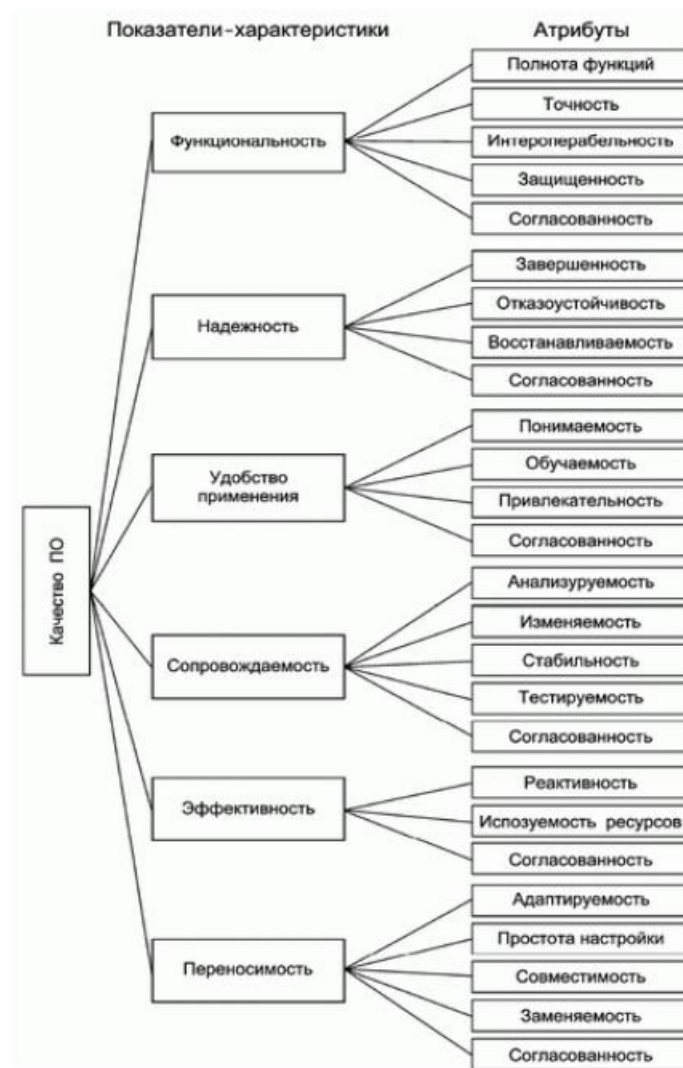


Рисунок 3.1 - Модель качества программного обеспечения

### 3.1 Статическая оценка качества разработанного программного обеспечения

Метрика программного обеспечения — мера, позволяющая получить численное значение некоторого свойства программного обеспечения или его спецификаций.

Набор используемых метрик включает:

- порядок роста (имеется в виду анализ алгоритмов в терминах асимптотического анализа и O-нотации);
- количество строк кода;

- цикломатическая сложность;
- анализ функциональных точек;
- количество ошибок на 1000 строк кода;
- степень покрытия кода тестированием;
- покрытие требований;
- количество классов и интерфейсов;
- связность.

Для каждой метрики обычно существуют ее эталонные показатели, указывающие, при каких крайних значениях стоит обратить внимание на данный участок кода. Метрики кода разделяются на категории и могут оценивать совершенно различные аспекты программной системы: сложность и структурированность программного кода, связность компонентов, относительный объем программных компонентов и др. Наиболее простая для понимания метрика – количество строк кода в программной системе, – хотя и элементарно вычисляется, но в совокупности с другими метриками может служить для получения формализованных данных для оценки кода. Например, можно построить соотношение между количеством строк кода в классе и количеством методов/свойств в классе, получив характеристику, показывающую, насколько методы данного класса являются объемными. Кроме того, такие оценки можно использовать в совокупности с метриками сложности для определения наиболее сложных участков в программном коде и принятия соответствующих мер.

В курсовом проекте для статической оценки качества разработанного программного обеспечения будут использоваться метрики Холстеда.

Основу метрики Холстеда составляют четыре измеряемых характеристики программы:

- $n1$  – число уникальных операторов программы, включая символы-разделители, имена процедур и знаки операций (словарь операторов);
- $n2$  – число уникальных операндов программы (словарь операндов);
- $N1$  – общее число операторов в программе;

- $N_2$  – общее число операндов в программе.

Опираясь на эти характеристики, получаемые непосредственно при анализе исходных текстов программ, М. Холстед вводит следующие оценки:

- Словарь программы  $n$  (формула 1);

$$n = n_1 + n_2 \quad (1)$$

- Длина программы  $N$  (формула 2);

$$N = N_1 + N_2 \quad (2)$$

- Теоретическая оценка длины программы  $N'$  (формула 3);

$$N' = n_1 * \log_2 n_1 + n_2 * \log_2 n_2 \quad (3)$$

- Объем программы  $V$  (формула 4);

$$V = N \log_2 n \quad (4)$$

- Уровень качества программирования  $L$ , основанный лишь на параметрах реальной программы без учета теоретических параметров (формула 5);

$$L = \frac{2 * n_2}{n_1 * N_2} \quad (5)$$

- Сложность понимания программы  $EC$  (формула 6);

$$EC = \frac{V}{2 * L} \quad (6)$$

- Трудоемкость кодирования программы  $D$  (формула 7);

$$D = \frac{1}{L} \quad (7)$$

- Время кодирования  $T$ , с (формула 8);

$$T = \frac{EC}{18} \quad (8)$$

- Информационное содержание  $I$  (формула 9);

$$I = \frac{V}{D} \quad (9)$$

- Уровень языка  $y'$  (формула 10);

$$y' = \frac{V}{D^2} \quad (10)$$



• Оценка необходимых интеллектуальных усилий при разработке программы Е (формула 11);

$$E = N' * \log_2\left(\frac{n}{L}\right) \quad (11)$$

На рисунке 3.2 представлены все вышеуказанные рассчитанные характеристики ПО для разработанного в ходе курсового проекта программного обеспечения.

n1	72
n2	69
N1	815
N2	212
n	141
N	1027
N'	865,72279
V	7332,3192
L	0,0090409
EC	405509,13
D	110,6087
T	22528,285
I	66,290622
y'	0,5993256
E	12058,542

Рисунок 3.2 – Результаты вычисления характеристик ПО

Качество реализованного ПО в большей степени определяется такими параметрами как информационное содержание, время кодирования и уровень языка.

На основе вычисленных параметров можно сделать вывод о том, что вычисленный в ходе выполнения статической оценки курсового проекта уровень языка соответствует среднему уровню, соответствующему языку C#. Исходя из того факта, что выбран язык среднего уровня, появляется ожидаемое существенное увеличение времени кодирования, отраженное в вычисленных метриках ( $T = 22528,285$ ). Сложность решаемой в ходе разработки ПО задачи определяется таким показателем, как информационное содержание программы ( $I$ ), значение которого приближенно равняется 66.

### 3.2 Динамическая оценка качества разработанного программного обеспечения

**Надежность программного обеспечения** - способность программного продукта безотказно выполнять определенные функции при заданных условиях в течение заданного периода времени с достаточно большой вероятностью.

Степень надежности характеризуется вероятностью работы программного продукта без отказа в течение определенного периода времени.

Существует 4 основные составляющие функциональной надежности программных систем:

- безотказность - свойство программы выполнять свои функции во время эксплуатации;
- работоспособность - свойство программы корректно (так как ожидает пользователь) работать весь заданный период эксплуатации;
- безопасность - свойство программы быть не опасной для людей и окружающих систем;
- защищенность - свойство программы противостоять случайным или умышленным вторжениям в нее.

Все модели надежности можно классифицировать по тому, какой из перечисленных процессов они поддерживают (предсказывающие, прогнозные, измеряющие и т.д.).

Некоторые модели, основанные на информации, полученной в ходе тестирования ПО, дают возможность делать прогнозы поведения ПО в процессе эксплуатации.

**Аналитические** модели дают возможность рассчитывать количественные показатели надежности, основываясь на данных о поведении программы в процессе тестирования (измеряющие и оценивающие модели).

**Эмпирические** модели базируются на анализе структурных особенностей программ. Они рассматривают зависимость показателей надёжности от числа межмодульных связей, количества циклов в модулях и т.д. Часто эмпирические модели не дают конечных результатов показателей надёжности, однако они включены в

классификационную схему, так как развитие этих моделей позволяет выявлять взаимосвязь между сложностью АСОД и его надежностью. Эти модели можно использовать на этапе проектирования ПО, когда осуществляется разбивка на модули и известна его структура.

Аналитические модели представлены двумя группами: динамические модели и статические. В динамических поведение ПС (появление отказов) рассматривается во времени. В статических моделях появление отказов не связывают со временем, а учитывают только зависимость количества ошибок от числа тестовых прогонов (по области ошибок) или зависимость количества ошибок от характеристики входных данных (по области данных). Статические модели принципиально отличаются от динамических прежде всего тем, что в них не учитывается время появления ошибок в процессе тестирования и не используется никаких предположений о поведении функции риска. Эти модели строятся на твердом статическом фундаменте.

Для использования динамических моделей необходимо иметь данные о появлении отказов во времени. Если фиксируются интервалы каждого отказа, то получается непрерывная картина появления отказов во времени (группа динамических моделей с непрерывным временем). С другой стороны, может фиксироваться только число отказов за произвольный интервал времени.

Для построения модели надежности разработанного программного обеспечения будет использоваться аналитическая статическая модель надежности Миллса.

**Модель Миллса** — способ оценки количества ошибок в программном коде, созданный в 1972 году программистом Харланом Миллсом. Использование этой модели предполагает необходимость перед началом тестирования искусственно вносить в программу некоторое количество известных ошибок. Ошибки вносятся случайным образом и фиксируются в протоколе искусственных ошибок. Специалист, проводящий тестирование, не знает ни количества ошибок, ни характера внесенных ошибок до момента оценки показателей надежности по модели Миллса. Предполагается, что все ошибки (как естественные, так и

искусственно внесенные) имеют равную вероятность быть найденными в процессе тестирования.

Предполагается, что в программном коде имеется заранее неизвестное количество ошибок (багов)  $N$ , требующее максимально точной оценки. Для получения этой величины вводятся в программный код  $M$  дополнительных ошибок, о наличии которых ничего не известно специалистам по тестированию. Далее предполагается, что после проведения тестирования было обнаружено  $n$  естественных ошибок (где  $n < N$ ) и искусственных (где  $m < M$ ). Тогда процент естественных и внесённых ошибок должен быть одинаков и выполняется соотношение, представленное в формуле 12:

$$\frac{n}{N} = \frac{m}{M} \quad (12)$$

Отсюда следует, что оценка полного количества естественных ошибок в коде равна (формула 13):

$$N = n \frac{M}{m} \quad (13)$$

Количество всё ещё не выловленных багов кода равно разности  $N - n$ .

Количественная оценка доверия модели соответствует следующему эмпирическому критерию  $C$  (формулы 14-15):

$$C = 1, \text{ если } n > N \quad (14)$$

$$C = \frac{M}{M+N+1}, \text{ если } n \leq N \quad (15)$$

Уровень значимости  $C$  оценивает вероятность, с которой модель будет правильно отклонять ложное предположение.

Рассчитаем все вышеуказанные характеристики ПО для разработанного в ходе курсового проекта программного обеспечения.

Получим следующие результаты:

- Количество внесенных искусственных ошибок  $M = 6$ ;
- Количество обнаруженных естественных ошибок  $n = 2$ ;
- Количество обнаруженных искусственных ошибок  $m = 3$ ;
- Оценка полного количества естественных ошибок  $N = 4$ ;

- Количество не обнаруженных ошибок  $N - n = 2$ ;
- Оценка доверия модели  $C = 0,55$ ;

Таким образом, исходя из результатов расчета всех характеристик модели надежности Миллса можно сделать вывод о том, что модель будет правильно отклонять ложное предположение с вероятностью в 0,55.

## **ЗАКЛЮЧЕНИЕ**

В ходе выполнения курсового проекта была изучена задача тестирования пользователей по заданным темам, спроектировано программное обеспечение для решения задачи тестирования пользователей по заданным темам, реализовано программное обеспечение для решения задачи тестирования пользователей по заданным темам, проведено комплексное тестирование разработанного программного обеспечения, а также оценено качество разработанного программного обеспечения.

Так как все задачи курсового проекта были выполнены, а требуемое программное обеспечение и требуемые процессы тестирования были реализованы, а также проведена оценка качества разработанного программного обеспечения и была построена его модель надежности курсовой проект можно считать выполненным.

## СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Албахари Дж. С# 6.0. Справочник. Полное описание языка [Текст] / Джозеф Албахари, Бен Албахари. — М.: «Вильямс», 2018. — 1040 с.
2. Бейзер Б. Тестирование чёрного ящика. Технологии функционального тестирования программного обеспечения и систем.[Текст] / Б. Бейзер. — СПб.: Питер, 2004. — 320 с.
3. Криспин Л. Гибкое тестирование: практическое руководство для тестировщиков ПО и гибких команд [Текст] / Лайза Криспин, Джанет Грегори. — М.: «Вильямс», 2010. — 464 с.
4. Майерс Гл. Искусство тестирования программ [Текст] / Гленфорд Майерс, Том Баджетт, Кори Сандлер, 3-е издание — М.: «Диалектика», 2012. — 272 с.
5. Плаксин М. Модель Миллса Тестирование и отладка программ для профессионалов будущих и настоящих. [Текст] / М. Плаксин — 2-е изд. — М.: БИНОМ, 2013. — 167 с.
6. Техническая документация Microsoft, цикл статей по «Windows Forms» [Электронный ресурс]. — Режим доступа: <https://docs.microsoft.com/ru-ru/dotnet/framework/winforms/> (дата обращения 01.04.2020).

**Приложение А**  
**(обязательное)**  
**Листинг программы**

**«Program.cs»**

```
using System;using System.Windows.Forms;
namespace Testing_Kurs{
    static class Program    {
        [STAThread]
        static void Main()    {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
    }
}
```

**«Form1.cs»**

```
using System;using System.Windows.Forms;
namespace Testing_Kurs{
    public partial class Form1 : Form    {
        public Form1()    {
            InitializeComponent();
        }
        private void Form1_Load(object sender, EventArgs e)    {
            LoadQuestions.LoadAllQuestions();
        }
        private void buttoEnterName_Click(object sender, EventArgs e)    {
            Model.userName = textBoxUserName.Text;
        }
        private void buttonStartTesting_Click(object sender, EventArgs e)    {
            TestingThemes testingThemes = new TestingThemes();
            testingThemes.ShowDialog();
        }
        private void buttonExit_Click(object sender, EventArgs e)    {
            Close();
        }
    }
}
```

**«TestingThemes.cs»**

```
using System;using System.Windows.Forms;
```



```

namespace Testing_Kurs{
    public partial class TestingThemes : Form    {
        public TestingThemes()    {
            InitializeComponent();
        }
        private void buttonMarvel_Click(object sender, EventArgs e)    {
            Model.testNumber = 1;
            Test test = new Test();
            test.ShowDialog();
        }
        private void buttonDC_Click(object sender, EventArgs e)    {
            Model.testNumber = 2;
            Test test = new Test();
            test.ShowDialog();
        }
        private void buttonSchool_Click(object sender, EventArgs e)    {
            Model.testNumber = 3;
            Test test = new Test();
            test.ShowDialog();
        }
        private void buttonBack_Click(object sender, EventArgs e)    {
            Close();
        }
    }
}

```

### «Test.cs»

```

using System;using System.Collections.Generic;using System.Linq;using System.Windows.Forms;
using static Testing_Kurs.Model;
namespace Testing_Kurs{
    public partial class Test : Form    {
        private List<QuestionClass> testQuestions = new List<QuestionClass>();
        public Test()    {
            InitializeComponent();
            rightAnswersCount = 0;
            switch(testNumber)    {
                case 1:
                    testQuestions = questionsMarvel.ToList();
                    labelTheme.Text = "Marvel";
                    LoadQuestion(0);
                    break;
                case 2:

```

```

        testQuestions = questionsDC.ToList();
        labelTheme.Text = "DC";
        LoadQuestion(0);
        break;
    case 3:
        testQuestions = questionsSchool.ToList();
        labelTheme.Text = "Школьная программа";
        LoadQuestion(0);
        break;
    }
}

private void LoadQuestion(int questionNumber)    {
    groupBoxQuestion.Text = testQuestions[questionNumber].Question;
    radioAnswer1.Text = testQuestions[questionNumber].Answers[0];
    radioAnswer2.Text = testQuestions[questionNumber].Answers[1];
    radioButtonAnswer3.Text = testQuestions[questionNumber].Answers[2];
    radioButtonAnswer4.Text = testQuestions[questionNumber].Answers[3];
}

private void buttonNextQuestion_Click(object sender, EventArgs e)    {
    // 1
    foreach (Control control in groupBoxQuestion.Controls)    {
        // 2
        if (control is RadioButton)    {
            // 3
            RadioButton radioButton = control as RadioButton;
            // 4
            if (radioButton.Checked)    {
                // 5
                int trueAnswerNumber = testQuestions[0].TrueAnswer;
                // 6
                if (radioButton.Text == testQuestions[0].Answers[trueAnswerNumber - 1])    {
                    // 7
                    rightAnswersCount++;
                }
            }
        }
    }
    // 8
    testQuestions.RemoveAt(0);
    // 9
    if (testQuestions.Count == 1)    {
        // 10

```

```

        LoadQuestion(0);
        buttonNextQuestion.Text = "Результат";
    }
    // 11
    else if (testQuestions.Count == 0)    {
        // 12
        Result result = new Result();
        result.ShowDialog();
        Close();
    }
    // 13
    else    {
        // 14
        LoadQuestion(0);
    }
}

private void buttonExitToMain_Click(object sender, EventArgs e)    {
    Close();
}
}
}

```

### «Result.cs»

```

using System;using System.Windows.Forms;using static Testing_Kurs.Model;
namespace Testing_Kurs{
    public partial class Result : Form    {
        public Result()    {
            InitializeComponent();
            label1.Text = $"Ваш результат, {userName}:";
            if (rightAnswersCount < 6)    {
                labelResult.Text = "Плохо, Вы ответили правильно менее, чем на 60% вопросов";
            }
            else if (rightAnswersCount > 5    && rightAnswersCount < 8)    {
                labelResult.Text = "Удовлетворительно, Вы ответили правильно более, чем на 60% вопросов";
            }
            else if (rightAnswersCount > 7    && rightAnswersCount < 9)    {
                labelResult.Text = "Хорошо, Вы ответили правильно более, чем на 70% вопросов";
            }
            else if (rightAnswersCount > 8)    {
                labelResult.Text = "Отлично, Вы ответили правильно более, чем на 80% вопросов";
            }
        }
    }
}

```

```

private void buttonExit_Click(object sender, EventArgs e)    {
    Close();
}
}
}

```

### «LoadQuestions.cs»

```

using System;using System.Collections.Generic;using System.IO;
using System.Linq;using System.Text;using static Testing_Kurs.Model;
namespace Testing_Kurs{
    class LoadQuestions    {
        public static void LoadAllQuestions()    {
            string path = @"..\..\Tests\Marvel.txt";
            questionsMarvel = GetQuestions(path).ToList();
            path = @"..\..\Tests\DC.txt";
            questionsDC = GetQuestions(path).ToList();
            path = @"..\..\Tests\School.txt";
            questionsSchool = GetQuestions(path).ToList();
        }
        public static List<QuestionClass> GetQuestions(string path)    {
            // 1
            List<QuestionClass> questions = new List<QuestionClass>();
            // 2
            using (StreamReader sr = new StreamReader(path, Encoding.UTF8))    {
                // 3
                while (!sr.EndOfStream)    {
                    // 4
                    string question = sr.ReadLine();
                    string answer1 = sr.ReadLine();
                    string answer2 = sr.ReadLine();
                    string answer3 = sr.ReadLine();
                    string answer4 = sr.ReadLine();
                    string trueAnswer = sr.ReadLine();
                    // 5
                    List<string> answers = new List<string>() { answer1, answer2, answer3, answer4 };
                    // 6
                    QuestionClass questionClass = new QuestionClass(question, answers, Convert.ToInt32(trueAnswer));
                    questions.Add(questionClass);
                }
            }
            // 7

```

```

        return questions;
    }
}
}

```

### «Model.cs»

```

using System;using System.Collections.Generic;using System.Linq;
namespace Testing_Kurs{
    class Model {
        public class QuestionClass {
            public string Question { get; private set; }
            public List<string> Answers { get; private set; }
            public int TrueAnswer { get; private set; }
            public QuestionClass(string question, List<string> answers, int trueAnswer) {
                Question = question;
                Answers = answers.ToList();
                TrueAnswer = trueAnswer;
            }
            public void PrintQuestion() {
                Console.WriteLine($"Q: {Question}");
                foreach(string answer in Answers) {
                    Console.WriteLine($"A: {answer}");
                }
                Console.WriteLine($"TA: {TrueAnswer}");
            }
        }
        public static string userName;
        public static int testNumber;
        public static int rightAnswersCount;
        public static List<QuestionClass> questionsMarvel;
        public static List<QuestionClass> questionsDC;
        public static List<QuestionClass> questionsSchool;
    }
}

```