ФЕДЕРАЛЬНОЕ АГЕНТСТВО ВОЗДУШНОГО ТРАНСПОРТА

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ ГРАЖДАНСКОЙ АВИАЦИИ (МГТУ ГА)»

Кафедра вычислительных машин, комплексов, систем и сетей

Л.А. Надейкина

ТЕХНОЛОГИЯ ПРОГРАММИРОВАНИЯ

Учебно-методическое пособие

по выполнению лабораторных работ № 1, 2, 3, 4

для студентов III курса
направления 09.03.01
очной формы обучения

Москва
2019

ББК 6Ф7.3
Н-17

Рецензент:
Черкасова Н.И. – канд. физ.-мат. наук, доц. каф. ВМКСС

Надейкина Л.А.
Н-17 Технология программирования: учебно-методическое пособие по выполнению лабораторных № 1, 2, 3, 4./ Л.А. Надейкина. – Воронеж: ООО «МИР», 2019. – 48 с.

Данное учебно-методическое пособие издается в соответствии с рабочей программой учебной дисциплины «Технология программирования» по учебному плану для студентов III курса направления 09.03.01 очной формы обучения.

В Учебном пособии рассматривается на базе языка С++ одна из основных парадигм современного программирования: обобщенное программирование. При объектно-ориентированном программировании основное внимание уделяется аспекту данных, а при обобщенном — алгоритмам.

Цель обобщенного программирования — создание кода, который не зависит от типов данных. Шаблоны — это средства С++, предназначенные для создания обобщенных программ.

Рассмотрены шаблоны функций, определяющих общий алгоритм семейства функций, и шаблоны классов, позволяющие определить общие свойства классов. Рассмотрена библиотека STL, которая содержит набор шаблонов, представляющих контейнеры, итераторы, объекты функций и алгоритмы.

Рассмотрено и одобрено на заседании кафедры 26.03.2019 г. и методического совета 19.03.2019 г.

В авторской редакции

Подписано в печать 25.03.2019 г.
Формат 60x84/16 Печ.л. 3 Усл. печ. л. 3,49
Заказ 446/090443 Тираж 30 экз.

Московский государственный технический университет ГА
125993 Москва, Кронштадтский бульвар, д.20
Отпечатано ООО «Мир»
394033, г. Воронеж, Ленинский пр-т 119А, лит. Я, оф. 215

© Московский государственный технический университет ГА, 2019

1 ЛАБОРАТОРНАЯ РАБОТА № 1

Разработка программ на C# с использованием интерфейсов для снижения сложности программ.

1.1 Цель лабораторной работы

Целью лабораторной работы является получение навыков проектирования и реализации программного обеспечения с использованием интерфейсов, позволяющих повышать гибкость и эффективность программы.

1.2 Теоретические сведения

Проблемы разработки сложных программных систем

Характерной чертой современных программных систем является высокий уровень сложности. Сложность программной системы обусловлена [1]:

- сложностью реальной предметной области и соответственно трудностью анализа предметной области и проектирования программной системы;
- трудностью управления процессом разработки;
- необходимостью обеспечить достаточную гибкость программы;
- неудовлетворительными способами описания больших дискретных систем.

Сложное программное обеспечение труднее анализировать, тестировать, отлаживать, сопровождать, но прежде всего его сложнее изучать и использовать. Издержки сложности, не устраненные во время разработки, резко проявляются после внедрения программы. Сложность создает источники ошибок, которые создают проблемы на протяжении всего срока службы программного обеспечения.

Можно выделить три источника сложности – это *сложности реализации*, степень трудности, с которой столкнется программист, пытаясь понять программу, для того чтобы создать ее ментальную модель или отладить; с другой стороны, потребители и пользователи склонны видеть сложность в самих понятиях *сложности интерфейса программы;* оба описанных выше вида сложности управляют третьим, более простым: общее количество строк кода в системе, то есть *размер кодовой базы*.

С источниками сложности необходимо бороться различными способами. Размер кодовой базы можно уменьшить с помощью лучших инструментальных средств. Сложность реализации можно уменьшить с помощью более тщательного выбора алгоритмов. Сложность интерфейса пользователя необходимо исправлять, тщательнее проектируя взаимодействие программы с пользователем, данный навык предполагает анализ эргономических характеристик и психологии пользователя.

За последние годы *объектно-ориентированная технология анализа и проектирования* (ООАП) проникла в различные разделы компьютерных наук. К ней относятся как к средству преодоления сложности, присущей многим реальным системам, и создания хорошей архитектуры программного обеспечения (ПО).

Архитектура программного обеспечения (software architecture) – это представление, которое даёт информацию о компонентах программного обеспечения, обязанностях отдельных компонентов и правилах организации связей между компонентами. Продуманная архитектура облегчает разработку и дальнейшее развитие программного обеспечения. Она служит базисом, каркасом создаваемой системы, интегрируя отдельные компоненты и создавая высокоуровневую модель системы.

Термин "SOLID" представляет собой акроним для набора практик проектирования программного кода и построения гибкой и адаптивной программы. Данный термин был введен известным американским специалистом в области программирования Робертом Мартином [2].

Пять основных принципов дизайна классов в проектировании, которые следует учитывать при написании кода, сокращенно называются SOLID и расшифровываются так:

- Single Responsibility Principle (Принцип единственной обязанности)
- Open Closed Principle (Принцип открытости/закрытости)
- Liskov's Substitution Principle (Принцип подстановки Барбары Лисков)
- Interface Segregation Principle (Принцип разделения интерфейса)
- Dependency Inversion Principle (Принцип инверсии зависимостей)

Принцип единственной обязанности заключается в том, что класс должен выполнять одну единственную задачу. Весь функционал класса должен быть целостным, обладать высокой связностью (high cohesion). Однако бывают такие классы, они еще называются "божественными", которые инкапсулируют в себе абсолютно всю функциональность. Применение подобных классов надо избегать. Класс следует применять только для одной задачи, например, - либо бизнес-логика, либо вычисления, либо работа с данными.

Принцип открытости/закрытости можно сформулировать так: система должна быть построена таким образом, что все ее последующие изменения должны быть реализованы с помощью добавления нового кода, а не изменения уже существующего.

Принцип подстановки Лисков является руководством по созданию иерархий наследования, предоставляющих возможность вместо базового типа подставить любой его подтип и избежать возможных проблем при применении полиморфизма.

Принцип разделения интерфейсов относится к тем случаям, когда классы имеют раздутый интерфейс, не все методы которого могут быть востребованы. Принцип разделения интерфейсов можно сформулировать так: клиенты не должны вынужденно зависеть от методов, которыми не пользуются. В этом случае интерфейс класса разделяется на отдельные части, которые составляют отдельные интерфейсы. Эти интерфейсы независимо друг от друга могут применяться и изменяться. Применение принципа разделения интерфейсов делает систему слабосвязанной, и ее легче модифицировать и обновлять.

Принцип инверсии зависимостей служит для создания слабосвязанных сущностей, которые легко тестировать, модифицировать и обновлять. Суть его в следующем. Модули верхнего уровня не должны зависеть от модулей нижнего

уровня. И те и другие должны зависеть от абстракций (интерфейсов). Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

Принципы SOLID - это ни догмы, которые надо обязательно применять при разработке, однако их использование позволит улучшить код программы, упростить возможные его изменения и поддержку.

Рассмотрим, критерии "хорошей архитектуры".

Эффективность системы. В первую очередь программа, конечно же, должна решать поставленные задачи и хорошо выполнять свои функции, причем в различных условиях. Сюда же можно отнести такие характеристики, как надежность, безопасность, производительность, способность справляться с увеличением нагрузки (масштабируемость).

Гибкость системы. С течением времени появляются новые требования встает необходимость изменять приложение. Архитектурное решение должно допускать внесение изменений. Изменение одного фрагмента должно практически не влиять на другие фрагменты.

Расширяемость системы. Архитектура должна позволять легко наращивать дополнительный функционал по мере необходимости.

Требования, чтобы архитектура системы обладала гибкостью и расширяемостью (то есть была способна к изменениям и эволюции) выделена в виде отдельного принципа SOLID — «Принципа открытости/закрытости»: программные сущности - классы, модули, функции и тому подобное - должны быть открытыми для расширения, но закрытыми для модификации.

То есть проектировать надо так, чтобы новая функциональность добавлялась путем написания нового кода и при этом существующий – не изменялся. Это так называемая *"плагинная архитектура"* (Plugin Architecture).

Соответственно, когда речь идет о построении архитектуры программы, под этим, главным образом, подразумевается декомпозиция программы на подсистемы (функциональные модули, сервисы, слои, подпрограммы) и организация их взаимодействия друг с другом и внешним миром. Причем, чем более независимы будут подсистемы, тем безопаснее разрабатывать каждую из них в отдельности не заботиться обо всех остальных частях.

Модуль – фрагмент программного текста, являющийся строительным блоком для физической структуры системы. Как правило, модуль состоит из интерфейсной части и части-реализации.

Модульность – свойство системы, которая может подвергаться декомпозиции на ряд внутренне связанных и слабо зависящих друг от друга модулей.

Деление на модули - подсистемы лучше всего производить исходя из тех задач, которые решает система. Основная задача разбивается на составляющие ее подзадачи, которые могут решаться или выполняться независимо друг от друга.

Чем слабее связанность модулей, тем легче писать, понимать, расширять, исправлять программу. *Можно сказать, что методы для уменьшения связанности, как раз и составляют основной "инструментарий архитектора".*

Один из принципов проектирования гласит, что при создании системы классов надо программировать на уровне ***интерфейсов,*** а не их конкретных реализаций. Под интерфейсами в данном случае понимаются не только типы, определенные с помощью ключевого слова ***interface***, а определение функционала без его конкретной реализации.

Если объекты обработки малосвязанные между собой, то для определения общего для всех них функционала лучше определить интерфейс.

Интерфейсы позволяют уменьшать связанность системы, так же как стоящий за ними принцип ***Инкапсуляция + Абстракция + Полиморфизм.***

Модули должны быть друг для друга "черными ящиками" (*Инкапсуляция*). Это означает, что один модуль не должен что-либо знать о внутренней структуре другого модуля.

Модули (подсистемы) должны взаимодействовать друг с другом лишь посредством интерфейсов, то есть, А*бстракций*, не зависящих от деталей реализации. Соответственно каждый модуль должен иметь четко определенный интерфейс или интерфейсы для взаимодействия с другими модулями.

В этом случае код будет работать одинаково с любой реализацией, соответствующей *контракту интерфейса*. Собственно, именно эта возможность работать с различными реализациями (модулями или объектами) через унифицированный интерфейс и называется полиморфизмом. *Полиморфизм* — это не только переопределение методов, но и — *взаимозаменяемость модулей-объектов с одинаковым интерфейсом, или - "один интерфейс, множество реализаций"*.

Благодаря интерфейсам и полиморфизму, как раз и достигается возможность модифицировать и расширять код, без изменения того, что уже написано (Open-Closed Principle).

Концепция интерфейсов включает в себя и обобщает почти все основные принципы SOLID.

1.3 Задание на выполнение лабораторной работы

Лабораторная работа выполняются в среде Microsoft Visual Studio на языке C# как консольные приложения.

1) Разработать интерфейс, определяющий функциональность стека с элементами типа ***object.*** Интерфейс может содержать только прототипы функций и свойства, которые в данной задаче только возвращают значения с помощью метода ***get().***

2) Реализовать стек на базе двух разных классов, хранящих элементы. Например, в первом классе - для хранения элементов использовать стандартный класс ArrayList, принадлежащий пространству имен System.Collection. Во втором из классов, реализующих интерфейс стека, применить для хранения элементов стека массив с элементами типа ***object.***

3) объявить статический метод для вывода на экран сведения об элементах стека.

4) В основной программе создав объекты классов, реализующих интерфейс стека, продемонстрировать их особенности и возможности статического метода для вывода элементов стека.

5) Варианты реализации интерфейсов:

- интерфейс стека - реализация посредством двусторонней очереди и списка;
- интерфейс очереди - реализация через двустороннюю очередь и список;
- интерфейс очереди с приоритетом – реализация посредством вектора и списка.

1.4 Порядок выполнения лабораторной работы

Стек – это классическая структура данных, для которой определены следующие функции: поместить элемент на вершину стека, извлечь элемент с вершины стека, получить значение элемента с вершины стека, не удаляя его из стека. А также в стеке имеется функция, проверяющая, не пуст ли стек.

В библиотеке обобщенных классов имеется класс обобщенных стеков, имеющих все перечисленные функции, но его не надо использовать.

Надо разработать программу так, чтобы, несмотря на различные реализации, стек можно применять, не обращая внимания на различия в реализациях, не замечая этого различия. Такую возможность обеспечивает применение ***интерфейса,*** определяющего функциональность целого семейства стеков.

1) Определить интерфейс стеков:

```
public interface IStack {
int Count {get;}                //число элементов в стеке
object Pop();                   //вытолкнуть элемент из стека
object Peek {get;}              //посмотреть верхний элемент
bool Push (object item);        //втолкнуть элемент в стек
string [] ToStringArray();      //массив записей об элементах
}
```

Интерфейс ***IStack*** не конкретизирует тип элементов стека, так как и параметр метода ***Push()*** и тип возвращаемого результата метода ***Pop()*** и значение свойства ***Peek*** имеют тип ***object*****,** так как тип ***object*****,** является базовым для всех типов языка **C#,** то это позволит хранить в стеке элементы любых типов.

В дополнение к традиционным для стека средствам в интерфейс включен прототип метода ToStringArray() для получения массива строк, содержащих сведения об включенных в стек элементах.

2) Создать реализацию стека, на базе класса хранения элементов ArrayList:

```
class StackList : IStack { // стек без ограничения объема.
ArrayList list;              //список элементов в стеке
public StackList() {         //конструктор
list = new ArrayList();
```

}

… // определить все функции в соответствие с интерфейсом IStack

};

- Конструктор создает пустой список и связывает его со ссылкой list.

- Ссылка list адресует растущий список.

- Свойство Count возвращает счетчик элементов списка.

- При извлечении элемента с вершины стека, метод Pop() вернет либо ссылку на элемент типа ***object*** и уменьшит количество элементов списка, либо значение типа ***null*** , если в стеке (списке) нет элементов.

- Свойство Peek работает похожим образом, но элемент из списка (из стека) не удаляется.

- Так как список для хранения элементов в этом классе не ограничен, метод Push(), поместив элемент в стек, всегда возвращает значение ***true***.

- в методе ToStringArray() создается массив строк и каждой строке массива присваивается результат применения метода ToString() к каждому элементу стека. При этом список стека сохраняется, но в массив строк записи помещаются в хронологическом порядке (первый вошел – первый в списке).

3) Создать реализацию стека, в котором для хранения элементов использовать массив фиксированных размеров с элементами типа ***object***:

```
class StackArray: IStack { // стек без ограничения объема.
object [] arr;      // элементы стека
int top             //число элементов стеке
public StackArray (int capacity) {      //конструктор
arr =new object[capacity];
top =0;
}
… // определить все функции в соответствие с интерфейсом IStack
};
```

- Параметр int capacity конструктора класса StackArray определяет потенциальную емкость стека - размер массива object [] arr для его элементов.

- Число элементов в стеке (int top*)* не может превысить емкость стека.

- Значение top увеличивается при добавлении в стек элемента и уменьшается при выталкивании элемента из стека.

- Если стек полон, то метод Push() возвращает значение ***false*** и пополнение стека не происходит.

- Классы StackList и StackArray очень похожи и функционально неразличимы до тех пор, пока стек класса StackArray не будет переполнен.

4) Определить основной класс программы.

5) В основном классе программы определить статический метод printStack() для вывода информации о стеке. Его основная особенность - применение параметра с типом интерфейса IStack. При вызове метода аргументом может быть ссылка на объект любого класса, реализовавшего интерфейс IStack.

```
static void printStack() (IStack ist){
foreach(var elem in ist.ToStringArray())
Console.WriteLine(elem)
}
```

6) В основном классе программы определить также статический метод Main(), в котором показать возможности метода RrintStack() и особенности стеков, формируемых как объекты классов StackList и StackArray.

- Создать «расширяемый стек» как объект класса StackList. С помощью метода Push() в стек занести элементы разных типов. Показать возможности класса StackList. Вывести элементы стека, используя метод printStack().

- Создать объект класса StackArray с ёмкостью 10 элементов. В стек занести 6 элементов разных типов и их значения вывести на экран методом printStack().

7) Представить диаграмму классов программы с реализацией интерфейса стеков.

1.5. Контрольные вопросы

1) Что такое технология программирования?
2) Программная инженерия и технология программирования.
3) Программное обеспечение. Программное средство.
4) Понятие «правильности программы». «Надежное» программное средство.
5) Программы «большие» и «маленькие». Особенности и свойства маленьких и больших программ.
6) Источники сложности современных программных систем.
7) Способы борьбы с источниками сложности.
8) Объектно-ориентированная технология анализа и проектирования (ООАП) как средство преодоления сложности и создания хорошей архитектуры ПО.
9) Архитектура программного обеспечения (software architecture).
10) Принципы и критерии создания хорошей архитектуры.
11) Модульность архитектуры и слабая связанность между модулями.
12) Интерфейсы. Основное назначение.
13) Различие интерфейсов и абстрактных классов.
14) Основные конструкции языка C#.

2 ЛАБОРАТОРНАЯ РАБОТА № 2

Разработка программ на C# с графическим интерфейсом пользователя.

2.1 Цель лабораторной работы

Целью лабораторной работы является

- во-первых, получение навыков создания удобного пользовательского интерфейса;

- во-вторых, получение навыков создания графических интерфейсов с помощью технологии WinForms, проектирования форм с размещения на них

управляющих элементов, а также освоение механизма управления программами с помощью событий.

2.2 Теоретические сведения

Основы создания удобного пользовательского интерфейса

Одним из важных показателей качества программного обеспечения является *удобство его использования*. Оно описывается с помощью таких характеристик, как

- понятность пользовательского интерфейса,
- легкость обучения работе с ним,
- трудоемкость решения определенных задач с его помощью,
- производительность работы пользователя с ПО,
- частота появления ошибок и жалоб на неудобства.

Для построения действительно удобных программ нужен учет контекста их использования, психологии пользователей, необходимости помогать начинающим пользователям и предоставлять все нужное для работы опытных пользователей. *Однако самым значимым фактором является то, помогает ли данная программа решать действительно значимые для пользователей задачи.*

Следующие *принципы* позволяют находить решения, повышающие удобство пользовательского интерфейса:

Принцип структуризации. Пользовательский интерфейс должен быть целесообразно структурирован. Близкие по смыслу, родственные его части должны быть связаны видимым образом, а независимые — разделены.

Человек воспринимает и осознает информацию, а также производит действия достаточно медленно по сравнению с компьютером. Сама «медленность» действий человека и его восприятия, а также соотношения затрат времени на различные действия должны учитываться при проектировании интерфейсов, рассчитанных на взаимодействие с человеком.

Глаз быстрее руки — человек гораздо быстрее узнает что-то, чем производит соответствующие действия. Поэтому, в частности, человеку часто удобнее работать с системами контекстной подсказки, предлагающими ему возможные варианты его дальнейшего ввода, чем набивать весь текст целиком самостоятельно.

Принципы отображения графического интерфейса. Принцип простоты. Наиболее распространенные операции должны выполняться максимально просто. При этом должны быть видимые ссылки на более сложные процедуры. Очевидность. Краткость. Обратимость - можно легко вернуться на предыдущую страницу.

Принцип видимости. Все функции и данные, необходимые для решения определенной задачи, должны быть видны, когда пользователь пытается ее решить.

Принцип обратной связи. Пользователь должен получать сообщения о действиях системы и о важных событиях внутри нее. Сообщения должны быть

информативными, краткими, однозначными и написанными на языке, понятном пользователю.

Принцип толерантности. Интерфейс должен быть гибким и терпимым к ошибкам пользователя. Ущерб от ошибок должен снижаться за счет возможности отмены и повтора действий и за счет разумной интерпретации любых разумных

действий пользователя и введенных им данных. По возможности, следует избегать обязывающего взаимодействия (модальных диалогов), основанного на ограничении свободы пользователя.

Принцип повторного использования. Следует стараться использовать многократно внутренние и внешние компоненты, обеспечивая тем самым унифицированность интерфейса и сходство между похожими его элементами.

Резюмируя вышесказанное, можно назвать шесть основных требований для создания удобного и функционального интерфейса пользователя:

1. В приложении должна быть цель — та задача, которую решает пользователь.
2. Нельзя отвлекать внимание от цели лишними элементами.
3. Элементы располагаются так, чтобы ими было удобно пользоваться в обычном для приложения контексте.
4. Связанные по смыслу элементы располагаются рядом друг с другом.
5. Элементы имеют иерархию по важности и располагаются согласно этой иерархии.
6. Интерфейс должен быть гибким и терпимым к ошибкам пользователя.

Создание графического приложения

Для создания графических интерфейсов пользователей с помощью платформы .NET применяются разные технологии - Window Forms, WPF, приложения для магазина Windows Store (для ОС Windows 8/8.1/10). Однако наиболее простой и удобной платформой до сих пор остается Window Forms или формы.

Для создания графического проекта потребуется среда разработки Visual Studio. Поскольку наиболее распространенная пока версия Visual Studio 2013, то для данного описания будет использоваться бесплатная версия данной среды Visual Studio Community 2013 .

Запустим Visual Studio и создадим проект графического приложения. Для этого в меню выберем пункт File (Файл) и в подменю выберем **New - > Project** (Создать - > Проект). После этого откроется диалоговое окно создания нового проекта (рис.1).

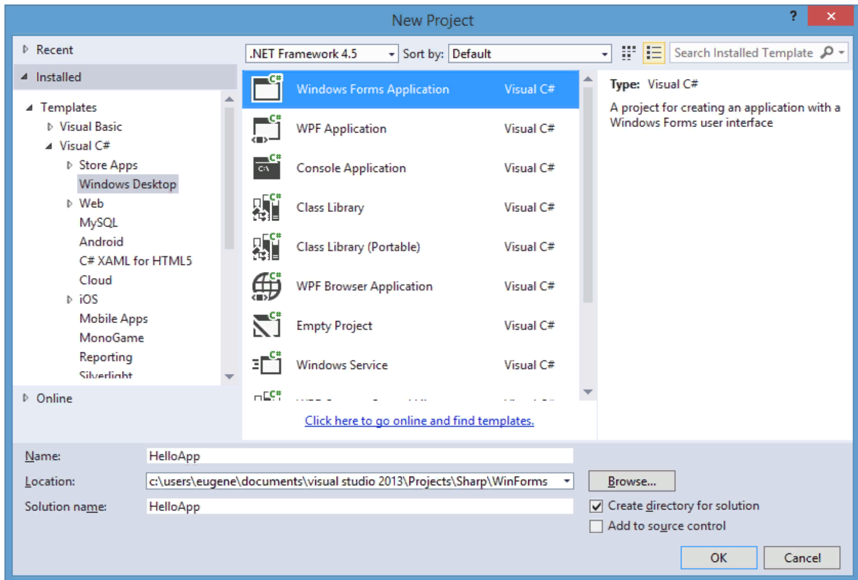


Рисунок 1. Окно создания нового проекта

В левой колонке выберем **Windows Desktop**, а в центральной части среди типов проектов - тип **Windows Forms Application** и дадим ему какое-нибудь имя в поле внизу. Например, назовем его *HelloApp*. После этого Visual Studio откроет наш проект с созданными по умолчанию файлами (рис.2).

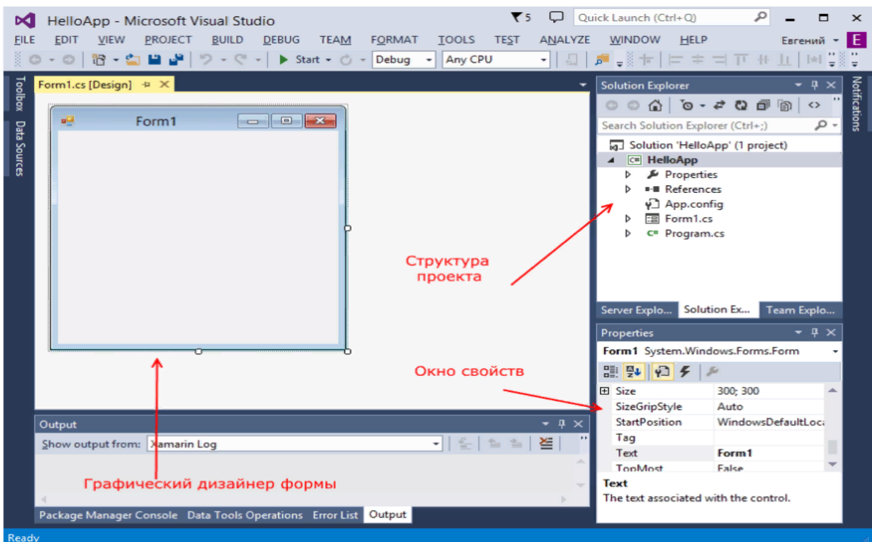


Рисунок 2. Окно проекта

Большую часть пространства Visual Studio занимает графический дизайнер, который содержит форму будущего приложения. Пока она пуста и имеет только заголовок Form1. Справа находится окно файлов решения/проекта - Solution Explorer (Обозреватель решений). Там и находятся все связанные с приложением файлы, в том числе файлы формы *Form1.cs*.

Внизу справа находится окно свойств - Properties. Так как в данный момент выбрана форма как элемент управления, то в этом поле отображаются свойства, связанные с формой. Теперь найдем в этом окне свойство формы Text и изменим его значение на любое другое.

Теперь перенесем на поле какой-нибудь элемент управления, например, кнопку. Для этого найдем в левой части Visual Studio вкладку Toolbox (Панель инструментов). Нажмем на эту вкладку, и откроется панель с элементами, откуда можно с помощью мыши перенести на форму любой элемент. Найдем среди элементов кнопку и, захватив ее указателем мыши, перенесем на форму (рис.3).

После перетаскивания элемент помечается надписью «button1», чтобы изменить надпись надо обратиться к спискам свойств этих элементов.

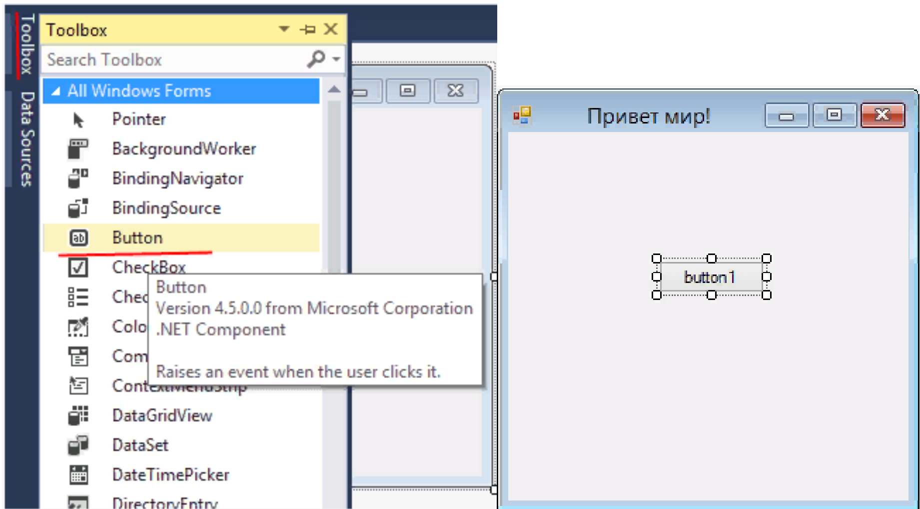


Рисунок 3. Панель выбора элемента управления и редактор экранной формы

Для этого надо щелкнуть левой клавишей мыши по изображению элемента и активировать пункт свойства.

Это визуальная часть. Теперь приступим к самому программированию. Добавим простейший код на языке C#, который бы выводил сообщение при нажатии кнопки. Для этого надо перейти в файл кода, который связан с этой формой. Если не открыт файл кода, можно нажать на форму правой кнопкой мыши и в появившемся меню выбрать View Code (Посмотреть файл кода, или нажать клавишу F7).

Однако воспользуемся другим способом, чтобы не писать много лишнего кода. Наведем указатель мыши на кнопку и щелкнем по ней двойным щелчком левой клавишей мыши. Автоматически попадаем в файл кода Form1.cs, в который добавится обработчик события «button1_Click» (нажатие на кнопку «button1»), и который выглядит так:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
namespace HelloApp
{   public partial class Form1 : Form
   {  public Form1()
      {  InitializeComponent();   }
       private void button1_Click(object sender, EventArgs e)
      {        }
   }}
```

Добавим вывод сообщения при нажатии кнопки, изменив код обработчика события «button1_Click» следующим образом:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
 namespace HelloApp
{   public partial class Form1 : Form
   { public Form1()
      { InitializeComponent();  }
       private void button1_Click(object sender, EventArgs e)
      {    MessageBox.Show("Привет");
      }
   }}
```

Чтобы запустить приложение в режиме отладки, нажмем на клавишу F5 или на зеленую стрелочку на панели Visual Studio. После этого запустится наша

форма с одинокой кнопкой. И если нажмем на кнопку на форме, то нам будет отображено сообщение с приветствием.

После запуска приложения студия компилирует его в файл с расширением exe. Найти данный файл можно в папке проекта в каталоге bin/Debug или bin/Release.

Внешний вид приложения является нам преимущественно через формы. Формы являются основными строительными блоками. Они предоставляют контейнер для различных элементов управления. А механизм событий позволяет элементам формы отзываться на ввод пользователя, и, таким образом, взаимодействовать с пользователем.

При открытии проекта в Visual Studio в графическом редакторе можно увидеть визуальную часть формы - ту часть, которую видим после запуска приложения и куда переносим элементы с панели управления. Но на самом деле форма скрывает мощный функционал, состоящий из методов, свойств, событий и прочее. Рассмотрим основные свойства форм.

При запуске простейшего приложения отобразится одна пустая форма. Однако даже такой простой проект с пустой формой имеет несколько компонентов (рис.4):

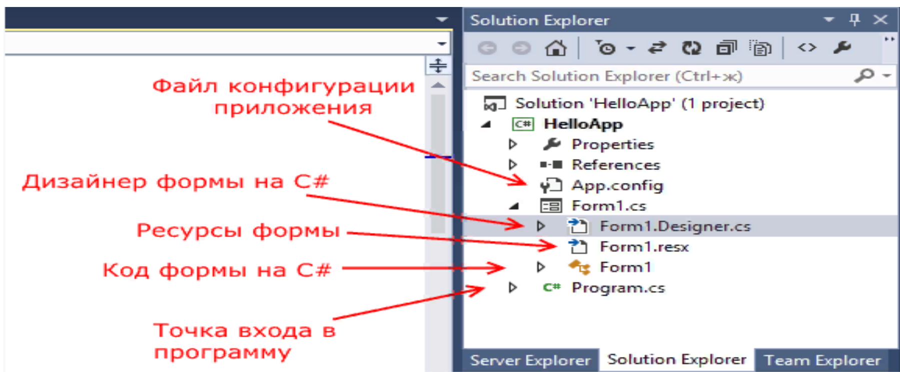


Рисунок 4. Файлы проекта

Стартовой точкой входа в графическое приложение является класс Program, расположенный в файле *Program.cs*:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using System.Windows.Forms;
 namespace HelloApp
{   static class Program
   {  [STAThread]
      static void Main()
      {   Application.EnableVisualStyles();
```

```
        Application.SetCompatibleTextRenderingDefault(false);
        Application.Run(new Form1());
    }
}}
```

Сначала системой в данном классе, запускается главный метод класса приложения Main(), в котором в свою очередь вызывается для данного приложения метод Run() (Application.Run(new Form1())), он и запускает приложение и форму Form1. Если нужно изменить стартовую форму в приложении на какую-нибудь другую, то надо изменить в вызове Form1 на соответствующий класс формы.

Основные свойства форм.

С помощью специального окна Propertie (рис.5) справа Visual Studio предоставляет нам удобный интерфейс для управления свойствами элемента:

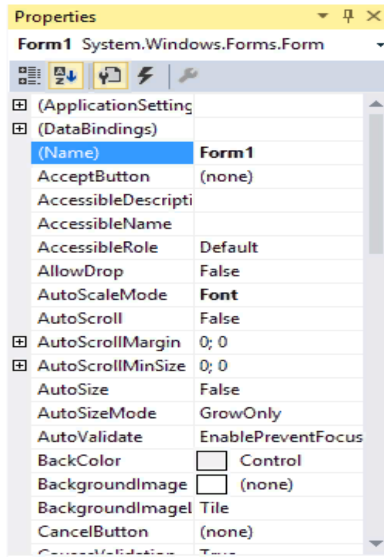


Рисунок 5. Окно свойств элемента

Большинство этих свойств оказывает влияние на визуальное отображение формы. Основные свойства:

Name: устанавливает имя формы - точнее имя класса, который наследуется от класса Form.

BackColor: указывает на фоновый цвет формы. Щелкнув на это свойство, мы сможем выбрать тот цвет, который нам подходит из списка предложенных цветов или цветовой палитры.

BackgroundImage: указывает на фоновое изображение формы.

BackgroundImageLayout: определяет, как изображение, заданное в свойстве **BackgroundImage**, будет располагаться на форме.

ControlBox: указывает, отображается ли меню формы. В данном случае под меню понимается меню самого верхнего уровня, где находятся иконка приложения, заголовок формы, а также кнопки минимизации формы и крестик. Если данное свойство имеет значение false, то мы не увидим ни иконку, ни крестика, с помощью которого обычно закрывается форма.

Cursor: определяет тип курсора, который используется на форме.

Enabled: если данное свойство имеет значение false, то она не сможет получать ввод от пользователя, то есть мы не сможем нажать на кнопки, ввести текст в текстовые поля и т.д.

Font: задает шрифт для всей формы и всех помещенных на нее элементов управления. Однако, задав у элементов формы свой шрифт, мы можем тем самым переопределить его.

ForeColor: цвет шрифта на форме

Icon: задает иконку формы

Location: определяет положение по отношению к верхнему левому углу экрана, если для свойства StartPosition установлено значение Manual

MaximizeBox: указывает, будет ли доступна кнопка максимизации окна в заголовке формы

MinimizeBox: указывает, будет ли доступна кнопка минимизации окна

Opacity: задает прозрачность формы

Size: определяет начальный размер формы

StartPosition: указывает на начальную позицию, с которой форма появляется на экране

Text: определяет заголовок формы

TopMost: если данное свойство имеет значение true, то форма всегда будет находиться поверх других окон

Visible: видима ли форма, если мы хотим скрыть форму от пользователя, то можем задать данному свойству значение false

Итак, чтобы в интегрированной среде разработки (например, в MS VS) разрабатывать программы с графическим интерфейсом пользователя, нужно:

1) научиться проектировать формы, размещая на них требуемые по смыслу задачи элементы пользовательского интерфейса (элементы управления) и компоненты;
2) понимать особенности управления программами с помощью событий;
3) уметь добавлять в создаваемую программу шаблоны (заготовки) обработчиков событий, возникающих при воздействии пользователя (прямо или косвенно) на элементы управления;
4) уметь дополнять заготовки обработчиков событий программным кодом, реализующим требования, предъявляемые к программе смыслом задачи.

На представленном ниже примере программы поясним особенности перечисленных выше пунктов, необходимых для разработки программы.

2.3 Задание на выполнение лабораторной работы

Разработать Windows-приложение в соответствием с вариантом задания.
Для создания графического интерфейса пользователя с помощью платформы .NET использовать технологию - Window Forms.

Для создания проекта использовать среду разработки Visual Studio и язык программирования C#.

При разработке проекта учитывать принципы создания удобного пользовательского интерфейса.

2.4. Пример выполнения лабораторной работы

Задание. Разработать Windows-приложение. В поле TextBox вывести в виде списка элементы массива строк. Отредактировав (изменив) список на экране, вывести его в диалоговое окно MessageBox. Обеспечить возможность восстановления начального состояния списка [3].

Настройка свойств элементов (рис.6):

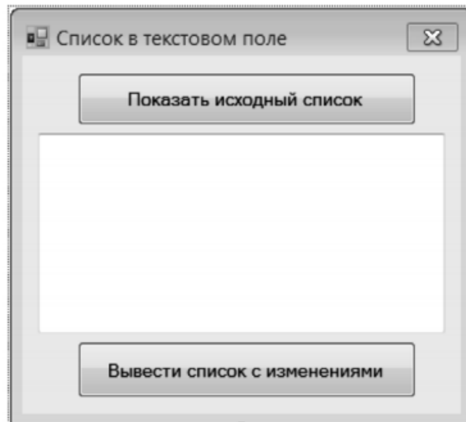


Рисунок 6. Изображение окна в редакторе форм

Form1.Text = Список в текстовом поле
Form1StartPosition = CenterScreen
button1.Text = Показать исходный список
button2.Text = Вывести список с изменениями
textbox1:Multiline = True
textbox1.Anchor = Top, Bottom, Left, Right

```
// prog.cs – редактируемый список в текстовом поле
using System.Windows.Forms;
namespace Program_1 {
public partial class Form1:Form {
public Form1( ) {
```

```
InitializeComponent( );
button2.Visible = false; // скрыть кнопку 2
}
string [] lines = new string[]
{“Каждый – ”,”Охотник – ”,”Желает – ”,”Знать – ”,
“Где – ”,”Сидит – ”, “Фазан – ” };
private void button1_Click(object sender, System.EventArgs e) {
textBox1.Lines = lines; // Вывести строки массива
button2.Visible = true; // показать кнопку 2
}
private void button2_Click(object sender, System.EventArgs e) {
string res = string.Join(“\n”, textBox1.Lines);
MessageBox.Show(“Результат изменений:\n”+res);
}
} // Form1
}
```

При создании объекта класса Form1 свойству Visible элемента button2 в конструкторе присваивается значение **false** и изображение кнопки button2 на форме не появляется (рис. 7).

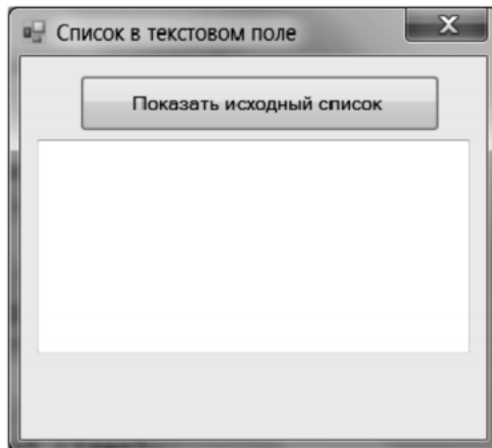


Рисунок 7. Форма после загрузки

В теле класса Form1 только одно поле lines – ссылка на инициализированный массив строк. При выполнении обработчика button1_Click() значение этой ссылки присваивается свойству Lines элемента (объекта) textBox1. Это приводит к выводу строк в текстовое поле.

Свойство Visible элемента button2 получает значение **true** и кнопка button2 становится видимой на форме (рис. 8).

Дополнение пользователем текста в окне textBox1 показано на рис. 8 слева. Справа – диалоговое окно MessageBox с результатами обработки текста.

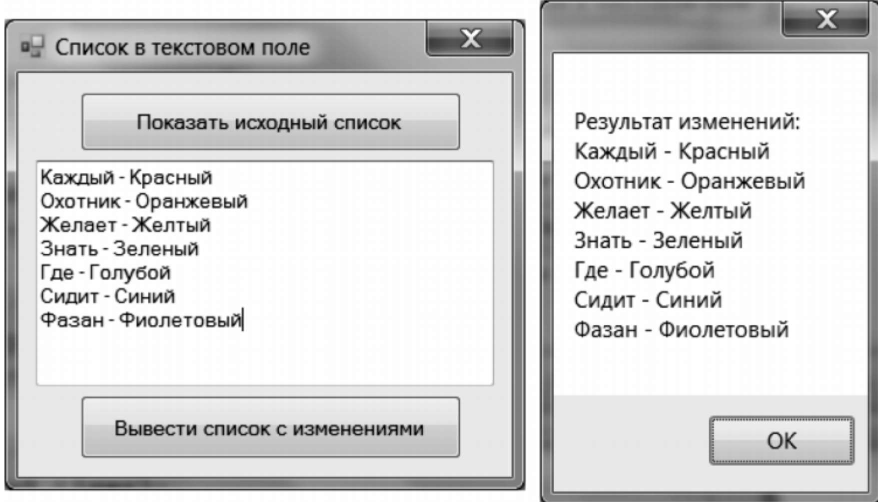


Рисунок 8. Редактирование текста и результат в диалоговом окне

2.5. Контрольные вопросы

1) Основные характеристики удобного интерфейса.

2) Основные факторы, которые надо учитывать при создании удобного интерфейса.

3) Психологические и физиологические факторы. Скоростные показатели деятельности человека. Внимание человека.

4) Понятность (Ментальная модель, Метафора, наглядность, стандарт). Память человека. Различные категории пользователей.

5) Факторы удобства использования и принципы создания удобного ПО

6) Методы разработки удобного программного обеспечения.

7) Проектирование формы в Windows Forms, размещение на них требуемые по смыслу задачи элементы пользовательского интерфейса (элементы управления) и компоненты.

8) Особенности управления программами с помощью событий.

9) Использование в создаваемой программе шаблонов (заготовки) обработчиков событий, возникающих при воздействии пользователя (прямо или косвенно) на элементы управления.

10) Изменять заготовки обработчиков событий программным кодом, реализующим требования, предъявляемые к программе смыслом задачи.

2.6. Варианты заданий лабораторной работы

Разработать Windows-приложение с удобным пользовательским интерфейсом:

1) Программа для просмотра изображений .

2) Написать программу, позволяющую создавать текстовый файл со списком факультетов. Каждая запись о факультете имеет три параметра:

- Название факультета, Ф.И.О. декана факультета, аббревиатура факультета.

Интерфейс программы должен состоять из двух окон:

- главное окно программы (просмотра файла);

- окно заполнения полей ввода (дополнение файла).

3) Написать программу-блокнот, позволяющую открывать, редактировать и сохранять текстовые документы.

Интерфейс программы должен иметь следующий вид (рис. 9):

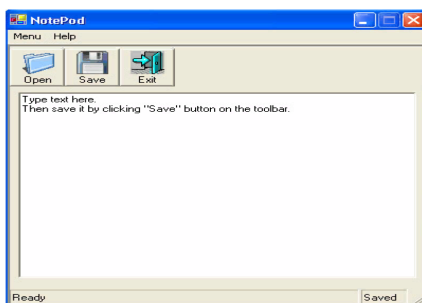


Рисунок 9. Интерфейс программы

4) Написать программу, предоставляющую возможность составить заказ на приобретение некоторого товара.

Интерфейс программы должен иметь следующий вид (рис.10):

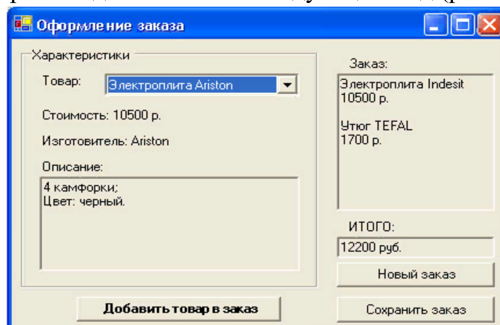


Рисунок 10. Интерфейс программы

5) Написать программу тестирования. Пользователю предлагается ответить на 10 вопросов. На каждый вопрос предусматривается по четыре варианта ответов.

Интерфейс программы должен состоять из трех окон: окно регистрации, окно тестирования, окно результатов

6) Написать программу "Учет пользователей". Программа хранит информацию о пользователях: имя, фамилию и адрес электронной почты. Изменять информацию пользователи могут только о себе.

Интерфейс программы должен состоять из трёх окон:
окно авторизации, окно создания новой учетной записи, главное окно программы

7) Поместить в центр формы одну кнопку и в её обработчике события «нажатие на кнопку» изменять размеры формы. В начале, при каждом нажатии на кнопку размеры формы уменьшать, но как только форма достигнет минимальных (заданных при разработке) размеров – увеличивать ее при нажатии на ту же кнопку. Когда форма достигнет максимальных размеров – переключить кнопку на уменьшение и т. д.

8) Периметр p правильного n-угольника, описанного около окружности радиуса r, равен 2*n*r*tg(PI/n).

Ввести значения n и r, проверить их корректность и вывести значение периметра. Проверяемые условия: n >= 3 и r > 0; отсутствие во входной строке нецифровых данных.

Размещение управляющих элементов на форме выполнить в относительных координатах. Ввести ограничения на минимизацию формы. (Изображения элементов не должны «налезать» друг на друга при уменьшении размеров формы.)

9) Написать программу, решающую квадратное уравнение. Интерфейс программы должен выглядеть, как показано на рис. 11.

3 ЛАБОРАТОРНАЯ РАБОТА № 3

Эволюционный подход к разработке программ.

3.1 Цель лабораторной работы

Целью лабораторной работы является:

- освоение принципов поэтапного проектирования и реализации приложения;

- овладение методологией интерактивности и постепенного увеличения функциональности программного продукта, когда решения, принятые на очередном этапе, существенно влияют на последующие этапы проектирования

3.2 Теоретические сведения

В современных методологиях программирования всегда предусматривается итеративность, проект развивается поэтапно, и решения, принятые на очередном этапе, могут существенно влиять на последующие этапы проектирования. На первых этапах создаются версии программного продукта, обладающие минимальной функциональностью, и эта функциональность нарастает при дальнейшей разработке. Можно сказать, что программа при таком подходе эволюционно развивается. Итеративность разработки и постепенное увеличение функциональности проектируемого продукта очень хорошо применимы при использовании интегрированных сред разработки, к которым относится MS VS.

Достоинством процесса создания программного обеспечения, построенного на основе эволюционного подхода, является то, что здесь спецификация может разрабатываться постепенно, по мере того как заказчик или сам разработчик осознает и сформулирует те задачи, которые должно решать программное обеспечение. Вместе с тем данный подход имеет и некоторые недостатки [4]:

1) Многие этапы процесса создания программного обеспечения не документированы. Но если система разрабатывается быстро, то экономически не выгодно документировать каждую версию системы.

2) Система часто получается плохо структурированной. Постоянные изменения в требованиях приводят к ошибкам и упущениям в структуре ПО. Со временем внесение изменений в систему становится все более сложным и затратным.

3) Часто требуются специальные средства и технологии разработки ПО. Это вызвано необходимостью быстрой разработки версий программного продукта.

Эволюционный подход наиболее приемлем для разработки небольших программных систем (до 100 000 строк кода) и систем среднего размера (до 500 000 строк кода) с относительно коротким сроком жизни. На больших долгоживущих системах заметно проявляются недостатки этого подхода [5].

Следуя эволюционным методикам создания программных продуктов, сформулируем рекомендации по выполнению разработки. Разработка должна выполняться по этапам (шагам), каждый из которых включает: проектирование (в том числе проектирование элементов визуального интерфейса); кодирование (включая синтаксическую и семантическую отладку кода) и тестирование.

1) Этапы разработки неформально делятся на два вида: исследовательские этапы и этапы собственно разработки, добавляющие создаваемому продукту новую функциональность в соответствии с техническим заданием (с условием задачи).

2) Каждый (даже первый) этап, независимо от его вида, предусматривает создание работающего программного продукта. На первом этапе создается программа с минимальной функциональностью.

3) В конце каждого этапа разработки выполняется анализ существующего (работоспособного) варианта программного продукта. Цель анализа – оценка возможности добавления средств (конструкций) для реализации новых, дополнительных требований, предусмотренных техническим заданием.

4) На основе результатов анализа либо выполняется реорганизация существующего варианта, либо осуществляется принятие новых проектных решений. В обоих случаях это новый этап разработки, включающий кодирование и тестирование.

5) После завершения каждого этапа разработки продукт, оставаясь работоспособным, приобретает новую (зачастую дополнительную) функциональность, либо изменяет уже имеющуюся функциональность, приближаясь к требованиям задания.

Таким образом, продукт постоянно находится в работоспособном состоянии и в него постоянно вносятся изменения.

Другими словами, программный продукт от шага к шагу, независимо от вида шагов эволюционирует, и его функциональность растет от минимальной до желаемой (указанной в техническом задании).

Покажем, как могут быть реализованы предложенные рекомендации на конкретном примере.

3.3 Задание на выполнение лабораторной работы

Разработать Windows-приложение в соответствием с вариантом задания. Для создания графического интерфейса пользователя с помощью платформы .NET использовать технологию - Window Forms.

Для создания проекта использовать среду разработки Visual Studio и язык программирования C#.

Использовать эволюционный подход при разработке программного обеспечения.

3.4. Пример разработки программы.

Задание:

- Определите матрицу с размерами 12 на 12, элементы которой принимают случайные значения 0 или 1.

- Визуализируйте матрицу. Выделяя (например, курсором мыши) конкретный элемент на изображении матрицы, «закрасьте» все смежные элементы, имеющие в матрице те же значения, что и выделенный (начальный). Смежными считать все элементы, имеющие на изображении общую сторону и примыкающие к нему непосредственно и опосредовано - через промежуточные элементы с теми же значениями.

Пример результата выполнения программы приведен на рис. 11.

1	0	1	1	0	0	0	1	0	1	0	0
1	1	1	0	1	1	0	0	1	0	1	1
1	0	1	0	0	0	0	1	0	0	1	0
0	1	0	1	0	1	1	0	1	0	1	0
0	1	1	0	0	1	1	0	1	0	0	1
1	0	1	1	1	1	1	0	0	1	0	0
1	0	1	1	1	0	1	1	1	0	1	0
1	0	0	0	0	0	1	1	1	0	0	0

Рисунок 11. Кластер ячеек с нулевыми значениями

Будем называть совокупность выделенных элементов кластером. Таким образом, кластер – это совокупность элементов матрицы, имеющих общие стороны и одинаковые значения.

Анализируя постановку задачи, отмечаем, что для ее решения необходимо сконструировать средства диалога (интерфейс пользователя) и разработать алгоритмы, обеспечивающие нужную функциональность программы.

Шаг 1. *Проектируем форму*

Следуя предложенной схеме эволюционного построения программ, начнем с шага, предполагающего создание программы с минимальной функциональностью.

Как уже сказано, каждый шаг должен включать три действия: проектирование, кодирование и тестирование. Проектирование на первом шаге решения нашей задачи сводится к выбору элементов интерфейса. В качестве основы выбираем стандартное окно Windows приложения (форму с заголовком «Выделение кластеров» в новом проекте с именем ***Clusters)***.

Для размещения приглашения пользователю поместим на форму элемент ***Label***, разместив на нем текст ″ Выделите ячейку: ″.

Для изображения матрицы используем элемент ***DataGridView***.

В качестве еще одного средства диалога с пользователем введем кнопку (элемент ***Button***) с надписью «Построить кластер». Отметим, что на первом шаге не предполагается обработки события «нажатие на кнопку», поэтому добавление элемента ***Button*** можно было бы отложить на второй шаг разработки. Однако в нашей несложной задаче интерфейс очень прост и визуальное конструирование формы может быть выполнено за один шаг разработки. Итак, минимальной функциональностью создаваемого программного продукта может быть форма с тремя элементами: ***Label**, **DataGridView**, **Button*** (класс с именем Form1).

Поместив на форму указанные элементы, необходимо явно задать значения некоторых из свойств.

Form1.Text = ″Выделение кластеров″
Form1.Font.Bold = True
Form1.StartPosition = CenterScreen
Form1.Size=392;400
Form1.MaximumSize=392;400
Form1.MinimumSize=392;400
Text = ″Выделите ячейку: ″
label1.Font.Bold = True
Text = ″Построить кластер″
button1.Font.Bold = True
DataGridView1.ColumnHeadersVisible = False (убрать заголовки столбцов);
DataGridView1.RowHeadersVisible = False (убрать названия строк);
DataGridView1.AutoSizeColumnMode = Fill (″растянуть″ строки по ширине элемента).
DataGridView1.Anchor = Top, Bottom, Left, Right.

Кодирование. Для достижения минимальной функциональности достаточно поместить в код класса Form1.cs обработчик события Form1_Load.

```
usingSystem;
using System.Windows.Forms;
namespace Clusters {
public partial class Form1 : Form {
int M=12, N=12; // размеры матрицы: M - строки, N - столбцы.
public Form1() {
InitializeComponent( );
```

```
}                                    private void Form1_Load(object sender,
EventArgs e) {
dataGridView1.ColumnCount = N;
dataGridView1.RowCount = M;
dataGridView1.Rows[0].Cells[0].Selected = false; // снять выделение
}
}}
```

Тестирование на первом шаге сводится к трансляции и выполнению программы. Результат выполнения программы в конце первого шага (форма на экране) показан на рис. 12.

Рисунок 12. Результат выполнения программы на первом шаге

Шаг 2.

В проект ***Clusters*** добавим класс матриц со случайными (0 или 1) значениями элементов. Присвоим классу имя ***RandomMatrix.***

```
namespace Clusters {
class RandomMatrix {
int mm, nn; // размеры матрицы
public sbyte[,] matrix; // ссылка на массив элементов
public RandomMatrix(int m, int n) { // конструктор
mm=m; nn=n;
matrix = new sbyte[m, n];
Random gen = new Random();
for (int i = 0; i < m; i++)
for(int j = 0; j < n; j++)
matrix[i, j] = (sbyte)gen.Next(2);
} // конструктор
}}
```

В Form1 добавим поле-ссылку на объект класса ***RandomMatrix***:

```
RandomMatrix matr; // матрица со случайными элементами 0, 1.
```

В конструктор формы Form1() добавим:

```
matr = new RandomMatrix(M, N);
```

В обработчик событий Form1_Load() добавим:

```
// поместить в таблицу значения элементов матрицы:
for (int i = 0; i < M; i++)
for (int j = 0; j < N; j++);
dataGridView1[j, i].Value = // первый индекс - столбец!
matr.matrix[i, j] == 0 ? “0” : “1”;
```

В результате выполнения программы на шаге 2 получим изображение заполненной матрицы, показанное на рис. 13**:**

Рисунок 13. Результат выполнения программы на шаге 2

Шаг 3.

В проект добавим класс ***Position*** (можно было бы использовать ***System.Prawing.Point***), объект которого представляет координаты элемента матрицы и ячейки сетки:

```
namespace Clusters {
class Position {
public int x, y;
public Position(int xi, int yi) {x=xi; y = yi;}
public Position( Position p) {x = p.x; y = p.y;}
}}
```

В код формы добавим новое поле:

Position pos; *// координаты выделенной ячейки*

Усложним обработчик события button1_Click():

```
private void button1_Click(object sender, EventArgs e) {
// Ищем выделенную ячейку (K - количество выделенных):
int K=0;
for (int i = 0; i < M; i++)
for (int j = 0; j < N; j++)
if (dataGridView1.Rows[i].Cells[j].Selected == true) {
pos = new Position(j, i); K++;
}
if (K == 0) { MessageBox.Show("Нет выделенной ячейки!");
return;
}
if (K > 1) { MessageBox.Show("Выделите одну ячейку!");
for (int i = 0; i < M; i++)
for (int j = 0; j < N; j++)
dataGridView1.Rows[i].Cells[j].Selected = false;
return;
}
else
MessageBox.Show("x="+pos.x+" y=" + pos.y);
} // button1_Click()
```

При выполнении программы с функциональностью, достигнутой на шаге 3, возможны три варианта результата (рис. 14).

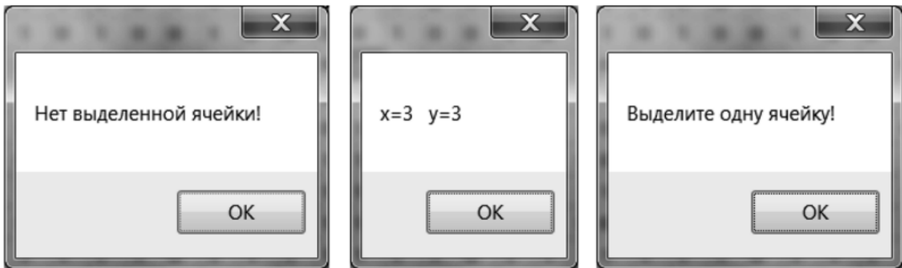


Рисунок 14. Результаты выполнения программы на шаге 3

Шаг 4.

В класс ***RandomMatrix*** добавим метод ***region()*** для формирования массива смежных (ближайших, соседних к выделённой) ячеек:

```
public Position[] region(Position p) { // ближайшие ячейки
Position [] res = null;
if (p.x > 0 & p.x < nn-1 & p.y > 0 & p.y < mm-1)
{ res = new Position[5];
res[0] = new Position(p);
```

```
res[1] = new Position(p.x+1, p.y);
res[2] = new Position(p.x, p.y-1);
res[3] = new Position(p.x-1, p.y);
res[4] = new Position(p.x, p.y+1);
}
if (p.x == 0 & p.y ==0) {res = new Position[3]
{new Position(p), new Position(p.x+1, p.y), new Position(p.x, p.y+1)};
}
if (p.x == nn-1 & p.y ==0) {res = new Position[3]
{new Position(p), new Position(p.x-1, p.y), new Position(p.x, p.y+1)};
}
if (p.x == nn-1 & p.y == mm-1) {res = new Position[3]
{new Position(p), new Position(p.x-1, p.y), new Position(p.x, p.y-1)};
}
if (p.x == 0 & p.y == mm-1) {res = new Position[3]
{new Position(p), new Position(p.x+1, p.y), new Position(p.x, p.y-1)};
}
if (p.x == 0 & p.y != 0 & p.y != mm-1) {res = new Position[4]
{new Position(p), new Position(p.x+1, p.y),
new Position(p.x, p.y+1), new Position(p.x, p.y-1)};
}
if (p.x == nn - 1 & p.y != 0 & p.y != mm-1) {res = new Position[4]
{new Position(p), new Position(p.x-1, p.y),
new Position(p.x, p.y+1), new Position(p.x, p.y-1)};
}
if (p.y == 0 & p.x != nn-1 & p.x != 0) {res = new Position[4]
{new Position(p), new Position(p.x+1, p.y),
new Position(p.x-1, p.y), new Position(p.x, p.y+1)};
}
if (p.y == mm-1 & p.x != nn-1 & p.x != 0) {res = new Position[4]
{new Position(p), new Position(p.x+1, p.y),
new Position(p.x-1, p.y), new Position(p.x, p.y-1)};
}
return res; // этот оператор будет заменён
} // region()
```

В обработчик события ***button1_Click()*** вносим дополнение:

```
else {
MessageBox.Show(“x=”+pos.x+” y=”+pos.y);
foreach (Position s in matr.region(pos))
dataGridView1.Rows[s.y].Cells[s.x].Selected = true;
}
```

После этих добавлений, нажатие на клавишу приведёт к выделению всех элементов, смежных с выбранным элементом, как показано на рис. 15.

Рисунок 15. Результаты выполнения программы на шаге 4

Шаг 5.

Дополним метод ***region()*** операторами для удаления окраски тех клеток, указывающих на элементы матрицы, значения которых отличны от помеченного элемента (рис. 16). Для этого вместо оператора *return res*, в метод ***region()*** помещаем код:

```
// Проверка найденных позиций:
int r=0;
for(int k=0; k < res.Length; k++) {
if (matrix[res[0].y, res[0].x] ==
matrix[res[k].y, res[k].x]) r++;}
Position [] temp = new Position[r];
for(int k=0, j=0; k < res.Length; k++)
if (matrix[res[0].y, res[0].x] == matrix[res[k].y, res[k].x])
temp[j++] = res[k];
return temp;
} // region()
```

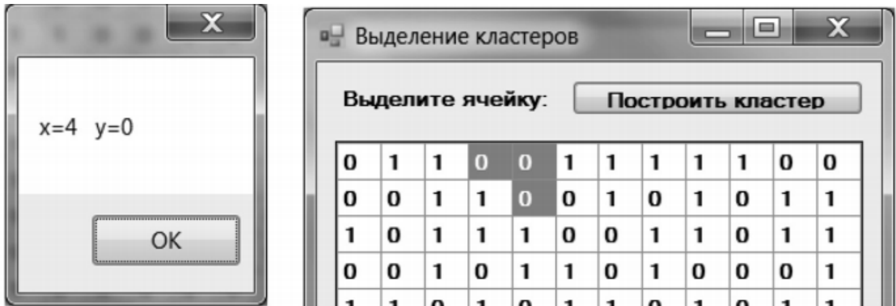


Рисунок 16. Результаты выполнения программы на шаге 5

Шаг 6.

Теперь можно дополнять программу средствами для выделения кластера, то есть построения и окрашивания всей совокупности одинаковых смежных ячеек, прилегающих прямо и косвенно (через соседей) к начальной (выделенной) ячейке.

На этом шаге придется заниматься разработкой алгоритма построения кластера. Для этого необходимо проанализировать каждого из соседей начального элемента. Так как у каждого из анализируемых элементов в свою очередь могут оказаться подходящие смежные элементы и т.д., то размер кластера заранее предсказать нельзя. Выбор элементов, принадлежащих кластеру, это типичная комбинаторная задача, в общем виде сводящаяся к требованию: «выделите все такие, и только такие элементы множества, которые удовлетворяют заданному условию».

Применим для решения нашей задачи один из вариантов основного метода комбинаторных вычислений – метода ветвей и границ, который еще называют методом перебором с возвратом. Основных подходов к реализации метода ветвей и границ два: 1) применение стека, сохраняющего выбранные, но еще не включенные в результат элементы, и 2) применение рекурсивного алгоритма. Остановимся на рекурсивном подходе. Пусть ***P*** – позиция, определяющая элемент матрицы, для которого должно быть выделено множество соседей, удовлетворяющих требованию принадлежности кластеру. Пусть ***R*** – множество позиций, уже включенных в кластер.

Пусть ***T*** – множество соседей элемента ***P***, удовлетворяющих требованию принадлежности кластеру, но еще не включенных в ***R*** и в предшествующие множества ***T***.

Пусть ***F(P, R)*** – рекурсивная процедура построения кластера ***R*** из позиции ***P***.

Псевдокод процедуры можно представить так:

F(P,R) {
Поместить позицию P в кластер R
Определить T(P) – множество соседей элемента P
Пометить P и все Pi из T как уже выбранные
Цикл по всем Pi из T
Вызов F(Pi, R)
Выход из F()
}

До обращения к процедуре ***P*** – задано, ***R*** – пусто. После выхода из процедуры ***F(P,R)*** в ***R*** – множество позиций, включенных в кластер. Процедура имеет одну особенность – в ней должна быть предусмотрена защита от повторного включения в очередное множество соседей уже выбранных ранее элементов. Поэтому при определении ***T (P)*** нужно выбирать во множество соседей элементы, соответствующие кластеру, но еще не включенные в рассмотрение (не помеченные ранее).

Для выбора множества соседей, пригодных для включения в кластер, хотелось бы применить разработанный нами метод ***RandomMatrix.region().*** Однако в нем не предусмотрена оценка «помеченности» элементов.

Помечать уже использованные элементы можно разными способами. С целью максимального использования без каких-либо изменений уже закодированных фрагментов нашей программы помечать выбранные элементы будем, изменяя значения элементов матрицы, на которой решается наша задача. Тем самым появляется возможность применить для получения множества ***T(P)*** метод ***RandomMatrix.region().*** По условию задачи элементы матрицы имеют значения 0 и 1, метод ***region()*** выбирает соседние элементы, имеющие то же значение, что и начальный. Если изменить значения уже использованных элементов матрицы, то они не будут считаться подходящими для кластера, и метод ***region()*** «проигнорирует» их присутствие по соседству с начальным элементом.

Техническое решение: будем помечать рассмотренный элемент, увеличивая соответствующее значение элемента матрицы на 2. Не останавливаясь на других деталях программной реализации алгоритма, включим в класс ***RandomMatrix*** следующий рекурсивный метод для построения кластера:

```
public void spot(Position p, ref Position[] res) {
int size = res.Length;
Position [] temp = res;
res = new Position [size+1];
temp.CopyTo(res,0);
res[size] = p;
Position [] path = region(p);
foreach (Position g in path)
matrix[g.y, g.x] += 2; // Помечаем выделенные элементы
for (int i=1; i<path.Length; i++) {
matrix[path[i].y, path[i].x] -= 2;
spot(path[i], ref res);
}
return;
} // spot()
```

Как сказано, в методе ***spot()*** уже рассмотренные элементы специальным образом помечаются. Для этого соответствующие элементы матрицы увеличиваются на 2. После этого они становятся «не похожими» на подключаемые к кластеру элементы. Тем самым исключается дублирование и возможное зацикливание. Однако для продолжения поиска новый начальный элемент приводится в исходное состояние (элемент матрицы уменьшается на 2). Принудительное «окрашивание» элементов, уже включенных в кластер, требует их восстановления для возможности повторных построений того же кластера (возможно из другого начального элемента). Это приходится явно выполнять в коде. Таким образом, в обработчик события button1_Click() вносим следующие изменения:

```
else { MessageBox.Show("x="+pos.x+" y="+pos.y);
Position [] cla = new Position[0];
matr.spot(pos, ref cla);
```

```
foreach (Position s in cla) {
dataGridView1.Rows[s.y].Cells[s.x].Selected = true;
matr.matrix[s.y, s.x] -=2;
} // foreach
} // else
```

Этот код должен заменить следующий, исключаемый из метода ***button1.Click(),*** фрагмент:

```
else {
MessageBox.Show("x="+pos.x+" y="+pos.y);
foreach (Position s in matr.region(pos))
dataGridView1.Rows[s.y].Cells[s.x].Selected = true;
}
```

Самое главное в новом коде – замена обращения к методу ***matr.region(pos)*** на вызов рекурсивного метода ***matr.spot(pos, ref cla).*** Различия между этими методами и пометка («окрашивание») уже рассмотренных элементов определили особенности нового заменяющего кода.

Пример результаты работы приложения представлен на рис. 17.

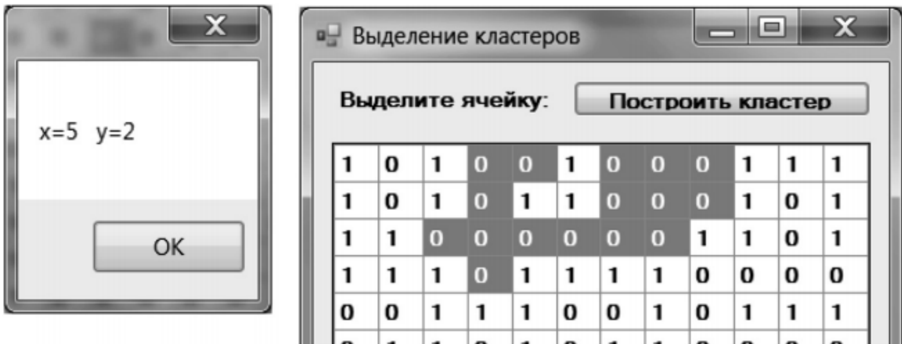


Рисунок 17. Результаты выполнения программы на шаге 6

3.5. Контрольные вопросы.

1.Жизненный цикл программного средства. Фазы жизненного цикла.
2.Модель жизненного цикла программных средств.
3.Каскадная модель процесса разработки
4. Инкрементная модель процесса разработки
5. RAD-модель процесса разработки
6.Спиральная модель процесса разработки
7.Прогностические и адаптивные процессы разработки программных средств.
8.Методология экстремального программирования.
9. Эволюционный подход к разработке программного обеспечения.

4 ЛАБОРАТОРНАЯ РАБОТА № 4

Визуальное моделирование в UML. Диаграмма вариантов использования на первом этапе разработки программного обеспечения.

4.1 Цель лабораторной работы

Целью лабораторной работы является:

- Получение навыков разработки концептуального представления предметной области и концептуальной модели системы в процессе ее проектирования с использование языка моделирования UML и исходной диаграммы Use Case.

- Получить навыки разработки программных продуктов с использованием Rational Software Architect (RSA), комплексного решения для проектирования, моделирования и разработки программного обеспечения.

4.2 Теоретические сведения

Язык UML (Unified Modeling Language,) представляет собой универсальный язык визуального моделирования, который разработан для спецификации, визуализации, проектирования и документирования компонентов программного обеспечения, бизнес-процессов и других систем.

Язык UML одновременно является простым и мощным средством моделирования, который может быть эффективно использован для построения концептуальных, логических и графических моделей сложных систем самого различного целевого назначения. Этот язык вобрал в себя наилучшие качества методов программной инженерии, которые с успехом использовались на протяжении последних лет при моделировании больших и сложных систем [6].

В рамках языка UML все представления о модели сложной системы фиксируются в виде специальных графических конструкций, получивших название диаграмм. В терминах языка UML определены следующие виды диаграмм: структурные диаграммы, диаграммы поведения, диаграммы взаимодействия.

Визуальное моделирование в UML можно представить, как некоторый процесс спуска от наиболее обшей и абстрактной концептуальной модели исходной системы к логической, а затем и к физической модели соответствующей программной системы [7]. Для достижения этих целей вначале строится модель в форме, так называемой ***диаграммы вариантов использования*** (диаграммы прецедентов, **use case diagram**), которая описывает функциональное назначение системы или, другими словами, то, что система будет делать в процессе своего функционирования. Диаграмма вариантов использования является исходным концептуальным представлением или концептуальной моделью системы в процессе ее проектирования и разработки.

Назначение диаграммы вариантов использования:

- Определить общие границы и контекст моделируемой предметной области на начальных этапах проектирования системы.

- Сформулировать общие требования к функциональному поведению проектируемой системы.

- Разработать исходную концептуальную модель системы для ее последующей детализации в форме логических и физических моделей.

- Подготовить исходную документацию для взаимодействия разработчиков системы с ее заказчиками и пользователями.

Суть данной диаграммы состоит в следующем: проектируемая система представляется в виде множества сущностей или актеров, взаимодействующих с системой с помощью, так называемых вариантов использования.

При этом ***актером (actor)*** или действующим лицом называется любая сущность, взаимодействующая с системой извне. Это может быть человек, техническое устройство, программа или любая другая система, которая может служить источником воздействия на моделируемую систему так, как определит сам разработчик. В свою очередь, ***вариант использования (use case)*** служит для описания сервисов, которые система предоставляет актеру.

Другими словами, каждый вариант использования определяет некоторый набор действий, совершаемый системой при ее взаимодействии с соответствующим актером. При этом ничего не говорится о том, каким образом будет реализовано взаимодействие актеров с системой.

Базовые элементы этого пакета — ***вариант использования (прецедент) и актер.***

Вариант использования

Конструкция или стандартный элемент языка UML *вариант использования* применяется для спецификации общих особенностей поведения системы или любой другой сущности предметной области без рассмотрения внутренней структуры этой сущности. Каждый вариант использования определяет последовательность действий, которые должны быть выполнены проектируемой системой при взаимодействии ее с соответствующим актером. Диаграмма вариантов может дополняться пояснительным текстом, который раскрывает смысл или семантику составляющих ее компонентов. Такой пояснительный текст получил название ***примечания*** или ***сценария***.

Отдельный вариант использования обозначается на диаграмме эллипсом, внутри которого содержится его краткое название или имя в форме глагола с пояснительными словами (рис.18)

Проверить состояние текущего счета

Рисунок 18. Графическое изображение варианта использования

Цель варианта использования заключается в том, чтобы определить законченный аспект или фрагмент поведения некоторой сущности В качестве такой сущности может выступать исходная система или любой другой элемент модели, который обладает собственным поведением, подобно подсистеме или классу в модели системы.

Варианты использования описывают не только взаимодействия между пользователями и сущностью, но также реакции сущности на получение отдельных сообщений от пользователей и восприятие этих сообщений за пределами сущности.

Множество вариантов использования в целом должно определять все возможные стороны ожидаемого поведения системы.

Примерами вариантов использования могут являться следующие действия: проверка состояния текущего счета клиента, оформление заказа на покупку товара, получение дополнительной информации о кредитоспособности клиента, отображение графической формы на экране монитора и другие действия.

Каждый выполняемый вариантом использования метод реализуется как неделимая транзакция, то есть выполнение сервиса не может быть прервано никаким другим экземпляром варианта использования.

Актеры

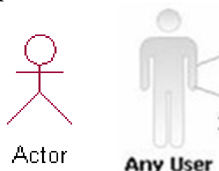
Актер представляет собой любую *внешнюю по отношению к моделируемой системе сущность*, которая взаимодействует с системой и использует ее функциональные возможности для достижения определенных целей или решения частных задач.

Примерами актеров могут быть: клиент банка, банковский служащий, продавец магазина, менеджер отдела продаж, пассажир авиарейса, водитель автомобиля, администратор гостиницы, сотовый телефон и другие сущности, имеющие отношение к концептуальной модели соответствующей предметной области.

В качестве актеров могут выступать другие системы, подсистемы проектируемой системы или отдельные классы.

Актер определяет некоторое согласованное множество ролей, в которых могут выступать пользователи данной системы в процессе взаимодействия с ней. В каждый момент времени с системой взаимодействует вполне определенный пользователь, при этом он выступает в одной из таких ролей.

Стандартным графическим обозначением актера на диаграммах является фигурка "человечка", под которой записывается имя актера:



Отношения на диаграмме вариантов использования

Между компонентами диаграммы вариантов использования могут существовать различные отношения.

Один актер может взаимодействовать с несколькими вариантами использования. В этом случае этот актер обращается к нескольким сервисам данной системы. В свою очередь один вариант использования может взаимодействовать с несколькими актерами, предоставляя для всех них свой сервис.

Варианты использования могут взаимодействовать друг с другом, если они определены для разных сущностей. И актеры могут взаимодействовать друг с другом

В языке UML имеется несколько стандартных видов отношений между актерами и вариантами использования:

- отношение ассоциации (association relationship),
- отношение расширения (extend relationship),
- отношение обобщения (generalization relationship),
- отношение включения (include relationship).

Отношение ассоциации

Отношение ассоциации является одним из фундаментальных понятий в языке UML и в той или иной степени используется при построении всех графических моделей систем в форме канонических диаграмм. Применительно к диаграммам вариантов использования оно служит для обозначения специфической роли актера в отдельном варианте использования. Другими словами, ассоциация специфицирует семантические особенности взаимодействия актеров и вариантов использования в графической модели системы. Таким образом, это отношение устанавливает, какую конкретную роль играет актер при взаимодействии с экземпляром варианта использования. На диаграмме вариантов использования, так же как и на других диаграммах, отношение ассоциации обозначается сплошной линией между актером и вариантом использования. Эта линия может иметь дополнительные условные обозначения, такие, например, как имя и кратность (рис. 19).

Рисунок 19. Пример графического представления отношения ассоциации между актером и вариантом использования

Кратность (multiplicity) ассоциации указывается рядом с обозначением компонента диаграммы, который является участником данной ассоциации. Кратность характеризует общее количество конкретных экземпляров данного компонента, которые могут выступать в качестве элементов данной ассоциации.

Для диаграмм вариантов использования наиболее распространенными являются четыре основные формы записи кратности отношения ассоциации:

- Целое неотрицательное число (включая цифру 0). Примером этой формы записи кратности ассоциации является указатель кратности "1" для актера "Клиент банка" (рис.19).

- Два целых неотрицательных числа, разделенные двумя точками и записанные в виде: *"первое число .. второе число"*. Пример такой формы записи кратности ассоциации - *"1..5"*..

- Два символа, разделенные двумя точками. При этом первый из них является целым неотрицательным числом или 0, а второй — специальным символом "*". Здесь символ "*" обозначает произвольное конечное целое

неотрицательное число, значение которого неизвестно на момент задания соответствующего отношения ассоциации.

- Единственный символ "*", который является сокращением записи интервала *"0..*"*. В этом случае количество отдельных экземпляров данного компонента отношения ассоциации может быть любым целым неотрицательным числом. При этом 0 означает, что для некоторых экземпляров соответствующего компонента данное отношение ассоциации может вовсе не иметь места. Если кратность отношения ассоциации не указана, то по умолчанию принимается ее значение, равное 1.

Отношение расширения

Отношение расширения отражает ***возможное*** присоединение одного варианта использования к другому в некоторой точке (*точке расширения*). При этом подчеркивается то, что расширяющий вариант использования выполняется лишь при определенных условиях и не является обязательным для выполнения основного прецедента. На диаграмме такой вид отношения изображается пунктирной стрелкой, направленной к расширяемому прецеденту, в отдельном разделе которого *может быть описана* ***точка расширения***, а ***условия расширения*** *могут быть приведены* в комментарии. Таким образом, расширение позволяет моделировать *необязательное поведение системы*, которое является *условным* и *не изменяет поведение* основного прецедента (рис 20).

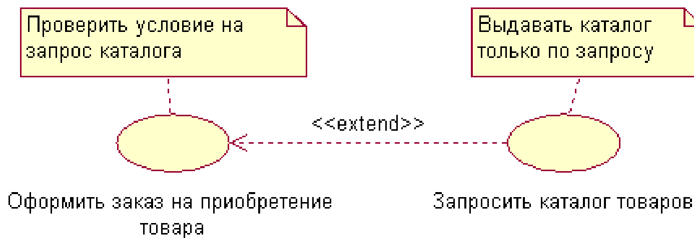


Рисунок 20. Отношение расширения варианта использования

Отношение включения

Отношение включения указывает на то, что поведение одного прецедента включается в некоторой точке в другой прецедент в качестве составного компонента. Особенности включения заключаются в том, что включаемый прецедент должен быть обязательным для дополняемого (включение должно быть безусловным, а дополняемый вариант использования без включения не сможет выполняться), то есть это отношение задает очень сильную связь. Графически данное отношение обозначается пунктирной линией со стрелкой (вариант отношения зависимости), направленной от базового варианта использования к включаемому варианту. При этом данная линия со стрелкой помечается ключевым словом «include» («включает») (рис. 21).

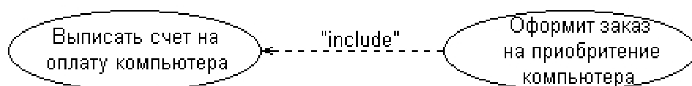


Рисунок 21. Отношения включения варианта использования

Отношение обобщения

Отношение обобщения служит для указания того факта, что некоторый вариант использования *A* может быть обобщен до варианта использования *B*. В этом случае вариант *A* будет являться специализацией варианта *B*. При этом *B* называется ***предком*** по отношению *A,* а вариант *A* - ***потомком*** по отношению к *B*. Следует подчеркнуть, что потомок наследует все свойства и поведение своего родителя, а также может быть дополнен новыми свойствами и особенностями поведения. Графически данное отношение обозначается сплошной линией со стрелкой в форме незакрашенного треугольника, которая указывает на родительский вариант использования (рис. 22). Эта линия со стрелкой имеет специальное название — *стрелка "обобщение".*

Рисунок 22. Отношения обобщения между вариантами использования

Между отдельными актерами также может существовать отношение обобщения (рис.23).

.

Рисунок 23. Отношение обобщения между актерами.

Пример диаграммы вариантов использования

В качестве примера рассмотрим процесс моделирования системы продажи товаров по каталогу, которая может быть использована при создании соответствующих информационных систем.

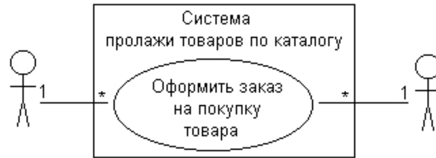
В качестве актеров данной системы могут выступать два субъекта, один из которых является продавцом, а другой — покупателем. Каждый из этих актеров взаимодействует с рассматриваемой системой продажи товаров по каталогу и является ее пользователем, т. е. они оба обращаются к соответствующему сервису "Оформить заказ на покупку товара".

Значения указанных на данной диаграмме кратностей отражают общие правила оформления заказов на покупку товаров:

- *Один продавец может участвовать в оформлении нескольких заказов*, в то же время каждый заказ может быть оформлен только одним продавцом, *который несет ответственность за корректность его оформления.*

- *С другой стороны, каждый покупатель может оформлять на себя несколько заказов, но, в то же время, каждый заказ должен быть оформлен на единственного покупателя, к которому переходят права собственности на товар после его оплаты.*

Первоначальная структура диаграммы может включать в себя только двух указанных актеров и единственный вариант использования:



На следующем этапе разработки данной диаграммы вариант использования "Оформить заказ на покупку товара" может быть уточнен на основе введения в рассмотрение четырех дополнительных вариантов использования.

Анализа процесса продажи товаров позволяет выделить в качестве отдельных сервисов такие действия:

- *как обеспечить покупателя информацией о товаре,*
- *согласовать условия оплаты товара и*
- *заказать товар со склада.*

С другой стороны, продажа товаров по каталогу предполагает наличие объекта — каталога товаров, который в некотором смысле не зависит от реализации сервиса по обслуживанию покупателей. Вполне резонно представить сервис "Запросить каталог товаров" в качестве самостоятельного расширяющего варианта использования (рис.24).

Рисунок 24. Дополнение сервисов

В рамках общей парадигмы ООАП дальнейшая детализация модели может выполняться в двух направлениях. В рамках системы продажи товаров может иметь самостоятельное значение и специфические особенности отдельная категория товаров — компьютеры.

В этом случае диаграмма может быть дополнена вариантом использования "Оформить заказ на покупку компьютера" и актерами "Покупатель компьютера" и "Продавец компьютеров", которые связаны с соответствующими компонентами диаграммы отношением обобщения (рис. 25).

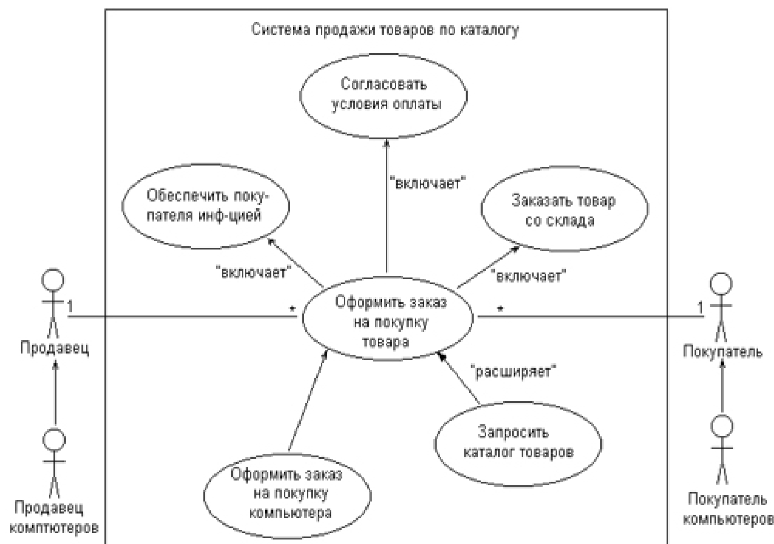


Рисунок 25. Диаграмма вариантов использования системы продажи товаров по каталогу

Разработка диаграммы вариантов использования в среде Rational Software Architect (RSA)

Rational Software Architect - это среда моделирования и разработки, которая использует *унифицированный* язык моделирования (UML) для проектирования архитектуры для приложений C++ и Java EE (JEE) Rational Software Architect построен на платформе Eclipse с открытым исходным кодом и включает в себя возможности, ориентированные на архитектурный анализ кода.

Проектирование приложения телефонной книги

В данном примере проектируется довольно простое приложение телефонной книги, в котором хранятся введенные пользователями номера телефонов.

Запуск Rational Software Architect

1. Запустите Rational Software Architect: В меню Windows выберите **Пуск** > **Программы** > **IBM Rational** > **IBM Rational Software Architect v9.0** > **Rational Software Architect**;
2. Открывается диалоговое окно с запросом на папку рабочего пространства. Нажмём “OK” для выбора настроек по умолчанию.

Создание UML-проекта

Создайте UML-проект с именем MyPhoneBookUMLProject:

1. В меню модуля выберите **File** > **New** > **Project > Other**;
2. Выберите **UML Project** и нажмите **Next**;
3. Введите MyPhoneBookUMLProject в качестве имени проекта и нажмите **Next**;
4. Задайте имя файла Phone Book UML ModelUML-модели, снимите флажок **Create a default diagram in the new model** и нажмите **Finish**.

Создание диаграммы прецедентов

Диаграмма прецедентов моделирует поведение системы и позволяет зарегистрировать требования. Диаграмма определяет взаимодействия между системой и ее действующими лицами и определяет область действия системы.

Действующее лицо

Представляет роль пользователя, взаимодействующего с системой. Пользователем может быть человек, организация, компьютер или другая внешняя система.

Прецедент.

Описывает функцию, которую выполняет система для достижения цели пользователя. Прецедент должен возвращать видимый результат, имеющий значение для пользователя системы.

Показанные в диаграмме прецеденты и действующие лица описывают, что делает система, и как это используют действующие лица, а не внутренний процесс работы системы. Чтобы связать действующее лицо и прецедент, для указания связи между двумя элементами модели можно создать отношение.

Предположим, что для приложения телефонной книги имеется только одно действующее лицо **Any User**, которое может выполнять следующие два прецедента относительно системы:

Добавление записи

Ввод уникального имени абонента и номера телефона с помощью пользовательского интерфейса, предоставляемого приложением. Система обрабатывает введенные данные и сохраняет их.

Поиск номера телефона

Получение номера телефона по вводу уникального имени абонента с помощью пользовательского интерфейса, предоставляемого приложением. Система находит номер телефона и возвращает его действующему лицу.

Создание диаграммы прецедентов со списком двух прецедентов:

1. В панели Model Explorer нажмите правой кнопкой мыши **Phone Book UML Model** и выберите **Add Diagram**> **Use Case Diagram** (рис.26);
2. Введите User Case Diagram в качестве имени сгенерированной диаграммы, заменив имя по умолчанию Diagram1. Теперь можно построить диаграмму прецедентов с помощью добавления с панели Palette на диаграмму различных элементов моделей (рис.27);

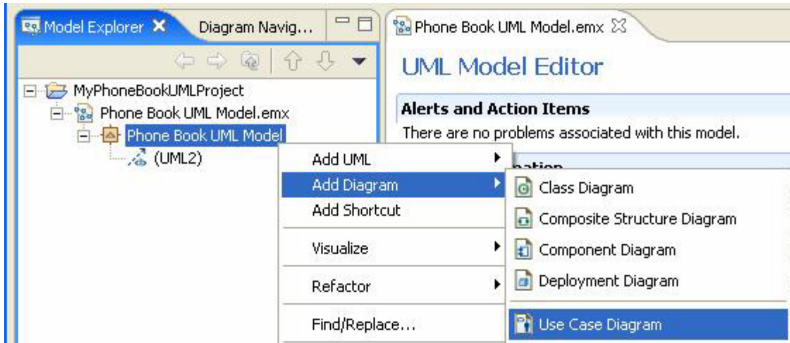


Рисунок 26. Добавление диаграммы прецедентов

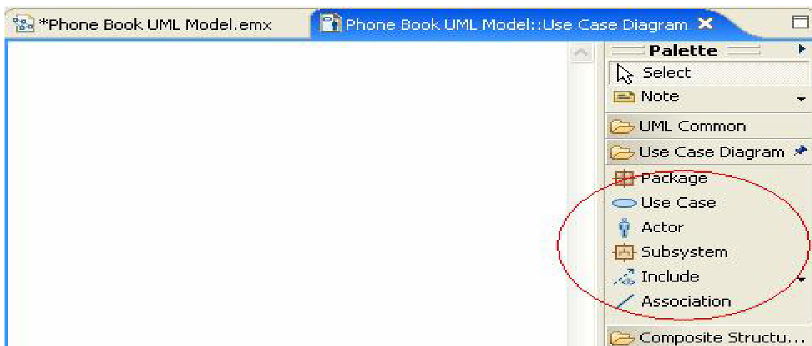


Рисунок 27. Добавление элементов модели

3. Выберите **Actor** в панели Palette, затем нажмите кнопку мыши в области диаграммы для создания действующего лица. Назовите его **Any User**;
4. Выберите **Use Case** в панели Palette, затем нажмите кнопку мыши в области диаграммы для создания прецедента. Назовите его **Add an entry**;
5. Таким же образом создайте другой прецедент и назовите его **Search for a phone number**;

6. Выберите **Association** в панели Palette. Начертите линию отношения от действующего лица **Any User** к прецеденту **Add an entry** для создания отношения между двумя элементами модели;
7. Таким же образом создайте другое отношение между действующим лицом **Any User** и прецедентом **Search for a phone number**;
8. Полностью диаграмма прецедентов должна выглядеть так, как показано на рис. 28. Нажмите **Ctrl-S** для сохранения диаграммы.

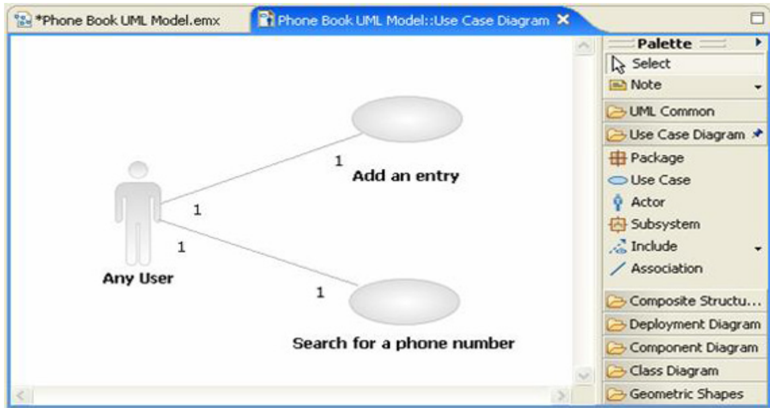


Рисунок 28. Созданная диаграмма прецедентов

Публикация проекта

Публикуя информацию о модели, ее можно совместно использовать с другими пользователями, не имеющими средства моделирования. Rational Software Architect поддерживает две функции публикации:

- Публикация моделей на Web-странице
- Публикация отчета информации о модели

Публикация проекта на Web-страницу:

1. Выберите **Phone Book UML Model** в панели Model Explorer. Выберите **Modeling** > **Publish** > **Web** (рис. 29);

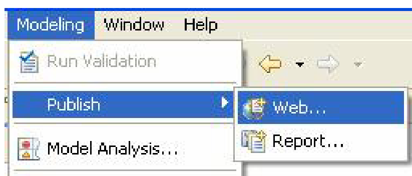


Рисунок 29. Публикация проекта на Web-странице

2. Укажите целевое местоположение создаваемых HTML-файлов, например, C:\HelloWorldSeries\RSA_Web в меню (Download), затем нажмите **OK**.

Модель публикуется в виде HTML-файлов, записываемых в указанное местоположение;

3. Откройте в Web-браузере файл C:\HelloWorldSeries\RSA_Web\index.html;

Рисунок 29. Опубликованная Web-страница

4. Нажмите ссылку **Phone Book UML Model**;
5. Просмотрите опубликованную модель, нажимая ссылки элементов и диаграммы.

4.3 Задание на выполнение лабораторной работы

Разработать диаграмму вариантов использования в соответствие с вариантом.

4.4 Контрольные вопросы

1 Какое назначение UML. Свойства. Основные элементы UML

2 Методология Rational Unified Process и ее содержание.

3 Диаграммы UML и их виды.

4 Основная идея - моделировать системы как наборы взаимодействующих объектов.

5 Сущности, отношения, диаграммы.

6 Диаграмма вариантов использования. Назначение. Базовые элементы. Отношения на диаграмме вариантов использования.

7 Виды ассоциаций. Кратность ассоциации

4.5 Варианты заданий лабораторной работы

1. <u>Ремонтная мастерская.</u> Основной вариант использования - заказ на ремонт. Несколько вариантов - включаемых, например, согласовать стоимость и расширяемых. Вариант – специализация ремонта (отношение - обобщение). Роли: клиент, приемщик, мастер, склад.

2. Университет. Основной вариант использования – услуга по получению образования. Несколько вариантов, включаемых, и расширяемых. Вариант – специализация обучения – обобщение Роли: студент, преподаватель, деканат, кафедра.

3. Телефонная станция. Вариант использования – услуга по обслуживанию. Роли: клиент, агент, почта. Несколько вариантов, включаемых, например, согласовать вид обслуживания и расширяемых. Вариант – специализации услуг (отношение - обобщение).

4. Агентство мобильной связи. Вариант использования – услуга по обслуживанию. Несколько вариантов, включаемых, например, предоставить информацию и расширяемых. вариант – специализации обслуживания – обобщение. Роли: клиент, терминал, оператор сотовой связи, агент, склад.

5. Образовательные платные курсы. Вариант использования – услуга по получению курса. Несколько вариантов, включаемых и расширяемых. Вариант – специализация обучения (отношение обобщения). Роли: клиент, агент, финансовый отдел, преподаватель.

6. Библиотека. Вариант использования - заказ на получения литературы. Несколько вариантов, включаемых, например, дать консультацию, расширяемых и вариант – специализация тематики (обобщение). Роли: клиент, библиотекарь, склад.

7. Склад. Вариант использования - заказ товаров. Несколько вариантов, включаемых, например, дать консультацию и расширяемых. Вариант – специализация склада (связь - обобщение). Роли: клиент, менеджер, кладовщик.

8. Спорт комплекс. Вариант использования - получение услуги тренировок. Роли: клиент, менеджер, тренер. Несколько вариантов, включаемых, например, дать консультацию, расширяемых и вариант – специализация тренировки (обобщение).

9. Турагентство. Вариант использования - выбор и бронирование оптимального тура. Роли: клиент, менеджер. Несколько вариантов, включаемых, например, дать консультацию, расширяемых и вариант – специализация поездок (обобщение).

10. Массажный салон. Вариант использования - получение услуги. Роли: клиент, менеджер, массажист. Несколько вариантов, включаемых, например, дать консультацию, расширяемых. Вариант – специализация (отношение вариантов - обобщение).

11. Мастерская по ремонту техники. Вариант использования - Заказ. Роли: клиент, приемщик, мастер. Несколько вариантов, включаемых, например, дать консультацию, и расширяемых. Вариант – специализация ремонта (отношение вариантов - обобщение).

12. Книжный магазин
Вариант использования - заказ литературы. Несколько вариантов, включаемых, например, дать консультацию, расширяемых. Вариант – специализация (отношение вариантов - обобщение
Роли: клиент, продавец, склад.

5. СПИСОК ЛИТЕРАТУРЫ

1. Мартин Р.С., Мартин М. Принципы, паттерны и методики гибкой разработки на языке C#. .- Москва: "Символ-Плюс", 2011.
2. Мартин Р. Чистый код: создание, анализ и рефакторинг. – Санкт Петербург: "Питер", 2016.
3. Подбельский В.В. Язык C#. Базовый курс, М: Финансы и статистика, 2015.
4. Вайнейкис Л.А. Технология программирования: Учеб. пособие. - М.: МГТУ ГА, 2008.
5. Брауде Э. Дж. Технология разработки программного обеспечения. СПб.: Питер, 2004
6. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++. Второе издание. М.: Бином, СПб.: Невский диалект, 2000
7. Д. Леффингуэлл, Д. Уидриг. Принципы работы с требованиями к программному обеспечению. Унифицированный подход. М.: Вильямс, 2002.

СОДЕРЖАНИЕ

1 Лабораторная работа № 1

Разработка программ на C# с использованием интерфейсов для снижения сложности

1.1 Цель лабораторной работы
1.2 Теоретические сведения
1.3 Задание на выполнение лабораторной работы
1.4 Порядок выполнения работы
1.5 Контрольные вопросы

2 Лабораторная работа № 2

Разработка программ на C# с графическим интерфейсом пользователя.

2.1 Цель лабораторной работы
2.2 Теоретические сведения
2.3 Задание на выполнение лабораторной работы
2.4 Пример выполнения лабораторной работы
2.5 Контрольные вопросы
2.6 Варианты заданий лабораторной работы

3 Лабораторная работа № 3

Эволюционный подход к разработке программ

3.1. Цель лабораторной работы
3.2. Теоретические сведения
3.3. Задание на выполнение лабораторной работы
3.4. Порядок выполнения работы
3.5. Пример выполнения лабораторной работы
3.6. Контрольные вопросы

4 Лабораторная работа № 4

Визуальное моделирование в UML. Диаграмма вариантов использования на первом этапе разработки программного обеспечения.

4.1. Цель лабораторной работы
4.2. Теоретические сведения
4.3. Задание на выполнение лабораторной работы
4.4. Контрольные вопросы
4.5. Варианты заданий лабораторной работы.

5 СПИСОК ЛИТЕРАТУРЫ