МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФГАОУ ВО «Севастопольский государственный университет»
Институт радиоэлектроники и информационной безопасности

УЧЕБНО-МЕТОДИЧЕСКОЕ ПОСОБИЕ

практикум по дисциплине

«Визуальное проектирование программ в среде Microsoft Visual Studio»

для студентов дневной и заочной форм обучения направлений подготовки
11.03.04 «Электроника и наноэлектроника»

Часть 2

Севастополь
2020 г.

УДК 004.4'236

Учебно-методическое пособие «Практикум по дисциплине «Визуальное проектирование программ в среде Microsoft Visual Studio» Часть 2 для студентов дневной и заочной форм обучения по направлению подготовки 11.03.04 «Электроника и наноэлектроника» / СевГУ; Сост. к.т.н. Д.Г. Мурзин – Севастополь: Изд-во СевГУ, 2020. – 132 с.

Целью учебно-методического пособия является оказание помощи студентам очной формы обучения в подготовке и выполнении практикума по дисциплине «Визуальное проектирование программ в среде Microsoft Visual Studio».

Рассмотрены и утверждены на заседании кафедры электронной техники (протокол № 1 от 30 января 2020 г.)

Рецензенты: к.т.н., доцент кафедры электронной техники Шевченко Н.В.

Ответственный за выпуск: заведующий кафедрой «Электронная техника», к.т.н., доцент Михайлюк Ю.П.

СОДЕРЖАНИЕ

ТРЕБОВАНИЯ К ОФОРМЛЕНИЮ ОТЧЕТА

Отчет о выполнении практической работы оформляется на стандартных листах формата А4. Расположение листов – вертикальное. Поля: левое 2 см, верхнее 2 см, нижнее 1 см, правое 1 см. Отчет следует оформлять в редакторе Microsoft Word.

Отчет должен включать в себя титульный лист с указанием номера и темы практической работы; цель работы и индивидуальное задание; блок-схему алгоритма программы с комментариями; анализ работы алгоритма и выводы (полученные результаты).

Блок-схема выполняется средствами редактора MS Visio и должна отражать весь алгоритм работы программы с комментариями. Если программа состоит из нескольких модулей, то для каждого модуля составляется отдельная блок-схема. В случае если в алгоритме определены объекты, то должна быть представлена структура каждого объекта.

При оформлении блок-схемы алгоритма программы следует придерживаться следующих правил:

1) элементы блок-схемы изображаются согласно ГОСТ;

2) линии изображаются со стрелками, если они (или какая-либо их часть) направлены вверх или влево;

3) пересечения линий не допускаются. В случае, когда возникает ситуация с пересечением линий, следует воспользоваться элементами «узел разрыва линий»;

4) линии должны соединяться с фигурами только сверху или снизу. Исключение составляет блок «Решение». Для него разрешается ответвление линий в любую сторону;

5) все надписи выполняются шрифтом одного размера; если надписи выполняются «от руки», то используют чертежный шрифт подходящего размера.

ВВЕДЕНИЕ

1. Предпосылки появления объектно-ориентированной парадигмы[1] программирования

Одной из основных причин поиска новых подходов к проектированию и реализации программ является их сложность:

1. Сложность объектов предметной области решаемой задачи, с которыми приходится иметь дело как при структурном описании на этапе проектирования системы, так и на этапе программной реализации;

2. Сложность управления процессом разработки программного обеспечения, в том числе и при коллективной разработке, требуют эффективной декомпозиции решаемой задачи с точки зрения скорости реализации конкретным конечным программистом.

3. Дискретная природа программных систем приводит к тому, что встает задача такого описания программных объектов, которое предотвратит взаимное влияние слабо связанных объектов друг на друга.

4. Большой объем требуемой для описания программной системы документации ставит задачу автоматизации документирования, в том числе и самодокументирование кода, без ущерба к пониманию сути решения задачи.

Для решения описанных выше проблем в области программирования были предложены различные парадигмы разработки программ. Применительно к программированию парадигмой называют устоявшуюся систему подходов к написанию компь-

[1] Парадигма - (от греч. paradeigma — пример, образец) — совокупность теоретических и методологических положений, принятых научным сообществом на известном этапе развития науки и используемых в качестве образца, модели, стандарта для научного исследования, интерпретации, оценки и систематизации научных данных, для осмысления гипотез и решения задач, возникающих в процессе научного познания.

ютерных программ. На сегодняшний день наиболее популярными являются следующие парадигмы: императивная[2], структурная[3], функциональная[4], логическая[5] и объектно-ориентированная. На практике все современные системы программирования используют несколько парадигм (концепция мультипарадигменного программирования).

2. Объектно-ориентированное программирование

Идея объектно-ориентированного подхода возникла как попытка обобщения образа восприятия человеком окружающей действительности. Процедурный и структурный подход предполагает указания того, что и в какой последовательности необходимо сделать для достижения цели. Альтернативным подходом является разбиение задачи на сущности, с которыми при ее решении приходится иметь дело программисту. Эти сущности получили название объектов, и именно вокруг их описания строится объектно-ориентированная система.

Объектно-ориентированное программирование (ООП) — парадигма программирования, в которой основными концепциями являются понятия объектов и классов.

---

[2] Императивное (процедурное) программирование является отражением архитектуры традиционных ЭВМ, предложенной фон Нейманом в 40-х годах. Программа состоит из последовательности операторов и предложений, управляющих последовательностью их выполнения.

[3] Структурный подход заключается в декомпозиции (разбиении) задачи на функциональные подсистемы, которые в свою очередь делятся на подфункции, подразделяемые на задачи и подзадачи.

[4] Программы, написанные в функциональном стиле, определяют математическую связь входных параметров и результата вычислений в виде каскадной цепочки вызовов функций.

[5] В качестве математического базиса языки логического программирования используют аксиоматическую систему исчисления предикатов первого порядка для описания предметной области и осуществления резолюционного логического вывода.

Объекты обладают поведением, состоянием, свойствами, которые в программе реализуются в виде функций и полей данных. Таким образом, объектно-ориентированная технология включает в себя возможности структурного подхода, но объектно-ориентированное проектирование в большей степени реализует модель реального мира и соответствует естественной логике человеческого мышления.

По мнению автора языка C++, Бьерна Страуструпа, различие между процедурным и объектно-ориентированным стилями программирования заключается в следующем: программа на процедурном языке отражает "способ мышления" процессора, а на объектно-ориентированном - способ мышления программиста. Отвечая требованиям современного программирования, объектно-ориентированный стиль программирования делает акцент на разработке новых типов данных, наиболее полно соответствующих концепциям выбранной области знаний и задачам приложения.

Основными концепциями объектно-ориентированного анализа и программирования являются принципы: абстракция, наследование, инкапсуляция и полиморфизм.

2.1 Принцип абстрагирования

Абстрагирование (abstraction) подразумевает собой процесс изменения уровня детализации программы, то есть выделение существенных характеристик некоторого объекта, отличающие его от всех других видов объектов и, таким образом, четкое определение его концептуальных границ с точки зрения наблюдателя.

Объекты реального мира, с которыми приходится иметь дело разработчику программной системы, зачастую многогранны, обладают большим числом характеристик и сложным поведением. При этом не все это многообразие существенно с точки зрения специфики решаемой задачи. Когда мы абстрагируемся от

проблемы, мы предполагаем игнорирование ряда подробностей с тем, чтобы свести задачу к более простой.

Задача абстрагирования и последующей декомпозиции типична для процесса создания программ. Декомпозиция используется для разбиения программ на компоненты, которые затем могут быть объединены, позволив решить основную задачу, абстрагирование же предлагает продуманный выбор таких компонент.

Последовательно выполняя то один, то другой процесс можно свести исходную задачу к подзадачам, решение которых известно. Для одного и того же моделируемого в программе объекта в зависимости от решаемой задачи необходимо учитывать различные свойства и характеристики, то есть рассматривать его на различных уровнях абстракции.

Например, если мы будем рассматривать объект «Файл» в контексте разработки текстового редактора, то нас будут интересовать такие параметры объекта, как тип представления информации в файле, методы чтения и записи информации из/в файл, используемые промежуточные буферы для хранения информации. Если этот же объект «Файл» рассматривать в контексте разработки файлового менеджера, то на первый план выходят имя файла, путь к файлу, атрибуты, права доступа и т.п. Выбор правильного набора абстракций для заданной предметной области представляет собой главную задачу объектно-ориентированного проектирования.

Абстракция определяет границу представления соответствующего элемента модели и применяется для определения фундаментальных понятий ООП, таких как класс и объект.

Класс представляет собой абстракцию совокупности реальных объектов, которые имеют общий набор свойств и обладают одинаковым поведением. Объект в контексте ООП рассматривается как экземпляр соответствующего класса.

2.2 Принцип наследования

Наследование заключается в процессе создания новых объектов (потомков) на основе уже имеющихся объектов (предков) с передачей их свойств и методов по наследству. Наследование позволяет модифицировать поведение объектов и придает объектно-ориентированному программированию исключительную гибкость.

Программный объект также может унаследовать от объекта-предка некоторые свойства и методы, а добавленные к этим унаследованным атрибутам собственные свойства и методы позволяют расширить функциональность по отношению к объекту-предку.

Преимущество принципа наследования заключается в повторном использовании кода, когда каждый новый объект не создается с нуля, а строится на фундаменте уже существующего объекта. При этом уменьшается как размер кода, так и сложность программы.

На практике, при реализации конкретной задачи, на наследование влияет используемый язык программирования, поскольку его семантика влияет на процесс декомпозиции. Так, например, язык программирования C++ допускает множественное наследование классов, когда некоторый вновь объявляемый класс объявляется потомком сразу нескольких уже существующих классов, в то время как в языках C# или Java множественное наследование запрещено (хотя разрешено множественное наследование интерфейсов).

Благодаря использованию принципа наследования в современные системы программирования включены библиотеки классов, которые представляют собой многоуровневые иерархические системы классов, описывающих элементы программного или пользовательского интерфейса прикладной программы. В основе подобных систем лежат базовые классы, которые обычно очень просты и являются обобщением свойств всех остальных классов библиотеки. Классы-потомки базовых дополняются

собственными свойствами, переопределяют или уточняют методы базовых, приобретая дополнительную функциональность. Они, в свою очередь, становятся основой для классов следующих уровней иерархии. В результате формируется гибкая и стройная, а главное – открытая для модификации и развития, система классов. Подобной библиотекой классов является стандартная библиотека классов платформы .NET Base Class Library фирмы Microsoft. Достоинством библиотек классов так же является и то, что программист получает возможность создавать собственные классы, не определяя их с нуля, а всего лишь доопределив ряд недостающих свойств для какого-либо стандартного класса библиотеки, выбранного в качестве базового.

Как уже было отмечено, наследование – не единственный способ описания отношений иерархии в ООП. Когда в качестве одного из полей класса используется экземпляр другого класса, мы имеем дело с агрегацией. Агрегация позволяет структурировать сложные объекты, собирая их как конструктор, из нескольких объектов - составных частей. Так, например, класс «Диалоговое окно» может агрегировать основные элементы пользовательского интерфейса окна в виде объектов «Кнопка», «Поле ввода», «Список выбора» и др.

2.3 Принцип инкапсуляции

Инкапсуляция есть объединение в едином объекте данных и кодов, оперирующих с этими данными. В терминологии объектно-ориентированного программирования данные называются членами данных (data members) объекта, а коды - объектными методами иди функциями-членами (methods, member functions).

Методы отвечают за поведение объекта, в них заключены те действия, которые объект может выполнить. Сообщения объекту поддерживаются с использованием механизма передачи параметров вызываемому методу. Члены данных отвечают за состояние объекта, их используют для фиксации тех характеристик, что были выделены на этапе абстрагирования. Инкапсуля-

ция является важным принципом ООП, организующим защиту информации от ненужных и случайных модификаций, что обеспечивает целостность данных и упрощает отладку программного кода после изменений.

Все компоненты объекта разделяются на интерфейс и внутреннюю реализацию. Интерфейс - это лицевая сторона объекта, способ работы со стороны его программного окружения – других объектов, модулей программы. В интерфейсной части описывается, что умеет делать объект. В противоположность интерфейсу, внутренняя реализация объекта представляет собой те компоненты класса, которые по замыслу разработчика класса не должны быть доступны извне. Реализация - это изнанка объекта, она определяет, как он выполняет задание, поступающее от интерфейсных компонент. При этом главным требованием принципа инкапсуляции, повторимся, является изоляция внутренней реализации объекта от окружения. Этим достигается целостность объекта при любых возможных внешних воздействиях на него.

Пример аналогии из реального мира: пользуясь смартфоном человек не задумывается что происходит, когда он начинает звонок и разговор с другим абонентом. Для него достаточно знать, что, набрав номер телефона он инициирует звонок и может разговаривать, используя микрофон для передачи голоса, и динамик для того, чтобы услышать говорящего.

2.4 Полиморфизм

Полиморфизм (греч. Poly - много, morfos - форма) — это свойство системы использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

Если взять пример из реального мира: Производится множество различных моделей сотовых телефонов, но интерфейс взаимодействия с пользователем у них практически одинаковый.

Если в объекте-потомке и объекте-родителе определены одноименные методы, имеющие разную кодовую реализацию (это называется перегрузкой метода в объекте-потомке), то вызов данного метода может быть привязан к его конкретной реализации в одном из родственных объектов только в момент выполнения программы. Это называется поздним связыванием, а методы, реализующие позднее связывание – полиморфными или виртуальными.

Принцип полиморфизма можно проиллюстрировать примером: для иерархии объектов – графических фигур (окружность, квадрат, треугольник и т.п.) можно определить виртуальную функцию draw(), отображающую фигуру. Объявление функции draw() виртуальной позволит обеспечить надлежащий отклик на событие с требованием отобразить ту или иную фигуру.

В языках программирования для реализации полиморфизма используются в том числе абстрактные классы и интерфейсы, которые служат начальной основой для переопределяемого кода в парадигме ООП.

Интерфейс — это разновидность обычного класса, но не содержащего код реализации свойств и методов. Интерфейс может определять имена методов и свойств, но не содержит их определение и реализация. Класс может наследовать несколько интерфейсов и должен реализовать код для всех методов этих интерфейсов. Из этого следует, что интерфейс регламентируется базовыми принципами и эти принципы настолько обобщены, что не имеет смысла для наследующего класса и каждый наследую-

щий, данный интерфейс, класс должен реализовать методы интерфейса самостоятельно для своей конкретной цели.

Абстрактный класс — это разновидность интерфейса и обычного класса и в отличии от интерфейса, может содержать в себе некоторую реализацию методов и может содержать еще абстрактные методы, которые в обязательном порядке должны быть реализованы в наследуемых классах.

3. Язык моделирования UML

Историю языков визуального моделирования, использующихся для создания моделей анализа и проектирования объектно-ориентированных систем, принято отсчитывать с начала 90-х годов, когда появились языки Booch (автор Г.Буч) и OMT (Object-ModelingTechnique Д.Рамбо). В 1995 г. объединение усилий авторов этих языков привело к разработке первой версии UML (Unified Modeling Language) – унифицированного языка моделирования. К настоящему времени усилиями консорциума OMG (Object Management Group) UML стал мировым стандартом моделирования объектно-ориентированных систем, в частности, язык UML принят в качестве международного стандарта ISO/IEC 19501:2005.

На сегодняшний день средства языка UML позволяют визуализировать, специфицировать, конструировать и документировать программные продукты, причем CASE-средства конвертируют UML-модели сразу в код на языках программирования, а также в структуру реляционной базы данных. Язык UML является языком графического моделирования, основная цель которого наладить коммуникацию как при общении с заказчиком, так и внутри самой команды разработчиков. Т.е. используется он для графического представления модели системы в виде UML-диаграмм. Нотация языка – элементы, из которых будут составляться диаграммы просты и понятны всем заинтересованным лицам и однозначно интерпретируемы. Нотация языка

UML использует для этого 4 вида элементов (фигуры, линии, надписи и значки), объединенных в две группы: сущности и отношения.

На основе их взаимодействия можно строить UML-диаграммы. Все фигуры диаграмм, в основном, плоские, исключение составляет только обозначение узлов инфраструктуры на диаграмме развертывания. Внутри любого элемента-сущности возможно отображение других элементов. Каждая линия на диаграмме (используются сплошные или пунктирные линии) должна обязательно соединяться своими концами с элементами-сущностями, т.е. «висящих» линий на диаграмме быть не должно. Пересечение линий допустимо, но для лучшей расшифровки диаграмм желательно количество пересечений минимизировать. Изображение фигур и значков минимально формализовано, и разработчики CASE-средств предлагают большую свободу действий при отри-совке диаграмм: заливку фигур цветом, использование теней, стилей и т.д.

Для целей визуализации представления объектов применяют одну из разновидностей диаграмм – статические диаграммы классов (class diagrams), которые используются для моделирования статической структуры классов системы и связей между ними. На таких диаграммах показывают классы, интерфейсы, объекты и кооперации, а также их отношения. На диаграмме классов каждый класс изображается прямоугольником, разделенным на три части. В первой указывается только имя класса, во второй перечисляются переменные класса, в третьей – методы класса (рис. 1). Иногда имеет смысл использовать сокращенное обозначение класса – прямоугольник, состоящий только из имени класса. Перед открытыми переменными и методами класса указывается знак + (вместо public), а перед закрытыми – знак – (вместо private). Кроме того, при описании переменных, методов и параметров методов сначала указывается имя, а затем после символа «:» тип. На диаграмме класса реализация метода не указывается (рис. 2).

а) б)

Рис. 1 – структура диаграммы классов (а – полное обозначение, б – сокращенное обозначение)

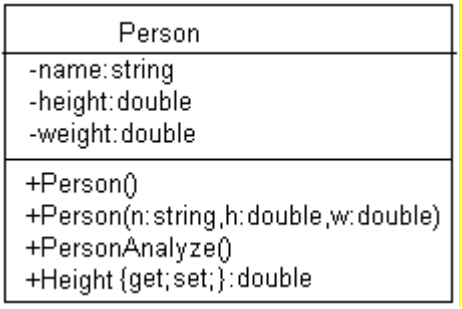
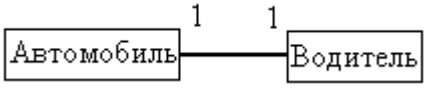


Рис. 2 – пример структуры диаграммы класса объекта

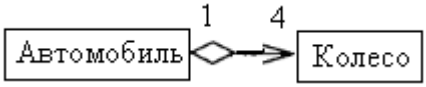
В языке UML сделана попытка классифицировать типы связей между объектами. Такая классификация существенно помогает в описании больших программных систем. Кроме того, существуют стандартные приемы реализации того или иного вида связи:

- Ассоциация – объект связан с другими объектами (знает об их существовании). Автомобиль – водитель.



- Композиция – объект (обязательно) состоит из других объектов (подобъектов) Подобъекты не могут существовать без объекта. Человек – сердце.

- Агрегация – объект (обязательно) состоит из других объектов (подобъектов). Подобъекты могут существовать самостоятельно или находиться в агрегации с другими объектами.



Таким образом, ассоциация – наиболее «слабый» тип связи, в наименьшей степени регламентирующий особенности связи.

Числа на концах линий связей называются кратностями связи.

Для реализации всех типов связей нужно в одном классе разместить переменную, ссылающуюся на объект (объекты) другого класса.

Для реализации ассоциации следует предусмотреть метод присваивания этой переменной ссылки на объект и метод, прерывающий (обнуляющий) эту связь. В конструкторе эту связь устанавливать не нужно.

Для реализации композиции следует в конструкторе класса создать объект и присвоить ссылку на него переменной. Открытый доступ к этому объекту реализовать только по чтению.

Для реализации агрегации следует в конструктор класса передать готовый объект и присвоить ссылку на него переменной. Реализовать также метод присваивания этой переменной ссылки на другой объект. Важно гарантировать невозможность присваивания этой переменной значения null.

ПРАКТИЧЕСКАЯ РАБОТА № 5

Тема: «Знакомство с элементами ООП с помощью динамического программирования визуальных компонентов.»

Цель работы – Изучить основные принципы реализации ООП в C#, получить навыки динамического программирования визуальных компонентов.

Теоретические сведения

5.1. Понятие класса в C#

Класс – это абстрактный тип данных, определяемый программистом. С помощью классов определяются свойства объектов. Объекты – это экземпляры класса.

Синтаксис объявления класса в C# следующий:

модификаторы class *имя-класса* {
    [элементы-класса]
}

Элементы класса – это данные и функции для работы с этими данными. Имя класса – это имя нового типа данных. Модификаторы определяют доступ к элементам класса.

Основные элементы класса:

1. Поле – обычная член-переменная, содержащая некоторое значение.
2. Константа – это поле, объявленное с модификатором const, значение которого изменить нельзя.
3. Метод – методы описывают функциональность класса.
4. Свойство – предоставляют защищённый доступ к полям.
5. Индексатор – это свойство-коллекция, отдельный элемент которого доступен по индексу.
6. Конструктор – осуществляет начальную инициализацию объекта (экземплярный конструктор) или класса (статический конструктор).
7. Финализатор – автоматически вызывается сборщиком мусора и содержит завершающий код для объекта.
8. Событие – представляют собой механизм рассылки уведомлений различным объектам – клиентам класса.

Модификаторы доступа:

1. private – элемент с этим модификатором доступен только в том типе, в котором определён.

2. protected – элемент виден в типе, в котором определён, и в наследниках этого типа[6].

3. internal – элемент доступен без ограничений, но только в той сборке, где описан.

4. protected internal – элемент виден в содержащей его сборке без ограничений, а вне сборки – только в наследниках типа (т.е. это комбинация protected или internal).

5. public – элемент доступен без ограничений как в той сборке, где описан, так и в других сборка, к которым подключается сборка с элементом.

По умолчанию в классе устанавливается спецификация доступа private. Спецификация доступа, отличная от private, должна указываться явно перед каждым членом класса. Для локальных переменных методов и операторных блоков модификаторы доступа не используются. При описании самостоятельного класса допустимо указать для него модификаторы public или internal (internal применяется по умолчанию).

Для удобного хранения классов возможно размещать каждый класс в отдельном файле. Но и в этом случае возможны ситуации, когда классы становятся очень большими. Для дальнейшего деления кода используются *разделяемые классы* (partial classes) – это классы, разбитые на несколько фрагментов, описанных в отдельных файлах с исходным кодом[7]. Все фрагменты разделяемого класса должны быть доступны во время компиляции, так как «сборку» типа выполняет компилятор. Модификаторы доступа должны быть одинаковыми у всех фрагментов. Для объявления разделяемого класса используется модификатор partial:

```
// файл part1.cs
partial class MyClass {
```

[6] Данный модификатор может применяться только в типах, поддерживающих наследование, то есть в классах.

[7] Разделяемыми могут быть не только классы, но структуры и интерфейсы.

```
    private int Field1;
    private string Field2;
}
// файл part2.cs
partial class MyClass {
    public void Method() { }
}
```

Для использования класса в программе после объявления необходима переменная класса – *объект*. Объект объявляется как обычная переменная: *имя-класса имя-объекта*; Объект – это та конструкция, которая поддерживает инкапсуляцию (объединяет в себе данные и код, работающий с ними).

Так как класс – ссылочный тип, то объекты должны быть инициализированы до непосредственного использования. Для инициализации объекта используется операция new – вызов конструктора класса. Если конструктор не описывался, применяется предопределённый конструктор без параметров с именем класса: *имя-объекта* = new *имя-класса*();
Инициализацию объекта можно совместить с его объявлением: *имя-класса (или* var*) имя-объекта* = new *имя-класса*();
Доступ к экземплярным элементам класса через объект осуществляется по синтаксису *имя-объекта.имя-элемента*.

5.1.1. Поля класса

Синтаксис объявления поля совпадает с оператором объявления переменной. Тип поля всегда должен быть указан явно. Если для поля не указано начальное значение, то поле принимает значение по умолчанию для соответствующего типа (для числовых типов – 0, для типа bool – false, для ссылочных типов – null). Модификатор readonly для поля запрещает изменение его после начальной установки. Поля с модификатором readonly похожи на константы, но имеют следующие отличия:

– тип поля может быть любым;

– значение поля можно установить при объявлении или в конструкторе класса;

– значение поля вычисляется в момент выполнения, а не при компиляции.

Пример описания полей и констант класса:

```
class MyClass {
        public          int         a;
        protected       float       b;
                        double      c;
        public  const   int         x = 25;
        public          char        simbol;
        private         decimal     sum;
    }
```

5.1.2. Методы класса

Методы в языке C# являются неотъемлемой частью описания таких пользовательских типов как класс или структура. Общий синтаксис описания метода:

модификаторы тип имя-метода(*[параметры]*) *тело-метода*

Здесь *тип* – это тип возвращаемого методом значения. Допустимо использование любого типа. В C# не существует процедур[8] – любой метод считается функцией. После имени метода всегда следует пара круглых скобок, в которых указывается список формальных параметров метода.

Список формальных параметров метода – это набор элементов, разделённых запятыми. Каждый элемент имеет следующий формат:

[модификатор] тип имя-формального-параметра [= значение]

Существуют четыре вида параметров:

1. *Параметры-значения* – объявляются без модификатора;
2. *Параметры, передаваемые по ссылке* – модификатор ref;

[8] Для «процедуры» в качестве тип возвращаемого значения указывается специальное ключевое слово void.

3. *Выходные параметры* – модификатором out;
4. *Параметры-списки* – модификатор params.

Параметры, передаваемые по ссылке и по значению, аналогичны C++. Выходные параметры подобны ссылочным, то есть при работе с ними в теле метода не создаётся копия фактического параметра. Параметры-списки позволяют передать в метод любое количество аргументов. Метод может иметь не более одного параметра-списка, который обязательно должен быть последним в списке формальных параметров. Тип параметра-списка объявляется как одномерный массив, и работа с таким параметром происходит в методе как с массивом. Каждый аргумент из передаваемого в метод списка ведёт себя как параметр, переданный по значению.

При объявлении метода для параметров-значений допустимо указать значение параметра по умолчанию (*опциональный параметр*), который располагается в конце списка формальных параметров метода. Для выхода из метода служит оператор return или оператор throw. Если тип метода не void, то после return обязательно указывается возвращаемое значение. Кроме этого, оператор return или оператор throw должны встретиться в таком методе во всех ветвях кода хотя бы один раз.

Примеры объявления методов.

1. Простейшее объявление метода-процедуры без параметров:

```
void HelloWorld() {
    Console.WriteLine("Hello World!");
}
```

2. Метод без параметров, возвращающий целое значение:

```
int ReturnInt() {
    Console.WriteLine("Return 5!");
    return 5;
}
```

3. Функция возвращает одно целочисленное значение:

```
int Minus(int x, int y) {
```

```
        return x - y;
    }
```

4. Метод функция возвращает через параметр a – 25 и как результат своей работы – 5:

```
    int Return2Val (out int a) {
        a = 25;
        return 5;
    }
```

5. Метод использует параметр-список:

```
    void PrintList(params int[] list) {
        foreach(int item in list)
            Console.WriteLine(item);
    }
```

6. Метод имеет опциональный параметр y:

```
    int WithOptional(int x, int y = 5) {
        return x - y;
    }
```

В экземплярных методах доступен параметр this[9] – это ссылка на текущий экземпляр, которая применяется для устранения конфликта имён[10]:

```
    class Student {
        private int age;
        private string name;
        public void SetAge(int age) {
            this.age=age>0?age:0;
        }
    }
```

C# позволяет выполнить в пользовательских типах *перегрузку методов*. Перегруженные методы имеют одинаковое имя, но разную сигнатуру. *Сигнатура* – это упорядоченный набор из модификаторов и типов формальных параметров.

---

[9] Метод класса – только для чтения, метод структуры – чтение и запись.
[10] Случай, когда имя элемента типа совпадает с именем параметра метода.

Для правильной работы перегруженных методов необходимо, чтобы Common Language Runtime (CLR) различала вызовы этих методов. Так, типы object и dynamic эквивалентны с точки зрения CLR, или , если одна версия метода содержит модификатор ref, а другая – out, то такой вызов неразличим CLR. Если различие двух версий перегруженного метода заключается только в модификаторе params, они не различимыми. Но если одна версия метода содержит модификатор ref или out, а другая нет, то методы различимы и такой код компилируется:

```
void Z(out int x) { . . . }
void Z(int x) { . . . }
```

При вызове метода вместо формальных параметров помещаются фактические *аргументы*[11]. Соответствие между параметром и аргументом устанавливается либо по позиции, либо используя синтаксис *именованных аргументов*:

имя-формального-параметра: *выражение-для-аргумента*

Пример вызова метода, содержащего три параметра:

```
int Sum(int x, int y = 3, int z = 5) {
   return x + y + z;
}
int r1=Sum(1, 2, 3); // передача по позиции
int r2= Sum (x:1, z:2, y:3);// именованные параметры
int r3=Sum(1, z:2, y:3);// комбинация этих способов
```

Если метод содержит опциональные параметры, использование именованных аргументов бывает необходимым:

```
int r4=Sum (1, z:2);
```

Метод с параметром-списком вызывается по-разному:

```
//передача в качестве аргумента массива целых чисел
SumList(new[] {1, 2, 3, 4});
SumList(1, 2); // передача двух аргументов
```

[11] Если аргументы заданы выражением, сначала производится вычисление этих аргументов.

```
SumList(1, 2, 3, 4); // передача четырех аргументов
SumList();// передача 0 аргументов
```

При использовании модификаторов ref или out для параметров[12] – они должны быть указаны и при вызове.

Разделяемые классы и структуры могут содержать *разделяемые методы*, которые состоят из двух частей: заголовка и реализации. Обычно эти части размещаются в различных частях разделяемого типа и подчиняются следующим правилам:

– объявление метода начинается с модификатора partial;
– метод обязан возвращать значение void;
– метод может иметь параметры, но out-параметры запрещены;
– метод неявно объявляется как private;
– разделяемые методы могут быть статическими или универсальными;
– вызов разделяемого метода нельзя инкапсулировать в делегат.

Пример описания разделяемого метода:

```
public partial class B {
   partial void Z(int x);
}
public partial class B {
   partial void Z(int x) {
      Console.WriteLine("Method Z");
   }
}
```

5.1.3. Свойства класса и индексаторы

Свойства класса позволяют осуществлять защищённый доступ[13] к полям. Поля класса объявляются с модификатором pri-

---

[12] Аргументы с такими модификаторами должны быть представлены переменными, а не литералами или выражениями и требуют абсолютное совпадение типов. В случае параметров-значений тип аргумента должен совпадать или неявно приводится к типу формального параметра.

[13] В C# непосредственная работа с полями не считается хорошим стилем.

vate, а для доступа к ним используются свойства. Базовый синтаксис описания свойства имеет следующий вид:

```
модификаторы тип-свойства имя-свойства {
    get { операторы }
    set { операторы }
}
```

Заголовок свойства подобен описанию обычного поля, при этом тип обычно совпадает с типом того поля. В специальном блоке содержатся методы для доступа к свойству: get[14] и set[15]. Если одна из частей отсутствует, то свойство будет только для чтения или только для записи. Свойства транслируются при компиляции в вызовы методов. В скомпилированный код класса добавляются методы со специальными именами get_*Name*() и set_*Name*(), где *Name* – это имя свойства[16]. Пример класса с свойством:

```
public class Student {
    private int _age;
    private string _name;
    public int Age {
        get { return _age; }
        set { _age = value>0?value:0;}
    }
    public string Name {
        get { return "My name is " + _name; }
        set { _name = value; }
    }
}
```

[14] Возвращаемое свойством значение - работает как функция

[15] Процедура, устанавливающая значение свойства, с передаваемым параметром value.

[16] Побочным эффектом трансляции является то, что пользовательские методы с данными именами допустимы в классе, только если они имеют сигнатуру, отличающуюся от методов, соответствующих свойству.

Обычно свойства имеют модификатор доступа public, но допускается применение и других модификаторов. Для методов свойства get и set допускается указывать модификаторы доступа. При этом действуют два правила: модификатор может быть только у одной из частей, и он должен понижать видимость части по сравнению с видимостью всего свойства:

```
public class MyClass {
   public int Prop {
      get { return 0; }
      private set { }
   }
}
```

В отличие от полей свойства не классифицируются как переменные. Поэтому свойство нельзя передать в качестве параметра ref или out.

Для упрощения описания свойств-«обёрток», в C# применяются автосвойства (auto properties):

```
public class MyClass {
   public string Prop { get; set; }
}
```

В этом случае компилятор сам сгенерирует необходимое поле класса, связанное со свойством. В автосвойстве должны присутствовать и часть get, и часть set[17]. При необходимости можно использовать модификаторы доступа:

```
public class MyClass {
   public string Prop { get; private set; }
}
```

Кроме скалярных свойств язык C# поддерживает индексаторы, с помощью которых осуществляется доступ к коллекции данных[18], содержащихся в объекте, с использованием синтакси-

---

[17] Начиная с C# 6.0 можно убрать «set»

[18] Коллекции предоставляют более гибкий способ работы с группами объектов по сравнению с массивами.

са для доступа к элементам массива (индекс в квадратных скобках). Базовый синтаксис объявления индексатора:

модификаторы тип this[*параметры*] { *get-и-set-блоки* }

Параметры индексатора служат для описания типа и имён индексов, применяемых для доступа к данным, и могут быть описаны как параметры-значения или как параметр-список (с использованием params). Также допустимо использование опциональных параметров и именованные индексы.

Пример класса, содержащего индексатор:

```
public class Student {
   private readonly int[] _marks = new int[5];
   public int this[int i]  {
      get {return Belongs(i, 0, 4) ? _marks[i] : 0;}
      set {
      if (Belongs(i, 0, 4) && Belongs(value, 1, 10))
            _marks[i] = value;
      }
   }
   private bool Belongs(int x, int min, int max) {
      return (x >= min) && (x <= max);
   }
}
var student = new Student();
student[1] = 8;
student[3] = 4;
for (int i = 0; i < 5; i++) {
   Console.WriteLine(student[i]);
}
```

Если необходимо использовать индексатор в пределах класса, применяют синтаксис this[*значение*].

В одном классе нельзя объявить два индексатора, у которых совпадают типы параметров. Можно объявить индексаторы, у

которых параметры имеют разный тип или количество параметров различается[19].

5.2. Статические элементы и методы расширения

В C# помимо экземплярных[20] элементов, существуют так же и статические элементы, которые применяются, когда требуется определить такой член класса, который будет использоваться независимо от всех остальных объектов этого класса. Доступ к члену класса организуется посредством объекта этого класса, но в то же время можно создать член класса для применения без ссылки на конкретный экземпляр объекта. Статические поля хранят информацию, общую для всех объектов, статические методы либо не используют поля, либо работают только со статическими полями. Чтобы объявить статический элемент, применяется модификатор static:

```
using System;
using System.Text;
namespace ConsoleApplication {
   class Circle {
   //методы, возвращающие площадь и длину круга
      public static double SqrCircle(int radius) {
         return Math.PI * radius * radius;
      }
      public static double LongCircle(int radius) {
         return 2 * Math.PI * radius;
      }
   }
   class Program {
      static void Main(string[] args) {
```

[19] Индексаторы транслируются в методы с именами get_Item() и set_Item(). Изменить имена методов можно, используя атрибут [IndexerName].

[20] Поля, методы и свойства классов, которые использовались при помощи объекта класса

```
            int r = 10;
            // Вызов методов из другого класса
            // без создания экземпляра объекта этого класса
            Console.WriteLine("Площадь круга радиусом {0} = 
{1:#.##}",r, Circle.SqrCircle(r));
            Console.WriteLine("Длина круга равна {0:#.##}", 
Circle.LongCircle(r));
            Console.ReadLine();
        }
    }
}
```

Для вызова статических элементов используется имя класса: Circle.SqrCircle(r);

Статическими могут быть сделаны поля, методы и обычные свойства, константа, описанная в классе, уже является статическим элементом. Индексатор класса не может быть статическим.

Если класс описан только с помощью статических элементов, то при описании класса указывается модификатор static и класс становится статическим:

```
public static class Pi {
    private static double _pi = 3,14159265358979323846264 3;
    public static double GetPi() {
        return _pi;
    }
}
```

Экземпляр статического класса не может быть создан и объявлен в программе, для них запрещено наследование. Доступ к открытым элементам статического класса доступны с помощью имени класса: Console.WriteLine(Pi.GetPi());

В пространстве имён System есть несколько полезных статических классов: для преобразования данных различных типов используют класс Convert, класс Math содержит набор математических функций, класс Console предназначен для чтения и за-

писи информации в консоль, класс Environment содержит свойства, описывающие окружение запуска программы.

Методы расширения[21] (extension method) позволяют добавлять новые методы в уже существующие типы без создания нового производного класса, при этом методами расширения могут быть только статические методы[22] статических классов. Методы расширения принимают параметр расширяемый тип с ключевым словом this перед именем типа:

```
public static ResultType ExtensionMethodExample
                                    (this ExtendedType value, ...)
public static string ToTranslit(this string str) { ... }
var s= "Привет"; var translitString = s.ToTranslit();
```

Количество параметров метода расширения не менее одного, но только первый можно указать с модификатором this. Соответственно, метод расширит тип первого параметра.

Методы расширения применимы к типу, как только импортируется пространство имён, содержащее класс с этими методами расширения. Если выполняется импорт нескольких пространств имён, содержащих классы с одинаковыми методами расширения для одного типа, то возникает ошибка компиляции. В этом случае методы расширения должны использоваться как обычные статические методы вспомогательных классов.

5.3. Конструкторы и инициализация объектов

Конструктор – это метод, который выполняет начальную инициализацию объекта или класса. Имя конструктора всегда совпадает с именем класса без указания на тип возвращаемого значения. Экземплярные конструкторы создают и инициализируют объект. Экземплярный конструктор производит размеще-

---

[21] Добавлены в C# 3.0. Методы расширения — это вид статического метода, которые вызываются как методы экземпляра в расширенном типе.

[22] Язык C# допускает только *методы* расширения, свойств и индексаторов расширения не существует.

ние объекта в динамической памяти и инициализацию полей объекта. Экземплярные конструкторы бывают двух видов: конструкторы по умолчанию и пользовательские конструкторы.

Конструктор по умолчанию – это всегда конструктор без параметров, который автоматически создаётся компилятором, если не описан собственный конструктор.

```
//MyClass не содержит описания конструктора
public class MyClass {
   public int Prop;
}
//вызов конструктора по умолчанию
var x = new MyClass ();
Console.WriteLine(x. Prop);//выводит 0
```

Пользовательские конструкторы создаются при описании класса. Класс может содержать несколько пользовательских конструкторов, которые обязательно должны различаться сигнатурой. Если в классе определён хотя бы один пользовательский конструктор, конструктор по умолчанию не создаётся. В пользовательском конструкторе возможно использовать значения параметров по умолчанию.

```
//MyClass содержит два пользовательских конструктора
public class MyClass {
   public int Prop;
   public MyClass () {
      Prop = 0;
   }
   public MyClass (int x) {
      Prop = x;
   }
}
//конструктор MyClass с значением параметра по умолчанию
public class MyClass {
   public int Prop;
```

```
    public MyClass(int x = 0) {
        Prop = x;
    }
}
```

Пользовательские конструкторы могут применяться для начальной инициализации readonly-полей или могут вызвать другой конструктор того же класса:

```
public MyClass () : this(10) { . . . }
```

Для вызова экземплярных конструкторов используется операция new, которая возвращает созданный объект (у объекта нельзя вызвать конструктор как метод).

```
// вызов конструкторов для создания объекта
var x = new MyClass ();
var y = new MyClass (5);
```

Статические конструкторы используются для начальной инициализации статических полей класса. Статический конструктор всегда объявляется с модификатором static без параметров. Область видимости у статических конструкторов не указывается. В теле статического конструктора возможна работа только со статическими полями и методами класса. Статический конструктор не может вызывать экземплярные конструкторы в начале своей работы.

```
public class MyClass {
    private static double _prop;
    static MyClass() {
        _ prop = 12;
    }
}
```

Статические конструкторы вызываются общеязыковой средой исполнения в следующих случаях[23]:

---

[23] Если класс содержит статический конструктор, компилятор C# генерирует код, выполняющий инициализацию статических полей класса непосредственно перед первым использованием класса. Без статического конструкто-

– перед созданием первого объекта класса или при первом обращении к элементу класса, не унаследованному от предка;

– перед первым обращением к статическому полю, не унаследованному от предка.

Обычно, при работе с объектами, сначала создаётся объект, а затем настраивается путём установки свойств:

```
var x = new MyClass();
x.Prop = 10;
```

C# позволяет совместить создание объекта с его настройкой. Для этого после параметров конструктора в фигурных скобках требуемым public-элементам класса присваиваются их значения (если конструктор не имел параметров, можно не указывать круглые скобки после его имени):

```
var x = new MyClass {Prop = 10};
```

Инициализация объектов действует в том числе и для классов-коллекций. Такой класс реализует интерфейс IEnumerable и имеет public-метод Add(), который вызывает компилятор, когда обрабатывает код инициализации:

```
var x = new List<int> {1, 2, 3, 4};
```

Инициализация коллекций без метода Add() возможна, если эти параметры записать в фигурных скобках:

```
var elements = new Dictionary<int, string> {{1, "Resistance"}, {2, "Capacitance"}, {3, "Inductance"}};
```

5.4. Контейнеры в Windows Forms

Для организации элементов управления в связанные группы существуют специальные элементы – контейнеры, такие как, Form, Panel, FlowLayoutPanel, SplitContainer, GroupBox. Использование контейнеров облегчает управление элементами, а также придает форме определенный визуальный стиль.

ра такая инициализация проводится в произвольный момент до использования класса.

Все контейнеры имеют свойство Controls, которое содержит все элементы данного контейнера. Когда происходит перенос элемента с панели инструментов на контейнер, элемент автоматически добавляется в данную коллекцию данного контейнера. Имеется возможность добавления элемента управления динамически с помощью кода в эту же коллекцию.

5.4.1 Компонент класса Form

Компонент класса `Form` (рис.5.1) представляет собой форму, на которой располагаются другие компоненты, и служит для создания основного либо дополнительных окон программы. Первая форма добавляется в проект автоматически при создании приложения, остальные добавляются, например, путем выбора пункта меню «Проект → Добавить форму Windows».

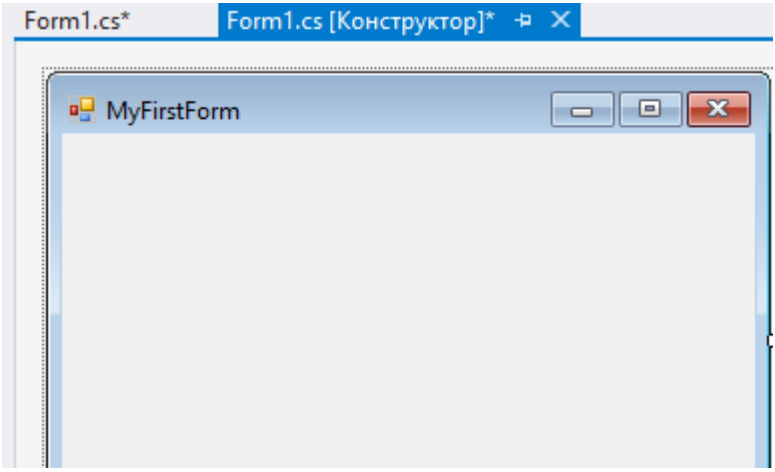


Рис. 5.1 – компонент класса Form

Ниже показан список основных свойств и событий компонента класса `Form`:

1. `AcceptButton` – ссылка на кнопку, которая будет нажиматься при нажатии клавиши «Enter»;

2. `CancelButton` – ссылка на кнопку, которая будет нажиматься при нажатии клавиши «Esc»;
3. `FormBorderStyle` – свойство, определяющее тип границы компонента. Значением «по умолчанию» является `Sizable`. Для диалоговых окон применяется значение `FixedDialog`;
4. `Text` – строка, описывающая заголовок формы;
5. `MainMenuStrip` – ссылка на компонент, формирующий основное меню формы;
6. `DialogResult` – свойство, позволяющее определить способ закрытия «модального» диалогового окна. Наиболее часто используемые значения: `None`, `OK`, `Cancel`;
7. `MinimizeBox` – логическое свойство, определяющее необходимость размещения кнопки минимизации у формы;
8. `MaximizeBox` – логическое свойство, определяющее необходимость размещения кнопки минимизации у формы;
9. `ShowInTaskbar` – логическое свойство, определяющее необходимость отображения формы на панели задач;
10. `Activated` – событие, возникающее при каждом получении формой фокуса ввода;
11. `FormClosing` – событие, возникающее перед закрытием формы и позволяющее заблокировать это закрытие путем присвоения свойству `Cancel` параметра `e` значения `true` (e.Cancel = true;);
12. `Load` – событие, возникающее один раз при создании формы (полезно для выполнения подготовительных действий);
13. `FormClosed` – событие, возникающее один раз перед закрытием формы;
14. `Resize` – событие, возникающее при изменении размера формы;

15. `Shown` – событие, возникающее при первом отображении формы. Все формы (кроме основной) отображаются по мере необходимости путем создания формы требуемого класса и ее программного вызова с помощью методов:
16. `Show` – отображение формы. При этом допускается одновременная работа и основной, и в вызванной форме;
17. `ShowDialog` – Отображение формы в виде диалогового окна (в «модальном» режиме)[24]. Результат завершения работы диалогового окна класса `DialogResult` возвращается как значение функции `ShowDialog`. На форме, работающей в «модальном» режиме, как правило размещаются две кнопки, у каждой из которых устанавливается свойство `DialogResult` в нужное значение:
    a. OK – кнопка подтверждения ввода данных в диалоговом окне;
    b. Cancel – кнопка отказа от ввода в диалоговом окне.

5.4.2 Динамическое добавление элементов

Очень часто требуется динамически создавать или удалять элементы в форме. Для динамического добавления элементов в форму необходимо создать событие загрузки формы, в котором будут создаваться новые элементы управления. Это можно сделать либо с помощью кода, либо визуальным образом. Для динамического добавления элементов необходимо создать обработчик события загрузки формы в файле кода:

```
private void Form1_Load(object sender, EventArgs e) {
    Button myButton = new Button();
    myButton.BackColor = Color.LightGray;
    myButton.ForeColor = Color.DarkGray;
    myButton.Location = new Point(10, 10);
    myButton.Text = "Click Me!";
```

[24] При таком отображении происходит блокирование всех остальных форм приложения (в том числе и основной) до тех пор, пока работа с диалоговым окном не будет завершена.

```
    this.Controls.Add(myButton);//добавление кнопки в
коллекцию элементов формы
}
```

С помощью метода `Controls.Remove()` можно удалить элемент с формы: `this.Controls.Remove(helloButton);`

5.4.3 Компонент класса GroupBox.

Компонент класса `GroupBox` (рис. 5.2) – это контейнер для размещения в нем других компонентов. Компоненты, расположенные в контейнере, отсчитывают свое положение от левой-верхней границы контейнера, при этом перемещение контейнера по экрану приводит к синхронному перемещению всех компонентов, расположенных внутри него. Кроме того, компоненты, расположенные внутри контейнера, не могут выйти за его границы, а компоненты, расположенные вне контейнера, не могут быть помещены внутрь его простым перетаскиванием на форме.

Свойство `Text` компонента класса `GroupBox` позволяет именовать группу компонентов, расположенных внутри него.

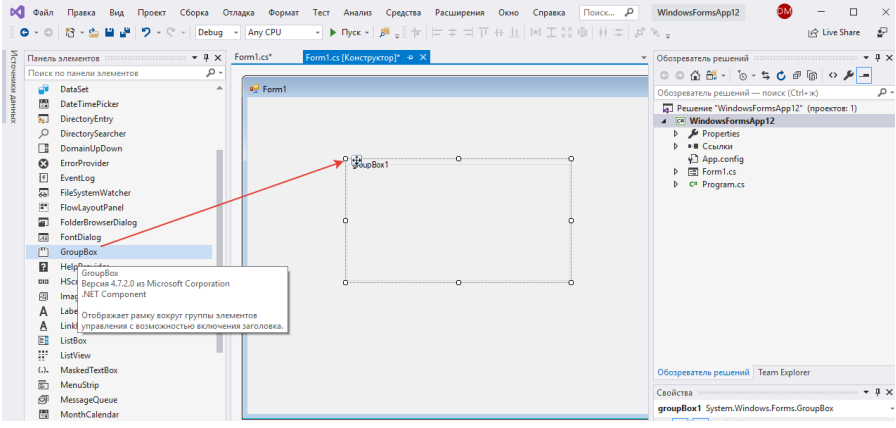


Рис. 5.2 – Использование компонента класса GroupBox.

5.4.3 Компонент класса Panel

Элемент `Panel` (рис. 5.3) представляет панель и также, как и `GroupBox`, объединяет элементы в группы. Для стилизации данного элемента используют цвет фона в свойстве `BackColor` и границы с помощью свойства `BorderStyle`, которое по умолчанию имеет значение `None`, то есть отсутствие границ.

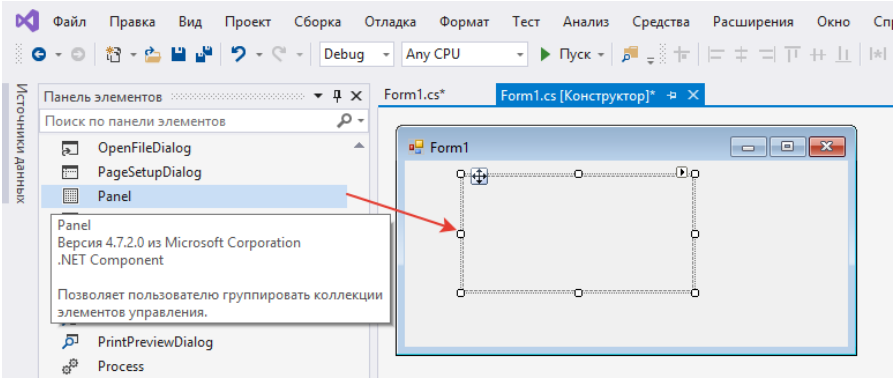


Рис. 5.3 – Использование компонента класса Panel.

В случае, если элементы выходят за границы панели, с помощью установки свойства `AutoScroll` в `true` создают прокрутку.

`GroupBox` и `Panel` имеют коллекции элементов, и имеется возможность динамически добавлять в эти контейнеры элементы. Например, пусть на форме есть элемент `GroupBox`, который имеет имя `groupBox1`:

```
private void Form1_Load(object sender, EventArgs e){
    Button myButton = new Button();
    myButton.BackColor = Color.LightGray;
    myButton.ForeColor = Color.Red;
    myButton.Location = new Point(10, 10);
    myButton.Text = Click Me!";
    groupBox1.Controls.Add(myButton);
}
```

Для указания расположения элемента в контейнере используется структура `Point`: `new Point(10, 10);`, которой в конструкторе передаются координаты размещения по осям `X` и `Y` относительно левого верхнего угла контейнера. Так же необходимо учитывать, что контейнером верхнего уровня является форма, а элемент `groupBox1` сам находится в коллекции элементов формы. В случае, если, необходимо удалить элемент, это осуществляется так: `this.Controls.Remove(groupBox1);`.

5.4.4 Компонент класса FlowLayoutPanel

Элемент `FlowLayoutPanel` наследуется от класса `Panel`, и соответственно содержит все его свойства. К дополнительным свойствам относят: изменение позиционирования и компоновки дочерних элементов при изменении размеров формы во время выполнения программы.

Свойство элемента `FlowDirection` позволяет задать направление, в котором направлены дочерние элементы. По умолчанию имеет значение `LeftToRight` - то есть элементы будут располагаться начиная от левого верхнего края. Следующие элементы будут идти вправо. Это свойство также может принимать следующие значения:

`RightToLeft` - элементы располагаются от правого верхнего угла в левую сторону

`TopDown` - элементы располагаются от левого верхнего угла и идут вниз

`BottomUp` - элементы располагаются от левого нижнего угла и идут вверх

Свойство `WrapContents` по умолчанию имеет значение `True`. Это позволяет переносить элементы, которые не умещаются в `FlowLayoutPanel`, на новую строку или в новый столбец. В случае `False` элементы не переносятся. Если свойство

AutoScroll равно true, к контейнеру добавляются полосы прокрутки.

5.4.5 Компонент класса TableLayoutPanel

Элемент TableLayoutPanel переопределяет панель и располагает дочерние элементы управления в виде таблицы, где для каждого элемента имеется своя ячейка. Если необходимо разместить в ячейке более одного элемента, то ячейку добавляется другой компонент TableLayoutPanel, в который затем вкладываются другие элементы.

Для задания числа строк и столбцов у таблицы используют свойства Rows и Columns соответственно. Выбрать один из этих пунктов можно в окне Properties (Свойства):

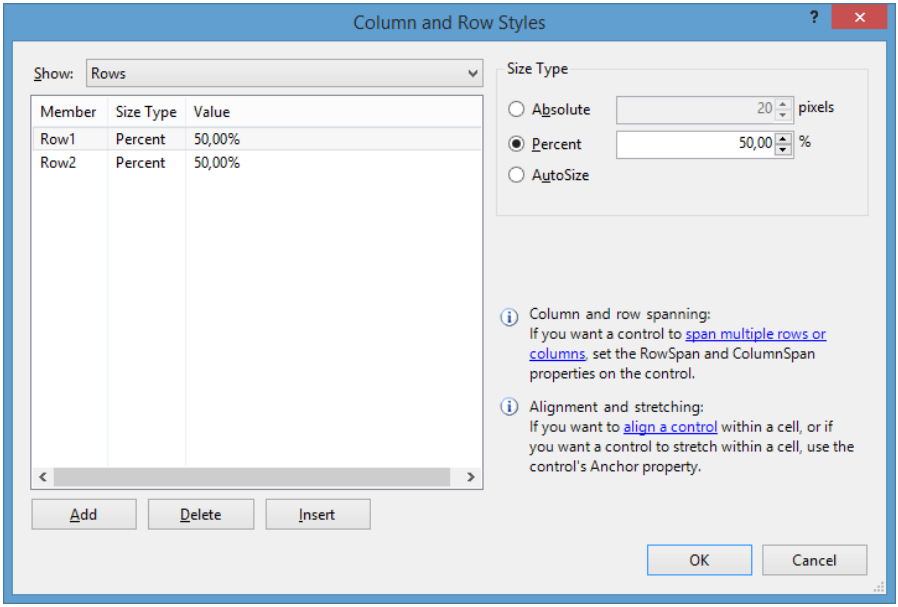


Рис. 5.4 – Использование компонента TableLayoutPanel.

В поле Size Type указывается размер столбцов/строк:

Absolute: абсолютный размер для строк или столбцов в пикселях;
Percent: относительный размер в процентах;
AutoSize: высота строк и ширина столбцов задаются автоматически в зависимости от размера самой большой в строке или столбце ячейки.

Для динамического изменения в коде значений столбцов и строк все столбцы представлены типом ColumnStyle, а строки - типом RowStyle. Для установки размера в ColumnStyle и RowStyle определено свойство SizeType, которое принимает одно из значений одноименного перечисления SizeType:

```
tableLayoutPanel1.RowStyles[0].SizeType =
SizeType.Percent;
tableLayoutPanel1.RowStyles[0].Height = 40;
tableLayoutPanel1.ColumnStyles[0].SizeType =
SizeType.Absolute;
tableLayoutPanel1.ColumnStyles[0].Width = 50;
```

Добавление элемента в контейнер TableLayoutPanel возможно, как в следующую свободную ячейку, так и возможно указать ячейку таблицы явно:

```
Button myButton = new Button();
// добавление кнопки в следующую свободную ячейку
tableLayoutPanel1.Controls.Add(myButton);
// добавление кнопки в ячейку (3,2)
tableLayoutPanel1.Controls.Add(myButton, 3, 2);
```

5.4.6 Компонент класса DataGridView

Компонент класса DataGridView (рис. 5.5) позволяет отображать таблицу, каждая из ячеек которой содержит некоторые данные.

Таблица может быть связана с некоторым источником данных (например, таблицей базы данных) или использоваться для ввода строковой информации. Данный компонент подходит для

организации работы с двухмерными (матрицами) или одномерными числовыми массивами.

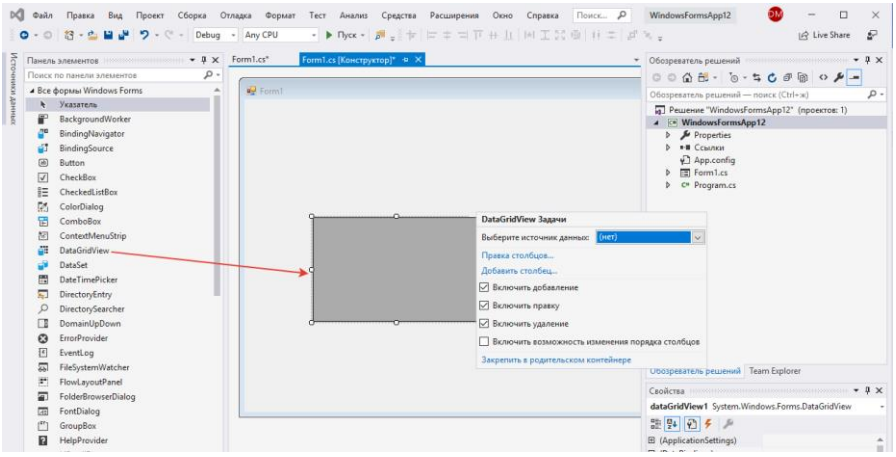


Рис. 5.5 – Использование компонента класса DataGridView.

Некоторые свойства данного компонента:

1) RowCount, ColumnCount (int) – свойства, с помощью которых можно определить или установить количество строк и столбцов, отображаемых в компоненте;

2) AllowUserToAddRows, AllowUserToDeleteRows (bool) – дает возможность добавлять и удалять строки;

3) RowHeadersVisible, ColumnHeadersVisible (bool) – определяют отображение столбца заголовков строк и столбцов;

4) [ACol, ARow] (DataGridViewCell) – доступ к ячейке[25] с номером столбца ACol и номером строки ARow;

5) ReadOnly (int) – при установке в значение true блокирует ручное изменение данных в компоненте;

6) DataSource (Object) – определяет источник данных, с которым связан компонент.

[25] Для доступа к данным ячейки необходимо обратиться к свойству Value (Object). Пример int i=Convert.ToInt32(dataGridView1[0,1].Value);

Нумерация строк и столбцов компонента начинается с нуля (0 .. RowCount-1).

5.4.7 Компонент класса NumericUpDown

Компонент класса NumericUpDown (рис. 5.6) используется для организации ввода чисел.

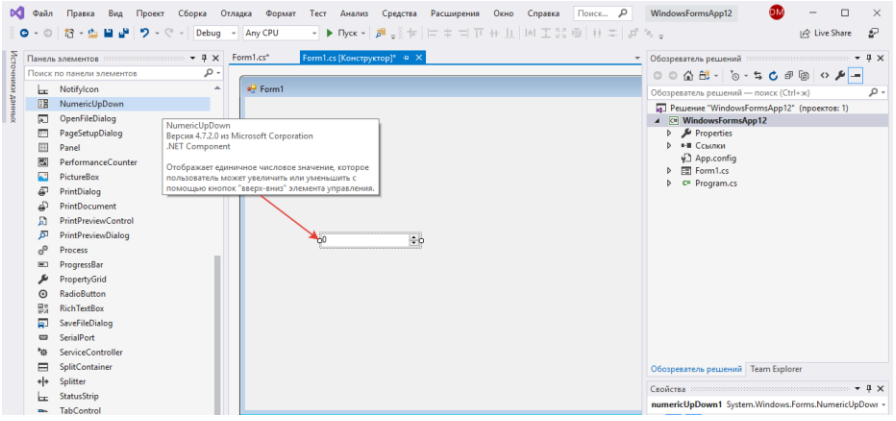


Рис. 5.6 – Использование компонента NumericUpDown.

Характерные свойства компонента:

1) Minimum, Maximum (decimal) – определяют диапазон значений, которые могут быть введены в данном компоненте;

2) DecimalPlaces (int) – определяет количество отображаемых десятичных разрядов;

3) Increment (decimal) – определяет, на сколько уменьшится или увеличивается текущее значение при использовании кнопок со стрелками;

4) Value (decimal) – определяет текущее значение.

Событие `ValueChanged` возникает при изменении значения свойства `Value`[26].

[26] Пример использования:

```
private void numeric UpDown1 ValueChanged(object sender, EventArgs e)
{ dataGridView1.RowCount=Convert.ToInt32(numericUpDown1.Value); }
```

5.4.8 Компонент класса PictureBox

Компонент класса PictureBox (рис. 5.7) представляет прямоугольную область, используемую для отображения рисунков и построения изображений программными средствами. Компонент расположен в разделе компонентов «Стандартные элементы управления». Компонент класса PictureBox представляет прямоугольную область, используемую для отображения рисунков и построения изображений программными средствами.

Для построения изображений на компоненте создаются объекты классов **Graphics**, **cPen**, **cBrush**. После изменения изображения требуется перерисовка компонента методом `Refresh`. Размеры компонента задаются с помощью свойства `Size`.

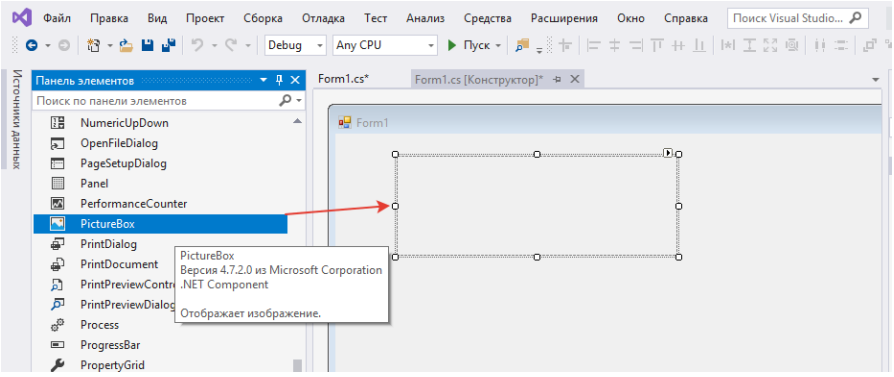


Рис. 5.7 – Использование компонента класса PictureBox.

Графический объект, отображаемый в поле компонента PictureBox, можно задать во время разработки формы или загрузить из файла во время работы программы.

Чтобы задать иллюстрацию во время создания формы, надо в строке свойства `Image` щелкнуть на кнопке с тремя точками и в появившемся стандартном окне «**Открыть**» выбрать файл иллюстрации. Среда разработки создаст битовый образ иллюстра-

ции и поместит его в файл ресурсов (таким образом, в дальнейшем файл иллюстрации программе будет не нужен). Загрузку иллюстрации из файла во время работы программы обеспечивает метод `FromFile`. В качестве параметра метода надо указать имя файла иллюстрации.

Например, инструкция

```
pictureBox1.Image = Image.FromFile("C:\Users\User\Pictures\Image-1.jpg");
```

обеспечивает загрузку и отображение иллюстрации, которая находится в файле C:\Users\User\Pictures\Image-1.jpg. Метод `FromFile` позволяет работать с файлами BMP, JPEG, GIF, PNG и других форматов.

Для установки изображения в PictureBox используется свойство `SizeMode`, которое принимает следующие значения:

- `Normal`: изображение позиционируется в левом верхнем углу PictureBox, и размер изображения не изменяется. Если PictureBox больше размеров изображения, то справа и снизу появляются пустоты, если меньше - то изображение обрезается
- `StretchImage`: изображение растягивается или сжимается таким образом, чтобы вместиться по всей ширине и высоте элемента PictureBox
- `AutoSize`: элемент PictureBox автоматически растягивается, подстраиваясь под размеры изображения
- `CenterImage`: если PictureBox меньше изображения, то изображение обрезается по краям и выводится только его центральная часть. Если же PictureBox больше изображения, то оно позиционируется по центру.
- `Zoom`: изменение размера иллюстрации таким образом, чтобы она занимала максимально возможную область компонента и при этом отображалась без искажения (с соблюдением пропорций).

Свойство `Image` - иллюстрация, которая отображается в поле компонента.

Свойство `Size` - Размер компонента. Уточняющее свойство `Width` определяет ширину компонента, `Height` — высоту.

Свойство `Image.PhysicalDimension` - свойство содержит информацию об истинном размере картинки, загруженной в поле компонента.

Свойство `Location` - положение компонента (области отображения иллюстрации) на поверхности формы. Уточняющее свойство `X` определяет расстояние от левой границы области до левой границы формы, уточняющее свойство `Y` — от верхней границы области до верхней границы клиентской области формы (нижней границы заголовка)

Свойство `Visible` - указывает, отображается ли компонент и, соответственно, иллюстрация на поверхности формы

Свойство `BorderStyle` - вид границы компонента:

- `None` — граница не отображается;
- `FixedSingle` — тонкая;
- `Fixed3D` — объемная.

Индивидуальные задания

Выберите индивидуальное задание из нижеприведенного списка согласно номеру в подгруппе. В качестве *f(x)* использовать по выбору: *sh(x)*, x^2, e^x. При проектировании интерфейса используйте несколько из описанных выше визуальных компонентов. В программе обязательно используйте динамическое создание/удаление визуальных компонентов и методы работы с объектами в C#.

1

$$a=\begin{cases}(f(x)+y)^2-\sqrt{f(x)y}, & xy\rangle 0\\ (f(x)+y)^2+\sqrt{|\mathrm{f(x)\ y}|}, & xy\langle 0\\ (f(x)+y)^2+1, & xy=0.\end{cases}$$

2

$$b=\begin{cases}\ln\left(f(x)\right)+\left(f(x)^2+y\right)^3, & x/y\rangle 0\\ \ln|f(x)/y|+\left(f(x)+y\right)^3, & x/y\langle 0\\ \left(f(x)^2+y\right)^3, & x=0\\ 0, & y=0.\end{cases}$$

3

$$c=\begin{cases}f(x)^2+y^2+\sin(y), & x-y=0\\ (f(x)-y)^2+\cos(y), & x-y\rangle 0\\ (y-f(x))^2+tg(y), & x-y\langle 0.\end{cases}$$

4

$$d=\begin{cases}(f(x)-y)^3+arctg(f(x)), & x\rangle y\\ (y-f(x))^3+arctg(f(x)), & y\rangle x\\ (y+f(x))^3+0.5, & y=x.\end{cases}$$

5

$$e=\begin{cases} i\sqrt{f(x)}, & i-\textit{нечетное},\ x\rangle 0 \\ i/2\sqrt{|f(x)|}, & i-\textit{четное},\ x\langle 0 \\ \sqrt{|if(x)|}, & \textit{иначе}. \end{cases}$$

6

$$g=\begin{cases} e^{f(x)-|b|}, & 0.5\langle xb\langle 10 \\ \sqrt{|f(x)+b|}, & 0.1\langle xb\langle 0.5 \\ 2f(x)^2, & \textit{иначе}. \end{cases}$$

7

$$s=\begin{cases} e^{f(x)}, & 1\langle xb\langle 10 \\ \sqrt{|f(x)+4*b|}, & 12\langle xb\langle 40 \\ bf(x)^2, & \textit{иначе}. \end{cases}$$

8

$$j=\begin{cases} \sin\left(5f(x)+3m|f(x)|\right), & -1\langle m\langle x \\ \cos\left(3f(x)+5m|f(x)|\right), & x\rangle m \\ \left(f(x)+m\right)^2, & x=m. \end{cases}$$

9

$$l=\begin{cases} 2f(x)^3+3p^2, & x\rangle|p| \\ |f(x)-p|, & 3\langle x\langle|p| \\ \left(f(x)-p\right)^2, & x=|p|. \end{cases}$$

10

$$k=\begin{cases} \ln\left(|f(x)|+|q|\right), & |xq|\rangle 10 \\ e^{f(x)+q}, & |xq|\langle 10 \\ f(x)+q, & |xq|=10 \end{cases}$$

11 $$m = \frac{\max\left(f(x), y, z\right)}{\min\left(f(x), y\right)} + 5.$$

12 $$n = \frac{\min\left(f(x) + y, y - z\right)}{\max\left(f(x), y\right)}.$$

13 $$p = \frac{\left|\min\left(f(x), y\right) - \max\left(y, z\right)\right|}{2}.$$

14 $$q = \frac{\max\left(f(x) + y + z, xyz\right)}{\min\left(f(x) + y + z, xyz\right)}.$$

15 $$r = \max\left(\min\left(f(x), y\right), z\right).$$

Порядок выполнения лабораторной работы

1) Запустите среду разработки Visual Studio.

2) Визуально спроектируйте форму основной программы в соответствие с индивидуальным заданием.

3) Выполните отладку и компиляцию программы. При необходимости исправьте ошибки компиляции.

4) Протестируйте программу на наличие программных ошибок.

5) Выполните отчет по практической работе.

Контрольные вопросы

1. Что понимается под термином «класс»?
2. Какие элементы определяются в составе класса?
3. Каково соотношение понятий «класс» и «объект»?
4. Что понимается под термином «члены класса»?
5. Какие члены класса Вам известны?
6. Какие члены класса содержат код?
7. Какие члены класса содержат данные?
8. Что понимается под термином «конструктор»?
9. Приведите синтаксис описания класса в общем виде.
10. Какие модификаторы типа доступа Вам известны?
11. В чем заключаются особенности доступа членов класса с модификатором public?
12. В чем заключаются особенности доступа членов класса с модификатором private?
13. В чем заключаются особенности доступа членов класса с модификатором protected?
14. В чем заключаются особенности доступа членов класса с модификатором internal?
15. Какое ключевое слово языка C# используется при создании объекта?
16. Приведите синтаксис создания объекта в общем виде.
17. Поясните динамическое добавление элементов.
18. Поясните общие характеристики и различия у компонентов Form, GroupBox, Panel, FlowLayoutPanel, TableLayoutPanel.
19. Поясните использование компонента класса DataGridView.
20. Поясните использование компонента класса PictureBox.

ПРАКТИЧЕСКАЯ РАБОТА № 6

Тема: «Знакомство с компонентами для создания меню и других элементов группировки навигации.»

Цель работы – Познакомится с некоторыми полезными компонентами, в том числе для создания меню и других элементов навигации в Windows приложениях. Рассмотреть взаимодействие форм в приложении.

Теоретические сведения

6.1. Размеры элементов и их позиционирование

С помощью свойства `Location` (в окне Свойств – рис. 6.1), задаются координаты верхнего левого угла элемента относительно контейнера, в котором размещен элемент.

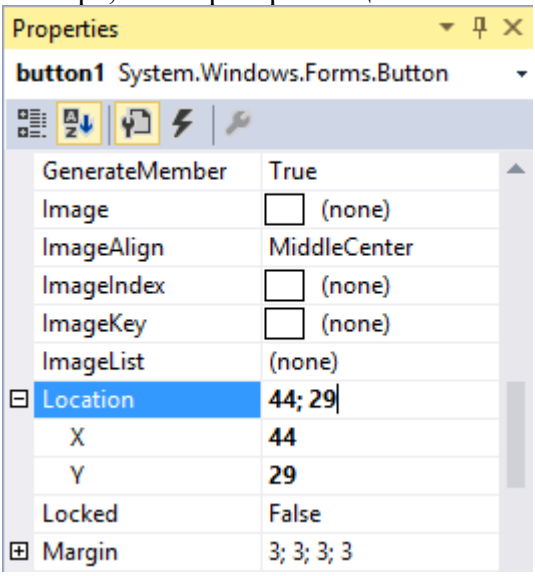


Рис. 6.1 – свойство Location

В момент переноса элемента с панели инструментов на форму данное свойство устанавливается автоматически. В коде так же можно установить значение программно:

```
private void Form1_Load(object sender, EventArgs e) {
     button1.Location = new Point(150, 150);
}
```

Размеры элемента задаются с помощью свойства `Size`:

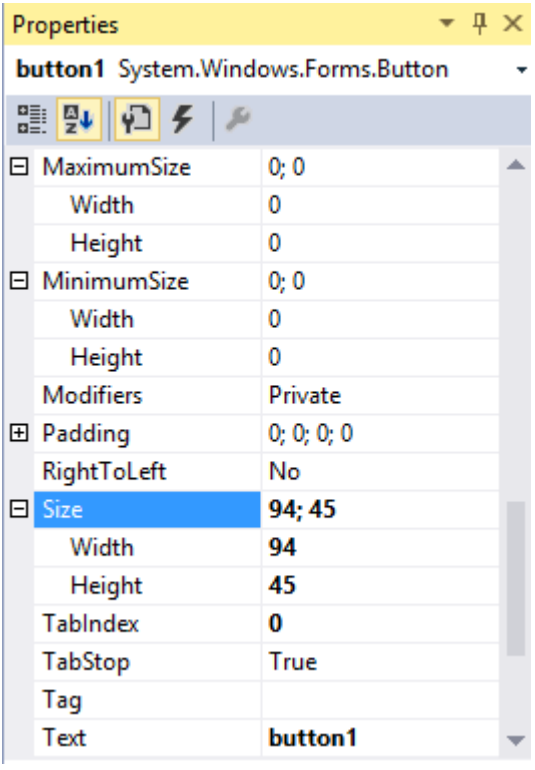


Рис. 6.2 – свойство Size

Дополнительные свойства `MaximumSize` и `MinimumSize` позволяют ограничить минимальный и максимальный размеры. В коде устанавливаются размеры следующим образом:

```
button1.Size = new Size { Width = 100, Height = 50 };
// установка свойств по отдельности
button1.Width = 150; button1.Height = 75;
```

С помощью свойства `Anchor` (рис. 6.3) определяется расстояние между одной из сторон элемента и стороной контейнера. По умолчанию у каждого добавляемого элемента это свойство равно `Top`, `Left`:

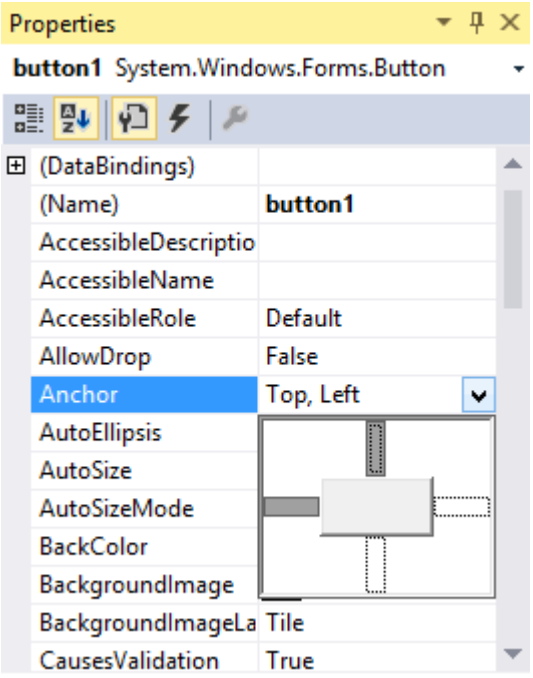


Рис. 6.3 – свойство Anchor

При растяжении формы влево или вверх, элемент сохраняет расстояние от левой и верхней границы элемента до границ контейнера. Можно задать четыре возможных значения для этого свойства или их комбинацию: `Top`, `Bottom`, `Left`, `Right`. Чтобы программно задать это свойство в коде, используется перечисление `AnchorStyles`:

```
button1.Anchor = AnchorStyles.Left;
// установка комбинации значений
button1.Anchor = AnchorStyles.Right | AnchorStyles.Top;
```

Свойство `Dock` (рис. 6.4)прикрепляет элемент к определенной стороне контейнера:

- None: значение по умолчанию
- Top: элемент прижимается к верхней границе контейнера
- Bottom: элемент прижимается к нижней границе контейнера
- Left: элемент прижимается к левой стороне контейнера
- Right: элемент прикрепляется к правой стороне контейнера
- Fill: элемент заполняет все пространство контейнера.

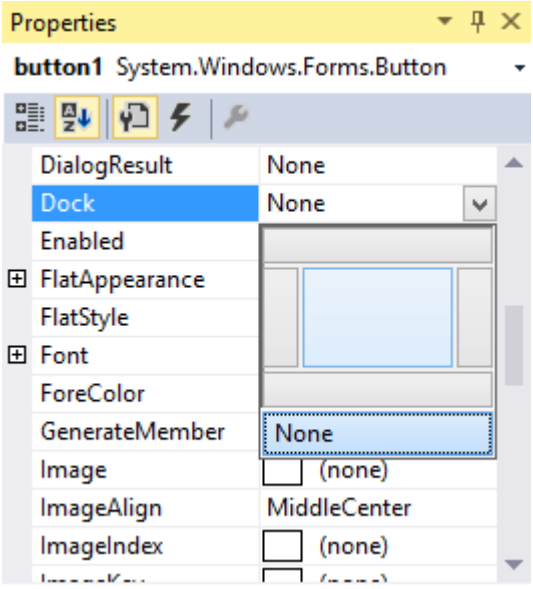


Рис. 6.4 – свойство Dock

6.2. Панель вкладок.

Компонент `TabControl` позволяет создать элемент управления в виде вкладок. Каждая вкладка является контейнером для размещения некоторого набора элементов управления, таких как кнопки, текстовые поля и др. Каждая вкладка представлена

классом `TabPage`. При переносе элемента `TabControl` с панели инструментов на форму по умолчанию создаются две вкладки - tabPage1 и tabPage2. Для настройки вкладки элемента `TabControl` используется свойство `TabPages`:

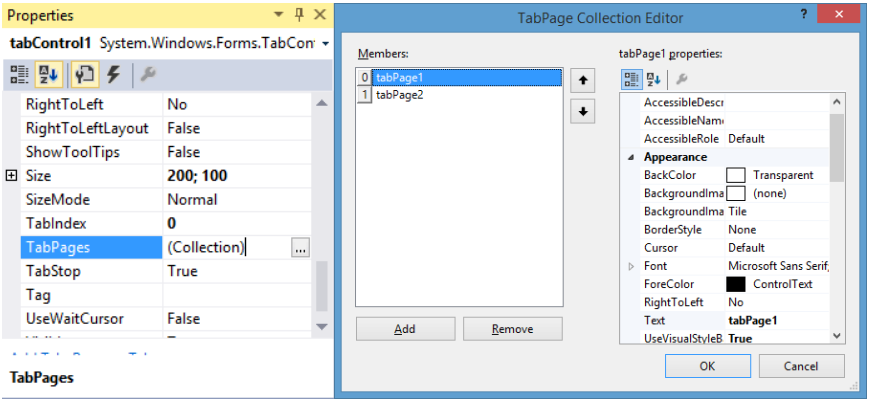


Рис. 6.5 – свойства и настройки TabControl

Каждая вкладка является панелью (рис. 6.6), на которую можно добавлять другие элементы управления, и заголовок, с помощью которого происходит переключение по вкладкам. Текст заголовка задается с помощью свойства `Text`.

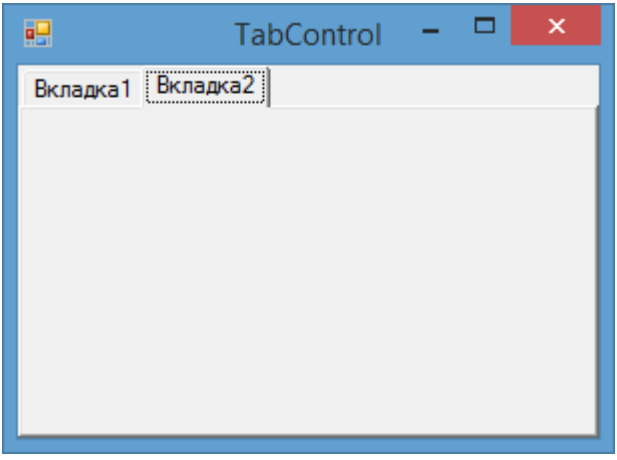


Рис. 6.6 – Результат применения компонента TabControl

Для программной манипуляции вкладками используется коллекция `tabControl1`:

```
//добавление вкладки
TabPage newTabPage = new TabPage();
newTabPage.Text = "My Tab";
tabControl1.TabPages.Add(newTabPage);
// удаление вкладки
// по индексу
tabControl1.TabPages.RemoveAt(0);
// по объекту
tabControl1.TabPages.Remove(newTabPage);
// изменение свойств
tabControl1.TabPages[0].Text = "First tab";
```

Компонент SplitContainer (рис. 6.7) позволяет создавать две разделенные сплитером панели:

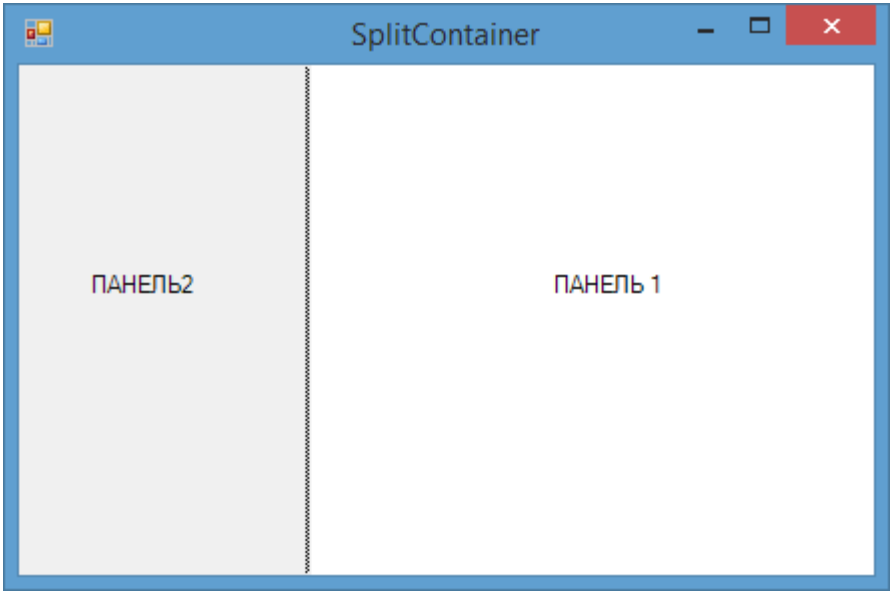


Рис. 6.7 – Компонент SplitContainer

Свойство `Orientation` задает горизонтальное или вертикальное отображение сплитера на форме (значения `Horisontal` и `Vertical` соответственно).

Для запрещения изменения положения сплиттера свойству `IsSplitterFixed` необходимо установить значение `true`. Свойство `FixedPanel` позволяет задать за одной панелью фиксированную ширину (при вертикальной ориентации сплиттера) или высоту (при горизонтальной ориентации сплиттера).

`SplitterDistance` позволяет изменить положение сплитера в коде, которое задает положение сплиттера в пикселях от левого или верхнего края элемента `SplitContainer`. Свойство `SplitterIncrement` задает шаг, на который будет перемещаться сплиттер при движении его с помощью клавиш-стрелок.

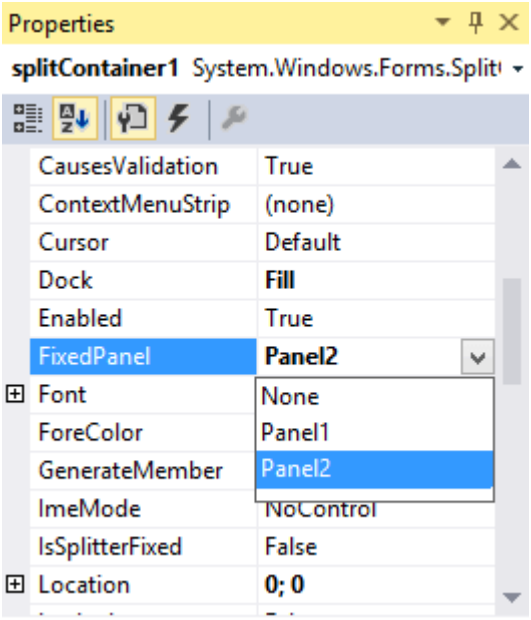


Рис. 6.8 – Свойства компонента SplitContainer

Чтобы скрыть одну из двух панелей, мы можем установить свойство `Panel1Collapsed` или `Panel2Collapsed` в `true`.

6.3. Меню и панели инструментов

6.3.1 Компонент класса ToolStrip

Элемент `ToolStrip` (рис. 6.9) представляет панель инструментов. Каждый отдельный элемент на этой панели является объектом `ToolStripItem`.

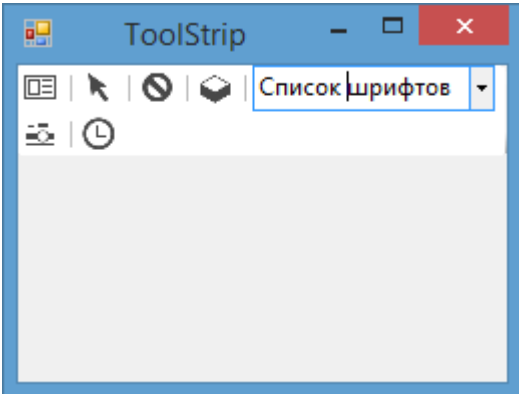


Рис. 6.9 – компонент класса ToolStrip

Основные свойства компонента ToolStrip связаны с его позиционированием на форме:

- **Dock**: прикрепляет панель инструментов к одной из сторон формы;
- **LayoutStyle**: задает ориентацию панели на форме (горизонтальная, вертикальная, табличная);
- **ShowItemToolTips**: указывает, будут ли отображаться всплывающие подсказки для отдельных элементов панели инструментов;
- **Stretch**: позволяет растянуть панель по всей длине контейнера.

В зависимости от значения свойства `LayoutStyle` панель инструментов может располагаться по горизонтали, или в табличном виде:

- **HorizontalStackWithOverflow**: расположение по горизонтали с переполнением[27];
- **StackWithOverflow**: элементы располагаются автоматически с переполнением;
- **VerticalStackWithOverflow**: элементы располагаются вертикально с переполнением;
- **Flow**: элементы располагаются автоматически, но без переполнения[28];
- **Table**: элементы позиционируются в виде таблицы.

Для значений HorizontalStackWithOverflow и VerticalStackWithOverflow, с помощью свойства `CanOverflow` можно задать поведение при переполнении. Если это свойство установлено в true (значение по умолчанию), то для элементов, не попадающих в границы ToolStrip, создается выпадающий список (рис. 6.10).

Панель ToolStrip может содержать объекты следующих классов:

- **ToolStripLabel**: текстовая метка на панели инструментов, представляет функциональность элементов Label и LinkLabel;
- **ToolStripButton**: аналогичен элементу Button.Также имеет событие Click, с помощью которого можно обработать нажатие пользователя на кнопку;
- **ToolStripSeparator**: визуальный разделитель между другими элементами на панели инструментов;

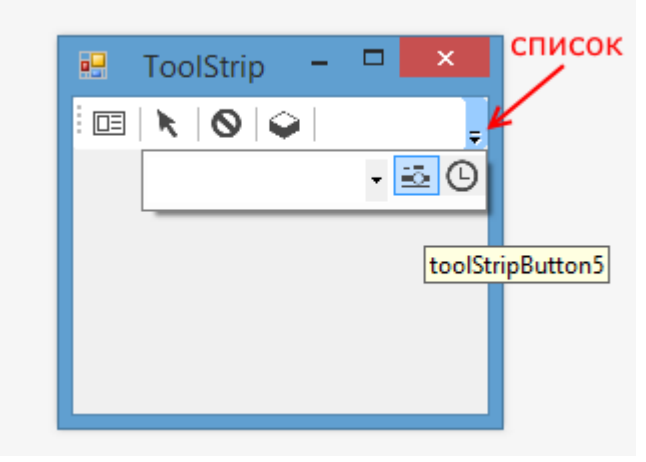
---

[27] Если длина панели превышает длину контейнера, то новые элементы, выходящие за границы контейнера, не отображаются, то есть панель переполняется элементами

[28] Если длина панели меньше длины контейнера, то выходящие за границы элементы переносятся, а панель инструментов растягивается, чтобы вместить все элементы

- **ToolStripToolStripComboBox**: подобен стандартному элементу ComboBox;
- **ToolStripTextBox**: аналогичен текстовому полю TextBox;
- **ToolStripProgressBar**: индикатор прогресса, как и элемент ProgressBar;
- **ToolStripDropDownButton**: представляет кнопку, по нажатию на которую открывается выпадающее меню.

Рис. 6.10 – выпадающий список ToolStrip



К каждому элементу выпадающего меню дополнительно можно прикрепить обработчик нажатия и обработать клик по этим пунктам меню. ToolStripSplitButton: объединяет функциональность ToolStripDropDownButton и ToolStripButton.
Добавить новые элементы можно в режиме дизайнера (рис. 6.11):

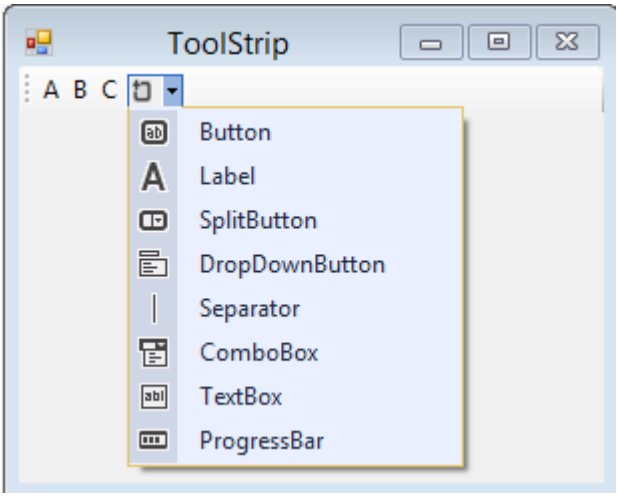


Рис. 6.11 – Режим визуального проектирования ToolStrip

При программном добавлении новых элементов, их расположение на панели инструментов будет соответствовать порядку добавления. Все элементы хранятся в ToolStrip в свойстве Items. Можно добавить в него любой объект класса ToolStripItem:

```
public partial class Form1 : Form {
    public Form1() {
        InitializeComponent();
         ToolStripButton clearBtn = new
ToolStripButton();
        clearBtn.Text = "Clear";
        // устанавливаем обработчик нажатия
        clearBtn.Click += btn_Click;
        toolStrip1.Items.Add(clearBtn);
    }
    void btn_Click(object sender, EventArgs e) {
        MessageBox.Show("Удаление");
    }
}
```

Элементы ToolStripButton, ToolStripDropDownButton и ToolStripSplitButton могут отображать как текст, так и изобра-

жения. Для управления размещением изображений в этих элементах имеются следующие свойства:

- **DisplayStyle**: определяет, будет ли отображаться на элементе текст, или изображение, или и то и другое;
- **Image**: указывает на изображение;
- **ImageAlign**: устанавливает выравнивание изображения относительно элемента;
- **ImageScaling**: указывает, будет ли изображение растягиваться, чтобы заполнить все пространство элемента;
- **ImageTransparentColor**: указывает, будет ли цвет изображения прозрачным.

Для текстового отображения надо указать значение Text, либо можно комбинировать два значения с помощью другого значения ImageAndText.

Все описанные выше значения хранятся в перечислении ToolStripItemDisplayStyle. Также можно установить свойства в коде c#:

```
ToolStripButton clearBtn = new ToolStripButton();
clearBtn.Text = "Поиск";
clearBtn.DisplayStyle =
ToolStripItemDisplayStyle.ImageAndText;
clearBtn.Image =
Image.FromFile(@"C:\Icons\search.png");
// добавляем на панель инструментов
toolStrip1.Items.Add(clearBtn);
```

6.3.2 Компонент класса MenuStrip

Для создания меню в Windows Forms применяется элемент `MenuStrip`. Данный класс унаследован от ToolStrip. Наиболее важные свойства компонента MenuStrip:

- **Dock**: прикрепляет меню к одной из сторон формы;
- **LayoutStyle**: задает ориентацию панели меню на форме. Может также, как и с ToolStrip, принимать следующие значения
- **Table**: элементы позиционируются в виде таблицы;

- **ShowItemToolTips**: указывает, будут ли отображаться всплывающие подсказки для отдельных элементов меню;
- **Stretch**: позволяет растянуть панель по всей длине контейнера;
- **TextDirection**: задает направление текста.

MenuStrip выступает своего рода контейнером для отдельных пунктов меню, которые представлены объектом ToolStripMenuItem.

Добавить новые элементы в меню можно в режиме дизайнера:

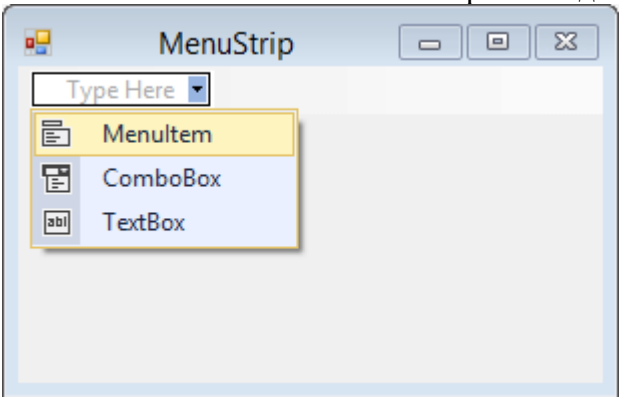


Рис. 6.12 – Режим визуального проектирования MenuStrip

Для добавления доступно три вида элементов: MenuItem (объект ToolStripMenuItem), ComboBox и TextBox. Меню же обычно содержит набор объектов ToolStripMenuItem.

Также мы можем добавить пункты меню в коде C#:

```
public partial class Form1 : Form {
    public Form1() {
        InitializeComponent();
        ToolStripMenuItem fileItem = new
ToolStripMenuItem("Файл");
        fileItem.DropDownItems.Add("Создать");
        fileItem.DropDownItems.Add(new
ToolStripMenuItem("Сохранить"));
```

```
        menuStrip1.Items.Add(fileItem);
        ToolStripMenuItem aboutItem = new
ToolStripMenuItem("О программе");
        aboutItem.Click += aboutItem_Click;
        menuStrip1.Items.Add(aboutItem);
    }
     void aboutItem_Click(object sender, EventArgs e) {
        MessageBox.Show("О программе");
    }
}
```

ToolStripMenuItem в конструкторе принимает текстовую метку, которая будет использоваться в качестве текста меню. Каждый подобный объект имеет коллекцию DropDownItems, которая хранит дочерние объекты ToolStripMenuItem. То есть один элемент ToolStripMenuItem может содержать набор других объектов ToolStripMenuItem. И таким образом, образуется иерархическое меню или структура в виде дерева (рис. 6.13).

Если передать при добавление строку текста, то для нее неявным образом будет создан объект ToolStripMenuItem: fileItem.DropDownItems.Add("Создать")

Назначив обработчики для события Click, мы можем обработать нажатия на пункты меню: aboutItem.Click += aboutItem_Click

Свойство CheckOnClick при значении true позволяет на клику отметить пункт меню. А с помощью свойства Checked можно установить, будет ли пункт меню отмечен при запуске программы.

Еще одно свойство CheckState возвращает состояние пункта меню - отмечен он или нет. Оно может принимать три значения: Checked (отмечен), Unchecked (неотмечен) и Indeterminate (в неопределенном состоянии)

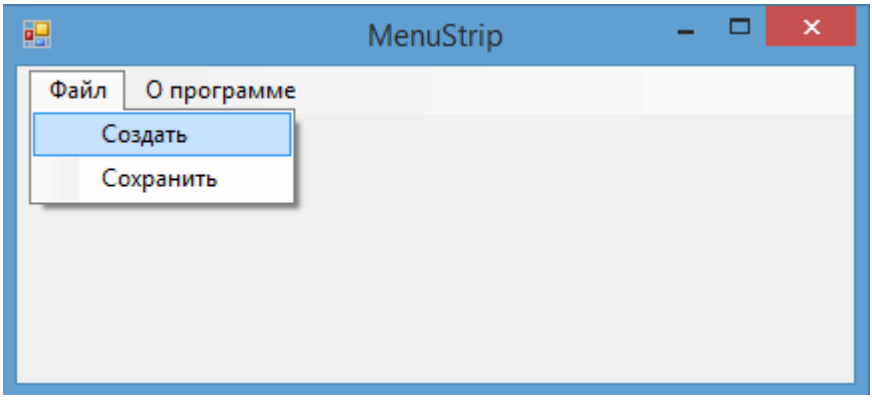


Рис. 6.13 – Результат создания меню

Например, создадим ряд отмеченных пунктов меню и обработаем событие установки / снятия отметки:

```
public partial class Form1 : Form {
    public Form1() {
        InitializeComponent();
         ToolStripMenuItem fileItem = new
ToolStripMenuItem("Файл");
         ToolStripMenuItem newItem = new
ToolStripMenuItem("Создать") { Checked = true,
CheckOnClick = true };
        fileItem.DropDownItems.Add(newItem);
        ToolStripMenuItem saveItem = new
ToolStripMenuItem("Сохранить") { Checked = true,
CheckOnClick = true };
        saveItem.CheckedChanged +=
menuItem_CheckedChanged;
        fileItem.DropDownItems.Add(saveItem);
        menuStrip1.Items.Add(fileItem);
    }
    void menuItem_CheckedChanged(object sender,
EventArgs e) {
        ToolStripMenuItem menuItem = sender as
ToolStripMenuItem;
        if (menuItem.CheckState == CheckState.Checked)
```

```
            MessageBox.Show("Отмечен");
        else if (menuItem.CheckState ==
CheckState.Unchecked)
            MessageBox.Show("Отметка снята");
    }
}
```

Если нам надо быстро обратиться к какому-то пункту меню, то мы можем использовать клавиши быстрого доступа. Для задания клавиш быстрого доступа используется свойство ShortcutKeys:

```
public partial class Form1 : Form {
    public Form1() {
        InitializeComponent();
        ToolStripMenuItem fileItem = new
ToolStripMenuItem("Файл");
        ToolStripMenuItem saveItem = new
ToolStripMenuItem("Сохранить") { Checked = true,
CheckOnClick = true };
        saveItem.Click+=saveItem_Click;
        saveItem.ShortcutKeys = Keys.Control | Keys.P;
        fileItem.DropDownItems.Add(saveItem);
        menuStrip1.Items.Add(fileItem);
    }
    void saveItem_Click(object sender, EventArgs e) {
        MessageBox.Show("Сохранение");
    }
}
```

Клавиши задаются с помощью перечисления Keys. В данном случае по нажатию на комбинацию клавиш Ctrl + P, будет срабатывать нажатие на пункт меню "Сохранить".

С помощью изображений мы можем разнообразить внешний вид пунктов меню. Для этого мы можем использовать следующие свойства:

- **DisplayStyle**: определяет, будет ли отображаться на элементе текст, или изображение, или и то и другое.
- **Image**: указывает на само изображение

- **ImageAlign**: устанавливает выравнивание изображения относительно элемента
- **ImageScaling**: указывает, будет ли изображение растягиваться, чтобы заполнить все пространство элемента
- **ImageTransparentColor**: указывает, будет ли цвет изображения прозрачным

Если изображение для пункта меню устанавливает в режиме дизайнера, то необходимо выбрать пункт Image, после чего откроется окно для импорта ресурса изображения в проект.

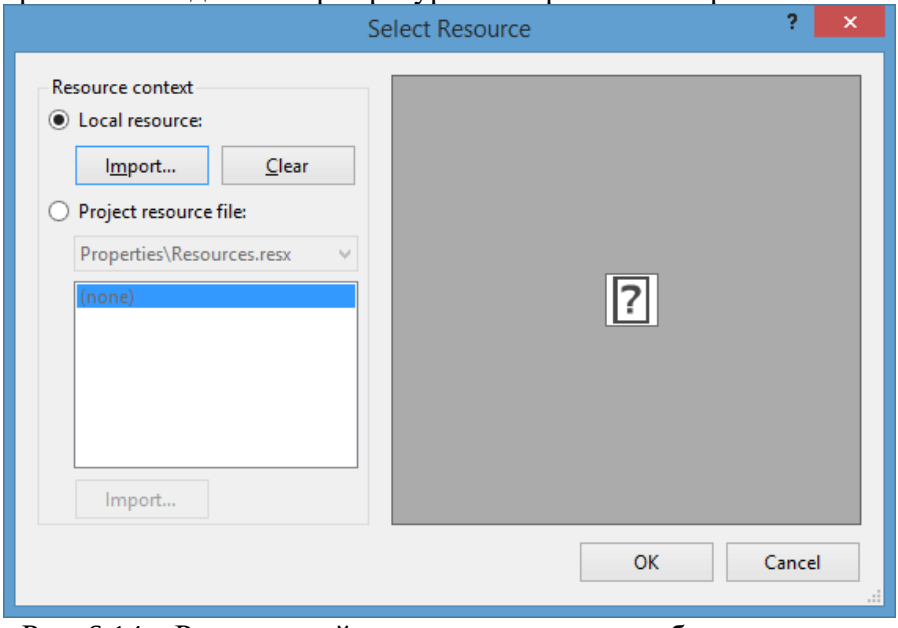


Рис. 6.14 – Режим дизайнера для установки изображения в меню.

Чтобы указать, как разместить изображение, у свойства DisplayStyle надо установить значение Image. Если необходимо, чтобы кнопка отображала только текст, то надо указать значение Text, либо можно комбинировать два значения с помощью

другого значения ImageAndText. По умолчанию изображение размещается слева от текста (рис. 6.15):

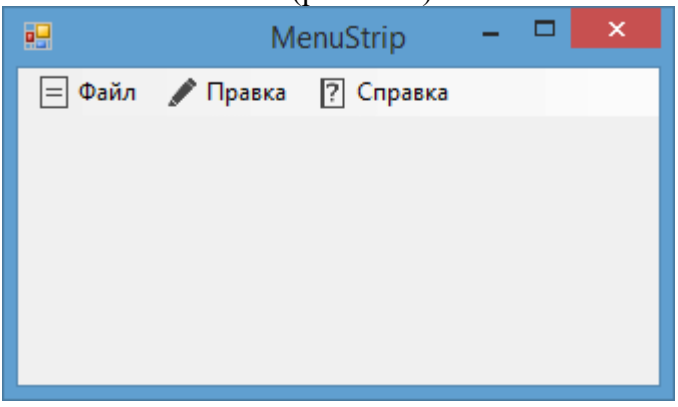


Рис. 6.15 – Оформление элементов меню.

Также можно установить изображение динамически в коде:

```
fileToolStripMenuItem.Image =
Image.FromFile(@"D:\Icons\0023\block32.png");
```

6.3.3 Компонент класса StatusStrip

StatusStrip представляет строку состояния, во многом аналогичную панели инструментов ToolStrip. Строка состояния предназначена для отображения текущей информации о состоянии работы приложения.

При добавлении на форму StatusStrip автоматически размещается в нижней части окна приложения (как и в большинстве приложений). Однако при необходимости мы сможем его иначе позиционировать, управляя свойством Dock, которое может принимать следующие значения:

- **Bottom**: размещение внизу (значение по умолчанию)
- **Top**: прикрепляет статусную строку к верхней части формы
- **Fill**: растягивает на всю форму
- **Left**: размещение в левой части формы
- **Right**: размещение в правой части формы

- **None**: произвольное положение

StatusStrip (рис. 6.16) может содержать различные элементы.

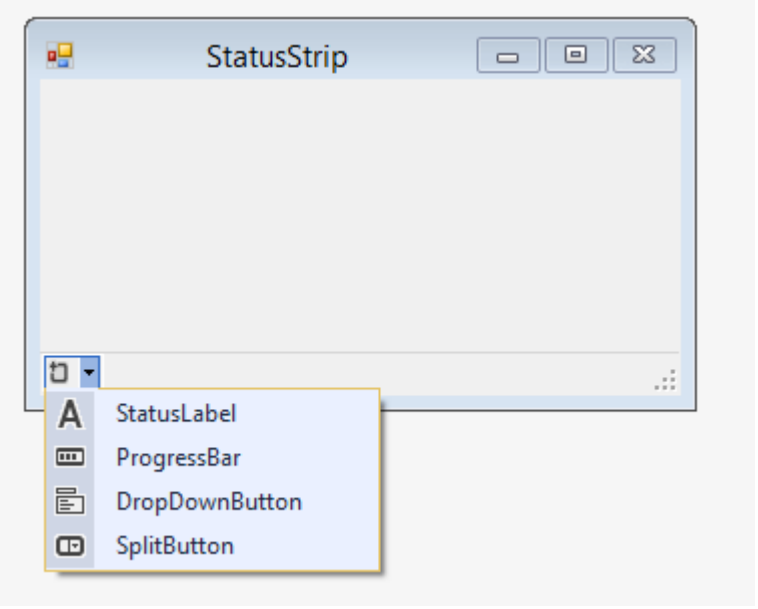


Рис. 6.16 – StatusStrip в режиме дизайнера.

В режиме дизайнера можно добавить следующие типы элементов:

- **StatusLabel**: метка для вывода текстовой информации. Представляет объект ToolStripLabel
- **ProgressBar**: индикатор прогресса. Представляет объект ToolStripProgressBar
- **DropDownButton**: кнопка с выпадающим списком по клику. Представляет объект ToolStripDropDownButton

- **SplitButton**: еще одна кнопка, во многом аналогичная DropDownButton. Представляет объект ToolStripSplitButton

Также можно управлять элементами программно.

```
public partial class Form1 : Form {
    ToolStripLabel dateLabel;
    ToolStripLabel timeLabel;
    ToolStripLabel infoLabel;
    Timer timer;
    public Form1() {
        InitializeComponent();
        infoLabel = new ToolStripLabel();
        infoLabel.Text = "Текущие дата и время:";
        dateLabel = new ToolStripLabel();
        timeLabel = new ToolStripLabel();
        statusStrip1.Items.Add(infoLabel);
        statusStrip1.Items.Add(dateLabel);
        statusStrip1.Items.Add(timeLabel);
        timer = new Timer() { Interval = 1000 };
        timer.Tick += timer_Tick;
        timer.Start();
    }
    void timer_Tick(object sender, EventArgs e) {
        dateLabel.Text =
DateTime.Now.ToLongDateString();
        timeLabel.Text =
DateTime.Now.ToLongTimeString();
    }
}
```

Здесь создаются три метки на строке состояния и таймер. После создания формы таймер запускается, и срабатывает его событие Tick, в обработчике которого устанавливаем текст меток (рис. 6.17).

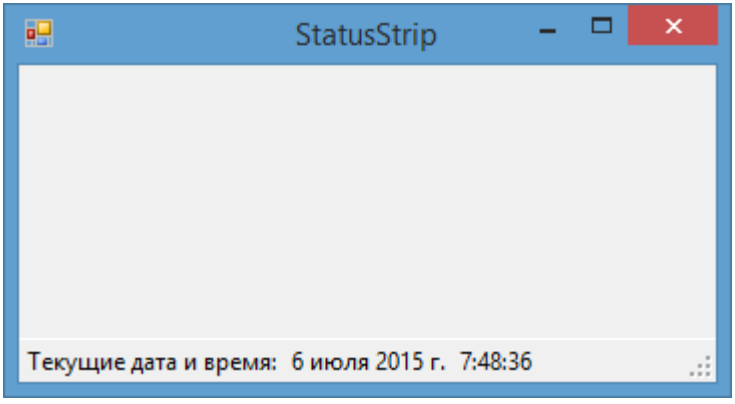


Рис. 6.17 – Пример работы компонента StatusStrip.

6.3.4 Компонент класса ContextMenuStrip

ContextMenuStrip представляет контекстное меню. Данный компонент во многом аналогичен элементы MenuStrip за тем исключением, что контекстное меню не может использоваться само по себе, оно обязательно применяется к какому-нибудь другому элементу, например, текстовому полю. Новые элементы в контекстное меню можно добавить в режиме дизайнера:

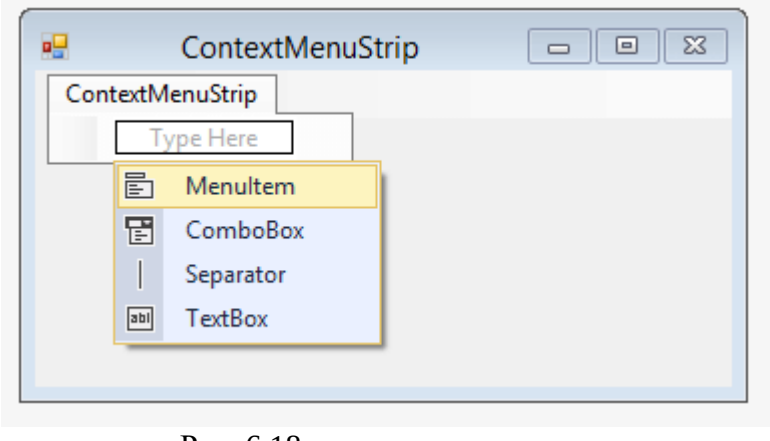


Рис. 6.18 – контекстное меню.

Свойство ContextMenuStrip позволяет ассоциировать контекстное меню с данным элементом. В случае с TextBox ассоциация происходит следующим образом:
textBox1.ContextMenuStrip = contextMenuStrip1.
И по нажатию на текстовое поле правой кнопкой мыши мы сможем вызвать ассоциированное контекстное меню.
Ниже показан пример программы в которой выполняется простейшая реализация функциональности copy-paste. В меню добавляется два элемента. А у текстового поля устанавливается многострочность, и оно растягивается по ширине контейнера.

```
public partial class Form1 : Form {
    string buffer;
    public Form1() {
        InitializeComponent();
        textBox1.Multiline = true;
        textBox1.Dock = DockStyle.Fill;
        // создаем элементы меню
        ToolStripMenuItem copyMenuItem = new
ToolStripMenuItem("Копировать");
        ToolStripMenuItem pasteMenuItem = new
ToolStripMenuItem("Вставить");
        // добавляем элементы в меню
        contextMenuStrip1.Items.AddRange(new[] {
copyMenuItem, pasteMenuItem });
        // ассоциируем контекстное меню с текстовым
полем
        textBox1.ContextMenuStrip = contextMenuStrip1;
        // устанавливаем обработчики событий для меню
        copyMenuItem.Click += copyMenuItem_Click;
        pasteMenuItem.Click += pasteMenuItem_Click;
    }
    // вставка текста
    void pasteMenuItem_Click(object sender, EventArgs
e) {
        textBox1.Paste(buffer);
    }
    // копирование текста
    void copyMenuItem_Click(object sender, EventArgs e)
{
```

```
        // если выделен текст в текстовом поле, то
копируем его в буфер
        buffer = textBox1.SelectedText;
    }
}
```

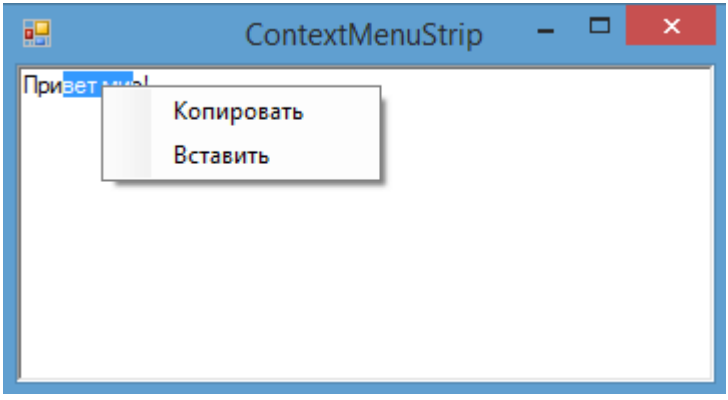


Рис. 6.19 – пример работы контекстного меню.

6.4 Обзор дополнительных полезных компонентов и возможностей.

6.4.1 Компонент класса Timer

Компонент класса Timer (рис. 6.20) служит для периодического выполнения некоторого набора команд.

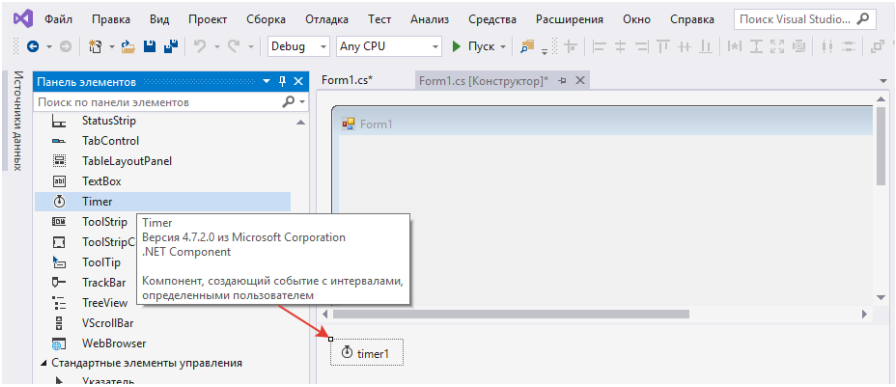


Рис. 6.20 – Использование компонента класса Timer.

Таймер создает низкоприоритетный поток команд, поэтому реальная максимальная частота срабатывания таймера около 30 раз в секунду. Некоторые свойства и события данного компонента:

- `Enabled` (`bool`) – свойство, определяющее состояние таймера. Значение `true` соответствует активному состоянию таймера, значение `false` – отключенному состоянию;
- `Interval` (`int`) – свойство, задающее периодичность срабатывания таймера в миллисекундах;
- `Tick` – событие, возникающее при срабатывании таймера. В данном событии описываются действия, которые требуется периодически выполнять.

6.4.2 Генератор случайных чисел

В библиотеке классов .NET Framework имеется класс Random (пространство имён System), который представляет собой генератор псевдослучайных чисел[29].

Текущая реализация Random класс основана на основе аддитивного генератора случайных чисел Дональда Кнута. Конструктор класса Random без параметов позволяет генерировать серию псевдослучайных чисел, начальный элемент которой строится на основе текущей даты и времени: Random Rnd = new Random();. Конструктор с параметром - public Random(int) обеспечивает возможность генерирования повторяющейся серии случайных чисел. Параметр конструктора используется для построения начального элемента серии случайных чисел.

Метод Next() перегружен (имеет три реализации) и при каждом вызове возвращает положительное целое, равномерно распределенное в некотором диапазоне, который и задается параметрами метода:

1) public int Next() - метод без параметров, выдает целые положительные числа во всем положительном диапазоне типа int ;

2) public int Next (int max) – метод с одним параметром целого типа, выдает целые положительные числа в диапазоне [0,max] ;

3) public int Next (int min, int max) - метод с двумя параметрами целого типа, выдает целые числа в диапазоне [min , max].

Метод public double NextDouble() - при каждом вызове выдает новое случайное число, равномерно распределенное в интервале [0,1].

---

[29] Алгоритмический метод получения случайных чисел, сгенерированных арифметическим способом. За эталон генератора случайных чисел принят такой генератор, который порождает последовательность случайных чисел с равномерным законом распределения в интервале (0; 1).

Метод public void NextBytes (byte[] buffer) - позволяет при одном обращении получать целую серию случайных чисел. Метод имеет параметр buffer - массив, который будет заполнен случайными числами в диапазоне [0, 255].

6.4.3 Компоненты классов OpenFileDialog, SaveFileDialog

Компоненты классов OpenFileDialog и SaveFileDialog представляют собой стандартные диалоговые окна, предназначенные для обеспечения возможности выбора файлов для загрузки и сохранения, соответственно. Компоненты имеют однотипные свойства, некоторые из которых приведены ниже:

1) AddExtension. Логическое свойство, определяющее, требуется ли добавлять расширение к имени файла, если у него расширение не задано;

2) DefaultExt. Строка, которая задает расширение файла «по умолчанию», т.е. расширение, которое автоматически подставляется к имени файла, если у него расширение не задано;

3) CheckFileExists. Логическое свойство, указывающее, должен ли существовать выбранный файл (как правило, установлено у диалогового окна открытия);

4) CheckPathExists. Логическое свойство, указывающее, должна ли существовать выбранная папка (как правило, установлено у диалоговых окон);

5) FileName. Строка, определяющая имя файла «по умолчанию» (до работы с диалоговым окном) и выбранный пользователем файл (после работы с диалоговым окном);

6) Filter. Определяет список типов файлов для быстрой фильтрации. Задается в виде строки, разделенной на блоки символами «|». Для одного фильтра требуется указать два блока: первый представляет собой текст, отображаемый пользователю, а второй – собственно фильтр отбора файлов. Например «Текстовые файлы (*.txt)|*.txt|Все файлы (*.*)|*.*» (без кавычек);

7) InitialDirectory. Строка, определяющая начальную папку, содержимое которой будет отображать диалоговое окно;

8) Title. Строка, задающая заголовок диалогового окна. У диалогового окна выбора файла для записи имеется свойство

9) OverwritePrompt (логическое), которое определяет необходимость выдачи дополнительного окна подтверждения при попытке перезаписать существующий файл.

6.4.4 Компоненты классов ColorDialog и FontDialog

ColorDialog (рис. 6.21) позволяет выбрать настройки цвета. Среди свойств ColorDialog следует отметить следующие:

- FullOpen: при значении true отображается диалоговое окно с расширенными настройками для выбора цвета;
- SolidColorOnly: при значении true позволяет выбирать только между однотонные оттенки цветов;
- Color: выбранный в диалоговом окне цвет.

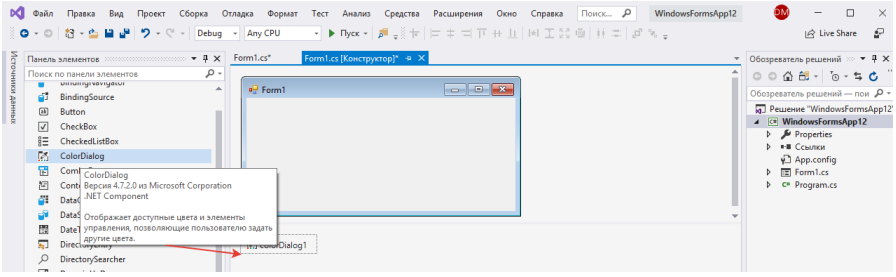


Рис. 6.21 – Использование компонента класса ColorDialog.

И при нажатии кнопку нам отобразится диалоговое окно, в котором можно установить цвет формы (рис. 6.22):

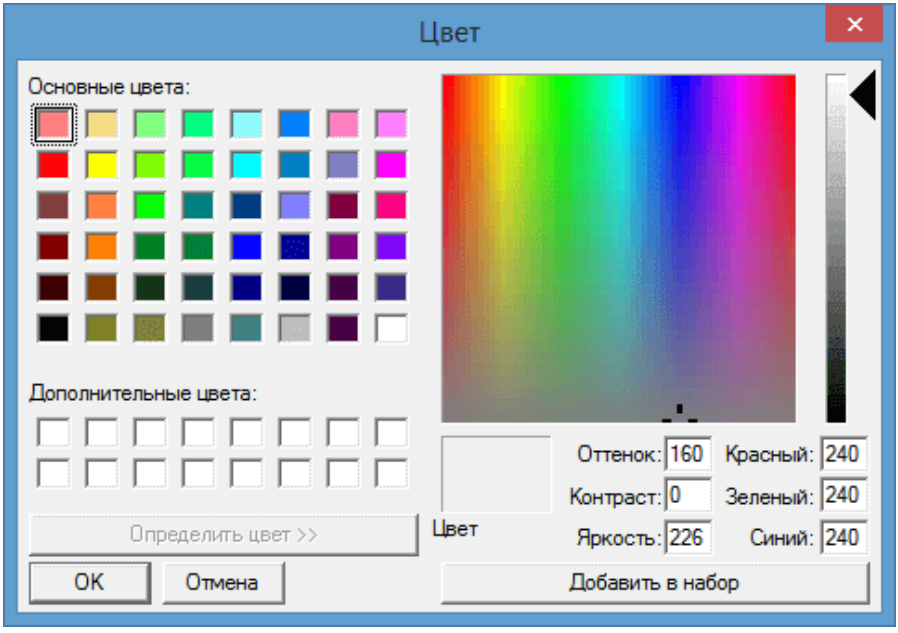


Рис. 6.22 – диалоговое окно ColorDialog.

Для выбора шрифта и его параметров используется FontDialog (рис. 6.23). Для его использования перенесем компонент с Панели инструментов на форму.

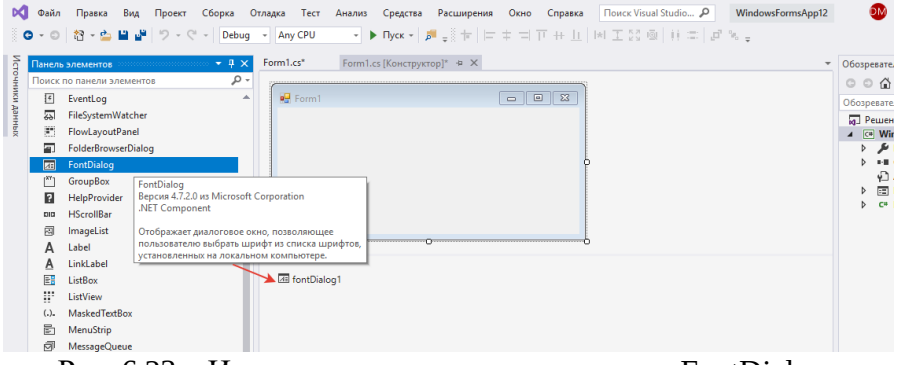


Рис. 6.23 – Использование компонента класса FontDialog.

FontDialog имеет ряд свойств, среди которых стоит отметить следующие:

- **ShowColor**: при значении true позволяет выбирать цвет шрифта
- **Font**: выбранный в диалоговом окне шрифт
- **Color**: выбранный в диалоговом окне цвет шрифта

Для отображения диалогового окна используется метод ShowDialog().

И если мы запустим приложение и нажмем на кнопку, то нам отобразится диалоговое окно, где мы можем задать все параметры шрифта. И после выбора установленные настройки будут применены к шрифту кнопки:

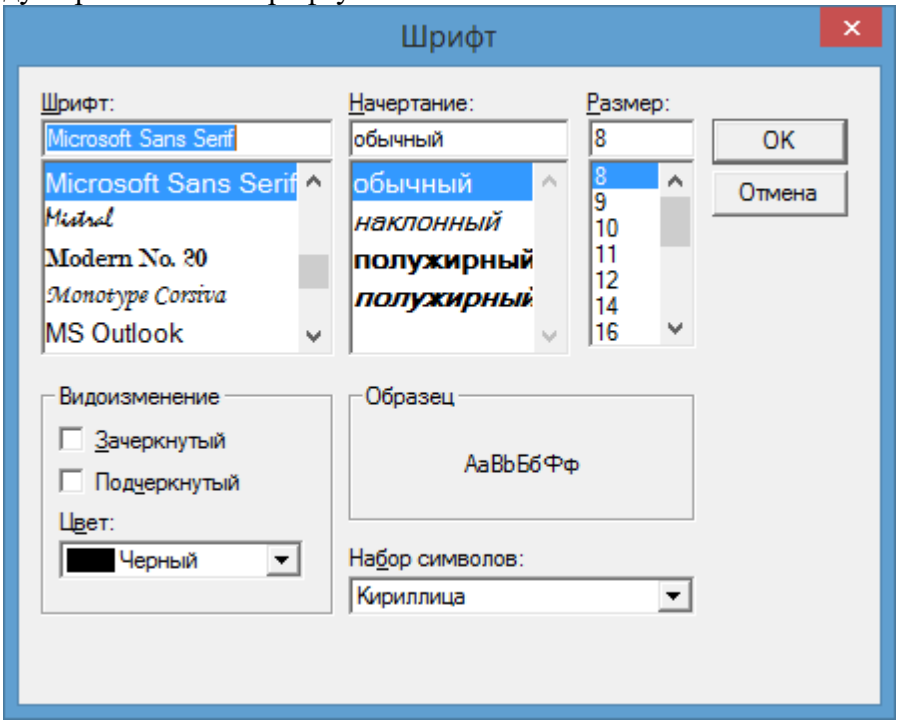


Рис. 6.24 – диалоговое окно FontDialog.

6.4.5 Компоненты класса WebBrowser

WebBrowser (рис. 6.25) предоставляет функции интернет-браузера, позволяя загружать и отображать контент из сети интернет. В то же время важно понимать, что данный элемент не является полноценным веб-браузером, и возможности по его настройки и изменению довольно ограничены.

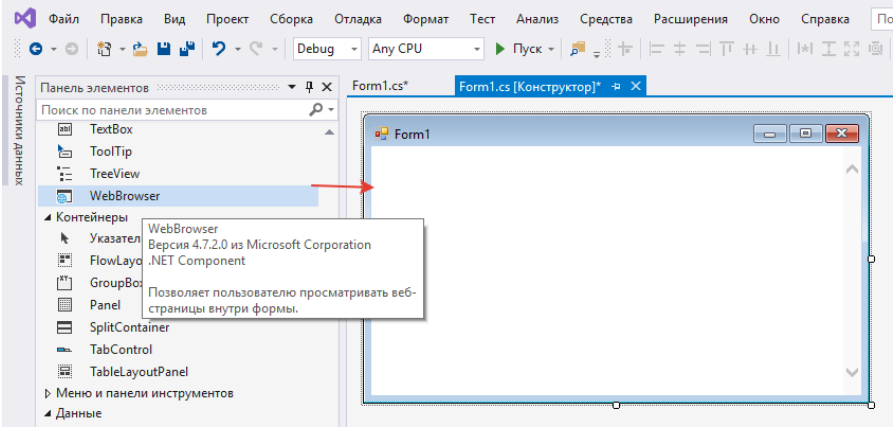


Рис. 6.25 – Использование компонента класса WebBrowser.

Рассмотрим основные его свойства:

- AllowWebBrowserDrop: при установке для данного свойства значения true можно будет с помощью мыши переносить документы в веб-браузер и открывать их.
- CanGoBack: определяет, может ли веб-браузер переходить назад по истории просмотров
- CanGoForward: определяет, может ли веб-браузер переходить вперед
- Document: возвращает открытый в веб-браузере документ
- DocumentText: возвращает текстовое содержание документа
- DocumentTitle: возвращает заголовок документа
- DocumentType: возвращает тип документа

- IsOffline: возвращает true, если отсутствует подключение к интернету
- ScriptErrorsSuppressed: указывает, будут ли отображаться ошибки javascript в диалоговом окне
- ScrollBarsEnabled: определяет, будет ли использоваться прокрутка
- URL: возвращает или устанавливает URL документа в веб-браузере

Кроме того, WebBrowser содержит ряд методов, которые позволяют осуществлять навигацию между документами:

- GoBack(): осуществляет переход к предыдущей странице в истории навигации (если таковая имеется)
- GoForward(): осуществляет переход к следующей странице в истории навигации
- GoHome(): осуществляет переход к домашней странице веб-браузера
- GoSearch(): осуществляет переход к странице поиска
- Navigate: осуществляет переход к определенному адресу в сети интернет

Таким образом, чтобы перейти к определенному документу, надо использовать метод Navigate:

```
// перейти к адресу в интернете
webBrowser1.Navigate("http://yandex.ru");
// открыть документ на диске
webBrowser1.Navigate("C://Images//24.png");
```

6.4.6 Компоненты классов DateTimePicker и MonthCalendar

Для работы с датами в Windows Forms имеются элементы DateTimePicker и MonthCalendar.

DateTimePicker (рис. 6.26) представляет раскрывающийся по нажатию календарь, в котором можно выбрать дату. собой эле-

мент, который с помощью перемещения ползунка позволяет вводить числовые значения.

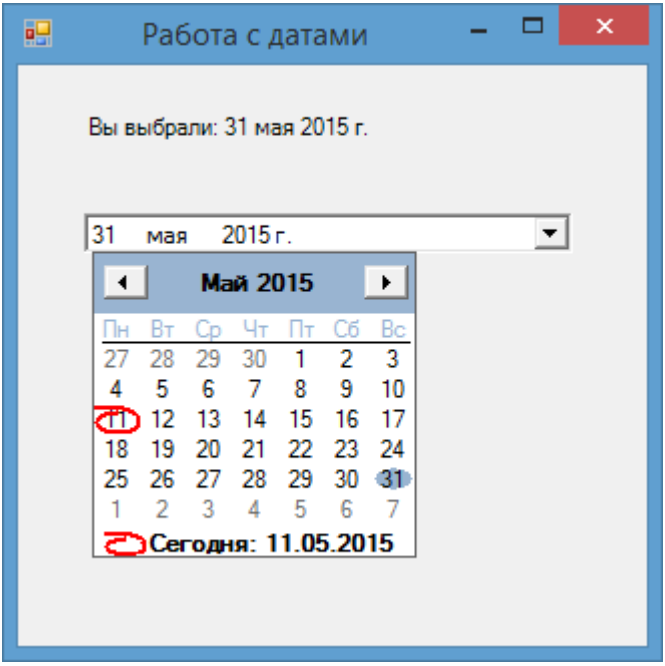


Рис. 6.26 – Использование компонента класса DateTimePicker.

Наиболее важные свойства DateTimePicker:

- Format: определяет формат отображения даты в элементе управления. Может принимать следующие значения:
- Custom: формат задается разработчиком
- Long: полная дата
- Short: дата в сокращенном формате
- Time: формат для работы с временем
- CustomFormat: задает формат отображения даты, если для свойства Format установлено значение Custom
- MinDate: минимальная дата, которую можно выбрать
- MaxDate: наибольшая дата, которую можно выбрать

- Value: определеяте текущее выбранное значение в DateTimePicker
- Text: представляет тот текст, который отображается в элементе

При выборе даты элемент генерирует событие ValueChanged. Например, обработаем данное собатие и присвоим выбранное значение тексту метки:

```
public partial class Form1 : Form {
    public Form1() {
        InitializeComponent();
        dateTimePicker1.Format =
DateTimePickerFormat.Time;

dateTimePicker1.ValueChanged+=dateTimePicker1_ValueChan
ged;
    }
    private void dateTimePicker1_ValueChanged(object
sender, EventArgs e) {
        label1.Text = String.Format("Вы выбрали: {0}",
dateTimePicker1.Value.ToLongTimeString());
    }
}
```

Свойство Value хранит объект DateTime, поэтому с ним можно работать, как и с любой другой датой. В данном случае выбранная дата преобразуется в строку времени.

С помощью MonthCalendar (рис. 6.27) также можно выбрать дату, только в данном случае этот элемент представляет сам календарь, который не надо раскрывать:

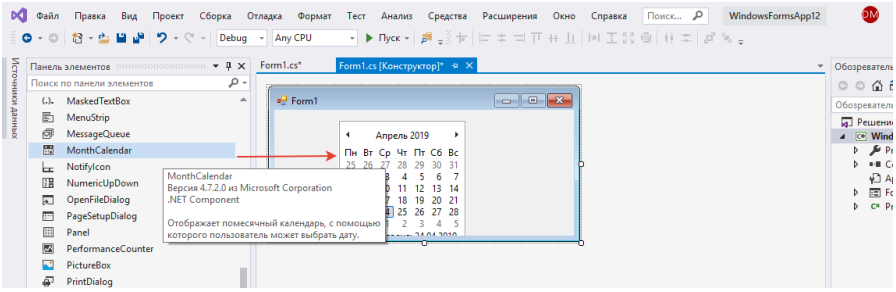


Рис. 6.27 – Использование компонента класса MonthCalendar.

Свойства выделения дат:

- AnnuallyBoldedDates: содержит набор дат, которые будут отмечены жирным в календаре для каждого года
- BoldedDates: содержит набор дат, которые будут отмечены жирным (только для текущего года)
- MonthlyBoldedDates: содержит набор дат, которые будут отмечены жирным для каждого месяца

Добавление выделенных дат делается с помощью определенных методов (как и удаление):

```
monthCalendar1.AddBoldedDate(new DateTime(2019, 05,
1));
monthCalendar1.AddBoldedDate(new DateTime(2019, 05,
2));
monthCalendar1.AddAnnuallyBoldedDate(new DateTime(2019,
05, 9));
monthCalendar1.AddMonthlyBoldedDate(new DateTime(2019,
05, 10));
```

Для снятия выделения можно использовать аналоги этих методов:

```
monthCalendar1.RemoveBoldedDate(new DateTime(2019, 05,
1));
monthCalendar1.RemoveBoldedDate(new DateTime(2019, 05,
2));
monthCalendar1.RemoveAnnuallyBoldedDate(new
DateTime(2019, 05, 9));
```

```
monthCalendar1.RemoveMonthlyBoldedDate(new
DateTime(2019, 05, 10));
```

Свойства для определения дат в календаре:

- MinDate: определяет минимальную дату для выбора в календаре
- MaxDate: задает наибольшую дату для выбора в календаре
- FirstDayOfWeek: определяет день недели, с которого должна начинаться неделя в календаре
- SelectionRange: определяет диапазон выделенных дат
- SelectionEnd: задает конечную дату выделения
- SelectionStart: определяет начальную дату выделения
- ShowToday: при значении true отображает внизу календаря текущую дату
- ShowTodayCircle: при значении true текущая дата будет обведена кружочком
- TodayDate: определяет текущую дату. По умолчанию используется системная дата на компьютере, но с помощью данного свойства мы можем ее изменить

Наиболее интересными событиями элемента являются события DateChanged и DateSelected, которые возникают при изменении выбранной в элементе даты.

```
void monthCalendar1_DateChanged(object sender,
DateRangeEventArgs e) {
   label1.Text = String.Format(Выборана дата: {0}",
e.Start.ToLongDateString());
}
```

6.4.7 Взаимодействие между формами

При использовании меню в приложениях, часто требуется более одной формы для отображения/управления данными. Чтобы добавить еще одну форму в проект, необходимо нажать на имя проекта в окне Solution Explorer (Обозреватель решений) правой кнопкой мыши и выбрать Add(Добавить)->Windows

Form (рис. 6.28). Для новой формы необходимо указать её имя, например, Form2.cs:

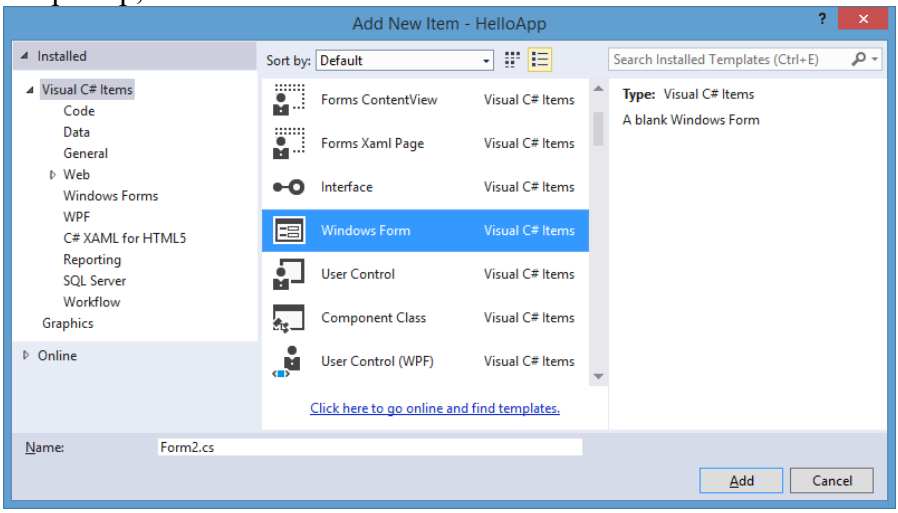


Рис. 6.28 – Добавление новой формы в проект.

При работе приложения необходимо осуществлять взаимодействие между этими формами.

Пусть первая форма по нажатию на кнопку будет вызывать вторую форму. С этой целью, добавим на первую форму Form1 кнопку и добавим в обработчик кода клика вызов второй формы (метод Show):

```
private void button1_Click(object sender, EventArgs e)
{
    Form2 newForm = new Form2();
    newForm.Show();
}
```

Следующим шагом является создание обратного перехода назад в первую форму. Для этого во второй форме необходимо передать сведения о первой форме. Данное действие можно осуществить с помощью передачи ссылки на форму в конструкторе.

```
namespace MyApp {
```

```
    public partial class Form2 : Form {
        public Form2() {
            InitializeComponent();
        }
         public Form2(Form1 f) {
            InitializeComponent();
            f.BackColor = Color.Yellow;
        }
    }
}
```

Здесь был добавлен здесь новый конструктор public Form2(Form1 f), в который передается ссылка на первую форму и описано действие установки ее фона в желтый цвет.

В коде первой формы напишем следующий код:

```
private void button1_Click(object sender, EventArgs e)
{
    Form2 newForm = new Form2(this);
    newForm.Show();
}
```

В данном примере ключевое слово this представляет ссылку на текущий объект - объект Form1, и при создании второй формы она будет получать ее (ссылку) и через нее управлять первой формой.

Теперь после нажатия на кнопку у будет создана вторая форма, которая сразу изменит цвет первой формы.

```
private void button1_Click(object sender, EventArgs e)
{
    Form1 newForm1 = new Form1();
    newForm1.Show();
    Form2 newForm2 = new Form2(newForm1);
    newForm2.Show();
}
```

При работе с несколькими формами надо учитывать, что одна из них является главной - которая запускается первой в файле Program.cs. Если в приложении одновременно открыто много различных форм, то при закрытии главной закрывается все приложение и вместе с ним все остальные формы.

Подробнее о способах передачи данных между формами можно посмотреть в электронном курсе, перейдя по ссылке http://moodle.sevsu.ru/mod/lesson/view.php?id=14011&pageid=14796.

Индивидуальные задания

Спроектируйте интерфейс приложения, используя несколько из описанных выше визуальных компонентов. В программе обязательно используйте более одной формы, создайте пользовательское меню, применяйте компоненты группировки элементов и их взаимодействие. При динамическом создании/удалении визуальных компонентов используйте методы работы с объектами в C#.

Порядок выполнения лабораторной работы

1) Запустите среду разработки Visual Studio.

2) Визуально спроектируйте форму основной программы в соответствие с индивидуальным заданием.

3) Выполните отладку и компиляцию программы. При необходимости исправьте ошибки компиляции.

4) Выполните отчет по практической работе.

Контрольные вопросы

1) Поясните особенности установки размеров элементов и их позиционирования.
2) Поясните особенность создания пользовательских меню.
3) Поясните особенность работы с статусной строкой.
4) Поясните особенность работы с таймером.
5) Для чего используются компоненты классов OpenFileDialog и SaveFileDialog?
6) Поясните особенность работы с компонентами выбора настроек цветов и шрифтов?
7) Для чего используются компонент WebBrowser?
8) Поясните возможности работы с датой и временем?

9) Как осуществляется взаимодействие между формами**?**

ПРАКТИЧЕСКАЯ РАБОТА № 7

Тема: «Технология Windows Presentation Foundation.»

Цель работы – Изучить возможности построения графических интерфейсов платформы Windows Presentation Foundation.

Теоретические сведения

7.1. Windows Presentation Foundation.

Windows Presentation Foundation (WPF) – это технология для построения клиентских приложений Windows, являющаяся частью платформы .NET. Технология WPF представляет собой подсистему для построения графических интерфейсов.

Основные преимущества WPF:

1. Собственные методы создания элементов. В Windows Forms классы для элементов управления делегируют функции отображения системным библиотекам, таким как user32.dll. В WPF любой элемент управления полностью строится самой WPF. Для аппаратного ускорения рендеринга[30] применяется технология DirectX (рис. 7.1).

2. Независимость от разрешения устройства вывода. Для указания размеров в WPF используется собственная единица измерения, равная 1/96 дюйма. Кроме этого, технология WPF ориентирована на использование векторных примитивов.

3. Декларативный пользовательский интерфейс. В WPF визуальное содержимое окна можно полностью описать на языке разметки XAML (англ. eXtensible Application Markup Language), основанному на XML (англ. eXtensible Markup Language).
При этом описание пользовательского интерфейса отделено от кода[31].

---

[30] Ре́ндеринг или отрисовка (англ. rendering — «визуализация») — термин в компьютерной графике, обозначающий процесс получения изображения по модели с помощью компьютерной программы.

[31] Дизайнеры могут использовать профессиональные инструменты (например, Microsoft Expression Blend), чтобы редактировать файлы XAML.

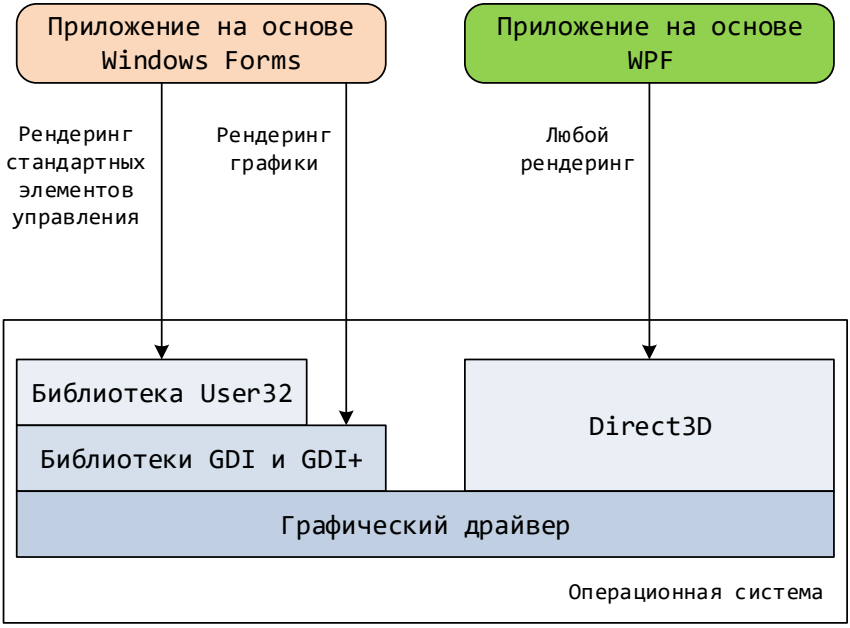


Рис. 7.1 – Рендеринг в приложениях на основе технологий Windows Forms и WPF.

4. Web-подобная модель компоновки. WPF поддерживает гибкий визуальный поток, размещающий элементы управления на основе их содержимого.

5. Стили и шаблоны. Стили стандартизируют форматирование и позволяют повторно использовать его по всему приложению. Шаблоны дают возможность изменить способ отображения любых элементов управления, даже таких основополагающих, как кнопки или поля ввода.

6. Анимация. В WPF анимация – определяется декларативными дескрипторами, и запускается в действие автоматически.

7. Приложения на основе страниц. Создание приложений с кнопками навигации, которые позволяют перемещаться по коллекции страниц. Специальный тип приложения XBAP (XAML

Browser Application — браузерное приложение XAML) — это приложения, которые выполняются внутри браузера.

В тоже время WPF имеет ряд ограничений:

– Несмотря на поддержку трехмерной визуализации, для создания приложений с большим количеством трехмерных изображений, прежде всего игр, лучше использовать другие средства - DirectX или специальные фреймворки, такие как Monogame или Unity;

– По сравнению с приложениями на Windows Forms объем программ на WPF и потребление ими памяти в процессе работы в среднем несколько выше.

7.2 Простейшее приложение WPF.

Рассмотрим простейшее приложение WPF. Пусть файл `Program.cs` содержит следующий код:

```
using System;
using System.Windows;
public class Program {
    [STAThread]
    public static void Main() {
        var myWindow = new Window();
        myWindow.Title = "My First WPF Program";
        myWindow.Content = "Hello, world!";
        var myApp = new Application();
        myApp.Run(myWindow);
    }
}
```

Пространство имён `System.Windows` содержит классы `Window` и `Application`, описывающее окно и приложение соответственно. Точка входа помечена атрибутом `[STAThread]`. Это обязательное условие для любого приложения WPF, оно связано с моделью многопоточности WPF. В методе `Main()` создаётся и настраивается объект окна, затем создаётся объект приложения. Вызов метода `Run()` приводит к отображению окна и запуску цикла обработки событий. Чтобы скомпилировать

приложение, необходимо указать ссылки на стандартные сборки `PresentationCore.dll`, `PresentationFramework.dll`, `System.Xaml.dll` и `WindowsBase.dll`.

В Visual Studio приложениям WPF соответствует отдельный шаблон проекта. Этот шаблон ориентирован на использование XAML, поэтому в случае однооконного приложения будет создан следующий набор файлов:

– файл `MainWindow.xaml.cs` на языке C# и `MainWindow.xaml` на языке XAML описывают класс `MainWindow`, являющийся наследником класса `Window`;

– файлы `App.xaml.cs` и `App.xaml` описывают класс `App`, наследник класса `Application`.

Ниже приведён файл `MainWindow.xaml` для простейшего окна:

```
<Window x:Class="WpfApplication.MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presenta
tion" xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="My First WPF Program" Height="400" Width="800">
  <!-- содержимое окна -->
  Hello, world!
</Window>
```

Компиляция проекта, созданного по шаблону WPF, происходит в два этапа. Вначале для каждого файла XAML генерируется два файла, сохраняемых в подкаталогах `obj\Debug` или `obj\Release` (в зависимости от цели компиляции):

1. файл с расширением `*.baml` (BAML-файл) – двоичное представление XAML-файла, внедряемое в сборку в виде ресурса;
2. файл с расширением `*.g.cs` – разделяемый класс, который соответствует XAML-описанию. Этот класс содержит поля для всех именованных элементов XAML и реализацию метода `InitializeComponent()`, загружающего BAML-данные из ресурсов сборки. Кроме этого, класс содержит метод, подключающий все обработчики событий.

На втором этапе сгенерированные файлы компилируются вместе с исходными файлами C# в единую сборку (рис. 7.2).

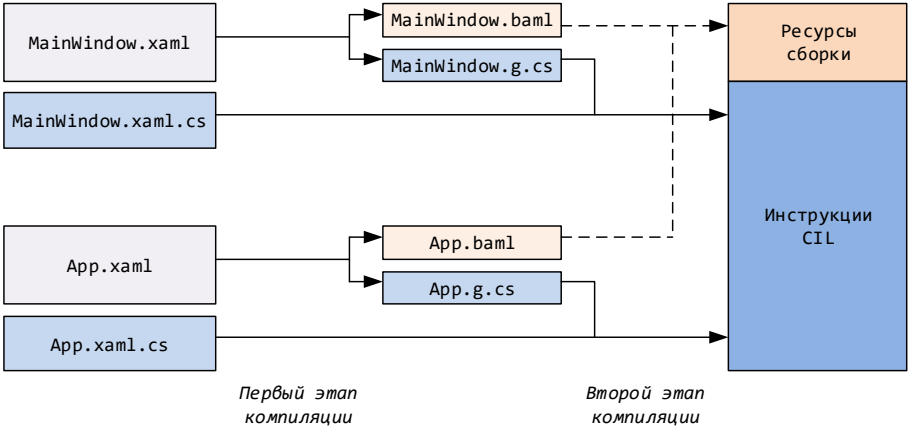


Рис. 7.2 – Компиляция приложения WPF в Visual Studio.

7.3 Введение в XML.

Расширенный язык разметки (eXtensible Markup Language, XML) широко распространен как платформо-независимый формат представления данных. XML используется в приложениях для моделирования частично структурированных и неструктурированных данных. XML представляет собой подмножество языка SGML (Standard Generalized Markup Language – стандартный обобщенный язык разметки[32]).

7.3.1 Объявление XML.

Документы XML начинаются с необязательного объявления XML, которое содержит обозначение версии XML, применяемой автором документа (1.0) и системы кодировки (UTF-8 соответствует стандарту Unicode), а также содержит сведения о том, имеется ли в документе ссылка на внешние объявления размет-

[32] Стандартизован ISO в 1986 году - ISO 8879:1986 Information processing—Text and office systems—Standard Generalized Markup Language (SGML)».

ки (standalone = 'yes' указывает, что в документе отсутствуют внешние объявления разметки):

```
<?xml version="1.1" encoding="UTF-8" standalone="yes"?>
```

7.3.2 Элементы XML.

Элементы XML, называемые также дескрипторами, представляются собой наиболее широко применяемую форму разметки. Первый элемент документа должен быть так называемым корневым элементом, который может содержать другие элементы (субэлементы). Каждый документ XML должен иметь один корневой элемент. Любой элемент начинается с начального дескриптора и оканчивается конечным дескриптором. Элемент XML чувствительны к регистру, поэтому элемент <DOC> отличный от элемента <doc>. Элемент может быть пустым, и в этом случае его можно сокращенно представить одним дескриптором, например, <DOC/>. Элементы должны быть правильно вложенными: `<doc> пример элемента </doc>`.

7.3.3 Атрибуты XML.

Атрибуты представляют собой пары “имя-значение”, которые содержат описательную информацию об элементе. Атрибуты помещаются внутри начального дескриптора после соответствующего имени элемента, а значение атрибута заключаются в кавычки. Каждый конкретный атрибут может присутствовать в дескрипторе только в одном экземпляре, тогда субэлемент в одном и том же дескрипторе могут повторяться. Каждый элемент может содержать любое количество атрибутов:

```
<termdef id="dt-dog" term="dog">.
```

Значения атрибутов не должны содержать символы «<» (левая угловая скобка), «>» (правая угловая скобка), «&» (амперсанд), «‘» (апостроф) и «“» (двойная кавычка). Данные символы кодируются последовательностями «<», «>», «&», «'», «"» соответственно.

7.3.4 Комментарии.

Комментарии заключаются в дескрипторы <! -- и -- > и могут содержать любые данные, кроме литеральной строки “--”. Комментарии могут помещаться между дескрипторами разметки в любом месте документа XML, но процессор XML не обязательно должен передавать комментарии приложению.

7.3.5 Разделы CDATA и команды обработки.

Раздел CDATA является для процессора XML указанием, что процессор должен игнорировать содержащиеся в этом разделе символы разметки и передавать заключенный в нем текст непосредственно приложению без интерпретации. Для передачи дополнительной информации приложению могут также применяться команды обработки. Команда обработки имеет форму, где служит для приложения идентификатором команды обработки. Поскольку команды зависят от приложения, любой документ XML может содержать несколько команд обработки, позволяющих передать разным приложениям команду на выполнение одинаковых действий.

7.3.6 Упорядочение.

В языке XML элементе рассматриваются как упорядоченные, таким образом, в языке XML следующие два фрагмента с переставленными элементами FNAME и LNAME считаются разными:

```
<NAME>
        <FNAME>John</FNAME>
        <LNAME>White</LNAME>
</NAME>
<NAME>
        <LNAME>White</LNAME>
        <FNAME>John</FNAME>
</NAME>
```

Но атрибуты в XML не являются упорядоченными, поэтому следующие два элемента XML считаются одинаковыми:

```
<NAME FNAME="John" LNAME="White"/>
<NAME LNAME="White" FNAME="John"/>
```

7.4 Расширяемый язык разметки приложений XAML.

Расширяемый язык разметки приложений – это язык для представления дерева объектов .NET, основанный на XML. Данные XAML превращаются в дерево объектов при помощи *анализатора XAML* (XAML parser). Основное назначение XAML – описание пользовательских интерфейсов в приложениях WPF.

Рассмотрим основные правила XAML. Документ XAML записан в формате XML. Кроме этого, XAML по умолчанию игнорирует лишние пробельные символы (однако это поведение изменяется установкой у элемента атрибута `xml:space="preserve"`).

Объектные элементы XAML описывают объект некоторого типа платформы .NET и задают значения открытых свойств и полей объекта. Имя элемента указывает на тип объекта.

Ниже показан пример описания на XAML объекта класса `Button`, а также реализация такого объекта на языке C#:

```
<!-- определение объекта в XAML -->
<Button Width="100">I am a Button</Button>
// определение объекта в коде
Button b = new Button();
b.Width = 100;
b.Content = "I am a Button";
```

В XAML пространству имён .NET ставится в соответствие пространство имён XML:

`xmlns:префикс="clr-namespace:пространство-имён"`

При необходимости указывается сборка, содержащая пространство имён:

`xmlns:префикс="clr-namespace:пространство-имён;assembly=имя-сборки"`

Для WPF зарезервировано два пространства имён XML:

1. `http://schemas.microsoft.com/winfx/2006/xaml/presentation` – является пространством имён по умолчанию и соответствует набору пространств имён .NET с типами WPF[33].
2. `http://schemas.microsoft.com/winfx/2006/xaml` – пространство имен XAML, которое включает различные служебные свойства XAML, которые позволяют влиять на то, как интерпретируется документ. Данное пространство имен отображается на префикс x. Это значит, что его можно применять, помещая префикс пространства имен перед именем элемента (как в <x:ИмяЭлемента>).

```
<!-- для корневого элемента Window заданы три пространства имён -->
<Window
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:sys="clr-namespace:System;assembly=mscorlib">
```

Информация пространства имен позволяет анализатору XAML находить правильный класс. Например, когда он просматривает элементы Window и Grid, то видит, что они помещены в пространство имен WPF по умолчанию. Затем он ищет соответствующие пространства имен .NET— до тех пор, пока не находит System.Windows.Window и System.Windows.Controls.Grid.

Для установки значений свойств объекта в XAML используют атрибуты XML:

```
<!—задаётся зеленый цвет фона для кнопки -->
<Button Background="Green" />
```

Элемент свойства вложен в объектный элемент описывается так: `<имя-типа.имя-свойства>`. Содержимое элемента свойства рассматривается как значение свойства:

```
<Button>
  <Button.Width>100</Button.Width>
  <Button.Background>Red</Button.Background>
</Button>
```

[33] System.Windows.*

Тип, соответствующий объектному элементу, может быть помечен атрибутом [ContentProperty] с указанием имени *свойства содержимого*. В этом случае анализатор XAML рассматривает содержимое объектного элемента как значение для свойства содержимого[34]. Поэтому следующие два фрагмента XAML эквиваленты:

```
<Button Content="Click me!" />
<Button>Click me!</Button>
```

Если тип реализует интерфейсы IList или IDictionary, при описании объекта этого типа в XAML дочерние элементы автоматически добавляются в соответствующую коллекцию. Например, свойство Items класса ListBox имеет тип ItemCollection, а этот класс реализует интерфейс IList:

```
<ListBox>
  <ListBox.Items>
    <ListBoxItem Content="Item 1" />
    <ListBoxItem Content="Item 2" />
  </ListBox.Items>
</ListBox>
```

Кроме этого, Items – это свойство содержимого для ListBox, а значит, приведённое XAML-описание можно упростить:

```
<ListBox>
  <ListBoxItem Content="Item 1" />
  <ListBoxItem Content="Item 2" />
</ListBox>
```

Механизм *расширений разметки* (markup extensions) позволяет вычислять значение свойства при преобразовании XAML в дерево объектов. Технически, любое расширение разметки – это класс, унаследованный от System.Windows.Markup.MarkupExtension и перекрывающий функцию ProvideValue(). Встретив расширение разметки, анализатор XAML генерирует код, который создаёт объект

[34] За исключением элементов свойств

расширения разметки и вызывает `ProvideValue()` для получения значения:

```
using System;
using System.Windows.Markup;
namespace MarkupExtensions {
 public class ShowTimeExtension : MarkupExtension {
  public string Header { get; set; }
  public ShowTimeExtension() { }
  public override object ProvideValue(IServiceProvider sp) {
   return string.Format("{0}: {1}", Header, DateTime.Now);
  }
 }
}
```

Если расширение разметки используется в XAML как значение атрибута, то оно записывается в фигурных скобках (если имя расширения имеет суффикс *`Extension`*, этот суффикс можно не указывать). В фигурных скобках также перечисляются через запятую аргументы конструктора расширения и пары для настройки свойств расширения в виде *`свойство=значение`*.

```
<Window xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:local="clr-namespace:MarkupExtensions">
  <StackPanel>
    <Button Content="{local:ShowTime}" />
    <Button Content="{local:ShowTime Header=Time}" />
  </StackPanel>
</Window>
```

Расширения разметки могут применяться как значения элементов свойств:

```
<Button>
  <Button.Content>
    <local:ShowTime Header="Time" />
  </Button.Content>
</Button>
```

В таблице 7.1 представлены стандартные расширения разметки.

Таблица 7.1 - Расширения разметки из `System.Windows.Markup`

Имя	Описание, пример использования
`x:Array`	Представляет массив. Дочерние элементы становятся элементами массива

	`<x:Array Type="{x:Type Button}">`
	`  <Button />`
	`  <Button />`
	`</x:Array>`

`x:Null`	Представляет значение `null` `Style="{x:Null}"`

`x:Reference`	Используется для ссылки на ранее объявленный элемент
	`<TextBox Name="customer" />`
	`<Label Target="{x:Reference customer}" />`

`x:Static`	Представляет статическое свойство, поле или константу `Height="{x:Static SystemParameters.IconHeight}"`
`x:Type`	Аналог применения оператора `typeof` из языка C#

Рассмотрим некоторые директивы анализатора XAML, применяемые в WPF. Анализатор генерирует код, выполняющий по документу XAML создание и настройку объектов. Действия с объектами (в частности, обработчики событий) обычно описываются в отдельном классе. Чтобы связать этот класс с документом XAML используется директива-атрибут `x:Class`. Этот атрибут применяется только к корневому элементу и содержит имя класса, являющегося наследником класса корневого элемента:

```
<!-- у корневого элемента Window задан атрибут x:Class -->
<Window x:Class="WpfApplication.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
```

Чтобы сослаться на объект в коде, этот объект должен иметь имя. Для указания имени объекта используется директива-атрибут `x:Name`:

```
    <Button x:Name="btn" Content="Click me!" />
```

Многие элементы управления WPF имеют свойство `Name`. Анализатор XAML использует соглашение, по которому задание

свойства Name эквивалентно указанию директивы-атрибута x:Name.
Существует возможность встроить фрагмент кода в XAML-файл. Для этого используется директива-элемент x:Code. Такой элемент должен быть непосредственно вложен в корневой элемент, у которого имеется атрибут x:Class.

```
<Window x:Class="WpfApplication.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

  <Button x:Name="btn" Click="btn_click" Content="Click me!" />
  <x:Code>
    <![CDATA[
      void btn_click(object sender, RoutedEventArgs e)
      {
          btn.Content = "Inline Code Works!";
      }
    ]]>
  </x:Code>
</Window>
```

Директива-атрибут x:Key применяется при описании дочерних элементов объекта-словаря, и позволяет указать ключ словаря для элемента:

```
<Window.Resources>
  <SolidColorBrush x:Key="borderBrush" Color="Red" />
  <SolidColorBrush x:Key="textBrush" Color="Black" />
</Window.Resources>
```

7.5 Организация приложений WPF

Приложения WPF допускают несколько способов организации. Самый распространённый вариант – приложение в виде набора окон. Альтернативой являются приложения на основе страниц.

7.5.1 Класс Window

Технология WPF использует для отображения стандартные окна операционной системы. Окно представлено классом

System.Windows.Window. Это элемент управления со свойством содержимого Content. Обычно содержимым окна является контейнер компоновки.

Рассмотрим основные свойства класса Window. Внешний вид окна контролируют свойства Icon, Title (заголовок окна) и WindowStyle (стиль рамки). Начальная позиция окна управляется свойствами Left и Top (координаты левого верхнего угла окна) или свойством WindowStartupLocation, принимающим значения из одноимённого перечисления: CenterScreen, CenterOwner, Manual. Чтобы окно отображалось поверх всех окон, нужно установить свойство Topmost. Свойство ShowInTaskbar управляет показом иконки окна на панели задач, а свойство TaskbarItemInfo типа System.Windows.Shell.TaskbarItemInfo даёт возможность настроить варианты отображения этой иконки.

Приведём пример описания окна – оно будет использоваться при рассмотрении контейнеров компоновки:

```
<Window x:Class="WpfLayout.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Height="220" Width="400"
  Icon="/WpfLayout;component/layout-icon.png"
  Title="WPF Layout Examples"
  WindowStartupLocation="CenterScreen"
  Topmost="True">
  <!-- здесь будем размещать содержимое окна -->
</Window>
```

Чтобы показать окно, используя код, нужно создать экземпляр окна и вызвать у него метод Show() или ShowDialog(). Первый метод отображает немодальное окно. Второй метод обычно применяется для модальных диалоговых окон – он ожидает закрытия окна и возвращает то значение, которое было установлено свойству окна DialogResult. Метод Hide() прячет окно, а метод Close() закрывает окно.

```
MyDialogWindow dialog = new MyDialogWindow();
```

```
// отображаем диалог и ждём, пока его не закроют
if (dialog.ShowDialog() == true) {
    // диалог закрыли, нажав на OK
}
```

Класс `Window` поддерживает отношение владения. Если отношение установлено, дочернее окно минимизируется или закрывается, когда соответствующие операции выполняет его владелец. Чтобы установить отношение, задайте у дочернего окна свойство `Owner`. Коллекция окна `OwnedWindows` содержит все дочерние окна (если они есть).

Опишем некоторые события, связанные с окном. Заметим, что класс `FrameworkElement`, который является базовым для класса `Window`, определяет события времени существования, генерируемые при инициализации (`Initialized`), загрузке (`Loaded`) или выгрузке (`Unloaded`) элемента. Событие `Initialized` генерируется после создания элемента и определения его свойств в соответствии с разметкой XAML. Это событие генерируется сначала вложенными элементами, затем их родителями. Событие `Loaded` возникает после того, как всё окно было инициализировано, и были применены стили и привязка данных. Событие `Loaded` сначала генерирует вмещающее окно, после чего его генерируют остальные вложенные элементы.

При изменении статуса активности окна генерируются события `Activated` и `Deactivated`. Событие `Closing` генерируется при попытке закрытия окна и даёт возможность отклонить эту операцию. Если закрытие окна все же происходит, генерируется событие `Closed`.

Ниже приведён листинг, демонстрирующий работу с некоторыми событиями окна. Обратите внимание – вместо назначения обработчиков событий используется перекрытие виртуальных методов, вызывающих события.

```
public partial class MainWindow : Window {
    public MainWindow() {
        InitializeComponent();
    }
```

```
    protected override void OnClosing(CancelEventArgs e) {
        base.OnClosing(e);
        if (MessageBox.Show("Do you want to exit?", "Exit",
                    MessageBoxButton.YesNo,
                    MessageBoxImage.Question) ==
MessageBoxResult.No) {
            e.Cancel = true;
        }
    }

    protected override void OnClosed(EventArgs e) {
         base.OnClosed(e);
         // здесь можно сохранить состояние окна
    }

    protected override void OnInitialized(EventArgs e) {
         base.OnInitialized(e);
         // здесь можно восстановить состояние окна
    }
}
```

7.5.2 Класс Application

Класс `System.Windows.Application` помогает организовать точку входа для оконного приложения WPF. Этот класс содержит метод `Run()`, поддерживающий цикл обработки сообщений системы для указанного окна до тех пор, пока окно не будет закрыто:

```
Window myWindow = new Window();
Application myApp = new Application();
myApp.Run(myWindow);
```

Свойство `StartupUri` класса `Application` служит для указания главного окна приложения. Если главное окно задано, метод `Run()` можно вызывать без аргументов:

```
Application app = new Application();
app.StartupUri = new Uri("MainWindow.xaml", UriKind.Relative);
app.Run();
```

При разработке в Visual Studio для каждого оконного приложения WPF создаётся класс, наследуемый от `Application`, который разделён на XAML-разметку и часть с кодом. Именно в разметке XAML задаётся `StartupUri`:

```
<Application x:Class="WpfLayout.App"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  StartupUri="MainWindow.xaml">
</Application>
```

Класс `Application` содержит несколько полезных свойств. При помощи статического свойства `Application.Current` можно получить ссылку на объект, представляющий текущее приложение. Коллекция `Windows` содержит все открытые окна приложения. Стартовое окно хранится в свойстве `MainWindow` (оно доступно для чтения и записи). Свойство `ShutdownMode` принимает значения из одноимённого перечисления и задаёт условие закрытия приложения. Словарь `Properties` позволяет хранить произвольную информацию с ключами любого типа и может использоваться для данных, разделяемых между окнами. События класса `Application` включают `Startup` и `Exit`, `Activated` и `Deactivated`, а также событие `SessionEnding`, генерируемое при выключении компьютера или окончании сессии Windows. События обычно обрабатываются путём перекрытия виртуальных методов, вызывающих их генерацию.

7.5.3 Приложения на основе страниц

В WPF можно создавать приложения в виде набора страниц с возможностью навигации между страницами. Отдельная страница представлена объектом класса `System.Windows.Controls.Page`. Любая страница обслуживается навигационным контейнером, в качестве которого могут выступать объекты классов `NavigationWindow` или `Frame`. Рассмотрим следующий пример. Создадим в Visual Studio WPF-приложение и добавим к проекту страницу `MainPage.xaml`:

```
<Page x:Class="WpfApplication.MainPage"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presenta
tion"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

```
  WindowHeight="220" WindowWidth="400"
  WindowTitle="Page Application">
  This is a simple page.
</Page>
```

В файле `App.xaml` установим `StartupUri="MainPage.xaml"`, чтобы выполнение приложения началось со страницы `MainPage.xaml`. Так как в качестве `StartupUri` указана страница, а не окно, при запуске приложения автоматически будет создан новый объект `NavigationWindow` для выполнения роли навигационного контейнера. Класс `NavigationWindow` наследуется от `Window` и выглядит как обычное окно, за исключением кнопок навигации, которые отображаются сверху (рис. 7.3).

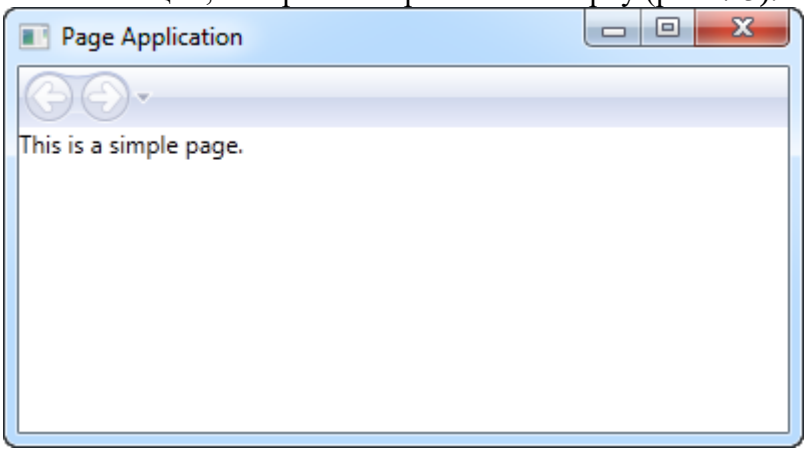


Рис. 7.3 – Страница в контейнере `NavigationWindow`.

Класс `Page` похож на упрощённую версию класса `Window`. Основные свойства класса `Page` перечислены в табл. 3.

Таблица 7.2 – Основные свойства класса `Page`.

Имя	Описание
`Background`	Фон страницы, который задаётся с помощью объекта `Brush`

`Content`	Элемент, который отображается на странице. Обычно в роли такого элемента выступает контейнер компоновки
`Foreground`, `FontFamily`, `FontSize`	Цвет, вид и размер текста внутри страницы. Значения этих свойств наследуются элементами внутри страницы
`WindowWidth`, `WindowHeight`, `WindowTitle`	Свойства определяют внешний вид окна `NavigationWindow`, в которое упаковывается страница
`NavigationService`	Возвращает ссылку на объект `NavigationService`, который можно использовать для программной навигации
`KeepAlive`	Определяет, должен ли сохраняться объект страницы после перехода пользователя на другую страницу
`ShowsNavigationUI`	Определяет, должны ли в обслуживающем страницу контейнере отображаться навигационные элементы управления
`Title`	Устанавливает имя, которое должно применяться для страницы в хронологии навигации

Отметим, что в классе `Page` нет эквивалентов для методов `Hide()` и `Show()`, доступных в классе `Window`. Если потребуется показать другую страницу, придётся воспользоваться одним из видов навигации. Навигационные контейнеры и класс `NavigationService` поддерживают метод `Navigate()`, который удобно использовать для программной навигации. Метод `Navigate()` принимает в качестве аргумента или объект страницы, или адрес страницы (обычно это имя XAML-файла):

```
// предполагается, что код расположен в методе страницы
// навигация по объекту страницы
PhotoPage nextPage = new PhotoPage();
this.NavigationService.Navigate(nextPage);
// или навигация по URI
var nextPageUri = new Uri("PhotoPage.xaml", UriKind.Relative);
this.NavigationService.Navigate(nextPageUri);
```

Гиперссылки – это простой способ декларативной навигации. В WPF гиперссылка представлена объектом класса `Hyperlink`, который внедряется в текстовый поток. Щелчки на ссылке можно обрабатывать при помощи события `Click`, или установить страницу перехода в свойстве `NavigateUri` (только если `Hyperlink` размещается на странице). При этом `NavigateUri` может указывать не только на объект `Page`, но и на веб-содержимое.

```
<TextBlock>
  Click <Hyperlink NavigateUri="PhotoPage.xaml">here</Hyperlink>
</TextBlock>
```

`NavigationWindow` поддерживает журнал показанных страниц и содержит две кнопки «назад» и «вперёд». Методы навигационного контейнера `GoBack()` и `GoForward()` позволяют перемещаться по журналу страниц, используя код.

Как было сказано выше, в качестве навигационного контейнера можно использовать элемент управления `Frame`. Он включает свойство `Source`, которое указывает на подлежащую отображению страницу. Так как `Frame` – обычный элемент управления, он может размещаться в заданном месте окна `Window` или страницы `Page`. В последнем случае образуются вложенные фреймы.

7.5.4 Работа с панелью задач Windows

Опишем некоторые возможности по работе с панелью задач Windows, доступные в приложениях WPF. *Список переходов* (jump list) – это мини-меню, которое открывается при щелчке правой кнопкой мыши на иконке приложения в панели задач.

Для работы со списком переходов в приложении WPF применяются классы `JumpList` (список переходов), `JumpPath` (путь к документу в списке переходов) и `JumpTask` (команда в списке) из пространства имён `System.Windows.Shell`:

```
// этот код размещён в конструкторе главного окна
// создаём и настраиваем команду для списка переходов
var jumpTask = new JumpTask();
jumpTask.Title = "Notepad";
jumpTask.Description = "Open Notepad";
jumpTask.ApplicationPath = @"%WINDIR%\system32\notepad.exe";
jumpTask.IconResourcePath = @"%WINDIR%\system32\notepad.exe";
// создаём сам список переходов, добавляем в него команду
var jumpList = new JumpList();
jumpList.JumpItems.Add(jumpTask);
// связываем список с текущим приложением
JumpList.SetJumpList(Application.Current, jumpList);
```

Класс `JumpTask` описывает команду, выполнение которой влечёт запуск заданного приложения. Класс имеет следующие свойства:

`CustomCategory` – категория, к которой относится команда.

`Title` – название команды;

`Description` – всплывающая подсказка;

`IconResourcePath` и `IconResourceIndex` – файл с иконкой и индекс иконки в файле для отображения в списке переходов;

`ApplicationPath` – путь к исполняемому файлу нужного приложения;

`WorkingDirectory` – рабочий каталог для приложения;

`Arguments` – аргументы командной строки для приложения.

У класса `JumpPath` для настройки доступны два свойства. Строка `CustomCategory` указывает категорию, к которой относится `JumpPath`. Свойство `Path` задаёт полный путь к файлу документа. При этом файл должен существовать, а его тип должен соответствовать типу файлов, за обработку которых отвечает данное приложение.

Для задания списка переходов в XAML используется присоединённое свойство `JumpList.JumpList`:

```
 <Application x:Class="WpfLayout.App"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="MainWindow.xaml">
    <JumpList.JumpList>
      <JumpList>
        <JumpTask Title="Notepad" Description="Open Notepad"
                  ApplicationPath="%WINDIR%\system32\notepad.exe"
                  IconResourcePath="%WINDIR%\system32\notepad.exe"
/>
        <JumpPath Path="C:\Pictures\Cat.jpg" />
      </JumpList>
    </JumpList.JumpList>
 </Application>
```

Класс `Window` имеет свойство `TaskbarItemInfo`, которое даёт доступ к иконке окна на панели задач. В следующем примере показан один из вариантов использования `TaskbarItemInfo` – на иконке отображается индикатор прогресса:

```
<Window.TaskbarItemInfo>
  <TaskbarItemInfo ProgressValue="0.3" ProgressState="Paused" />
</Window.TaskbarItemInfo>
```

7.5.5 Компоновка.

В WPF *компоновка* (layout) – это процесс размещения визуальных элементов на поверхности родительского элемента. Компоновка состоит из двух фаз:

1. *Фаза измерения* (measure). В этой фазе родительский контейнер запрашивает желаемый размер у каждого дочернего элемента, которые, в свою очередь, выполняют фазу измерения рекурсивно.
2. *Фаза расстановки* (arrange). Родительский контейнер сообщает дочерним элементам их истинные размеры и позицию, в зависимости от выбранного способа компоновки.

7.6 Размер и выравнивание

Рассмотрим некоторые свойства элементов WPF, связанные с процессом компоновки. Свойство `Visibility`, определённое

в классе UIElement, управляет видимостью элемента. Это свойство принимает значение из перечисления System.Windows.Visibility: Visible – элемент виден на визуальной поверхности; Collapsed – элемент не виден на визуальной поверхности и не участвует в процессе компоновки; Hidden – элемент не виден на визуальной поверхности, но участвует в процессе компоновки («занимает место»).

В классе FrameworkElement определён набор свойств, ответственных за размер, отступы и выравнивание отдельного элемента (табл. 7.3).

Таблица 7.3 – Свойства размера, отступа и выравнивания

Имя	Описание
HorizontalAlignment	Определяет позиционирование дочернего элемента внутри контейнера компоновки, если доступно пространство по горизонтали. Принимает значения из одноимённого перечисления: Center, Left, Right, Stretch
VerticalAlignment	Определяет позиционирование дочернего элемента внутри контейнера компоновки, когда доступно дополнительное пространство по вертикали. Принимает значения из одноимённого перечисления: Center, Top, Bottom или Stretch
Margin	Добавляет пространство вокруг элемента. Это экземпляр структуры System.Windows.Thickness с отдельными компонентами для верхней, нижней, левой и правой стороны
MinWidth и MinHeight	Устанавливает минимальные размеры элемента. Если элемент слишком велик, он будет усечён

`MaxWidth` и `MaxHeight`	Устанавливает максимальные размеры элемента. Если контейнер имеет свободное пространство, элемент не будет увеличен сверх указанных пределов, даже если свойства `HorizontalAlignment` и `VerticalAlignment` установлены в `Stretch`
`Width` и `Height`	Явно устанавливают размеры элемента. Эта установка переопределяет значение `Stretch` для свойств `HorizontalAlignment` и `VerticalAlignment`. Однако размер не будет установлен, если выходит за пределы, заданные в `MinWidth` и `MinHeight`

При установке размеров в XAML можно указать единицу измерения: `px` (по умолчанию) – размер в единицах WPF (1/96 дюйма); `in` – размер в дюймах; `cm` – размер в сантиметрах; `pt` – размер в пунктах (1/72 дюйма).

```
<Button Width="100px" Height="0.5in" />
<Button Width="80pt" Height="2cm" />
```

В `FrameworkElement` свойства `Width` и `Height` установлены по умолчанию в значение `double.NaN`. Это означает, что элемент будет иметь такие размеры, которые нужны для отображения его содержимого. В разметке XAML значению `double.NaN` для свойств размера соответствует строка `"NaN"` или (более предпочтительно) строка `"Auto"`. Также в классе `FrameworkElement` определены свойства только для чтения `ActualWidth` и `ActualHeight`, содержащие действительные отображаемые размеры элемента после фазы расстановки. Следующий пример демонстрирует компоновку с элементами, у которых установлены некоторые свойства размера и позициони-

рования. Обратите внимание на различные способы указания свойства `Margin`:
– одно значение: одинаковые отступы для всех четырёх сторон;
– два значения: отступы для левой/правой и верхней/нижней сторон;
– четыре числа: отступы для левой, верхней, правой и нижней стороны.

```
    <StackPanel>
      <Button HorizontalAlignment="Left" Content="Button 1"
/>
      <Button HorizontalAlignment="Right" Content="Button
2" />
      <Button Margin="20" Content="Button 3" />
      <Button Margin="5,10,100,10" Content="Button 4" />
      <Button Margin="5,10" MaxWidth="100" Content="Button
5" />
    </StackPanel>
```

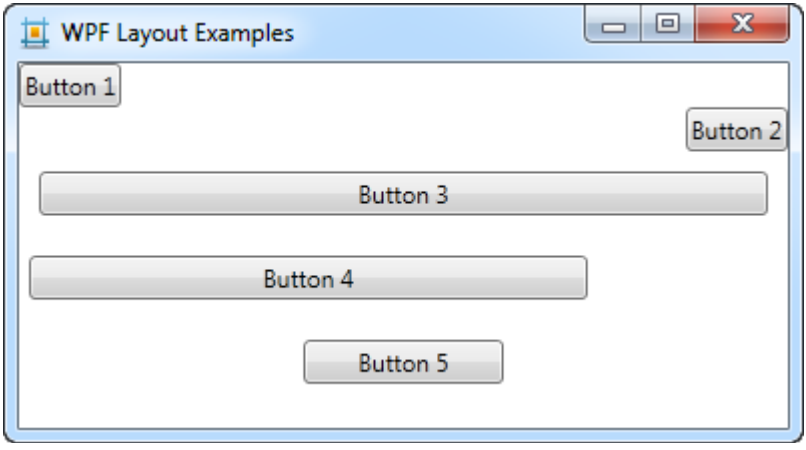


Рис. 7.4 – Использование свойств размера и позиционирования.

В элементах управления, унаследованных от класса `Control`, определены свойства отступа и выравнивания для содержимого. Выравнивание осуществляют свойства `HorizontalContentAlignment`, `VerticalContentAlignment`. Они поддерживают те же значения, что и свойства `HorizontalAlignment` и `VerticalAlignment`. Свойство `Padding` позволяет вставить пустое пространство между краями элемента управления и краями содержимого. Его тип и способ задания аналогичны свойству `Margin`.

7.7 Встроенные контейнеры компоновки

Контейнер компоновки – это класс, реализующий определённую логику компоновки дочерних элементов. Технология WPF предлагает ряд стандартных контейнеров компоновки:

1. `Canvas`. Позволяет элементам позиционироваться по фиксированным координатам.
2. `StackPanel`. Размещает элементы в горизонтальный или вертикальный стек. Контейнер обычно используется в небольших секциях сложного окна.
3. `WrapPanel`. Размещает элементы в сериях строк с переносом. Например, в горизонтальной ориентации `WrapPanel` располагает элементы в строке слева направо, затем переходит к следующей строке.
4. `DockPanel`. Выравнивает элементы по краю контейнера.
5. `Grid`. Встраивает элементы в строки и колонки невидимой таблицы.

Все контейнеры компоновки являются панелями, которые унаследованы от абстрактного класса `System.Windows.Controls.Panel`. Этот класс содержит несколько полезных свойств:

`Background` – кисть, используемая для рисования фона панели. Кисть нужно задать, если панель должна принимать события мыши (как вариант, это может быть прозрачная кисть).

`Children` – коллекция элементов, находящихся в панели. Это первый уровень вложенности – другими словами, это элементы, которые сами могут содержать дочерние элементы.

`IsItemsHost` – булево значение. Устанавливается в `true`, если панель используется для показа элементов в шаблоне спискового элемента управления.

`ZIndex` – присоединённое свойство класса `Panel` для задания высоты визуального слоя элемента. Элементы с большим значением `ZIndex` выводятся поверх элементов с меньшим значением. `Canvas` – контейнер компоновки, реализующий позициони-

рование элементов путём указания фиксированных координат. Для задания позиции элемента следует использовать присоединённые свойства `Canvas.Left`, `Canvas.Right`, `Canvas.Top`, `Canvas.Bottom`. Эти свойства определяют расстояние от соответствующей стороны `Canvas` до ближайшей грани элемента[35]. Использование контейнера `Canvas` демонстрируется в следующем примере:

```
<Canvas>
<Button Background="Red" Content="Left=0, Top=0" />
<Button Canvas.Left="20" Canvas.Top="20" Background="Orange"
Content="Left=20, Top=20" /><Button Canvas.Right="20"
Canvas.Bottom="20" Background="Yellow" Content="Right=20,
Bottom=20" /><Button Canvas.Right="0" Canvas.Bottom="0"
Background="Lime" Content="Right=0, Bottom=0" /><Button
Canvas.Right="0" Canvas.Top="0" Background="Aqua" Content="Right=0,
Top=0" /><Button Canvas.Left="0" Canvas.Bottom="0"
Background="Magenta" Content="Left=0, Bottom=0" />
</Canvas>
```

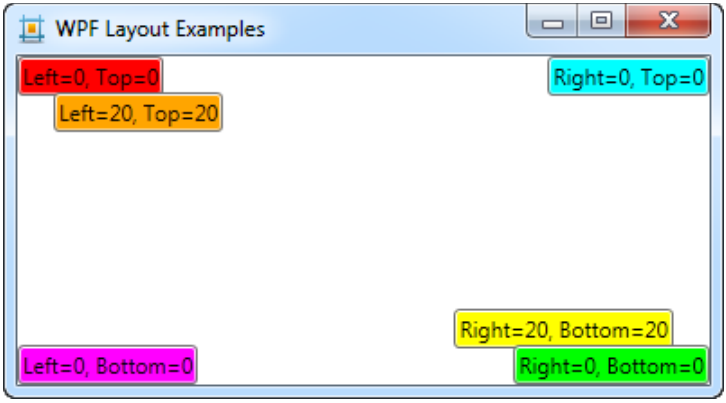


Рис. 7.5 – Кнопки в контейнере `Canvas`.

[35] При одновременной установке `Canvas.Left` имеет преимущество перед `Canvas.Right`, а `Canvas.Top` – перед `Canvas.Bottom`.

`StackPanel` – популярный контейнер компоновки, который размещает дочерние элементы последовательно, по мере их объявления в контейнере. Свойство `Orientation` управляет направлением размещения дочерних элементов и принимает значения из одноимённого перечисления: `Vertical` (по умолчанию) или `Horizontal` (рис. 7.6).

```
<StackPanel Orientation="Horizontal">
  <Button Background="Red" Content="One" />
  <Button Background="Orange" Content="Two" />
  <Button Background="Yellow" Content="Three" />
  <Button Background="Lime" Content="Four" />
  <Button Background="Aqua" Content="Five" />
  <Button Background="Magenta" Content="Six" />
</StackPanel>
```

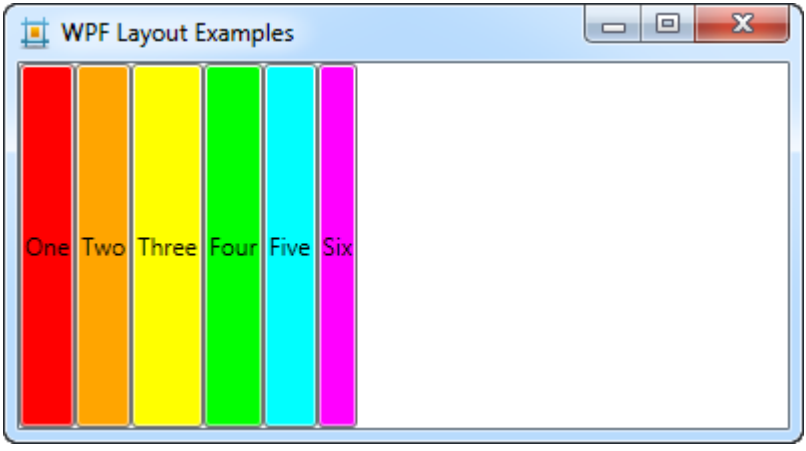


Рис. 7.6 – `StackPanel` с горизонтальной ориентацией.

`WrapPanel` – это контейнер компоновки, который во многом аналогичен `StackPanel`. Однако `WrapPanel` использует автоматический перенос элементов, для которых не хватает вертикального (горизонтального) пространства, в новый столбец (строку). `WrapPanel` поддерживает несколько свойств настройки:

`Orientation` – свойство аналогично одноименному свойству у `StackPanel`, но по умолчанию использует значение `Horizontal`.
`ItemHeight` – единая мера высоты для всех дочерних элементов. В рамках заданной единой высоты каждый дочерний элемент располагается в соответствие со своим свойством `VerticalAlignment` или усекается.
`ItemWidth` – единая мера ширины для всех дочерних элементов. В рамках заданной единой ширины каждый дочерний элемент располагается в соответствие со своим свойством `HorizontalAlignment` или усекается.
По умолчанию, свойства `ItemHeight` и `ItemWidth` не заданы (имеют значение `double.NaN`). В этой ситуации ширина столбца (высота строки) определяется по самому широкому (самому высокому) дочернему элементу.

```
<WrapPanel ItemHeight="80">
  <Button Width="60" Height="40" Background="Red" Content="One" />
  <Button Width="60" Height="20" Background="Orange"
          Content="Two" />
  <Button Width="60" Height="40" Background="Yellow"
          Content="Three" />
  <Button Width="60" Height="20" VerticalAlignment="Top"
          Background="Lime" Content="Four" />
  <Button Width="60" Height="40" Background="Aqua" Content="Five"
/>
  <Button Width="60" Height="20" Background="Magenta"
          Content="Six" />
</WrapPanel>
```

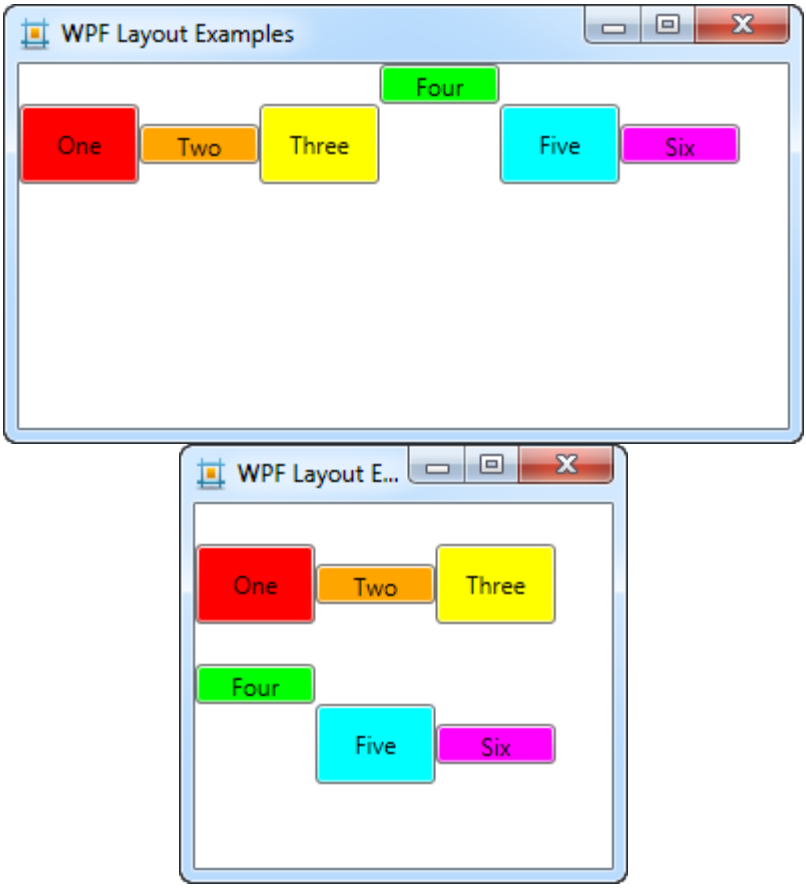


Рис. 7.7 – Элементы на `WrapPanel` для разной ширины окна.

`DockPanel` (*док*) реализует *примыкание* (docking) дочерних элементов к одной из своих сторон. Примыкание настраивается при помощи присоединённого свойства `DockPanel.Dock`, принимающего значения `Left`, `Top`, `Right` и `Bottom` (перечисление `System.Windows.Controls.Dock`). Примыкающий элемент растягивается на всё свободное пространство дока по вертикали или горизонтали (в зависимости от стороны примыкания), если у элемента явно не задан размер. Порядок дочерних элементов в

доке имеет значение. Если в `DockPanel` свойство `LastChildFill` установлено в `true` (это значение по умолчанию), последний дочерний элемент занимает всё свободное пространство дока.

```
<DockPanel>
  <Button DockPanel.Dock="Top" Background="Red" Content="1 (Top)"
/>
  <Button DockPanel.Dock="Left" Background="Orange" Content="2
(Left)" />
  <Button DockPanel.Dock="Right" Background="Yellow" Content="3
(Right)" />
  <Button DockPanel.Dock="Bottom" Background="Lime" Content="4
(Bottom)" />
  <Button Background="Aqua" Content="5 (Fill)" />
</DockPanel>
```

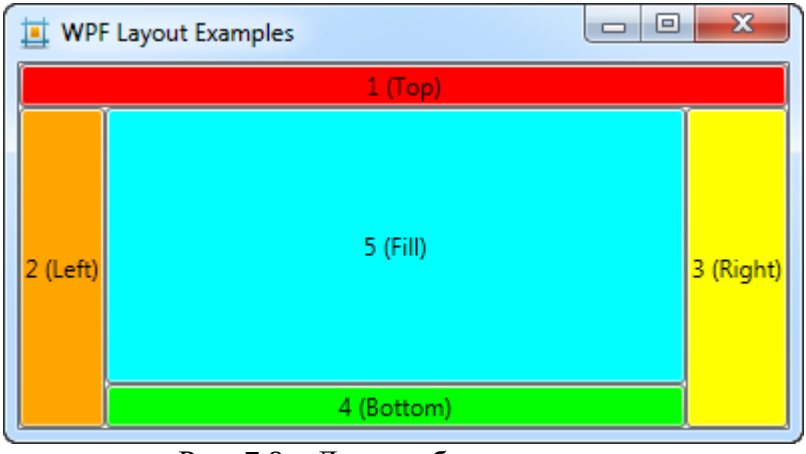


Рис. 7.8 – Док с набором кнопок.

`Grid` – один из наиболее гибких и широко используемых контейнеров компоновки. Он организует пространство как таблицу с настраиваемыми столбцами и строками. Дочерние элементы размещаются в указанных ячейках таблицы.

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="100" />
    <ColumnDefinition Width="Auto" />
    <ColumnDefinition Width="*" />
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition Height="*" />
    <RowDefinition Height="2*" />
  </Grid.RowDefinitions>
   <Button Grid.Column="0" Grid.Row="0"
           Background="Red" Content="One" />
   <Button Grid.Column="1" Grid.Row="0" Width="60"
           Background="Orange" Content="Two" />
   <Button Grid.Column="2" Grid.Row="0"
           Background="Yellow" Content="Three" />
   <Button Grid.Column="0" Grid.Row="1"
           Background="Lime" Content="Four" />
   <Button Grid.Column="1" Grid.Row="1" Grid.ColumnSpan="2"
           Background="Aqua" Content="Five" />
</Grid>
```

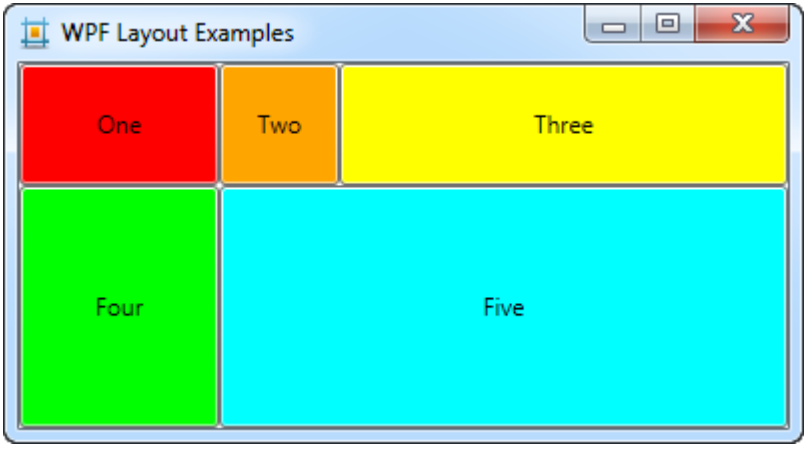


Рис. 7.9 – Демонстрация контейнера `Grid`.

При настройке `Grid` необходимо задать набор столбцов и строк с помощью коллекций `ColumnDefinitions` и `RowDefinitions`. Для столбцов может быть указана ширина,

для строк – высота. При этом допустимо использовать абсолютное значение, подбор по содержимому или *пропорциональный размер*. В последнем случае применяется символ * и необязательный коэффициент пропорциональности. В примере, приведённом выше, высота второй строки равна удвоенной высоте первой строки, независимо от высоты окна.

Дочерние элементы связываются с ячейками `Grid` при помощи присоединённых свойств `Grid.Column` и `Grid.Row`. Если несколько элементов расположены в одной ячейке, они наслаиваются друг на друга. Один элемент может занять несколько ячеек, определяя значения для присоединённых свойств `Grid.ColumnSpan` и `Grid.RowSpan`.

Контейнер `Grid` позволяет изменять размеры столбцов и строк при помощи перетаскивания, если используется элемент управления `GridSplitter`. Вот несколько правил работы с этим элементом:

1. `GridSplitter` должен быть помещён в ячейку `Grid`. Лучший подход заключается в резервировании специального столбца или строки для `GridSplitter` со значениями `Height` или `Width`, равными `Auto`.
2. `GridSplitter` всегда изменяет размер всей строки или столбца, а не отдельной ячейки. Чтобы сделать внешний вид `GridSplitter` соответствующим такому поведению, растяните его по всей строке или столбцу, используя присоединённые свойства `Grid.ColumnSpan` или `Grid.RowSpan`.
3. Изначально `GridSplitter` настолько мал, что его не видно. Дайте ему минимальный размер. Например, в случае вертикальной разделяющей полосы установите `VerticalAlignment` в `Stretch`, а `Width` – в фиксированный размер.
4. Выравнивание `GridSplitter` определяет, будет ли разделительная полоса горизонтальной (используемой для изменения размеров строк) или вертикальной (для изменения размеров

столбцов). В случае горизонтальной разделительной полосы установите `VerticalAlignment` в `Center` (что принято по умолчанию). В случае вертикальной разделительной полосы установите `HorizontalAlignment` в `Center`.
Ниже приведён фрагмент разметки, использующей горизонтальный `GridSplitter`:

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="100" />
    <ColumnDefinition Width="Auto" />
    <ColumnDefinition Width="*" />
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition Height="*" />
    <RowDefinition Height="Auto" />
    <RowDefinition Height="2*" />
  </Grid.RowDefinitions>
  <GridSplitter Grid.Column="0" Grid.Row="1" Grid.ColumnSpan="3"
HorizontalAlignment="Stretch" Height="3" VerticalAlignment="Center"
/>
</Grid>
```

Классы `RowDefinition` и `ColumnDefinition` обладают строковым свойством `SharedSizeGroup`, при помощи которого строки или столбцы объединяются в *группу, разделяющую размер*. Это означает, что при изменении размера одного элемента группы (строки или столбца), другие элементы автоматически получают такой же размер. Разделение размеров может быть выполнено как в рамках одного контейнера `Grid`, так и между несколькими `Grid`. В последнем случае необходимо установить свойство `Grid.IsSharedSizeScope` в значение `true` для внешнего контейнера компоновки:

```
<Grid IsSharedSizeScope="True">
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto" SharedSizeGroup="myGroup" />
    <ColumnDefinition />
    <ColumnDefinition SharedSizeGroup="myGroup" />
  </Grid.ColumnDefinitions>
  <!-- определение элементов Grid -->
</Grid>
```

7.8 Прокрутка и декорирование содержимого

Необходимость в прокрутке возникает, если визуальное содержимое выходит за границы родительского элемента[36]. *Панель прокрутки* – это элемент управления содержимым `ScrollViewer`. Его свойства `VerticalScrollBarVisibility` и `HorizontalScrollBarVisibility` управляют видимостью полос прокрутки и принимают значение из перечисления `ScrollBarVisibility`:

1. `Visible` – полоса прокрутки видима, даже если в ней нет необходимости.
2. `Auto` – полоса прокрутки появляется только тогда, когда содержимое не помещается в визуальных границах панели прокрутки.
3. `Hidden` – полоса прокрутки не видна, но прокрутку можно выполнить в коде или используя клавиатуру.
4. `Disabled` – полоса прокрутки не видна, прокрутку выполнить нельзя.

Элемент `ScrollViewer` имеет методы для программной прокрутки. Вертикальная прокрутка выполняется при помощи методов `LineUp()`, `LineDown()`, `PageUp()`, `PageDown()`, `ScrollToHome()`, `ScrollToEnd()`, а горизонтальная прокрутка – при помощи `LineLeft()`, `LineRight()`, `PageLeft()`, `PageRight()`, `ScrollToHorizontalOffset()`, `ScrollToLeftEnd()`, `ScrollToRightEnd()`.

Одной из особенностей `ScrollViewer` является возможность участия содержимого в процессе прокрутки. Такое содержимое должно быть представлено объектом, реализующим интерфейс

---

[36] Классы, унаследованные от `UIElement`, обладают булевым свойством `ClipToBounds`. Если в родительском элементе это свойство установлено в `true`, визуальное содержимое дочерних элементов отсекается при выходе за границы родителя.

IScrollInfo. Кроме этого, необходимо установить свойство ScrollViewer.CanContentScroll в значение true. Интерфейс IScrollInfo реализуют всего несколько элементов. Одним из них является контейнер StackPanel. Его реализация интерфейса IScrollInfo обеспечивает *логическую прокрутку*, которая осуществляет переход от элемента к элементу, а не от строки к строке.

```
<ScrollViewer CanContentScroll="True">
  <StackPanel>
    <Button Height="100" Content="1"/>
    <Button Height="100" Content="2"/>
    <Button Height="100" Content="3"/>
    <Button Height="100" Content="4"/>
  </StackPanel>
</ScrollViewer>
```

Декораторы обычно служат для того, чтобы графически разнообразить и украсить область вокруг объекта. Все декораторы являются наследниками класса System.Windows.Controls.Decorator. Большинство декораторов – это специальные классы, предназначенные для использования вместе с определёнными элементами управления. Есть два общих декоратора, применять которые имеет смысл при создании пользовательских интерфейсов: Border и Viewbox. Декоратор Border принимает вложенное содержимое и добавляет к нему фон или рамку. Для управления Border можно использовать свойства размера и отступа, а также некоторые особые свойства:

Background – задаёт фон декоратора с помощью объекта Brush.

BorderBrush, BorderThickness – свойства задают цвет и ширину рамки. Чтобы показать рамку, нужно задать оба свойства. У рамки можно отдельно настроить ширину каждой из четырёх сторон.

CornerRadius – позволяет закруглить углы рамки. Можно отдельно настроить радиус закругления каждого угла.

```
<Border Margin="20" Padding="10" VerticalAlignment="Top"
        Background="LightYellow" BorderBrush="SteelBlue"
        BorderThickness="10,5,10,5" CornerRadius="5">
  <StackPanel>
    <Button Margin="5" Content="One" />
    <Button Margin="5" Content="Two" />
    <Button Margin="5" Content="Three" />
  </StackPanel>
</Border>
```

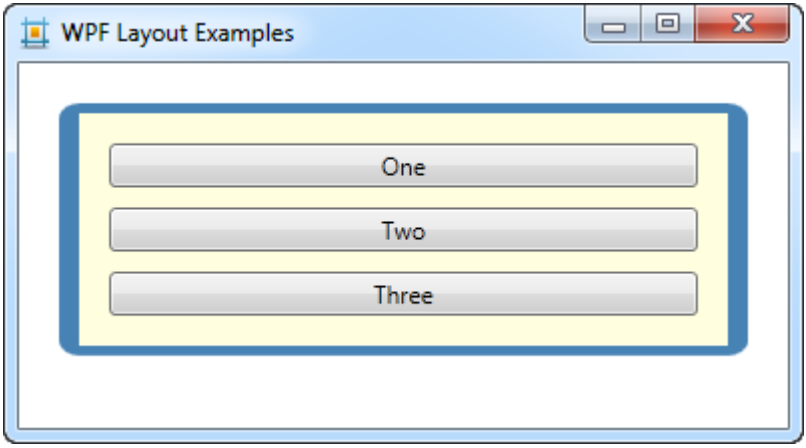


Рис. 7.10 – Декоратор `Border`.

Декоратор `Viewbox` масштабирует своё содержимое так, чтобы оно умещалось в этом декораторе. По умолчанию `Viewbox` выполняет масштабирование, которое сохраняет пропорции содержимого. Это поведение можно изменить при помощи свойства `Stretch`. Например, если свойству присвоить значение `Fill`, содержимое внутри `Viewbox` будет растянуто в обоих направлениях. Кроме этого, можно использовать свойство `StretchDirection`. Оно управляет масштабированием, когда содержимое достаточно мало (или слишком велико), чтобы уместиться в `Viewbox`.

7.9 Обзор элементов управления WPF

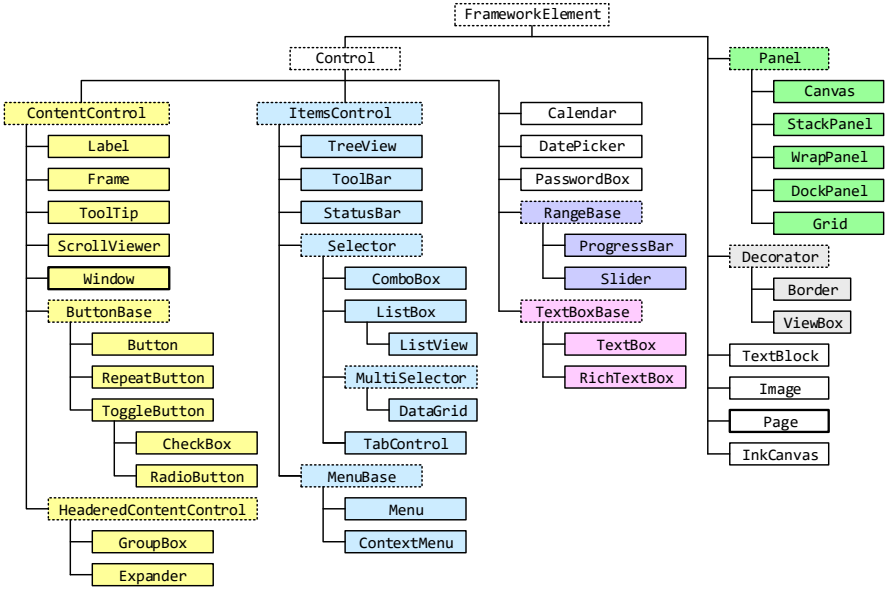


Рис. 7.11 – Стандартные элементы WPF.

Индивидуальные задания

Спроектируйте интерфейс приложения windows калькулятор, используя проект WPS.

Порядок выполнения лабораторной работы

1) Запустите среду разработки Visual Studio.

2) Визуально спроектируйте форму основной программы в соответствие с индивидуальным заданием.

3) Выполните отладку и компиляцию программы. При необходимости исправьте ошибки компиляции.

4) Выполните отчет по практической работе.

Контрольные вопросы

1) Поясните особенности установки размеров элементов и их позиционирования.
2) Поясните особенность создания пользовательских меню.
3) Поясните особенность работы с статусной строкой.
4) Поясните особенность работы с таймером.
5) Для чего используются компоненты классов OpenFileDialog и SaveFileDialog?
6) Поясните особенность работы с компонентами выбора настроек цветов и шрифтов?
7) Для чего используются компонент WebBrowser?
8) Поясните возможности работы с датой и временем?
9) Как осуществляется взаимодействие между формами**?**

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Казанский, А. А. Программирование на Visual C# 2013 : учебное пособие для СПО / А. А. Казанский. — М. : Издательство Юрайт, 2018. — 191 с. — (Серия : Профессиональное образование). — ISBN 978-5-534-02721-1.
2. Снетков, В.М. Разработка приложений на C# в среде Visual Studio 2005 [Электронный ресурс] : учебное пособие / В.М. Снетков. — Электрон. дан. — Москва : , 2016. — 956 с. — Режим доступа: https://e.lanbook.com/book/100467 .
3. Программирование на C++/C# в Visual Studio .NET 2003: Пособие / Понамарев В.А. - СПб:БХВ-Петербург, 2015. - 340 с. ISBN 978-5-9775-1224-4 - Режим доступа: http://znanium.com/catalog/product/939605
4. Марчуков, А.В. Работа в Microsoft Visual Studio [Электронный ресурс] : учебное пособие / А.В. Марчуков, А.О. Савельев. — Электрон. дан. — Москва : , 2016. — 384 с. — Режим доступа: https://e.lanbook.com/book/100439 .
5. Microsoft® Visual Studio 2010: Практическое руководство / Голощапов А.Л. - СПб:БХВ-Петербург, 2011. - 543 с. ISBN 978-5-9775-0617-5 - Режим доступа: http://znanium.com/catalog/product/354994
6. Visual C# 2010 на примерах: Практическое руководство / Зиборов В.В. - СПб: БХВ-Петербург, 2011. - 423 с. ISBN 978-5-9775-0698-4 - Режим доступа: http://znanium.com/catalog/product/355304