МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ, МОЛОДЕЖИ РЕСПУБЛИКИ КРЫМ

ОТДЕЛ ОБРАЗОВАНИЯ, МОЛОДЕЖИ И СПОРТА АДМИНИСТРАЦИИ ЧЕРНОМОРСКОГО РАЙОНА РЕСПУБЛИКИ КРЫМ

**МУНИЦИПАЛЬНОЕ БЮДЖЕТНОЕ ОБЩЕОБРАЗОВАТЕЛЬНОЕ УЧЕРЕЖДЕНИЕ «ЧЕРНОМОРСКАЯ СРЕДНЯЯ ШКОЛА №1 ИМ. НИКОЛАЯ КУДРИ» МУНИЦИПАЛЬНОГО ОБРАЗОВАНИЯ ЧЕРНОМОРСКИЙ РАЙОН РЕСПУБЛИКИ КРЫМ**

**Разработка приложения с графическим пользовательским интерфейсом для настольных компьютеров**

Работа обучающегося 11-А класса

Мельника Павла Юрьевича

Руководитель проекта

Ященко Николай Григорьевич

Работа допущена к защите «\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2023 год Подпись руководителя проекта \_\_\_\_\_\_ (\_\_\_\_\_\_)

Черноморское, 2023г.

Содержание

[Введение 3](#_Toc1)

[РАЗДЕЛ I Теоретическая часть 5](#_Toc2)

[I.1 План работы 5](#_Toc3)

[I.2 Сравнение технологий кроссплатформенной разработки приложений для ПК 6](#_Toc4)

[I.3 Объектно-ориентированное программирование 12](#_Toc5)

[I.4 Шаблон проектирования MVVM 15](#_Toc6)

[I.5 Реактивное программирование 17](#_Toc7)

[I.6 Система контроля версий 19](#_Toc8)

[I.7 Среда разработки 21](#_Toc9)

[I.8 Сериализация данных 23](#_Toc10)

[I.9 Разметка Markdown 24](#_Toc11)

[РАЗДЕЛ II Практическая часть 26](#_Toc12)

[I.10 Начало работы 26](#_Toc13)

[I.11 Создание макета представления 28](#_Toc14)

[I.12 Создание модели объекта 29](#_Toc15)

[I.13 Создание коллекции объектов 30](#_Toc16)

[I.14 Привязка к интерфейсу 31](#_Toc17)

[I.15 Интеракции. Добавление новых элементов 33](#_Toc18)

[I.16 Удаление элементов 35](#_Toc19)

[I.17 Редактирование элементов 36](#_Toc20)

[I.18 Интерфейс, унификация действий с объектом 37](#_Toc21)

[I.19 Поиск 38](#_Toc22)

[I.20 Реактивная ViewModel объекта 39](#_Toc23)

[I.21 Подробности задач 40](#_Toc24)

[I.22 Идентификатор объектов 41](#_Toc25)

[I.23 Реализация Markdown 42](#_Toc26)

[Выводы 43](#_Toc27)

[Список использованных источников 44](#_Toc28)

Введение

Темой индивидуального проекта является разработка приложения с графическим пользовательским интерфейсом. GUI (Graphical User Interface) или ГИП (графический интерфейс пользователя) — программная оболочка, которая предоставляет пользователю удобный интерфейс для работы с программой. Она визуализирует многие компоненты взаимодействия с приложением в виде графических объектов.

К преимуществам графического интерфейса относится наличие более дружелюбной (с англ. User-Friendly) системы управления программным обеспечением по сравнению с интерфейсом командной строки.

Разрабатываемым приложением будет «список задач». Реализация данной темы важна в наше время, так как сейчас у людей часто много дел и планов, которые нужно запомнить или записать, а также организовать эту информацию. Бумажные носители можно смело назвать устаревшими, так как у них имеется множество недостатков:

* Расход конечных природных ресурсов на бумагу и принадлежности для письма
* Необходимость разборчиво писать от руки
* Сложность быстрого обмена информацией
* Громоздкость при наличии больших объемов данных
* Сложность поиска среди доступной информации
* Сложность сортировки, организации и анализа данных

В то же время у хранения данных в электронном виде множество преимуществ:

* Информацию можно легко и быстро передавать в любую точку мира через интернет
* Можно осуществлять поиск нужной информации с помощью компьютерных программ, удобно структурировать данные и выполнять их анализ
* Хранение большого количества информации не занимает много места в реальном мире, не требует расхода природных ресурсов.
* ременную карту памяти размером с ноготь можно поместить содержимое всех библиотек мира
* Данные в электронном виде легко изменять и обновлять, в то время как данные на бумаге после первой же ошибки нужно переписывать заново, если нужно созранить приемлемый вид документа

На основе исследования, проведенного СберСтрахование жизни и Rambler&Co в 2022 году, для ведения заметок исключительно бумагой пользуются лишь 21% россиян. 14% используют только электронные носители, 5% стараются все запоминать, и оставшиеся 60% сочетают эти способы. Таким образом, 74% населения не обходится без хранения данных в электронном виде в повседневной жизни, поэтому данное приложение актуально.

По сравнению с другими похожими проектами можно выделить следующие отличительные черты: очень простой, понятный и приятный дизайн интерфейса, выделение выполненных задач, а также расширенные возможности форматирования текста.

**Цель работы** – создание графического приложения «Список задач»

**Предмет** исследования — разработкаGUI-приложений для ПК

**Объект** исследования — приложение «Список задач»

**Задачи**:

1. Определить актуальность темы, определить требования к продукту и разработать план работы.
2. Изучить и сравнить способы и технологий разработки компьютерных приложений
3. Обучиться разработке на выбранной платформе и разработать приложение «Список задач»

**Методы**, примененные при работе над проектом:

* Сравнение

# РАЗДЕЛ I Теоретическая часть

## План работы

План работы над проектом:

1. Определение технологии выполнения работы.

На этом этапе будет проведено сравнение технологий и средств, доступных для использования при работе над проектом.

1. Разработка первого прототипа.

После проведения спецификации и определения используеиых технологий, будут изучены способы разработки и создана первая версия продукта, так называемый **MVP** (англ. Minimal Viable Product — минимальный жизнеспособный продукт) с базовым функционалом. **CRUD** — акроним, обозначающий четыре базовые функции, используемые при работе с базами данных: создание (англ. create), чтение (read), модификация (update), удаление (delete). Является стандартной классификацией функций по манипуляции данными. Начальной целью будет создание такого приложения.

1. Добавление новых функций

К базовым функциям CRUD будут добавлены некоторые дополнительные, которые станут отличительными особенностями программы

## Сравнение технологий кроссплатформенной разработки приложений для ПК

Что вообще из себя представляет кроссплатформенная разработка? Если нужно, чтобы ваше приложение работало сразу на нескольких операционных системах с минимальными затратами человеко-ресурса, то скорее всего вы прибегнете именно к этому. Вы разрабатываете одну программу, пишете один код, а он запускается на всех поддерживаемых платформах.

В чем же преимущество кроссплатформенной разработки? Во-первых, нужно задействовать меньше людей (не нужно содержать .Net разработчика, Swift/objective C разработчика и всех к ним прилагающихся), во вторых вы охватываете большее количество устройств, а значит больше людей смогут пользоваться вашей программой.

На помощь приходят уже созданные Фреймворки и технологии для разработки кроссплатформенных GUI. Ниже представлены некоторые из них.

**Electron JS**

Electron JS — это JavaScript фреймворк, позволяющий вам написать полноценную программу используя Web Технологии — HTML & CSS, JS. Electron JS представляет собой движок Chromium в котором и исполняется весь ваш код. У Electron'a есть один, но достаточно серьезный недостаток — большое потребление памяти, как физической, так и оперативной. Но если посмотреть с другой стороны: сегодня многие популярные приложения написаны на Electron'e — Slack, Skype, Discord, VSCode, Atom, Postman, Insomnia и т.д. А с учетом непрекращающегося роста мощности компьютеров все реже приходится слышать от пользователей, что «ваш хром съел всю мою память». Высокое потребление памяти не будет играть большой роли если продукт будет хорош в своей сфере, будет грамотно написан код и распределены процессы.

Плюсы Electron JS:

* Использование наработок из Web
* Качественная документация
* Поддержка сообщества и GitHub

Недостатки:

* Высокое потребление памяти (физическая и ОЗУ) из-за того, что программа требует отдельный запущенный браузер для работы. Отсюда же следует плохая нативность, т. е. cложность или отсутствие доступа к системным функциям.
* Легко написать плохой код и сложно его поддерживать и развивать из-за слабой динамической типизации JavaScript
* Написание логики (Backend) желательно на языке, отличном от языка интерфейса приложения (Frontend). Обычно серьезные приложения пишут на JavaScript в качестве фронтенда и C#, Java, Ruby или Python для бэкенда, опять же из-за типизации JS. Для этого нанимают разных специалистов в этих областях. Если для написания интерфейсов JavaScript прижился, то делать на его основе взаимодействие с базами данных и какой-либо их анализ желающих мало.

**QT/QML**

Qt — мощный набор инструментов для создания кроссплатформенных приложений на языке С++ и Python (а также Ruby, PHP, C# и др. но они поддерживаются сообществом). Qt уже достаточно старый фреймворк, но он продолжает активно развиваться и на нем написаны такие программы как: 2ГИС для Android, Kaspersky Internet Security, Virtual Box, VLC Media Player, Opera и другие. Известное многим линукс-пользователям окружение рабочего стола KDE Plasma тоже написано с использованием Qt. Qt имеет среду разработки — Qt Creator, которая включает в себя Qt Designer с помощью которого можно создавать графический интерфейс. Визуальное создание интерфейса позволяет легко и просто создавать интерфейс, перетаскивая различные виджеты (выпадающие списки, кнопки, переключатели) на форму. Qt имеет обширную документацию, активную поддержку сообщества и множество других преимуществ.

* Визуальный редактор интерфейса
* Огромное количество модулей в составе фреймворка
* Поддержка большого количества языков программирования (однако официально только Python и C++, остальные поддерживаются сообществом).
* Низкий порог вхождения

Недостатки QT:

* Большой вес приложения
* Высокая стоимость коммерческой лицензии
* Сложность написания программы на C++, возможная нестабильность при использовании других «побочных» языков программирования
* Сложность распространения программы на Microsoft Windows из-за зависимостей.

**GTK**

GTK на ряду с Qt является одной из самых популярных библиотек разработки элементов интерфейса для X Window Systems. Разработан Gnome Foundation и GNU. С использованием этой библиотеки написаны многие окружения рабочего стола и оконные менеджеры Linux (Gnome, xfce MATE, Cinnamon, AfterStep, Marco и многие другие), и программы, например: Chromium, FireFox, MonoDevelop, Gimp и другие. GTK написан на языке C, но существуют обертки для других языков программирования (C, C++, Python, Java и другие). Gtk имеет конструктор для визуальной разработки интерфейсов — Glade.

Преимущества GTK:

* Поддержка большого количества языков
* Большое и активное сообщество
* Много популярных проектов и примеров
* Визуальный редактор

Недостатки:

* Как и с QT, данный фреймворк используется в основном на Linux. Там он установлен на большинстве систем или поставляется как зависимость для многих программ, поэтому его не нужно отдельно поставлять и устанавливать вместе с программой. На Windows же QT или GTK нужно устанавливать вручную отдельно от программы или делать установщик для программы, что снижает ее портативность.
* Основной язык разработки — C++.

**AvaloniaUI**

AvaloniaUI — кроссплатформенный фреймворк для разработки пользовательского интерфейса на основе языка разметки XAML. Достаточно молодой и активно развивающийся фреймворк для кроссплатформенной разработки, разработанный Стивеном Кирком и поддерживаемый .NET Foundation и сообществом. Он поддерживается на Windows, Linux и macOS. Сама Avalonia основана на WPF/UWP. Код пишется на языке C# и языке разметки XAML. Благодаря XAML Avalonia позволяет создавать гибкие и стилизованные интерфейсы. Имеет хорошую документацию, хоть и не большое, но очень приветливое рускоязычное сообщество. Для Visual Studio существует расширение, которое содержит шаблоны проектов и элементов управления. С ними вы легко вникнете в суть и сразу сможете приступить к разработке интерфейсов.

AvaloniaUI является open-source проектом, который активно развивается и совершенствуется. Однако это не мешает уже использовать его в каких-либо крупных проектах.

Исходный код Avalonia полностью открыт — его можно найти в официальном GitHub-репозитории Avalonia, выпущенной под лицензией MIT.

Avalonia написана на .NET Standard и на 100% совместима со всеми версиями .NET Core и .NET Framework. Авалония очень быстро развивается людьми по всему миру, которые работают с клиентами и знают, что им нужно. В Avalonia есть поддержка Visual Studio и Rider для создания файлов XAML и XAML intellisense, хотя Avalonia XAML intellisense все еще отстает от того, что может предложить WPF XAML intellisense. Avalonia позволяет создавать приложения, которые выглядят и ведут себя одинаково на разных платформах, но также поддерживает настройку для конкретной платформы. Несколько популярных проектов с открытым исходным кодом .NET даже перешли с WPF на Avalonia, чтобы сделать свои продукты мультиплатформенными.

Преимущества AvaloniaUI:

* Хорошая нативность и поддержка системных декораций
* Активное и приветливое сообщество, как русскоязычное, так и англоязычное (чаты, форумы)
* Заимствование и сходство с WPF (Если вы работали с WPF вам будет легко разобраться с Авалонией, и при возникновении трудностей, код написанный для WPF скорее всего заработает у вас на авалонии)
* Стили (стили в Авалонии имеют сходство с CSS, что упрощает написание этих самых стилей). Это позволяет легко изменить дизайн программы, чтобы он соответствовал времени.

**WxWidgets/WxPython**

wxWidgets (ранее известная как wxWindows) — это кросс-платформенная библиотека инструментов с открытым исходным кодом для разработки кроссплатформенных на уровне исходного кода приложений, в частности для построения графического интерфейса пользователя (GUI).Она разработана не только для того, чтобы создавать GUI. Она также имеет набор классов для работы с графическими изображениями, HTML, XML документами, архивами, файловыми системами, процессами, подсистемами печати, мультимедиа, сетями, классы для организации многопоточности, отладки, отправки дампов и множество других инструментов. Имеет обертку для языка Python -WxPython. С использованием этой библиотеки написанны такие программы как: FileZilla, AudaCity, BitTorrent и другие.

Преимущества WxWidgets:

* Простое написание интерфесов
* Поддержка разных языков программирования

Недостатки:

* Мало современной информации в интернете
* Не очень активное сообщество
* Примитивные интерфейсы

**Tkinter**

Tkinter — кросс-платформенная событийно-ориентированная графическая библиотека разработанная Гвидо ван Россумом (создатель языка Python), позволяет написать вам простой, но функциональный интерфейс для вашего проекта на Python. По сути Tkinter представляет из себя пакет для Python, предназначенный для работы с библиотекой Tk. Библиотека Tk содержит компоненты графического интерфейса пользователя (graphical user interface — GUI), написанные на языке программирования Tcl и существуют реализации для разных языков (Tkinter приведен для примера).

Преимущества:

* Простое написание интерфесов
* Позволяет вам быстро написать простой GUI для вашей Python программы

Недостатки:

* Примитивные интерфейсы
* Для одного языка (но непосредственно Tk существует для разных языков)

**Окончательный выбор**

Все перечисленные технологии по-своему хороши, однако у многих нашлись недостатки. Например, NodeJS использует браузер для работы и потребляет много ресурсов, а также использует язык JavaScript со слабой динамической типизацией, которая осложняет разработку приложения в перспективе. Tk и WxWidgets используют интерпретируемый Python с хоть и строгой, но динамической типизацией, и сами фреймворки довольно старые и примитивные. У GTK и QT же основным языком является C++, который является относительно низкоуровневым требует навыков в работе напрямую с памятью. Для ОС и других программ, нацеленных на максимальную производительность, это плюс, однако для разработки проектов, у которых в приоритете полезная нагрузка приложения, это недостаток. Также нам не удалось наладить распространение библиотек вместе с приложением для работы на Windows.

Для разработки программы был выбран фреймворк AvaloniaUI, платформа .NET и язык С# — читается как «Си шарп». Это язык программирования от компании Microsoft. Изначадьно его создали для проектов под Windows, но не так давно данный язык стал универсальным. На этом языке пишут игры, десктопные программы, веб-сервисы, нейросети и графику для метавселенных. Он и стал решающим в выборе данной платформы.

Преимущества языка С#:

* Независим от платформы

Разработчику не потребуется адаптировать приложение под различные платформы и системы. За человека это будет выполнять специальная виртуальная машина, которая уже вшита NET Framework. Получается, что один и тот же код удастся запускать на различных гаджетах. Например, на телефонах, серверах, компьютерах и т. д.

* Хорошая совместимость с Windows т. к. платформа начинала путь в той же компании
* Сборщик мусора.

Для того чтобы приложение работало хорошо и без лагов, память требуется время от времени очищать от различных ссылок, кэша, ненужных объектов и т. д. В С# этот процесс осуществляется автоматическим путем. Специалисту не нужно следить за тем, как расходуется память, бороться с её утечками или убирать мертвые куски кода.

* Строжайшая статическая типизация.

Как только специалист объявит переменную в С#, потребуется сначала указать, что именно в ней лежит – число, массив, строка или какой-либо другой конкретный тип информации. Таким способом придется немного дольше разрабатывать программу, но код будет работать правильно и предсказуемо. Числа будут взаимодействовать с числами, строки со строками и т.д. В языках, у которых слабая динамическая типизация свободы больше, но специалист может случайно пропустить ошибку. Её можно будет заметить только во время выполнения программы, в так называемом runtime. Если такое произойдет в готовом продукте, то это может привести к серьезным последствиям.

Стоит отметить, что код со статической строгой типизацией легче читается и понимается, чем, например, код на Lua или JavaScript. Если посмотреть на определение метода, то и без документации понятно, что нужно в нее передать и какой результат будет получен. Эту информацию не нужно держать в голове и передавать комментариями в коде, т. к. типизация может в некотором плане заменить документацию.

* Огромное сообщество.

На данном языке пишут множество специалистов во всем мире. В социальных сетях и различных сообществах можно задавать разные вопросы, обсуждать сложные темы или искать готовые решения.

* Синтаксический сахар.

В С# существует множество различных сокращений кода, не нарушающих логику приложения. Опытные специалисты называют подобные приемы «синтаксическим сахаром» — они позволяют писать код намного понятнее, проще и красивее. Например, LINQ позволяет гибко работать с коллекциями, заменяя традиционные циклы лаконичными цепочками вызовов. Также он используется для работы с реляционными базами данных и с реактивными потоками данных, что очень удобно: одними и теми же функциями можно работать с различными множествами данных.

## Объектно-ориентированное программирование

**ООП** (Объектно-Ориентированное Программирование) стало неотъемлемой частью разработки многих современных проектов. Это подход, при котором программа рассматривается как набор объектов, взаимодействующих друг с другом. У каждого есть свойства и поведение. ООП ускоряет написание кода и делает его более читаемым.

Идеология объектно-ориентированного программирования (ООП) разрабатывалась, чтобы связать поведение определенного объекта с его классом. Людям проще воспринимать окружающий мир как объекты, которые поддаются определенной классификации (например, разделение на живую и неживую природу).

**Зачем нужно ООП**

До ООП в разработке использовался другой подход — процедурный. Программа представляется в нем как набор процедур и функций — подпрограмм, которые выполняют определенный блок кода с нужными входящими данными. Процедурное программирование хорошо подходит для легких программ без сложной структуры. Но если блоки кода большие, а функций сотни, придется редактировать каждую из них, продумывать новую логику. В результате может образоваться много плохо читаемого, перемешанного кода — «спагетти-кода» или «лапши».

В отличие от процедурного, объектно-ориентированное программирование позволяет вносить изменения один раз — в объект. Именно он — ключевой элемент программы. Все операции представляются как взаимодействие между объектами. При этом код более читаемый и понятный, программа проще масштабируется.

Чтобы сделать код проще, программу разбивают на независимые блоки — объекты. В реальной жизни это может быть стол, чашка, человек, книга, здание и многое другое. В программировании объекты — это структуры данных: пользователь, кнопка, сообщение. У них, как и у реальных предметов, могут быть свойства: цвет, содержание или имя пользователя. А чтобы объединить между собой объекты с похожими свойствами, существуют классы.

**Класс**— это «шаблон» для объекта, который описывает его свойства. Несколько похожих между собой объектов, например профили разных пользователей, будут иметь одинаковую структуру, а значит, принадлежать к одному классу. Каждый объект — это экземпляр какого-нибудь класса.

Такой подход помогает строить сложные системы более просто и естественно благодаря тому, что вся предметная область разбивается на объекты и каждый из них слабо связан с другими объектами. Слабая связанность возникает вследствие соблюдения трех принципов: инкапсуляции, наследования и полиморфизма.

**Инкапсуляция**— сокрытие поведения объекта внутри него. Объекту «водитель» не нужно знать, что происходит в объекте «машина», чтобы она ехала. Это ключевой принцип ООП.

Один из основных механизмов, обеспечивающих инкапсуляцию — модификаторы доступа членов класса private и public. Они регулируют то, доступен ли член класса снаружи объекта, или только изнутри, если он обеспечивает инкапсулированную логику объекта, которую в клиентском коде нельзя вызывать. Как сокращенная форма этого механизма, в C# у класса есть поля и свойства.

**Поля** — приватные переменные, к которым можно получить доступ только изнутри объекта, т. е. из его методов.

**Свойства** — сокращенная форма написания методов Get и Set для каждого поля. Свойство выглядит как переменная, которая представляет какие-либо поля объекта наружу. Например, в Get свойства «Длина» отрезка можно написать разность координат его начальной и конечной точки, а Set вообще не определять. Тогда снаружи будет выглядеть, будто у отрезка есть переменная «Длина», хотя на самом деле в ней инкапсулирована логика, которая выполняется при каждом получении значения свойства. При этом свойство «Длина» будет только для чтения, так как у него нет метода Set, который задействуется при присвоении свойству значения. Очевидно, что длина отрезка зависит от координат его точек, и никто не должен иметь возможности просто так изменить это значение. Это позволяет избежать ошибок при программировании, ограничивая свободу взаимодействия с объектом до нужной степени.

В методе Set можно выполнять некоторую проверку данных перед передачей значения полю. Например, на сайте есть объект «Форма регистрации», в которой нужно ввести email. Не помешает проверить email как минимум на наличие собачки, а вообще можно даже вызвать отправку письма с подтверждением прямо в Set.

**Наследование**.

Есть объекты «человек» и «водитель». У них есть явно что-то общее. Наследование позволяет выделить это общее в один объект (в данном случае более общим будет человек), а водителя — определить как человека, но с дополнительными свойствами и/или поведением. Например, у водителя есть водительские права, а у человека их может не быть.

Если наследовать класс от другого, тогда он повторит свойства и методы родительского. После этого можно будет использовать любые методы, применимые к тому же классу «человек», и с классом «водитель», наследованным от него. Это называется приведением типов или upcast.

В C# нельзя наследовать класс от нескольких классов одновременно, поскольку результат может быть непредсказуемым. Как более правильное решение для указания некого общего типа данных был введен еще один механизм — интерфейсы. Они позволяют лишь по наличию некоторых реализованных членов класса определить, подходит ли данный класс для использования в том или ином случае. Класс может наследовать, помимо одного родительского класса, сколько угодно интерфейсов.

**Полиморфизм**— это переопределение поведения. Можно снова рассмотреть «человека» и «водителя», но теперь добавить «пешехода». Человек умеет как-то передвигаться, но как именно, зависит от того, водитель он или пешеход. То есть у пешехода и водителя схожее поведение (метод движения), но реализованное по-разному: один перемещается ногами, другой — на машине.

**Преимущества ООП:**

* **Модульность**

Объектно-ориентированный подход позволяет сделать код более структурированным, в нем легко разобраться стороннему человеку. Благодаря инкапсуляции объектов уменьшается количество ошибок и ускоряется разработка с участием большого количества программистов, потому что каждый может работать независимо друг от друга.

* **Гибкость**

ООП-код легко развивать, дополнять и изменять. Это обеспечивает независимая модульная структура. Взаимодействие с объектами, а не логикой упрощает понимание кода. Для модификации не нужно погружаться в то, как построено ПО. Благодаря полиморфизму можно быстро адаптировать код под требования задачи, не описывая новые объекты и функции.

* **Экономия времени**

Благодаря полиморфизму и наследованию можно не писать один и тот же код много раз. Это ускоряет разработку нового функционала. Интерфейсы и классы в ООП могут легко преобразовываться в подобие библиотек, которые можно использовать заново в новых проектах. Также ООП экономит время при поддержке и доработке приложения.

* **Надежность**

Программу сложно сломать, так как инкапсулированный код недоступен извне, а также в подавляющем большинстве случаев языки парадигмы ООП имеют строгую статическую типизацию.

**Недостатки ООП:**

* **Сложный старт**

Чтобы пользоваться ООП, нужно сначала изучить теорию и освоить процедурный подход, поэтому порог входа относительно высокий.

* **Снижение производительности**

Объектно-ориентированный подход немного снижает производительность кода в целом. Программы работают несколько медленнее из-за особенностей доступа к данным и большого количества сущностей.

## Шаблон проектирования MVVM

**MVVM** (Model-View-ViewModel) — шаблон проектирования, концентрирующий внимание на разделении бизнес-логики и интерфейса программы. Данный шаблон широко используется в приложениях на платформах WPF/UWP/Xamarin/Avalonia.

**“Что” от “чего” отделяет?**

При использовании вышеприведенных фреймворков, приложение делится на два слоя:

Бизнес-логика приложения, в паттерне первая буква “M” (Model). В данном слое описывается логика и основные задачи приложения. Взаимодействие с файловой системой, базой данных, API, описание сущностей системы и т.п. Часто общение с различными источниками данных выделяют в отдельную под-часть (Services).

Интерфейс – в паттерне буква V (View) описывается с помощью языка разметки XAML.

Именно эти два слоя и призван разделить паттерн с помощью добавления еще одного – модель представления (ViewModel). ViewModel — связывающий слой между Model и View с помощью технологии привязки (Binding). Для понятия Binding, введем понятие свойства (Property) – изменяемое поле данных во ViewModel. Простыми словами, с помощью binding, все property, описанные в ViewModel доступны для View. Важным, также является изменяемость property – под этим следует, что любые изменения во View или Model о которых «узнает» ViewModel будут автоматически изменены в зависимости от того, откуда пришли изменение (ввод текстового поля, получение ответа от API и т.п)

Представление отвечает за отображение элементов пользовательского интерфейса на экран и зависит от конкретной операционной системы, а модель представления позволяет двум описанным выше слоям взаимодействовать, адаптируя слой модели для взаимодействия с пользователем – человеком.

**Зачем разделять?**

Для создания независимых частей приложения. Используя MVVM, вы гарантируете, что слой Model ничего не знает о View, это же применимо для ViewModel. В свою очередь это дает следующие преимущества:

* Использовать единожды написанную логику в других проектах, изменяя только View.

Например, вы разрабатываете приложение для Android / iOS на платформе Xamarin и вам необходимо сделать десктоп-версию. Используя MVVM, бизнес логика приложения не изменится и достаточно будет ее добавить в проект с View, написанным на платформах WPF/UWP.

* Unit-тесты.

Независимость Model и ViewModel позволяет писать Unit-тесты, не обращая внимание на особенности интерфейса. Unit-тесты — это тестирование отдельных частей приложения. Оно позволяет узнать, какая часть программы приводит к сбою и требует доработки. Очевидно, что при разделении модели и представления позволяет легко узнать, что именно та или иная часть не работает.

* В случае редизайна приложения, необходимость изменения логики работы минимальна или отсутствует.

## Реактивное программирование

Реактивное программирование — это подход к разработке ПО, который строится на реагировании на поток событий и на распространении событий. При этом модель реакции на события предполагает возможность простого распространения этих или трансформированных событий далее по системе. Ярким примером реализации реактивного подхода может служить таблица Excel. В ней существует цепочка вычислений, разделённая на несколько ячеек: при изменении значения одной из ячеек в цепочке значения в зависимых ячейках пересчитываются автоматически.

Таким образом, реактивное программирование позволяет преобразовывать поток информации, например, группируя изменения в строке поиска по времени, чтобы не перегружать сервер или компьютер пользователя обработкой каждой ввеленной буквы, и асинхронно на него реагировать.

**Шаблон разработки Observer**

Наблюдатель — самый распространеный шаблон разработки реактивных приложений

Шаблон разработки наблюдателя позволяет подписчику регистрироваться у поставщика и получать от него уведомления.

Он подходит для любого сценария, в котором требуется использование push-уведомлений.

Шаблон определяет поставщика (также называемого субъектом или наблюдаемым) и ноль, одного или несколько наблюдателей.

Наблюдатели регистрируются у поставщика, и при возникновении предварительно заданного условия, события или изменения состояния поставщик автоматически уведомляет все наблюдатели путем вызова одного из их методов.

В вызове метода поставщик также может предоставить наблюдателям сведения о текущем состоянии.

В .NET конструктивный шаблон наблюдателя применяется путем реализации универсальных интерфейсов Observable<T> и IObserver<T>.

Параметр универсального типа представляет тип, который предоставляет сведения об уведомлении.

**Применение шаблона**

Шаблон разработки наблюдателя подходит для распределенных push-уведомлений, так как он поддерживает четкое разделение двух разных компонентов или уровней приложения, таких как уровень источника данных (бизнес-логика) и уровень пользовательского интерфейса (отображение).

Шаблон можно реализовывать во всех случаях, когда поставщик использует обратные вызовы для предоставления текущих сведений клиентам.

Для реализации шаблона необходимы перечисленные ниже элементы:

* Поставщик или субъект, то есть объект, отправляющий уведомления наблюдателям.

Поставщик — это класс или структура, реализующие интерфейс IObservable<T>.

Поставщик должен реализовывать единственный метод, IObservable<T>.Subscribe, вызываемый наблюдателями, которым требуется получать уведомления от поставщика.

* Наблюдатель, то есть объект, получающий уведомления от поставщика.

Наблюдатель — это класс или структура, реализующие интерфейс IObserver<T>.

**Механизм, который позволяет поставщику отслеживать наблюдатели.**

Как правило, поставщик использует объект контейнера, например объект System.Collections.Generic.List<T>, для хранения ссылок на реализации IObserver<T>, которые подписаны на уведомления.

Использование контейнера хранилища в этих целях позволяет поставщику обрабатывать от нуля до неограниченного числа наблюдателей.

Порядок, в котором наблюдатели получают уведомления, не определен.

Поставщик может использовать любой метод для определения порядка.

**Реализация IDisposable для удаления наблюдатели по завершении уведомления.**

Наблюдатели получают ссылку на реализацию IDisposable из метода Subscribe, поэтому они также могут вызывать метод IDisposable.Dispose, чтобы отменить подписку, прежде чем поставщик завершит отправку уведомлений.

**ReactiveUI**

ReactiveUI — полноценный MVVM-фреймворк: bindings, routing, message bus, commands и прочие слова, которые есть в описании почти любого MVVM-фреймворка, есть и тут. Применяться он может практически везде, где есть .NET: WPF, Windows Forms, UWP, Windows Phone 8, Windows Store, Xamarin.

ReactiveUI построен вокруг реактивной модели программирования и использует Reactive Extensions (Rx). Обычно даже в тех местах, где проявляется «реактивность», код можно довольно легко прочитать и понять, что произойдет.

## Система контроля версий

Система контроля версий – это система, записывающая изменения в файл или набор файлов в течение времени и позволяющая вернуться позже к определенной версии. Мы хотим гибко управлять некоторым набором файлов, откатываться до определенных версий в случае необходимости. Можно отменить те или иные изменения файла, откатить его удаление, посмотреть кто что-то поменял. Как правило системы контроля версий применяются для хранения исходного кода, но это необязательно. Они могут применяться для хранения файлов совершенно любого типа.

Как хранить различные версии файлов? Люди пришли к такому инструменту как системы контроля версий не сразу, да и они сами бывают очень разные. Предложенную задачу можно решить с применением старого доброго copy-paste, локальных, централизованных или распределенных систем контроля версий.

**Copy-paste**

Известный метод при применении к данной задаче может выглядеть следующим образом: будем называть файлы, добавляя версию, возможно с добавлением времени создания или изменения.

Данный способ является очень простым, но он подвержен различным ошибкам: можно случайно изменить не тот файл, можно скопировать не из той директории (ведь именно так переносятся файлы в этой модели).

**Локальная система контроля версий**

Следующим шагом в развитии систем контроля версий было создание локальных систем контроля версий. Они представляли из себя простейшую базу данных, которая хранит записи обо всех изменениях в файлах.

Одним из примеров таких систем является система контроля версий RCS, которая была разработана в 1985 году (последний патч был написан в 2015 году) и хранит изменений в файлах (патчи), осуществляя контроль версий. Набор этих изменений позволяет восстановить любое состояние файла. RCS поставляется с Linux.

Локальная система контроля версий хорошо решает поставленную перед ней задачу, однако ее проблемой является основное свойство — локальность. Она совершенно не преднезначена для коллективного использования.

**Централизованная система контроля версий**

Централизованная система контроля версий предназначена для решения основной проблемы локальной системы контроля версий.

Для организации такой системы контроля версий используется единственный сервер, который содержит все версии файлов. Клиенты, обращаясь к этому серверу, получают из этого централизованного хранилища. Применение централизованных систем контроля версий на протяжении многих лет являлась стандартом. К ним относятся CVS, Subversion, Perforce.

Такими системами легко управлять из-за наличия единственного сервера. Но при этом наличие централизованного сервера приводит к возникновению единой точки отказа в виде этого самого сервера. В случае отключения этого сервера разработчики не смогут выкачивать файлы. Самым худшим сценарием является физическое уничтожение сервера (или вылет жесткого диска), он приводит к потерю кодовой базы.

Несмотря на то, что мода на SVN прошла, иногда наблюдается обратный ход — переход от Git'а к SVN'у. Дело в том, что SVN позволяет осуществлять селективный чекаут, который подразумевает выкачку лишь некоторых файлов с сервера. Такой подход приобретает популярность при использовании монорепозиториях, о которых можно будет поговорить позже.

**Распределенная система контроля версий**

Для устранения единой точки отказа используются распределенные системы контроля версий. Они подразумевают, что клиент выкачает себе весь репозиторий целиком заместо выкачки конкретных интересующих клиента файлов. Если умрет любая копия репозитория, то это не приведет к потере кодовой базы, поскольку она может быть восстановлена с компьютера любого разработчика. Каждая копия является полным бэкапом данных.

Все копии являются равноправными и могут синхронизироваться между собой. Подобный подход очень напоминает (да и является) репликацией вида master-master.

К данному виду систем контроля версий относятся Mercurial, Bazaar, Darcs и Git. Последняя система контроля версий и будет рассмотрена нами далее более детально.

**История Git**

В 2005 году компания, разрабатывающая систему контроля версий BitKeeper, порвала отношения с сообществом разработчиков ядра Linux. После этого сообщество приняло решение о разработке своей собственной системы контроля версий. Основными ценностями новой системы стали: полная децентрализация, скорость, простая архитектура, хорошая поддержка нелинейной разработки.

## Среда разработки

Интегрированная среда разработки (IDE) — это многофункциональная программа, которая поддерживает многие аспекты разработки программного обеспечения. Интегрированная среда разработки Visual Studio — это стартовая площадка для написания, отладки и сборки кода, а также последующей публикации приложений. Помимо стандартного редактора и отладчика, которые есть в большинстве сред IDE, Visual Studio включает в себя компиляторы, средства автозавершения кода, графические конструкторы и многие другие функции для улучшения процесса разработки.

**Средства повышения производительности**

Вот несколько популярных возможностей Visual Studio, которые повышают производительность при разработке программного обеспечения:

* Волнистые линии и быстрые действия

Волнистые линии обозначают ошибки или потенциальные проблемы кода прямо во время ввода. Эти визуальные подсказки помогают немедленно устранить проблемы, не дожидаясь появления ошибок во время сборки или выполнения. Если навести указатель мыши на волнистую линию, на экран будут выведены дополнительные сведения об ошибке. Также в поле слева может отображаться лампочка, указывающая на наличие сведений о быстрых действиях для устранения ошибки.

* Автодополнение кода

При вводе первых букв названия функции или переменной, Visual Studio предложит варианты дополнения этого названия с учетом контекста, т. е. При вызове метода, требующего строку аргументом, первым вариантом высвечивается строковая переменная.

* Очистка кода

Можно одним нажатием кнопки отформатировать код и применить к нему исправления, предложенные параметрами стиля кода, соглашениями в файле .editorconfig и (или) анализаторами Roslyn. Очистка кода, которая сейчас доступна только для кода C#, помогает устранять проблемы в коде перед переходом к его проверке.

* Рефакторинг

Рефакторинг включает в себя такие операции, как интеллектуальное переименование переменных и функций, извлечение одной или нескольких строк кода в новый метод и изменение порядка параметров методов.

* IntelliSense

IntelliSense — это набор возможностей, отображающих сведения о коде непосредственно в редакторе и в некоторых случаях автоматически создающих небольшие отрывки кода. По сути, это встроенная в редактор базовая документация, которая избавляет от необходимости искать информацию в других источниках. Функции IntelliSense зависят от языка. Дополнительные сведения см. в руководствах по IntelliSense для C# , IntelliSense для Visual C++, IntelliSense для JavaScript и IntelliSense для Visual Basic.

* Поиск в Visual Studio

Иногда может казаться, что в Visual Studio слишком много меню, действий и свойств. Чтобы быстро находить функции интегрированной среды разработки или элементы кода, в Visual Studio представлен единый компонент поиска.

* CodeLens

CodeLens помогает находить ссылки на код, изменения кода, связанные с кодом ошибки, рабочие элементы, проверки кода и модульные тесты — не выходя из редактора.

* Перейти к определению

Функция «Перейти к определению» позволяет перейти к расположению, где определена выбранная функция или тип.

* Показать определение

В окне Показать определение можно отобразить метод или определение типа, не открывая отдельный файл.

## Сериализация данных

**Сериализация** — это преобразование объекта или дерева объектов в какой-либо формат с тем, чтобы потом эти объекты можно было восстановить из этого формата. Используется, например, для сохранения состояния программы (то есть, некоторых её объектов) между запусками. Или для передачи данных между различными экземплярами программы (или различными программами), например, по сети.

Главная идея состоит в том, что сериализованный формат — набор байт или строка, которую можно легко сохранить на диск или передать другому процессу или, например, по сети, в отличие от самого объекта. А значит, задача сохранения объекта/группы объектов при этом сводится к простой задаче сохранения набора байт или строки.

**JSON** — один из популярных форматов для сериализации, он текстовый, легковесный и легко читается человеком.

Т.е. есть определенный набор данных, который хранится в массиве или он представляет собой некий объект. Если провести процесс сериализации для этого набора данных, то этот набор данных мы представим в виде строки по какому-то определенному набору правил.

В зависимости от применяемого набора правил, на выходе мы получим некую текстовую строку, которая будет оформлена по каким-то правилам. Это набор правил называется форматом. Чаще всего при сериализации данных они формируются в формате json.

Дело в том, что с текстовыми строками может работать практически любое устройство. Если мы представляем набор данных в виде текстовой строки, эту строку может обрабатывать любое устройство (персональный компьютер, телефон или планшет). Все эти устройства могут прочитать строку и десериализовать эту строку, т. е. провести обратный процесс сериализации.

С помощью этого формата программы на разных языках программирования также могут обмениваться данными.

В .NET есть библиотека Newtonsoft.Json для сериализации и десериализации объектов из строк JSON, которую мы будем использовать в проекте.

## Разметка Markdown

**Markdown** — язык разметки текстов. Такие тексты легко писать и можно читать даже в сыром виде. Их можно без труда сконвертировать в HTML. Большинство программистов предпочитают Markdown для написания документации, описаний своих проектов, написания блогов и так далее.

**Как пользоваться языком разметки**

Человек пишет текст в любом редакторе и добавляет в него специальные комбинации стандартных символов. Это разметка, она показывает, где текст должен быть жирным, где — заголовком, а в каком месте нужен список. С помощью символов можно разметить заголовки, выделения текста и даже сложные конструкции вроде таблиц. Специальные преобразователи формируют на основе разметки готовый документ. Это похоже на использование HTML, но Markdown проще, текст в нем более читаемый. Преобразователи или изначально есть в текстовом редакторе, например в приложении для заметок, или скачиваются отдельно.

Основной синтаксис. Он очень простой: текст «оборачивается» в символы или они ставятся перед строкой. Вот так, например, будут выглядеть заголовки:

# Заголовок H1 (первого уровня, например название текста)

## Заголовок H2 (второго уровня, заголовки)

### Заголовок H3 (третьего уровня, промежуточные подзаголовки)

И так далее, вплоть до заголовков шестого уровня с шестью решетками перед строкой. На практике подзаголовки ниже H3 практически не используют.

Можно применить базовое форматирование текста. Буквицу или рукописный шрифт вставить не получится, но выделить важные участки — легко.

\*Курсив\* или \_курсив\_

\*\*Жирный текст\*\* или \_\_жирный текст\_\_

\*\*\*Жирный и подчеркнутый текст\*\*\*

> Цитата (угловая скобка ставится перед каждой строкой цитаты)

— Пункт маркированного списка

1. Первый пункт нумерованного списка

2. Второй пункт нумерованного списка (и так далее)

Вставка материалов. Разметка позволяет вставлять в текст разные элементы: ссылки и изображения, которые потом «подтянутся» в документ, когда применится форматирование.

[текст\_ссылки](любая ссылка) для вставки ссылок

[подпись к картинке](ссылка на картинку) для вставки картинок в текст

\*\*\*, — или \_\_\_ — горизонтальная линия на всю строку, которая отделяет часть текста от другой.

**Реализация**

В проекте планируется возможность написания заметок на языке разметки Markdown с последующим их преобразованием и правильным отображением благодаря библиотеке Markdown.Avalonia. Она реализует многие функции Markdown: заголовки, выделение текста, цитаты, именованные ссылки, нумерованные и ненумерованные списки, строки и блоки программного кода, таблицы, разделители, цитаты, выравнивание текста по краям и даже отображение изображений, причем как локальных, так и из интернета. Данная библиотека позволяет отображать Markdown из любых строковых переменных, а также из разметки Markdown прямо внутри объекта XAML.

# РАЗДЕЛ II Практическая часть

## Начало работы

**Установка Visual Studio**

Чтобы облегчить написание, а также тестирование и отладку программного кода нередко используют специальные среды разработки, в частности, Visual Studio. Рассмотрим создание приложений на C# с помощью бесплатной и полнофункциональной среды Visual Studio Community 2022, которую можно загрузить с официального сайта Microsoft.

После загрузки запустим программу установщика. В открывшемся окне нам будет предложено выбрать те компоненты, которые мы хотим установить вместе с Visual Studio. Стоит отметить, что Visual Studio — очень функциональная среда разработки и позволяет разрабатывать приложения с помощью множества языков и платформ. В нашем случае нам будет интересовать прежде всего C# и .NET.

Чтобы добавить в Visual Studio поддержку проектов для C# и .NET 7, в программе установки среди рабочих нагрузок можно выбрать только пункт «Разработка классических приложений .NET» . Можно выбрать и больше опций или вообще все опции, однако стоит учитывать свободный размер на жестком диске — чем больше опций будет выбрано, соответственно тем больше места на диске будет занято.

**Создание проекта**

Давайте создадим программу на базе фреймворка AvaloniaUI.

Запустим среду разработки Visual Studio. Сначала нужно установить пакет для разработки приложений AvaloniaUI с помощью пакетного менеджера Nuget. Он содержит библиотеки, необходимые для работы будущей программы.

Также не помешает установить расширение Avalonia для Visual Studio. Оно включает в себя конструктор XAML, который можно использовать для предварительного просмотра XAML в процессе его написания.

XAML — это декларативный язык разметки. Как и в случае с моделью программирования .NET, XAML упрощает создание пользовательского интерфейса для приложения .NET. Можно создать видимые элементы пользовательского интерфейса в декларативной XAML-разметке, а затем отделить определение пользовательского интерфейса от логики выполнения, используя файлы кода программной части, присоединенные к разметке, называемые code-behind с помощью определений разделяемых классов. Язык XAML напрямую представляет создание экземпляров объектов в конкретном наборе резервных типов, определенных в сборках. В этом заключается его отличие от большинства других языков разметки, которые, как правило, представляют собой интерпретируемые языки без прямой связи с системой резервных типов.

Далее можно создать новый проект. Откроется окно «Создание проекта» с отображением нескольких шаблонов проектов. Шаблон содержит основные файлы и параметры, которые требуются для определенного типа проекта.

Чтобы найти шаблон, можно ввести ключевые слова в поле поиска. Выберем шаблон Avalonia.MVVM. Убедимся, что в окне Дополнительные сведения в раскрывающемся меню «Целевая платформа» указана новейшая версия .NET 7.0, а затем нажмем «Создать». Это простейшее приложение «Hello World», в котором вызывается окно с надписью «Hello World!» в окне.

Файлы проекта отображаются справа в окне интегрированной среды разработки Visual Studio в окне с названием Обозреватель решений. Код C# для нашего приложения открывается в центральном окне редактора, который занимает большую часть пространства.

Мы можем видеть, что есть каталоги для каждой из концепций шаблона MVVM (модели, представления и модели представлений), а также несколько других файлов и каталогов:

* Каталог Assets содержит ресурсы для нашего приложения, такие как значки и растровые изображения. Файлы, размещенные в этом каталоге, будут автоматически включены в приложение в качестве ресурсов.
* Папка Models в настоящее время пуста, но, как следует из названия, мы будем размещать в ней наши модели.
* Каталог ViewModels предварительно заполнен базовым классом для моделей представления и моделью представления для главного окна приложения.
* Каталог Views пока содержит только главное окно приложения.
* В файле App.axaml будут размещены стили и шаблоны XAML, которые будут применяться ко всему приложению.
* Файл Program.cs — это точка входа для выполнения приложения. Здесь мы можем настроить параметры платформы для Авалонии, если это необходимо.
* Файл ViewLocator.cs используется для поиска представлений для моделей представлений. Это будет объяснено более подробно позже.

Код автоматически выделяется цветом для обозначения таких элементов, как ключевые слова и типы. Найти код можно по номерам строк.

Небольшие вертикальные пунктирные линии в коде указывают, какие фигурные скобки соответствуют друг другу. Чтобы свернуть или развернуть блоки кода, можно использовать небольшие рамки со знаками минус и плюс соответственно. Эта функция структурирования кода позволяет скрыть ненужный код на экране. Также доступно множество других меню и окон инструментов.

Запустить приложение можно, выбрав в главном меню Visual Studio пункты Отладка>Запуск без отладки. Теперь у нас есть выполняемое приложение.

## Создание макета представления

Будем работать с файлом MainWindow.axaml, который описывает макет представления программы на языке XAML. Это очень похоже на HTML-разметку веб-страницы. Попробуем добавить несколько элементов в окно

В Avalonia нельзя просто так вставлять элементы, представляющие данные в главное окно. Оно может содержать лишь один элемент, и в качестве этого элемента мы можем использовать одну из множества панелей.

Панели — объекты, которые напрямую не содержат в себе данные. Они служат для того, чтобы организовывать элементы внутри себя в ту или иную форму. Например, StackPanel может содержать любое количество элементов и отображать их в один ряд по горизонтальной или вертикальной оси.

Grid (сетка, разметка) — наиболее полезная панель. Она позволяет задавать некоторое количество рядов и столбцов, образуя таблицу. При этом также можно настроить, в сколько места в тех или иных единицах измерения отводится под каждую колонку или столбец. Также можно использовать значение auto, чтобы ряд или столбец занимали ровно достаточно места для содержания своих элементов. Еще есть значение \*, которое занимает все доступное место (можно указать также 2\* или другое число, чтобы задать пропорции), оставшееся после более конкретных значений, например числовых или auto. Внутри самой панели, т. е. между тегами в свободном порядке размещаются элементы, и каждому из них указывается свойство Grid.Row и Grid.Column, чтобы определить, по каким координатам в таблице будет располагаться элемент — представление данных или другая панель.

Давайте создадим панель Grid в главном окне, так как она всегда позволит нам расширить или уточнить макет представления программы. Внутри таблицы поместим элемент ScrollViewer. Как можно догадаться из названия, он позволяет прокручивать свое содержимое, если оно не помещается полностью в панели. Это нам понадобится, потому что иначе мы не сможем иметь больше 5-10 задач одновременно, либо придется разбивать их на страницы. Внутри создадим список, добавив теги ListBox. Теперь у нас есть пока еще пустой список.

Также заблаговременно на второй ряд сетки поместим горизонтальную StackPanel. В нее мы позднее будем добавлять кнопки для взаимодействия со списком задач. Первый ряд обозначим звездочкой \* , так как список задач должен занимать как можно большее пространство, а второй ряд с кнопками обозначим auto, чтобы он занимал только необходимое ему количество места.

## Создание модели объекта

Создавать программу мы будем по шаблону MVVM, поэтому нам потребуется создать модель объекта «Задача»

Так как C# — объектно-ориентированный язык программирования, мы можем создать класс. Он позволит нам быстро и безошибочно в будущем создавать объекты с определенными свойствами Нам потребуется создать класс для наших задач. Назовем его «ToDoItem».

Нам потребуются такие свойства, как заголовок (Title) задачи а также индикатор того, была ли она выполнена. Для этого добавим к классу ToDoItem свойство Content типа string (строка) и свойство IsDone типа bool. Bool — логический тип. Он может принимать только два значения: правда или ложь. Стоит различать свойства и поля. Свойства, по аналогии с реальным миром, это то, что мы можем наблюдать. Свойства публичны и их можно получить из объекта в любой части программы. Поля же приватны. К ним можно получить доступ только из конструктора и методов класса, и они используются для хранения той информации, которую необходимо преобразовать перед выводом или проверить, например, пароль, с помощью приватного свойства. Также они нужны, чтобы не допустить изменения той информации, изменение которой извне может повлечь за собой сбой программы. Мы создаем свойства, так как нам потребуется их выводить на экран, но поля также будут использоваться в проекте.

Также классу нужен конструктор. Конструктор — это публичная функция которая будет использоваться для создания объекта. По умолчанию у каждого класса уже есть стандартный конструктор, однако он требует напрямую вводить значения для каждого свойства. Это может быть неудобным. Например, было бы странно каждый раз уточнять, что только что созданная задача еще не выполнена. Поэтому в конструкторе можно задавать свойству IsDone значение false по умолчанию. Также конструктор может проводить различные преобразования входных данных или проверять их валидность перед тем, как создать из них новый объект.

## Создание коллекции объектов

Во многих приложениях требуется создавать группы связанных объектов и управлять ими. Существует два способа группировки объектов: создать массив объектов и создать коллекцию.

Массивы удобнее всего использовать для хранения фиксированного числа строго типизированных объектов и работы с ними.

Коллекции предоставляют более гибкий способ работы с группами объектов. В отличие от массивов, коллекция, с которой вы работаете, может расти или уменьшаться динамически при необходимости. Некоторые коллекции допускают назначение ключа любому объекту, который добавляется в коллекцию, чтобы в дальнейшем можно было быстро извлечь связанный с ключом объект из коллекции.

Коллекция является классом, поэтому необходимо объявить экземпляр класса перед добавлением в коллекцию элементов.

Если коллекция содержит элементы только одного типа данных, можно использовать один из классов в пространстве имен System.Collections.Generic. Универсальная коллекция обеспечивает строгую типизацию, так что в нее нельзя добавить другие типы данных. При извлечении элемента из универсальной коллекции не нужно определять или преобразовывать его тип данных.

Будем использовать универсальный класс List<T>, который позволяет работать со строго типизированными списками объектов. Здесь T в угловых скобках обозначает тип данных этой коллекции, т.е. в нашем случае это будет List<ToDoItem>. Назовем его Items.

Для того, чтобы добавить элементы в коллекцию, можно использовать метод Add(). Теперь мы можем создать несколько задач, создав переменную и передав ей конструктор объекта задачи с помощью кодового слова new и класса ToDoItem.

## Привязка к интерфейсу

Мы создали коллекцию в модели, однако она никак не связана с пользовательским интерфейсом.

Теперь нам понадобится модель представления, представляющая список. Это класс, который будет предоставлять данные для нашего представления. Перейдем к модели представления главного окна MainWindowViewModel. Она находится в каталоге ViewModels в вашем проекте.

Давайте подключим библиотеку DynamicData, которая предоставляет методы для отслеживания изменений в коллекциях. Это библиотека, которая позволяет использовать всю мощь реактивных расширений при работе с коллекциями. Реактивные расширения из коробки не предоставляют оптимальных способов для работы с изменяющимися наборами данных, и задача DynamicData — исправить это. Современные приложения довольно сложны, и зачастую возникает необходимость создавать проекции коллекций —  фильтровать, трансформировать или сортировать элементы. DynamicData была разработана как раз чтобы избавиться от того невероятно сложного кода, который потребовался бы нам для управления динамически меняющимися наборами данных. Инструмент активно развивается и дорабатывается, и уже сейчас поддерживается более 60 операторов для работы с коллекциями.

Можно было бы использовать и ObservableCollection, встроенную в C# — всякий раз, когда коллекция меняется, публиковать новую коллекцию с изменившимися элементами. Но это чревато серьёзными проблемами с производительностью приложения, особенно, если в нашей коллекции не десяток элементов, а сотня, а то и несколько тысяч.

Вместо классического IObservable<T>, DynamicData определяет операции над IObservable<IChangeSet<T>> и IObservable<IChangeSet<TValue, TKey>>, где IChangeSet — чанк, содержащий информацию об изменении коллекции — тип изменения и элементы, которые были затронуты. Такой подход позволяет существенно улучшить производительность кода для работы с коллекциями, написанного в реактивном стиле. При этом, IObservable<IChangeSet<T>> всегда можно трансформировать в обычный IObservable<IEnumerable<T>>

В нашей ViewModel будем считывать коллекцию задач и добавлять все её элементы в SourceList, коллекцию из библиотеки DynamicData, которая будет отправлять уведомления о изменениях в ней нашему графическому интерфейсу.

DynamicData использует встроенные типы .NET для отображения данных во внешний мир. С помощью мощных операторов DynamicData мы можем преобразовать IObservable<IChangeSet<ToDoItem>> в ReadOnlyObservableCollection нашей модели представления.

Далее нам потребуется при помощи метода Bind подключить к этой коллекции обычную ReadOnlyObservableCollection из пространства имен System.Collections.ObjectModel, к которой уже будет привязан список в интерфейсе.

ListBox, однако, не имеет никакого понятия о том, как должен выглядеть на экране созданный нами объект класса ToDoItem. Давайте обозначим это с помощью шаблона данных ListBox.ItemTemplate. Он позволяет указать, какие графические элементы создавать для каждого объекта в коллекции и какие свойства объекта к ним привязывать.

Создадим шаблон, состоящий из уже знакомой Grid, содержащей CheckBox и TextBlock. Первый - это флажок или галочка, которую можно видеть во многих приложениях. Он может принимать 2 значения: true и false. Второй же просто отображает текст. Логично, что к флажку мы привяжем логическое свойство объекта IsDone, а к блоку текста строковое свойство Title.

Теперь мы можем в коде добавить несколько элементов в список и увидеть их в интерфейсе. Это, однако, не позволяет пользователю никак взаимодействовать с данными. Это так называемый Hard-Coding — практика разработки программного обеспечения, при которой данные встраиваются непосредственно в исходный код программы или другого исполняемого объекта, а не получаются из внешних источников или генерируются во время выполнения.

Жестко закодированные данные обычно можно изменить только путем редактирования исходного кода и повторной компиляции исполняемого файла.

## Интеракции. Добавление новых элементов

Давайте дадим пользователю возможность создавать новые элементы и добавлять их в коллекцию. Нам придется принимать пользовательский ввод, и специально для этого в библиотеке ReactiveUI есть Interactions.

Иногда для выполнения кода ViewModel необходимо запросить пользовательский ввод. Например, при редактировании данных, перед удалением файла или после возникновения ошибки. Отображение интерактивного диалога из модели представления — простое решение, но оно привязывает модель представления к конкретной платформе пользовательского интерфейса и затрудняет или делает невозможным тестирование приложения. Interactions — это решение проблемы приостановки выполнения модели представления до тех пор, пока пользователь не предоставит необходимую информацию.

Интеракции принимают ввод и производят вывод. Представления используют ввод для обработки интеракции. Модель представления получает результат взаимодействия. Например, модели представления может потребоваться запросить у пользователя подтверждение перед удалением файла. интеракции взаимодействие, он может передать путь к файлу в качестве входных данных и получить логическое значение в качестве вывода, указывающее, можно ли удалить файл.

Типы ввода и вывода Interaction<TInput, TOutput> являются параметрами универсального типа и, следовательно, находятся под контролем программиста; нет никаких ограничений относительно того, что вы можете использовать в качестве типа ввода или вывода.

Давайте создадим интеракцию, которая будет принимать на вход Unit (ничего), а после пользовательского ввода выводить объект ToDoItem т. е. Interaction<Unit,ToDoItem>. Добавим эту интеракцию в качестве свойства нашей MainWindowViewModel. Также нам потребуется сделать команду класса ReactiveCommand, чтобы привязать к этой команде кнопку или другой элемент интерфейса. Эта команда будет активировать интеракцию и ожидать вывода, после чего добавлять этот вывод, т.е. новый объект ToDoItem в коллекцию SourceList.

Однако, интеракция ничего не знает о том, как создават, например, всплывающее окно и брать из него какие-либо значения. Для этого нужно создать обработчик (Handler) интеракции.

Делать это нужно уже не в ViewModel, а в коде непосредственно MainWindow, т.е. View. Добавим в конструктор MainWindow, т.е. в код, который будет выполняться при создании главного окна программы, обработчик интеракции. Для этого используем метод RegisterHandler(). В обработчике нам нужно будет открывать новое окно, но у нас такого окна пока что нет, так что давайте его создадим Нам понадобится создать новое представление модель представления для диалогового окна создания задачи. Для этого в Avalonia есть шаблоны некоторых элементов программы, таких как окна. Создадим окно по шаблону и получим пустое окно с правильно настроенными классами и пространствами имен. В этом окне создадим поле ввода (TextBox) и привяжем его к свойству модели представления. Также добавим несколько кнопок для создания задачи и отмены операции. Эти кнопки нам потребуется привязать к реактивным командам. Обе команды будут закрывать окно, но одна из них будет возвращать новый объект задачи, а другая Unit, то есть ничего. Очевидно, что первая команда — создание задачи, а вторая — отмена.

Теперь вернемся к главному окну. В обработчике интеракции мы теперь можем создать диалоговое окно классф NewItemView, которое мы только что создали. Для этого в Avalonia есть метод ShowDialog(), который применяется к окну и принимает в качестве аргументов возвращаемое значение и родительское окно, т.е. главное окно программы. В коде NewItemView не забудем подписаться на выполнение команд во ViewModel, чтобы закрывать окно и в качестве вывода отправлять результат команды.

Теперь мы можем нажать кнопку «Добавить», ввести в ней текст и эта задача добавится в наш список. Стоит, однако, подумать о том, что пользователь может не ввести ничего в поле и нажать кнопку «принять». Нам это не нужно: во-первых, у задачи не может быть пустым единственное поле, потому что в этом нет смысла. Также само поле у нас не может быть пустым, и если такое произойдет, то программа выдаст исключение NullReferenceException и вылетит.

Для того, чтобы этого не допустить, привяжем свойство IsActive кнопки принятия задачи к значению поля ввода. IsActive управляет тем, работает ли кнопка. Когда она неактивна, ее нельзя нажать и она отображается неактивной. Мы, естественно, не можем привязать логический тип к строковому.

Нам понадобится Converter (Преобразователь). В Avalonia это класс, который реализует два метода: Convert и ConvertBack, Они могут преобразовывать входное значение к нужному виду, будь то другой формат данных или логическое условия, как в нашем случае. Наследуем наш конвертатор от интерфейса IValueConverter. Это позволит системе понять, что его можно использовать в качестве такового. Также это не позволит приложению скомпилироваться, пока мы не реализуем необходимые для работы преобразователя методы. Добавим метод Convert и простое условие в качестве его реализации. Если входная строка пустая или состоит только из пробелов и переносов строки, будем возвращать значение ложь (false), а в противном случае правда (true). В этом поможет встроенный в класс строки метод string.IsNullOrWhiteSpace(), который делает как раз то, что нам нужно. Нам необходимо реализовать метод ConvertBack, однако в него мы просто добавим вывод исключения NotSupportedException, чтобы было понятно, в чем проблема, если кто-то попытается вызвать этот метод и чтобы программа при этом не скомпилировалась, предотвращая тем самым ошибки во время работы (runtime) программы.

Теперь мы можем привязать строку содержания поля ввода к логическому параметру IsActive кнопки, чтобы не позволить пользователю ломать программу своими действиями.

## Удаление элементов

Одной базовой функции нам не хватает. Это функция удаления элементов. Реализовать ее будет не так сложно. У списка ListBox в Avalonia есть свойство SelectedIndex, которое позволит нам получить индекс выбранного элемента. Привяжем это свойство к свойству в ViewModel главного окна. Теперь создадим реактивную команду, которая будет удалять элемент по индексу с помощью метода SourceList.RemoveAt().

Здесь тоже не обойдется без предотвращения неправильных действий пользователя. Если в таком виде оставить программу, то если пользователь нажмет кнопку удаления, привязанную к данной реактивной команде, то программа выдаст исключение. Привяжем активность кнопки к SelectedIndex. В этот раз конвертер писать самим не придется: в Avalonia уже есть встроенный, который проверяет, есть ли значение у переменной. Он называется IsNull. Null — значение, которое обозначает отсутствие значение. Его можно получить, если, например, объявить переменную, но не присвоить ей значения. Именно таким будет значение SelectedIndex, если в данный момент не выбран никакой элемент в списке.

Укажем встроенный конвертер для привязки, и теперь пользователь не сможет нажать кнопку, которая приведет к аварийному завершению программы.

## Редактирование элементов

Мы, конечно, можем удалять элементы и создавать новые, однако это крайне неудобно. Тогда пользователю придется переписывать весь текст, если он захочет исправить ошибку или как-то дополнить заметку. Создадим уже знакомую нам интеракцию, только в этот раз она будет принимать в качестве ввода уже существующую модель объекта, а потом изменять нужные свойства и возвращать ее для замены. Добавим в интерфейс еще одну кнопку «Изменить» и привяжем ее к соответствующей реактивной команде. Также нам понадобится создать еще одно представление окна, как мы это делали с добавлением задач, а также зарегистрировать обработчик событий в главном окне. Не забудем привязать активность и этой кнопки к тому же свойству ListBox.SelectedIndex, чтобы и тут избежать сбоев в работе.

## Интерфейс, унификация действий с объектом

Заметим, что для создания и изменения объектов используется одинаковый графический интерфейс, однако нам пришлось его писать 2 раза. Давайте сделаем, чтобы можно было использовать одинаковую разметку и некоторую логику для обоих действий. Таким образом нам не придется выполнять двойную работу при изменении разметки окна создания и изменения задачи.

Для этого создадим новую View. Назовём её ItemActionView. Скопируем в файл ItemActionView.axaml разметку окна EditItemView.axaml, а в ItemActionView.axaml.cs, то есть code-behind, скопируем логику EditItemView.axaml.cs.

Однако, не все так просто. Компилятор требует указать ViewModel для окна, а у нас ViewModel редактирования и создания задач все-таки немного отличаются. Однако, мы можем указать также интерфейс, а от этого интерфейса наследовать наши уже существующие ViewModel’и.

Создадим интерфейс IitemActionViewModel. В нем обозначим команды для кнопок, которые необходимо будет реализовать при наследовании: AcceptItemCommand и CancelCommand. Теперь мы можем указать этот интерфейс в качестве типа ViewModel для ItemActionView.

Как же будет определяться, какую ViewModel когда использовать? Все очень просто. В регистрации обработчика интеракции мы уже назначали вьюмодель. Нам нужно всего лишь подменить EditItemView и NewItemView на ItemActionView, а остальное оставить как есть. Так как EditItemViewModel и NewItemViewModel реализуют интерфейс IitemActionViewModel, мы можем спокойно назначить их как DataContext названным выше представлениям.

## Поиск

Когда пунктов в списке станет много, понадобится возможность находить задачи по названию. Для этого в библиотеке DynamicData есть метод Filter. Он позволяет применить к коллекции фильтр, как следует из названия. Таким образом мы сможем довольно легко реализовать поиск.

Для начала добавим поле для ввода TextBox в главное окно приложения. Для этого потребуется лишь добавить в Grid один ряд сверху.

Теперь создадим метод MakeFilter, возвращающий делегат Func, принимающий объект ToDoItem и возвращающий логическое значение в зависимости от содержания строки поиска. Если строка поиска пуста, метод будет возвращать функцию, которая возвращает true для любого элемента. В противном случае, метод вернёт функцию, возвращающую истину в том случае, если заголовок задачи содержит строку поиска. Это будет происходить при помощи метода Contains. К нему добавим аргумент IgnoreCase; Тогда сравнение не будет чувствительным к регистру.

Давайте создадим наблюдаемый объект filter. Поставим его отслеживать поток событий об изменениях строки поиска. Далее применим к потоку данных метод Select, позволяющий преобразовывать данные в событии. в метод Select передадим метод MakeFilter. Теперь у нас есть наблюдаемый объект, который на каждое обновление строки поиска возвращает функцию. Этот наблюдаемый объект поместим в метод Filter нашей динамической коллекции.

Таким образом, при обновлении строки поиска будет обновляться функция, которая определяет, при каких условиях нужно отображать элемент. Теперь в нашей программе есть поиск, который позволит пользователю легко и быстро найти нужную заметку.

## Реактивная ViewModel объекта

У класса ToDoItem поля и свойства самые обычные, и, как выше было сказано, в таких переменных нельзя отслеживать информацию, так как они не присылают уведомления о своем изменении. Нам понадобится их отслеживать, например, чтобы после переименования объект пропадал из отфильтрованного списка.

Для этого можно было бы реализовать интерфейс INotifyPropertyChanged с помощью метода RaiseAndSetIfChanged из ReactiveUI, однако это противоречит концепции MVVM. Если мы, например, в будущем отправим такой реактивный объект на сервер для обработки, то нам потребуется на сервере также скачивать библиотеки ReactiveUI. Они не будут приносить пользу, зато будут занимать место на диске и замедлять код отправкой уведомлений.

Мы можем сделать обертку (Wrapper) для модели ToDoItem. Назовем наш новый класс ItemViewModel и наследуем его от ReactiveObject, как это делается с ViewModel окон. Теперь сделаем все его свойства отслеживаемыми с помощью RaiseAndSetIfChanged. В конструкторе будем принимать объект ToDoItem и присваивать его свойства новому объекту ItemViewModel. Также добавим метод GoBack (), чтобы преобразовывать ItemViewModel обратно в ToDoItem, когда это понадобится.

Теперь мы можем отслеживать свойства объекта. Это нам понадобится в дальнейшей разработке. Немного изменим методы сериализации, чтобы при чтении из Json переводить объекты в класс ItemViewModel, а при сохранении обратно.

## Подробности задач

Пользователям не помешает еще одно поле для полезной информации в заметке; если расписывать что-то подробно в единственном поле, то список быстро растянется и его станет трудно воспринимать. Добавим текстовый блок «Описание» для каждой задачи, которое будет раскрываться по нажатию на стрелочку возле названия задачи в GUI. Таким образом можно будет и быстро просмотреть названия задач, и при необходимости углубиться и посмотреть их подробности.

Очевидно, нам также потребуется реализовать это у ToDoItem и ItemViewModel. Просто добавим одно свойство Content. К нему в DataTemplate привяжем новый текстовый блок. С раскрытием на кнопку, однако, придется сложнее.

Добавим в DataTemplate еще один флажок справа, ни к чему его не привязывая. К его состоянию привяжем видимость текста Content. Это работает, однако выглядит непрезентабельно: мы ведь хотели сделать стрелочку.

Для этого переопределим ControlTemplate у флажка. Теперь мы можем сменить то, как он представляет свои свойства, такие, как IsChecked. Сменим флажок на элемент «Image», и привяжем его к состоянию IsChecked самого флажка. Очевидно, логический тип не может стать изображением, поэтому сделаем конвертер, который будет выдавать адрес иконки в зависимости от значения на входе.

Также добавим поле для ввода подробностей в интерфейс создания и редактирования задач и будем передавать его значение в конструктор задач.

Теперь у нас есть раскрывающиеся подробности, однако не всем задачам они нужны. Для некоторых достаточно нескольких слов, так что стрелочка и обязательное поле для ввода будем лишним. Для таких случаев мы можем сделать свойство Content Nullable, то есть разрешить ему не принимать никакое значение и быть пустым. Для этого просто допишем знак вопроса после типа свойства. В интеракции создания и изменения добавим механизм, который будет делать Content Null, если пользователь оставит в поле только пробелы.

Осталось только разобраться со стрелочкой, которая вводит в заблуждение, появляясь рядом с задачами с пустым полем подробностей. Привяжем видимость стрелочки к свойству Content с уже знакомым нам конвертером IsNotNull. Теперь стрелочка не будет отображаться, если Content задачи не задан.

## Идентификатор объектов

Поскольку объекты хранятся в SourceList, т. е. нумерованном списке, они идентифицируются по порядковому номеру в списке. При поиске же порядок и количество элементов изменяется, из-за чего поведение функций редактирования и удаления становилось непредсказуемым. Для того, чтобы это исправить необходимо предоставить каждому объекту свой уникальный идентификатор, который не будет зависеть от визуального расположения объекта на экране.

Если мы собираемся выдавать объектам уникальный идентификатор, так называемый первичный ключ, то нам подойдет SourceCache вместо SourceList. Это еще одна динамическая коллекция, которая больше напоминает стандартный словарь Dictionary<T, TKey>.

Где же взять ключ? Давайте создадим метод KeyGen () в ViewModel. Он будет перебирать числа от 0 до предела UInt32, т. е. до 4294967295. Здесь мы использовали именно UInt32, поскольку он вместо 2 миллиардов отрицательных значений обычного Int32 включает дополнительные 2 миллиарда положительных. При переборе метод будет проверять, есть ли в коллекции SourceCache объект с таким ключом, и возвращать первый незанятый ключ. Далее добавим ItemViewModel свойство Key, а в конструкторе будем принимать еще и число, присваемое Key.

Вот нам и пригодилась отдельная ItemViewModel. Ведь в модели для хранения или отправки объектов на сервер необязательно хранить еще и ключ. Ключ необходим только во время работы графического интерфейса программы. Таким образом мы не нарушаем схему MVVM.

Теперь при добавлении нового элемента в список будем вызывать метод KeyGen, чтобы выдавать ID задаче. Никаких ошибок при взаимодействии с объектами не будет возникать, вне зависимости от порядка и положения объектов в интерфейсе.

## Реализация Markdown

Было бы полезно добавить возможность форматирования заметок; выделять их курсивом, жирным текстом или зачеркивать, а также добавлять гиперссылки и изображения. Для этого есть библиотека Markdown.Avalonia, позволяющая отображать документы Markdown прямо внутри программы на Avalonia.

Интегрировать данный язык разметки с программой не так сложно. Понадобится лишь сменить в DataTemplate элемент, отображающий Content заметки. Сейчас это свойство выводится обычным текстом с помощью TextBlock. Сменим его на md:MarkdownScrollViewer.

Теперь мы можем писать заметки на языке Markdown. Стоит заметить, что данный язык разметки вообще не навязчивый, т. е. никто не мешает писать обычным текстом, не пользуясь разметкой, пользователям, которые не знают Markdown или не хотят в конкретный момент использовать разметку.

# Выводы

Данный проект был направлен на изучение и разработку компьютерных программ с графическим интерфейсом.

Во время работы над проектом были изучены и сравнены технологии разработки приложений для ПК, а также получены практические навыки в разработке программы на платформе Microsoft.NET Core и GUI-фреймворке AvaloniaUI.

Все библиотеки и ниструменты обновлялись до новейших версий, чтобы получить доступ к новым функциям и исправлениям ошибок. На момент окончания работы над проектом эти версии:

* .NET Core 7.0
* AvaloniaUI 11.0.0-preview5
* ReactiveUI 18.4.26
* DynamicData 7.13.1
* Markdown.Avalonia 11.0.0-a9

В результате работы над проектом был получен конечный продукт в виде приложения «Список задач» для распространеных ОС настольных компьютеров; Microsoft Windows, Linux и MacOS.

Продукт реализует базовые функции CRUD: создание, чтение, редактирование и удаление данных. В качестве дополнительного функционала присутствует возможность написания заметок на языке разметки Markdown, а также инкрементный поиск, позволяющий в реальном времени фильтровать коллекцию объектов. Еще имеется возможность расширенного просмотра свойств объектов в выпадающем текстовом блоке.

Окончательный размер исходного кода программы составил:

* 556 строк в 17 файлах C#
* 270 строк разметки XAML в 3 файлах
* 19 строк комментариев

Числа могут казаться огромными для не слишком сложного проекта, однако стоит отметить, что некоторая часть — шаблонный код, который генерируется вместе с новым проектом, например, одни только пространства имен XML занимают примерно 10 строк в каждом файле. Также были расставлены пустые строки и применены программы форматирования кода, которые разбивают длинные строки и цепочки вызовов функций на несколько строк для лучшей читаемости. Не стоит забывать, что Visual Studio значительно облегчает написание кода на таком многословном языке программирования как C#. Некоторые методы в названии имеют до 5 слов, однако их не нужно было полностью писать каждый раз благодаря автодополнению.

Программа запускается на компьютере разработчика менее чем за секунду и занимает 200 МБ оперативной памяти. Приложение протестировано и работает и выглядит идентично на ОС Linux.

# Список использованных источников

3 книги, 4 статьи, 6 интернет-ресурсов:

* Язык программирования C# 7 и платформы.NET и.NET Core, 8-е изд. — Эндрю Троелсен, Филипп Джепикс (2018)
* Паттерны проектирования на платформе.NET – Сергей Тепляков (2015)
* CLR via C#. Программирование на платформе Microsoft.NET Framework 4.5 на языке C# – Джеффри Рихтер (2022)
* <https://tproger.ru/translations/reactive-programming/> — статья о принципах реактивного программирования
* <https://stackoverflow.com/> — популярнейший сайт, где разработчики ищут помощи других в решении технических задач
* <https://habr.com/ru/amp/post/552872/> — сравнение технологий кроссплатформенной разработки
* <https://habr.com/ru/post/528614/> — статья о системах контроля версий
* <https://www.reactiveui.net/docs/> — официальная документация фреймворка ReactiveUI
* <https://www.codeproject.com/Articles/5308645/Multiplatform-UI-Coding-with-AvaloniaUI-in-Easy-Sa> — обучающая статья с примерами кода для AvaloniaUI
* <https://www.dotnetperls.com/>  — обучающий сайт с примерами встроенных функций C# и многих других языков программирования.
* <https://learn.microsoft.com/en-us/docs/> — официальная документация Microsoft по C#
* <https://docs.avaloniaui.net/> — официальная документация AvaloniaUI
* <https://nbj.ru/blogz/Sber/60208/> — исследование о предпочтениях россиян в способах хранения информации