**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ**

**БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

**ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ**

**Кафедра технологий программирования**

**РАЗРАБОТКА DESKTOP ПРИЛОЖЕНИЯ**

**НА БАЗЕ WPF**

Отчет по производственной практике

|  |
| --- |
| Доскоча Романа Дмитриевича |
| студента 3 курса, специальность  «прикладная информатика» |
| Руководитель практики: |
| доцент, |
| В. В. Горячкин |

|  |  |
| --- | --- |
|  |  |

Минск, 2022

**СОДЕРЖАНИЕ**

[**ВВЕДЕНИЕ** 3](#_Toc113753911)

[**Глава 1 Введение в платформу WPF** 3](#_Toc113753912)

[**1.1** **Общие сведения** 3](#_Toc113753913)

[**1.2 Особенности технологии** 3](#_Toc113753914)

[**1.3 Data Binding** 3](#_Toc113753915)

[**1.4 Стили** 4](#_Toc113753916)

[**1.5 Шаблоны элементов управления** 4](#_Toc113753917)

[**1.6 Графика** 5](#_Toc113753918)

[**Глава 2. Разработка Desktop приложения** 6](#_Toc113753919)

[**2.1 Реализация всплывающего баннера Sutisfaction survey** 6](#_Toc113753920)

[**2.2 Перенос общего функционала под спецификацию MVVM паттерна.** 8](#_Toc113753921)

[**2.3 Best Practice AsyncCommand** 11](#_Toc113753922)

[**2.4 Реализовать подход one-instance приложения.** 14](#_Toc113753923)

[**ЗАКЛЮЧЕНИЕ** 17](#_Toc113753924)

[**ПРИЛОЖЕНИЯ** 18](#_Toc113753925)

[*Приложение А* 18](#_Toc113753926)

[*Приложение Б* 19](#_Toc113753927)

# **ВВЕДЕНИЕ**

Программирование настольных приложений положило начало эре разработки программного обеспечения и долгое время удерживало лидирующие позиции в этой сфере. Однако с быстрым развитием технологий многие предприятия и пользователи начали больше полагаться на веб-приложения, к которым можно получить доступ через Интернет.

Новые тенденции заставляют перейти весь мир на использование веб приложений, и, казалось бы, что настольные приложения уже по не многу исчерпывают свои потребности, но все же остаются те случаи, когда без их не обойтись, такими случаями могут быть:

* корпоративные приложения — компании, которым требуются приложения с расширенными функциональными возможностями, которые должны выполнять различные сложные задачи и обрабатывать данные из нескольких отделов;
* приложения в фоновом режиме — иногда пользователям нужны приложения для работы в фоновом режиме, например, служба Steam для запуска игр или загрузчик файлов;
* низкоуровневое программное обеспечение — программное обеспечение, обеспечивающее правильную работу компьютера, например, драйверы для Linux или Windows;
* игры со сложным пользовательским интерфейсом — игры, требующие высокой производительности, занимающие много места и сильно зависящие от графического процессора, могут разрабатываться только как настольные приложения;
* программное обеспечение для бизнеса — некоторым компаниям необходимо собирать и хранить большие объемы конфиденциальных данных; для этого они выбирают настольную разработку, поскольку она позволяет хранить все данные на жестких дисках компьютеров, а не отправлять их на внешние серверы.

Производственная практика проходила в компании **EPAM Systems**.

**EPAM Systems** — американская ИТ-компания, основанная в 1993 году. Штаб-квартира компании расположена в Ньютауне, штат Пенсильвания, а её отделения представлены более чем в 40 странах мира.

**Признание**

* В 2013 году компания вошла в списки Forbes «25 самых быстрорастущих технологических компаний Америки» и «20 самых быстрорастущих технологических звёзд Америки»
* В 2014 году EPAM названа лидером в отчёте «The Forrester Wave: Software Product Development Services, Q1 2014». По мнению Forrester, «EPAM опередила всех других поставщиков по продвижению инноваций и содействию в создании новых инновационных продуктов».
* По итогам 2016 года EPAM занял 3-е место в рейтинге CNews «Крупнейшие ИТ-компании России 2016», а также 1-е место среди разработчиков ПО в рейтинге РБК+ российских ИТ-компаний (3-е место в общем списке).
* В 2017 году EPAM снова включен в ежегодный рейтинг Forbes «25 самых быстрорастущих публичных технологических компаний Америки».
* В 2021 EPAM Systems совместно с Mars Incorporated были объявлены победителями в номинации «Leader of the Pack: Retail» премии «Acquia Engage Award 2021». Наградами в данной номинации отмечаются цифровые решения и продукты, отличающиеся высоким уровнем функциональности, интеграции, производительности и удобства для пользователя.

В рамках производственной практики были выданы следующие задания:

1. Реализация всплывающего баннера Sutisfaction survey
2. Перенос общего функционала под спецификацию MVVM паттерна.
3. Применение паттерна AsyncCommand для эффективной работы с асинхронными задачами.
4. Реализовать подход one instance приложения.

# **Глава 1 Введение в платформу WPF**

# **Общие сведения**

Windows Presentation Foundation – это платформа пользовательского интерфейса, которая создает настольные клиентские приложения. Платформа разработки WPF поддерживает набор функций разработки приложений, включая модель приложения, ресурсы, элементы управления, графику, макет, привязку данных, документы и безопасность.

WTF является частью .NET, поэтому, если вы ранее создавали приложения с помощью .NET, используюя Windows Forms, опыт программирования должен быть знакомым. WPF использует расширяемый язык разметки приложений (XAML) для обеспечения декларативной модели прикладного программирования.

# **1.2 Особенности технологии**

В основе WPF лежит векторная система визуализации, не зависящая от разрешения устройства вывода и созданная с учётом возможностей современного графического оборудования. WPF предоставляет средства для создания визуального интерфейса, включая язык XAML (eXtensible Application Markup Language), элементы управления, привязку данных, макеты, двухмерную и трёхмерную графику, анимацию, стили, шаблоны, документы, текст, мультимедиа и оформление.

XAML представляет собой язык декларативного описания интерфейса, основанный на XML. Также реализована модель разделения кода и дизайна, позволяющая кооперироваться программисту и дизайнеру. Кроме того, есть встроенная поддержка стилей элементов, а сами элементы легко разделить на элементы управления второго уровня, которые, в свою очередь, разделяются до уровня векторных фигур и свойств/действий. Это позволяет легко задать стиль для любого элемента, например, Button (кнопка).

Графической технологией, лежащей в основе WPF, является DirectX, в отличие от Windows Forms, где используется GDI/GDI+. Производительность WPF выше, чем у GDI+ за счёт использования аппаратного ускорения графики через DirectX.

Также существует урезанная версия CLR, называющаяся WPF/E, она же известна как Silverlight.

# **1.3 Data Binding**

Это гибкий механизм, который позволяет через расширения разметки XAML связывать различные данные (от значений свойств элементов управления до общедоступных свойств, реализующих поля базы данных через Entity Framework). Привязка данных представлена классом Binding, который в свою очередь унаследован от MarkupExtension, что позволяет использовать привязки не только в коде, но и в разметке:

**<StackPanel** Orientation="Horizontal"**>**

**<Slider**

x:Name="slider"

Minimum="1"

Maximum="100"

Value="60"**/>**

**<TextBox**

Text="{Binding ElementName=slider, Path=Value}"**/>**

**</StackPanel>**

Помимо основного класса Binding в WPF реализовано еще несколько механизмов привязок:

* **MultiBinding** — позволяет создавать множественные привязки, указывая несколько элементов
* **TemplateBinding** — используется в шаблонах для связывания свойства элемента внутри шаблона со свойством элемента, к которому применен шаблон
* **PriorityBinding** — ранжирует список привязок и выбирает из них свойство (согласно приоритету) к которому будет применена привязка. Если привязка, имеющая наивысший приоритет успешно возвращает значение, то нет необходимости обрабатывать другие привязки в списке.

# **1.4 Стили**

Позволяют создавать стилевое оформление элементов и, как правило, используются только в разметке:

**<Button>**

**<Button.Style>**

**<Style** TargetType="Button"**>**

**<Setter**

Property="FontSize"

Value="20"**/>**

**<Setter**

Property="Foreground"

Value="LimeGreen"**/>**

**</Style>**

**</Button.Style>**

**</Button>**

Если стиль задается в ресурсах (например в словаре ресурсов), то можно использовать атрибут x:Key для указания уникального ключа. Затем в элементе управления, к которому необходимо применить стиль, нужно использовать расширение разметки StaticResource для связывания с этим стилем. Если использовать этот прием, то стили не будут нагромождать разметку.

# **1.5 Шаблоны элементов управления**

Позволяют менять графическое оформление элементов и представлены классом ControlTemplate. В отличие от стилей, можно менять не только графическое представление элемента, но и его структуру. При этом шаблон элемента управления задается через свойство Template.

Простой пример круглой кнопки:

**<Button**

Content="Hey!"

Background="LimeGreen"

Foreground="White"**>**

**<Button.Template>**

**<ControlTemplate** TargetType="Button"**>**

**<Grid>**

**<Ellipse**

Fill="{TemplateBinding Background}"

Stroke="{TemplateBinding BorderBrush}"

Stretch="Fill"**/>**

**<ContentPresenter**

VerticalAlignment="Center"

HorizontalAlignment="Center"**/>**

**</Grid>**

**</ControlTemplate>**

**</Button.Template>**

**</Button>**

# **1.6 Графика**

WPF представляет обширный, масштабируемый и гибкий набор графических возможностей:

* Графика, не зависящая от разрешения и устройства. Основной единицей измерения в графической системе WPF является аппаратно-независимый пиксель, который составляет 1/96 часть дюйма независимо от фактического разрешения экрана.
* Дополнительная поддержка графики и анимации. WPF упрощает программирование графики за счет автоматического управления анимацией. Разработчик не должен заниматься обработкой сцен анимации, циклами отрисовки и билинейной интерполяцией
* Аппаратное ускорение. Графическая система WPF использует преимущества графического оборудования, чтобы уменьшить использование ЦП.

WPF предоставляет библиотеку общих двухмерных фигур, нарисованных с помощью векторов, таких, как прямоугольники и эллипсы, а также графические пути. И в своей функциональности фигуры реализуют многие возможности, которые доступны обычным элементам управления.

Двухмерная графика в WPF включает визуальные эффекты: градиенты, точечные рисунки, чертежи, рисунки с видео, поворот, масштабирование и наклон.

WPF также включает возможности трехмерной отрисовки, интегрированные с двухмерной графикой, что позволяет создавать более яркий и интересный пользовательский интерфейс.

# **Глава 2. Разработка Desktop приложения**

# **2.1 Реализация всплывающего баннера Sutisfaction survey**

**Задача**:

Реализовать всплывающее окно Sutisfaction survey баннер.

**Требования**:

1. Реализовать по образу UI/UX команды
2. Sutisfaction survey баннер должен показываться пользователю в течении 2 недель после установки приложения.
3. Если пользователь перейдет по ссылке баннера, тот должен исчезнуть в дальнейшем для текущего пользователя
4. У каждого пользователя свой баннер

**Реализация:**

1. реализация UI составляющей
2. Имплементация функционала
3. Тестирование

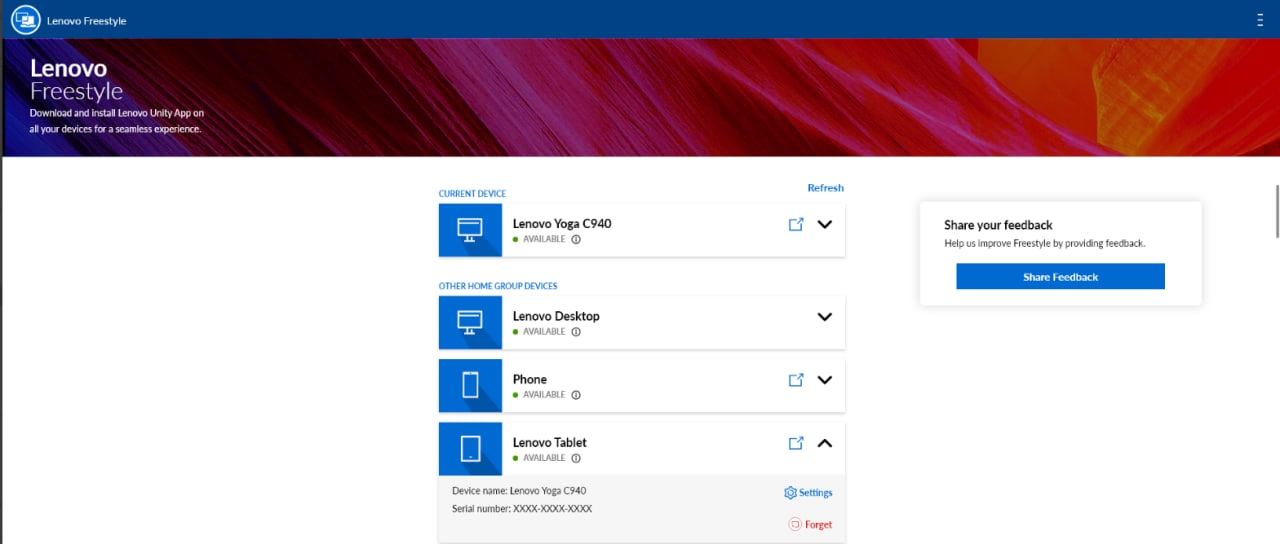


Рисунок 2.1.1 Общий план приложения

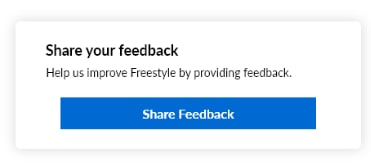


Рисунок 2.1.2 Sutisfaction survey баннер

Основной сложностью в создании UI была полупрозрачная тень подсветка. Так как такой тени нет в стандартных свойствах я решил это следующим способом: добавить на задний слой прямоугольник светло-серого цвета чуть большего за сам баннер, и применить к нему эффект размытия, таким образом можно воссоздать этот эффект. Код реализации UI части можно найти в приложении А.

Следующим этапом является реализация функционала и поведения. Кнопка ссылается на сайт отзывов, я добавил при нажатии пользователем переход по ссылке на этот сайт. Далее стаяла задача временного появления баннера в течении 2 недель с момента установки приложения.

Сначала я решил, что можно найти событие часов компьютера и подписавшись на его изменение можно управлять таким образом появлением. Но в ходе исследования выяснилось, что это невозможно, существует только события ручного изменения времени.

public static event EventHandler? TimeChanged ;

Я придумал другое решение для этого нам понадобится некий таймер, который будет проверять состояние наступления определённой даты и изменять появления баннера. Например, каждую секунду таймер проверяет текущее время и сравнивает его с 2-недельным периодом.

В .NET существует несколько видом таймеров.

1. **System.Threading.Timer** - Ставит задачу в пул потоков, поэтому подходит для тех, случаев, когда задача выполняется по таймеру в фоновом режиме. Например, ваш объект изменяет параметр Х, но на форме этого не видно.

2. **System.Windows.Threading.DispatcherTimer** - вызывает исполнение метода в том же потоке, что и был вызван. Благодаря этому, обеспечивается доступ к элементам графического интерфейса, но при этом сюда лучше не помещать сложную вычислительную логику.

Исходя из описаний документации можно сделать вывод что в нынешней ситуации нам подойдет **DispatcherTimer** ибо нужно скрывать баннер – это изменения UI объекта.

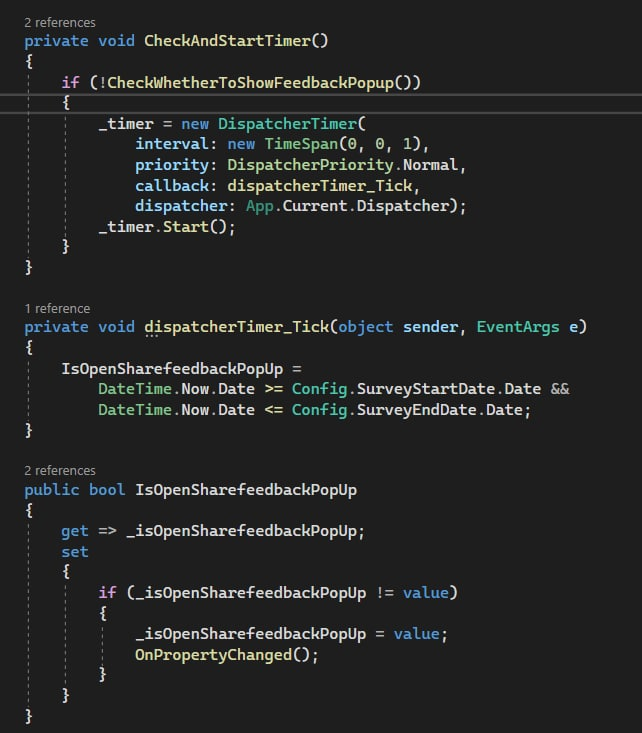


Рисунок 2.1.3 Код использования таймера

Еще одно требование к баннеру что если конкретный пользователь переходит по ссылке баннера, то сам баннер должен скрываться из вида. Но если пользователь перелогинится под другим аккаунтом, то баннер должен снова появится. Эта задача решается с помощью введение флага значение которой записывается в файл настроек, который уникален для каждого пользователя, тем самым решая это требование.

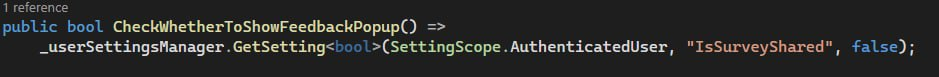


Рисунок 2.1.4 Проверка флага IsSurveyShared

Тестирование показало правильную работу, и команда QA закрыла тикет.

# **2.2 Перенос общего функционала под спецификацию MVVM паттерна.**

**Задача**:

Рефакторинг кода под спецификацию MVVM паттерна.

**Реализация:**

1. Поиск мест с взаимодействия View и Model
2. Перенос связи по MVVM
3. Проверка правильности работы

Паттерн **MVVM (Model-View-ViewModel)** позволяет отделить логику приложения от визуальной части (представления). Данный паттерн является архитектурным, то есть он задает общую архитектуру приложения.

Данный паттерн был представлен Джоном Госсманом (John Gossman) в 2005 году как модификация шаблона Presentation Model и был первоначально нацелен на разработку приложений в WPF. И хотя сейчас данный паттерн вышел за пределы WPF и применяется в самых различных технологиях, в том числе при разработке под Android, iOS, тем не менее WPF является довольно показательной технологией, которая раскрывает возможности данного паттерна.

MVVM состоит из трех компонентов: модели (Model), модели представления (ViewModel) и представления (View).



**Рисунок 2.2.1 Схема MVVM**

**Model**

Модель описывает используемые в приложении данные. Модели могут содержать логику, непосредственно связанную этими данными, например, логику валидации свойств модели. В то же время модель не должна содержать никакой логики, связанной с отображением данных и взаимодействием с визуальными элементами управления.

Нередко модель реализует интерфейсы INotifyPropertyChanged или INotifyCollectionChanged, которые позволяют уведомлять систему об изменениях свойств модели. Благодаря этому облегчается привязка к представлению, хотя опять же прямое взаимодействие между моделью и представлением отсутствует.

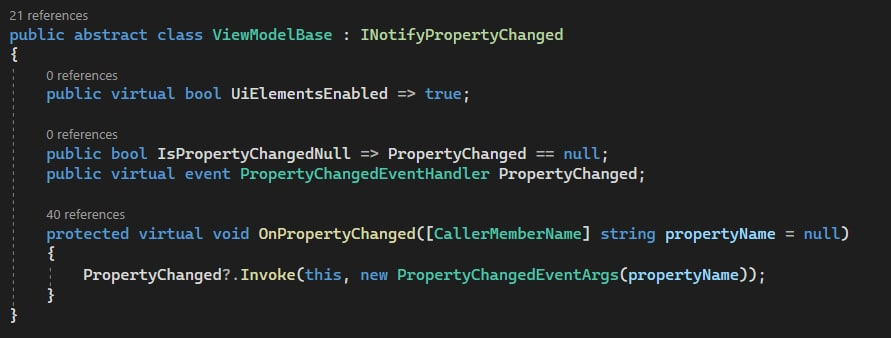


Рисунок 2.2.2 Базовый класс для всех моделей ViewModelBase

**View**

View или представление определяет визуальный интерфейс, через который пользователь взаимодействует с приложением. Применительно к WPF представление - это код в xaml, который определяет интерфейс в виде кнопок, текстовых полей и прочих визуальных элементов.

Хотя окно (класс Window) в WPF может содержать как интерфейс в xaml, так и привязанный к нему код C#, однако в идеале код C# не должен содержать какой-то логики, кроме разве что конструктора, который вызывает метод InitializeComponent и выполняет начальную инициализацию окна. Вся же основная логика приложения выносится в компонент ViewModel.

Однако иногда в файле связанного кода все может находиться некоторая логика, которую трудно реализовать в рамках паттерна MVVM во ViewModel.

Представление не обрабатывает события за редким исключением, а выполняет действия в основном посредством команд.

**ViewModel**

ViewModel или модель представления связывает модель и представление через механизм привязки данных. Если в модели изменяются значения свойств, при реализации моделью интерфейса INotifyPropertyChanged автоматически идет изменение отображаемых данных в представлении, хотя напрямую модель и представление не связаны.

ViewModel также содержит логику по получению данных из модели, которые потом передаются в представление. И также VewModel определяет логику по обновлению данных в модели.

Поскольку элементы представления, то есть визуальные компоненты типа кнопок, не используют события, то представление взаимодействует с ViewModel посредством команд.

Например, пользователь хочет сохранить введенные в текстовое поле данные. Он нажимает на кнопку и тем самым отправляет команду во ViewModel. А ViewModel уже получает переданные данные и в соответствии с ними обновляет модель.

Итогом применения паттерна MVVM является функциональное разделение приложения на три компонента, которые проще разрабатывать и тестировать, а также в дальнейшем модифицировать и поддерживать.

# **2.3 Best Practice AsyncCommand**

**Задача**:

Понять и решить проблему недоступности кнопки refresh после сна компьютера. Эта проблема случается изредка.

**Реализация:**

1. Исследование проблемы такого поведения.
2. Анализ найденного решения на специфику применения.
3. Внедрение найденного решения для успешного устранения.
4. Тестирование.

В WPF любая кнопка состоит из ее UI представления и описания функционала по средству описания специального атрибута Command. В данном случае описывается команда Command="{Binding LoadDevicesAsyncCommand}".

В классе вьюмодели мы можем найти описание функции выполнения команды LoadDevicesAsyncCommand. Это так называемый MVVM подход. Инициализация команды

LoadDevicesAsyncCommand = new AsyncCommand(LoadDevicesAsync);

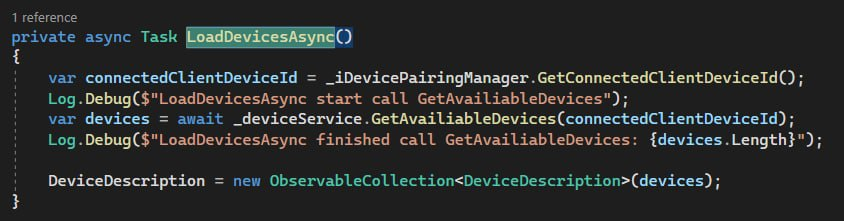


Рисунок 2.3.1 Реализация команды refresh кнопки

Основная сложность состояла в том что я не мог воспроизвести данный баг на своем устройстве. Пришлось пересмотреть реализацию более глубоко и понять что в принципе влияет на блокировку кнопки.

Все дело оказалось в методе CanExecute(). Это метод который находится в реализации класса AsyncCommand он непосредственно участвует в процессе асинхронного выполнения команды. Этот метод возвращает булевское значение показывающее доступность выполнения данной команды, если этот метод возвращает false то кнопка будет заблокирована на время, пока метод не вернет true.

Допустим если пользователь нажимает на кнопку авторизации несколько раз подряд команда не успевает выполнится и CanExecute() возвращает отрицательное значение. Это сделано для сохранения работы приложения от лишних вычислительных процессов.

Я предположил, что этот метод может зациклиться в потоках и все время возвращать false что приведет к извечной блокировке кнопки. Эта проблема может случаться в изредка, потому что состояние гонки не управляемый процесс.

Так как класс AsyncCommand отдан на разработку каждому разработчику самостоятельно и не имеет библиотечной реализации, то не факт что наша реализация предусмотрела все тестовые случаи.

Я начал искать best practice по написанию AsyncCommand класса, который смог бы предусмотреть все случаи и возможно это смогло бы решить мою проблему. Майкрософт предоставляет материалы best practice по данной теме. **Async Programming: Patterns for Asynchronous MVVM Applications**

Расскажу основные идеи, которые были описаны в данной практике. Универсального решения для асинхронной команды, подходящего всем, пока не существует. Сообщество разработчиков все еще изучает асинхронные шаблоны пользовательского интерфейса.

В частности, не существует идеальной истории с несколькими асинхронными операциями. Недостатки есть как в рабочих очередях, так и в уведомлениях, и мне кажется, универсальный UX еще предстоит разработать. По мере того как все больше пользовательских интерфейсов становятся асинхронными, все больше умов будут думать об этой проблеме, и революционный прорыв может быть не за горами.

Прежде чем начать, взгляните на интерфейс ICommand:

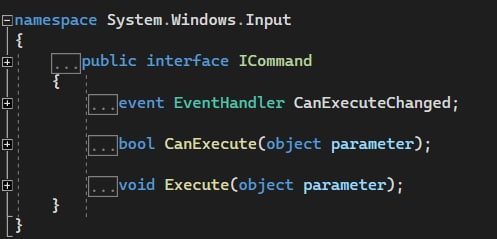


Рисунок 2.3.2 Интерфейс ICommand

Что бы реализовать свой IAsyncCommand лучше свести код к минимуму в асинхронном методе void и предоставить вместо него асинхронный метод Task, который содержит реальную логику. Эта практика делает код более тестируемым. Имея это в виду, я предлагаю следующий интерфейс в качестве асинхронного командного интерфейса

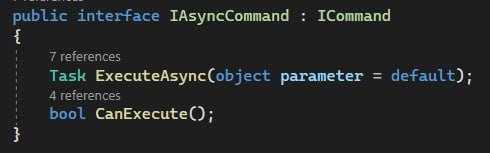


Рисунок 2.3.3 Интерфейс ICommandAsync

Теперь реализуем класс AsyncCommand который и будет служить основой в асинхронных командах код можно найти в приложении Б. Одной из самых главных деталей является CanExecuteChanged и базовый класс передает реализацию CanExecuteChanged классу CommandManager. Он ожидает результата, чтобы гарантировать, что любые исключения в логике асинхронной команды будут правильно переданы в основной цикл потока пользовательского интерфейса

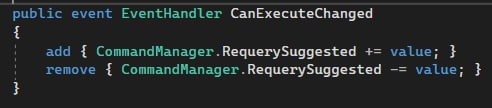


Рисунок 2.4.4 Событие CanExecuteChanged

Основной проблемой старой реализации было не правильное использование CommandManager в связке с CanExecuteChanged, вот их возможности:

1. CanExecuteChanged уведомляет любые источники команд (такие как a Button), когда возвращаемое, CanExecute изменилось. Источники команд заботятся об этом, потому что им обычно необходимо соответствующим образом обновлять свой статус (например, a Button отключит себя, если CanExecute()вернет false).
2. Событие CommandManager.RequerySuggested возникает всякий раз, когда CommandManager считает, что что-то изменилось, что повлияет на возможность выполнения команд. Например, это может быть смена фокуса. Оказывается, это событие вызывает много срабатываний.

Таким образом, по сути, этот фрагмент кода гарантирует, что всякий раз, когда диспетчер команд считает, что возможность выполнения команды изменилась, команда будет инициирована, CanExecuteChanged даже если она на самом деле не изменилась.

После тестирования данного подхода тестер выявивший баг сообщил что проблема больше не появляется.

# **2.4 Реализовать подход one-instance приложения.**

**Задача**:

Реализовать подход one-instance приложения.

**Требования**:

1. При повторном запуске любым способом приложения нужно открывать тот же экземпляр приложения, то есть не создавать копии окон

**Реализация:**

1. Исследования методов в реализации данной функциональности
2. Написание кода реализации
3. Тестирование

Я нашел очень хороших блог посвященный этой теме и попытался реализовать данный подход. Идея заключается в создании так называемого именованного мьютекса, объекта синхронизации. У него есть метод Mutex.WaitOne. Он имеет перегрузку, которая указывает время ожидания. Поскольку мы на самом деле не хотим синхронизировать наш код (более того мы просто проверяем, используется ли он в данный момент), мы используем перегрузку с двумя параметрами: Mutex.WaitOne(Timespan timeout, bool exitContext).

Ожидание одного процесса возвращает true, если он может войти, и false, если это не так. В этом случае мы вообще не хотим ждать; если наш мьютекс используется, пропустите его и двигайтесь дальше, поэтому мы передаем TimeSpan.Zero 0 миллисекунд и устанавливаем для exitContext значение true, чтобы мы могли выйти из контекста синхронизации, прежде чем пытаться получить блокировку. Используя это, мы заключаем наш код Application.Run во что-то вроде этого:

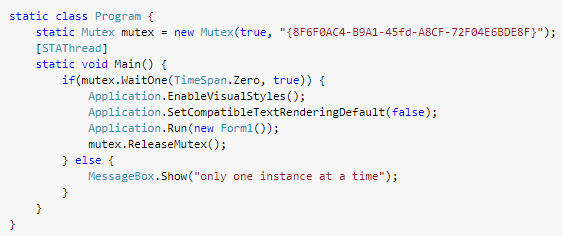


Рисунок 2.4.1 Основная идея мьютекса

Вместо того, чтобы показывать окно сообщения, я решил использовать небольшой Win32, чтобы уведомить мой запущенный экземпляр о том, что кто-то забыл, что он уже запущен (поместив себя поверх всех других окон). Для этого я использовал функцию Win32 SetForegroundWindow

[DllImport("user32")]

static extern IntPtr SetForegroundWindow(IntPtr hWnd);

Выводит поток, создавший указанное окно, на передний план и активирует окно. Ввод с клавиатуры направлен в окно, а для пользователя изменены различные визуальные подсказки. Система назначает немного более высокий приоритет потоку, создавшему окно переднего плана, чем другим потокам.

В параметры этой функции входит дескриптор окна рабочего приложения, для того что бы найти его можно воспользоваться еще одной Win32 функцией FindWindow

[DllImport("user32", CharSet = CharSet.Unicode)]

static extern IntPtr FindWindow(string cls, string win);

Которая по названию окна приложения может найти и вернуть дескриптор окна приложения.

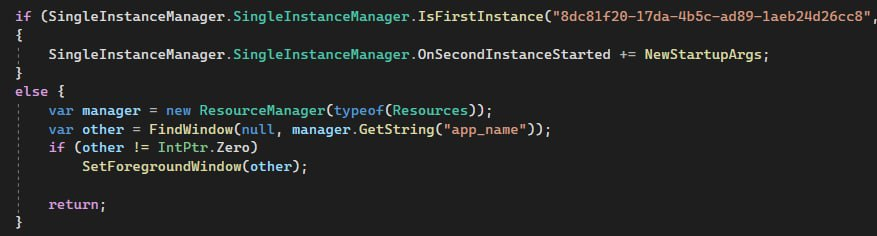


Рисунок 2.4.2 Проверка на One Instance приложение

Тестирование прошло успешно, задача может считаться закрытой.

# **ЗАКЛЮЧЕНИЕ**

В ходе практики на предприятии ИООО ЕПАМ Системз были выполнены все задания, исправлены баги в реализации десктоп-приложения, разработаны новые UI компоненты, и проведен рефакторинг команд на MVVM паттерн. Разработка велась на языке программирования C# под платформой WPF.

Кроме того, были изучения аспекты использования соответствующих инструментов (язык программирования С# под WPF, языка разметки XAML, работа с JIRA и совместной работе в команде) на практике, в промышленной разработке.

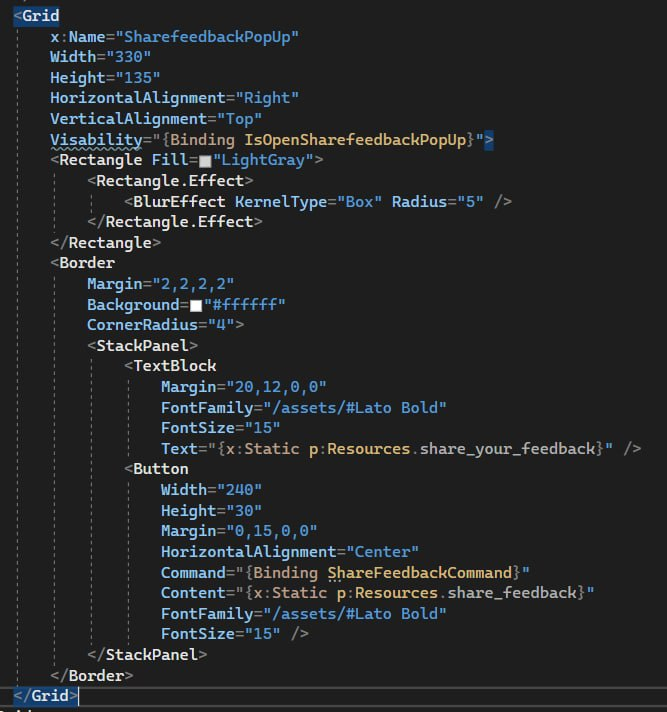
Цели, поставленные на период производственной практики, достигнуты. Ход выполнения заданий проконтролирован руководителем со стороны предприятия.

Разработка для настольных ПК будет продолжать развиваться и улучшаться в будущем. Это подтверждается тем фактом, что с выпуском Windows 10 Microsoft пересмотрела и реформировала свои инструменты разработки для настольных ПК и запустила платформу под названием Universal Windows Platform (UWP). Тем временем он по-прежнему поддерживает свое предыдущее решение для разработки Windows Presentation Foundation (WPF).

Некоторые эксперты прогнозируют, что в ближайшем будущем мы столкнемся с новым типом приложений, которые объединят лучшие возможности веб-разработки и разработки для настольных компьютеров. Таким образом, эти приложения будут легко доступны в Интернете и будут быстро разрабатываться, оставаясь при этом безопасными и мощными.

# **ПРИЛОЖЕНИЯ**

# *Приложение А*



# *Приложение Б*

