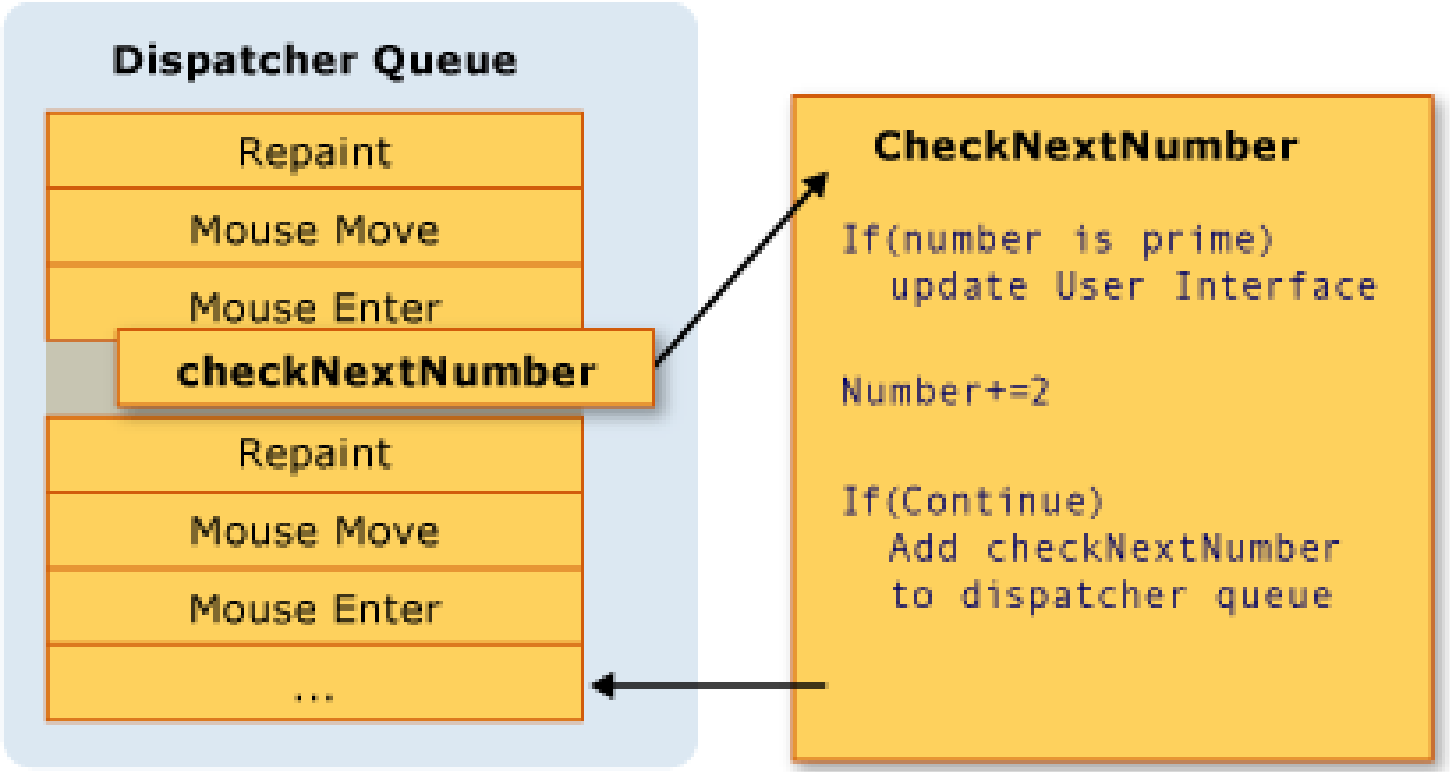
Программирование и обработка графического интерфейса

Лекция №3: Интерфейсы, делегаты, события, async, await

Dispatcher Thread – поток UI

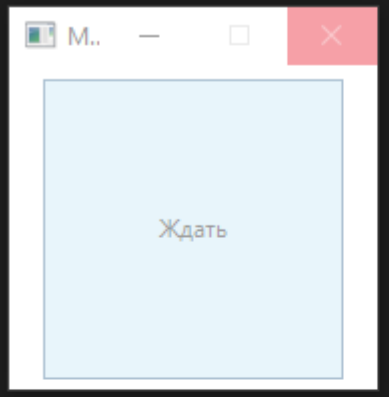
В WPF есть ОДИН главный поток

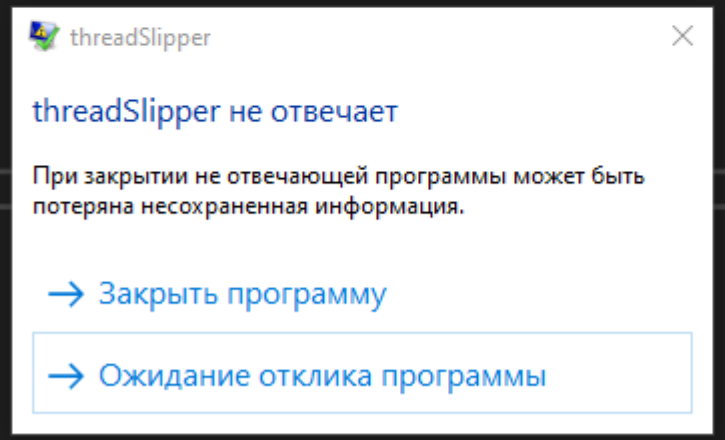
Только этот поток может менять элементы интерфейса



Dispatcher Thread – поток UI

```
namespace threadSlipper;

/// <summary>
/// Interaction logic for MainWindow.xaml
/// </summary>
Ссылок: 2
public partial class MainWindow : Window
{
    Ссылок: 0
    public MainWindow()
    {
        InitializeComponent();
    }

    Ссылок: 1
    private void ButtonBase_OnClick(object sender, RoutedEventArgs e)
    {
        Thread.Sleep(millisecondsTimeout: 20000);
    }
}
```





Dispatcher Thread – поток UI

- Когда мы запускаем «сложную логику» (имитация Sleep(20000)), наш главный поток ждет выполнения этой логики.
- Наш поток ждет завершения этого блока кода.
- Результат: окно не перерисовывается, не отвечает на клики, висит и показывает «Не отвечает».

Интерфейсы

- Интерфейс – абстрактный тип данных, описывающий методы, которые должны быть реализованы классом или структурой. Может содержать свойства и поля.

Он определяет, что класс должен делать, но не содержит реализации методов.

Интерфейсы

- Интерфейс определяется с помощью ключевого слова **Interface**

```
//интерфейс для логгирования
public interface ILogger
{
    void Log(string message);
    void Error(string errorMessage);
    string Name { get; }
}
```

Интерфейсы

- Интерфейсы могут наследоваться от других интерфейсов.

```
public interface IPrintable
{
    void Print();
}

public interface IEditable : IPrintable
{
    void Edit();
}
```

Интер

```
Ссылок: 0
class test : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler? PropertyChanged;

    int _myProperty;
    Ссылок: 1
    public int MyProperty
    {
        get => _myProperty;
        set
        {
            if (_myProperty != value)
            {
                _myProperty = value;
                OnPropertyChanged(nameof(MyProperty));
            }
        }
    }

    Ссылок: 1
    private void OnPropertyChanged(string propertyName)
    {
        PropertyChanged?.Invoke(sender: this, e: new PropertyChangedEventArgs(propertyName));
    }
}
```

Делегаты

Делегаты - типы, которые представляют ссылки на методы. Они позволяют хранить и вызывать методы как объекты. Делегат может ссылаться на любой метод с подходящей сигнатурой.

Для объявления делегата используется ключевое слово delegate, после которого идет возвращаемый тип, название и параметры:

```
[доступность] delegate [возвращаемый тип] Название([параметры]);
```

```
delegate void Message();

private delegate string Decode(string param1, int param2);
```

Делегаты

Делегаты могут хранить ссылки на несколько методов.

```
delegate void Message();
Ссылок: 1
void Hello() => Console.WriteLine("Hello");
Ссылок: 1
void HowAreYou() => Console.WriteLine("How are you?");
Ссылок: 0
public MainWindow()
{
    InitializeComponent();
    Message message = Hello;
    //теперь message указывает на два метода
    message += HowAreYou;
    //Выведется сначала "Hello", затем "How are you?"
    message();
}
```

Делегаты

Invoke - метод, который используется для синхронно вызова методов, поддерживаемых объектом делегата.

Вызывающий код должен ожидать завершения вызова прежде чем продолжить сво работу.

```
delegate int Operation(int x, int y);
delegate void Message();

Ссылок: 1
void Hello() => Console.WriteLine("Hello");
Ссылок: 1
int Add(int x, int y) => x + y;

Ссылок: 0
public MainWindow()
{
    InitializeComponent();

    Message mes = Hello;
    mes.Invoke();

    Operation op = Add;
    int n = op.Invoke(3, 4);
    Console.WriteLine(n);
}
```

Делегаты

Делегаты могут быть переданы в метод в качестве аргумента.

```
delegate int Operation(int x, int y);
Ссылок: 1
int Add(int x, int y) => x + y;
Ссылок: 1
int Subtract(int x, int y) => x - y;
Ссылок: 1
int Multiply(int x, int y) => x * y;
Ссылок: 3
void DoOperation(int a, int b, Operation operation)
{
    Console.WriteLine(operation(a, b));
}
Ссылок: 0
public MainWindow()
{
    InitializeComponent();
    DoOperation(5, 4, Add);
    DoOperation(5, 4, Subtract);
    DoOperation(5, 4, Multiply);
}
```

События

События - сообщение, которое возникает в различных точках исполняемого кода при выполнении определённых условий. События предназначены для того, чтобы предусмотреть реакцию программного обеспечения.

Событие - тип делегата, предназначенный для реализации событийной модели. События позволяют объектам оповещать других о том, что произошло определенное действие, например, нажатие кнопки, завершение процесса или изменение состояния.

События

Событие объявляется с помощью ключевого слова event:

```
// Определяем делегат
public delegate void Notify();

Ссылок: 1
public class Process
{
    // Объявляем событие
    public event Notify ProcessCompleted;

    Ссылок: 0
    public void StartProcess()
    {
        // Генерация события
        ProcessCompleted?.Invoke();
    }
}
```

События

Добавление и удаление обработчика события через оператор += и -=

```
Ссылок: 0
public MainWindow()
{
    InitializeComponent();
    Process process = new Process();

    // Подписка на событие
    process.ProcessCompleted += OnProcessCompleted;

    process.StartProcess();
}
//Метод для подписки
Ссылок: 1
public static void OnProcessCompleted()
{
    Console.WriteLine("Process completed!");
}
```

Мини итог

Делегаты представляют собой типы, которые хранят ссылки на методы. Они полезны в ситуациях, когда требуется передать метод в качестве аргумента или сохранить метод для последующего вызова.

Пример: Когда нужно передать метод в качестве параметра, чтобы другой метод мог вызвать его.

Мини итог

События создаются на основе делегатов, но предназначены для других целей. События обеспечивают механизм подписки и уведомления, часто используются в контексте взаимодействия между объектами:

Если нужно уведомить другие части программы об изменении состояния или завершении какого-либо действия.

Используется в пользовательских интерфейсах (UI). Например, нажатие кнопки, изменение текста, перемещение мыши и т. д.

Мини итог

Используйте делегаты, когда вам нужно передавать или сохранять методы для вызова в будущем.

Используйте события, когда нужно уведомлять другие объекты о произошедших изменениях или действиях.

ВАЖНО: Вызвать срабатывания события может только объект владелец события.

Часть 2 пример

- Проблема: Жестко связанный код и зависший интерфейс.

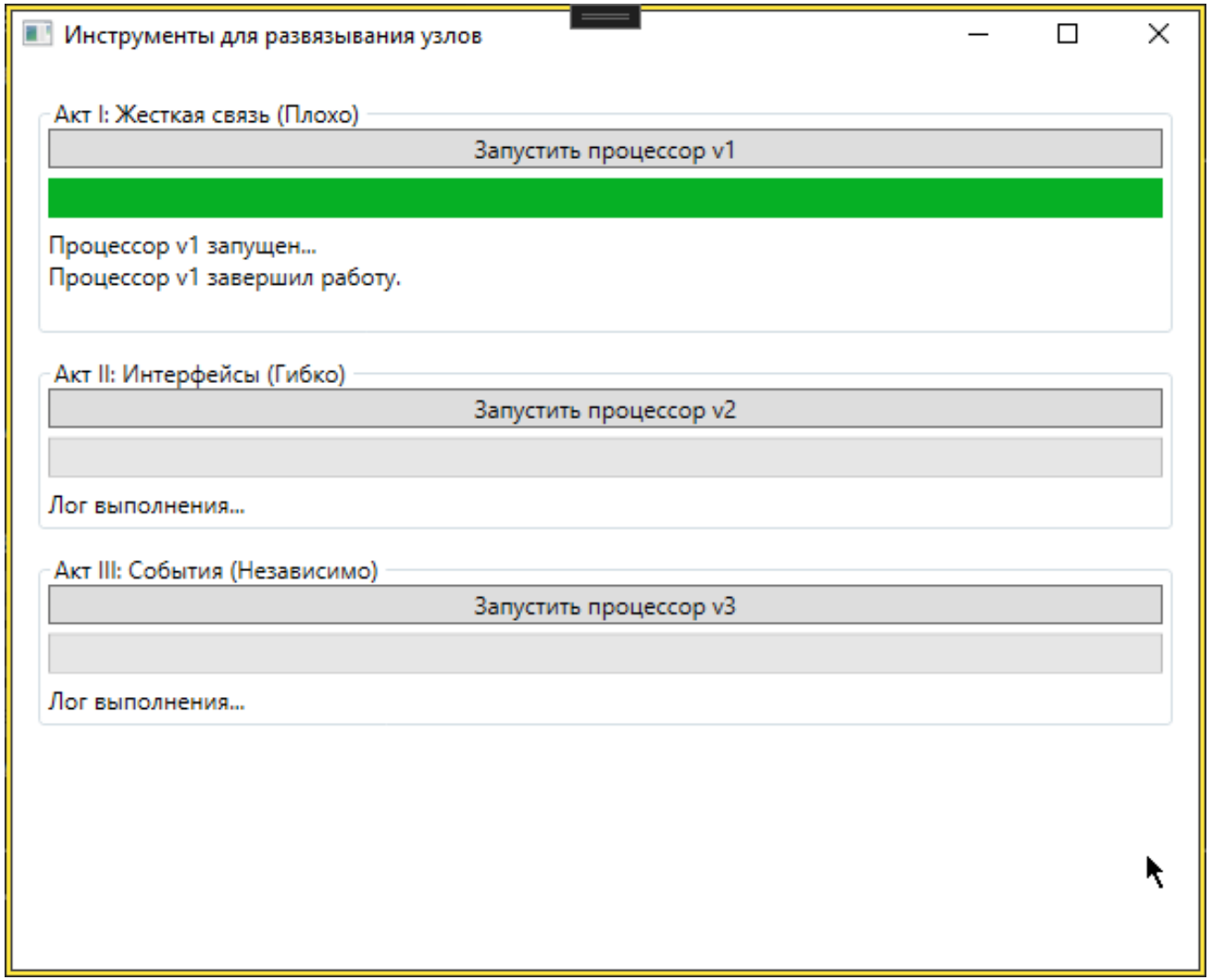
Решение №1: Интерфейсы (гибкость).

Решение №2: События (независимость).

Решение №3: async/await (отзывчивость).

Цель: Заставить кнопку «Запустить» выполнять долгий процесс и в реальном времени показывать прогресс на ProgressBar и в логе.

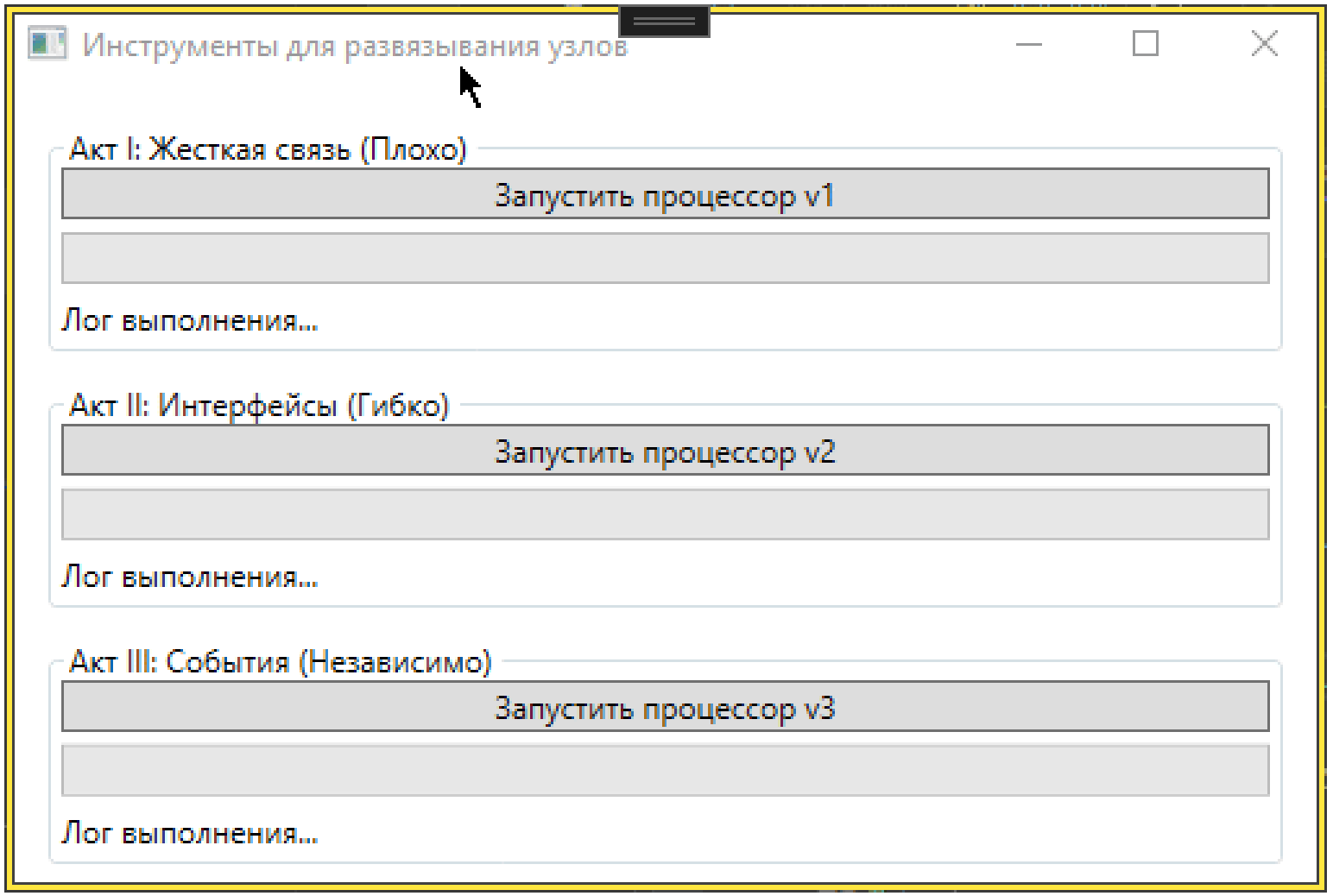
Акт 1


Инструменты для развязывания узлов
Акт I: Жесткая связь (Плохо)
Запустить процессор v1
Процессор v1 запущен...
Процессор v1 завершил работу.
Акт II: Интерфейсы (Гибко)
Запустить процессор v2
Лог выполнения...
Акт III: События (Независимо)
Запустить процессор v3
Лог выполнения...

```
// Версия 1: Мозг, прибитый гвоздями к UI
Ссылок: 2
public class DataProcessor_V1
{
    // ЖЕСТКАЯ ЗАВИСИМОСТЬ от конкретных WPF-контролов
    private readonly ProgressBar _progressBar;
    private readonly TextBlock _logTextBlock;

    Ссылок: 1
    public DataProcessor_V1(ProgressBar progressBar, TextBlock logTextBlock)
    {
        _progressBar = progressBar;
        _logTextBlock = logTextBlock;
    }

    Ссылок: 1
    public void ProcessData()
    {
        _logTextBlock.Text += "Процессор v1 запущен...\n";
        for (int i = 0; i <= 100; i += 10)
        {
            Thread.Sleep(millisecondsTimeout: 50); // Имитация работы

            // Обновление UI из рабочего потока - так делать нельзя без диспетчера!
            _progressBar.Dispatcher.Invoke(() => _progressBar.Value = i);
        }
        _logTextBlock.Text += "Процессор v1 завершил работу.\n";
    }
}
```

```
// --- АКТ I: Запуск плохого кода ---
Ссылок: 1
private void StartV1Button_Click(object sender, RoutedEventArgs e)
{
    LogV1.Text = ""; // Очищаем лог
    // Мы передаем КОНКРЕТНЫЙ UI-элемент в логику.
    // Это плохо, потому что теперь логика зависит от UI.
    var processorV1 = new DataProcessor_V1(ProgressBarV1, LogV1);
    processorV1.ProcessData();
}
```

Акт 1


Инструменты для развязывания узлов
Акт I: Жесткая связь (Плохо)
Запустить процессор v1
Лог выполнения...
Акт II: Интерфейсы (Гибко)
Запустить процессор v2
Лог выполнения...
Акт III: События (Независимо)
Запустить процессор v3
Лог выполнения...

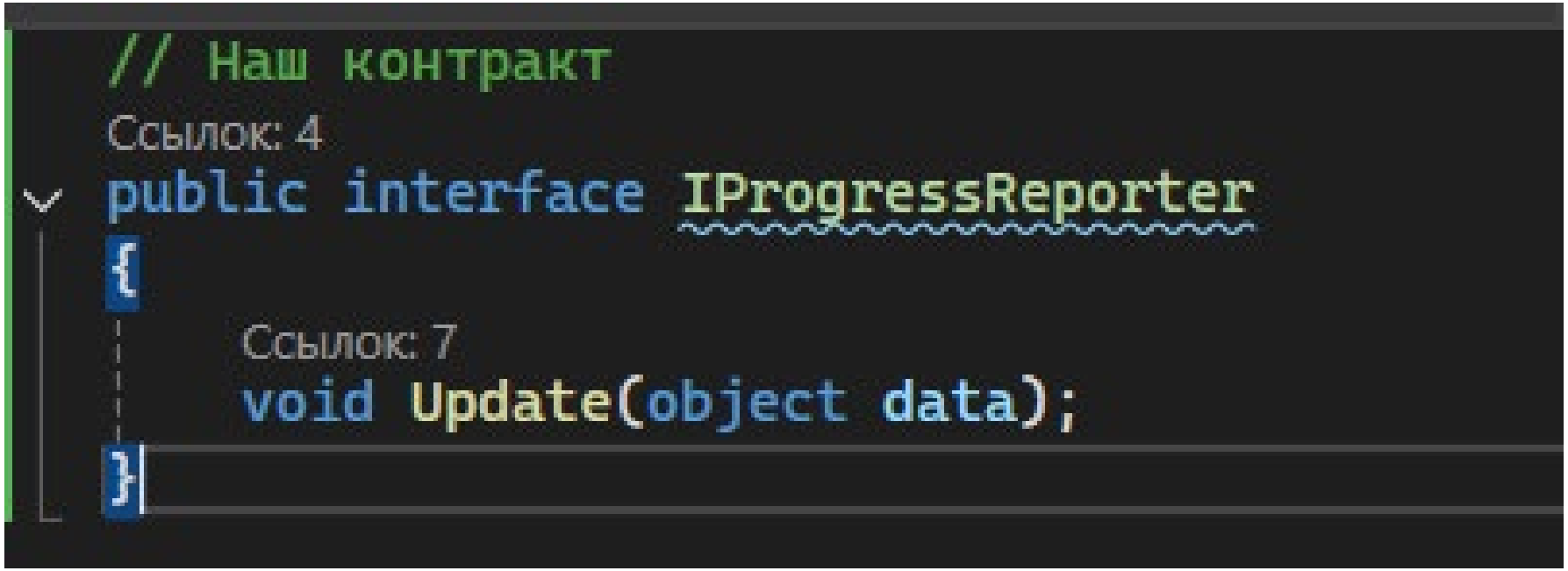
Акт 1 - вывод

- Наш код плохо спроектирован, а для пользователя он неработоспособен. Мы заблокировали поток UI и заставили ждать пользователя, не объяснив ему что случилось и не сломалась ли программа.

Акт 2

- Проблема: Наш **DataProcessor** прибит гвоздями к **ProgressBar**.
- Решение: Заставить его говорить не с конкретным классом, а с абстрактной ролью «Тот, кто умеет сообщать о прогрессе».
- Суть: Интерфейс — это контракт. Не важно кто ты, главное подпишись здесь и выполняй обещания

Акт 2


// Наш контракт
Ссылок: 4
public interface IProgressReporter
{
Ссылок: 7
void Update(object data);
}

Акт 2

```
// Адаптер для ProgressBar
Ссылок: 3
public class WpfProgressBarReporter : IProgressReporter
{
    private readonly ProgressBar _progressBar;
    Ссылок: 2
    public WpfProgressBarReporter(ProgressBar progressBar) { _progressBar = progressBar; }

    Ссылок: 5
    public void Update(object data)
    {
        if (data is int percentage)
        {
            _progressBar.Dispatcher.Invoke(() => _progressBar.Value = percentage);
        }
    }
}
```

Акт 2

```
// Адаптер для TextBlock
Ссылок: 3
public class WpfTextBlockReporter : IProgressReporter
{
    private readonly TextBlock _textBlock;
    Ссылок: 2
    public WpfTextBlockReporter(TextBlock textBlock) { _textBlock = textBlock; }

    Ссылок: 5
    public void Update(object data)
    {
        if (data is string message)
        {
            _textBlock.Dispatcher.Invoke(() => _textBlock.Text += message + "\n");
        }
    }
}
```

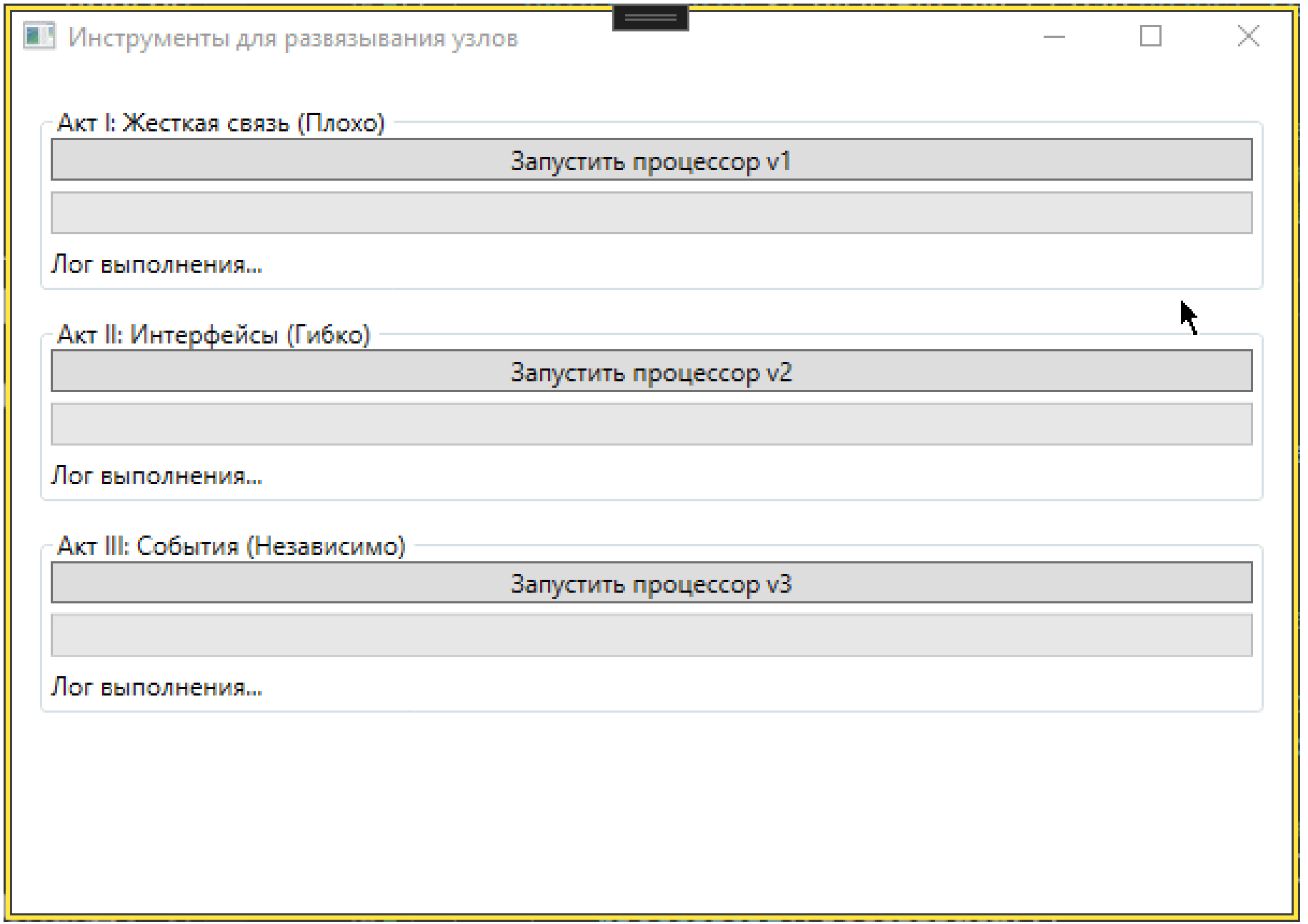
Акт 2

```
public class DataProcessor_V2
{
    private readonly IProgressReporter _reporter;

    Ссылок: 2
    public DataProcessor_V2(IProgressReporter reporter) { _reporter = reporter; }

    Ссылок: 2
    public void ProcessData()
    {
        _reporter.Update(data: "Процессор v2 запущен...");
        for (int i = 0; i <= 100; i += 10)
        {
            Thread.Sleep(millisecondsTimeout: 50);
            _reporter.Update(i); // Передаем int
        }
        _reporter.Update(data: "Процессор v2 завершил работу.");
    }
}
```

Акт 2

```
// --- АКТ II: Запуск гибкого кода ---
Ссылок: 1
private void StartV2Button_Click(object sender, RoutedEventArgs e)
{
    LogV2.Text = "";
    // Создаем два разных репортера - для ProgressBar и для TextBlock.
    // Они реализуют один и тот же интерфейс IProgressReporter,
    // но делают это по-разному.
    var progressBarReporter = new WpfProgressBarReporter(ProgressBarV2);
    var textBlockReporter = new WpfTextBlockReporter(LogV2);

    // Подсовываем репортер для ProgressBar в процессор
    var processor1 = new DataProcessor_V2(progressBarReporter);
    processor1.ProcessData();

    // А теперь подсовываем репортер для TextBlock
    var processor2 = new DataProcessor_V2(textBlockReporter);
    processor2.ProcessData();

    // Можно даже использовать оба репортера одновременно!
}
```

Акт 2


Инструменты для развязывания узлов
Акт I: Жесткая связь (Плохо)
Запустить процессор v1
Лог выполнения...
Акт II: Интерфейсы (Гибко)
Запустить процессор v2
Лог выполнения...
Акт III: События (Независимо)
Запустить процессор v3
Лог выполнения...

Акт 2 ввод

- Мы сделали код более гибким и тестируемым. Наш процессор больше не знает про WPF. Это огромная победа для архитектуры.
- Но проблему пользователя мы не решили. Интерфейсы отвечают за структуру, а не за поведение во времени. Нам нужен другой инструмент

Акт 3

- Проблема: **DataProcessor** всё ещё сам решает, кому и когда сообщать. Он активно вызывает метод репортера.
- Решение: Сделать процессор полностью независимым. Он должен просто «кричать в пустоту» о своих успехах, а кому интересно — пусть слушают.
- Суть: Событие — это система оповещений. Издатель (DataProcessor) генерирует новость. Подписчики (ProgressBar, Логгер) её получают. Они ничего не знают друг о друге.

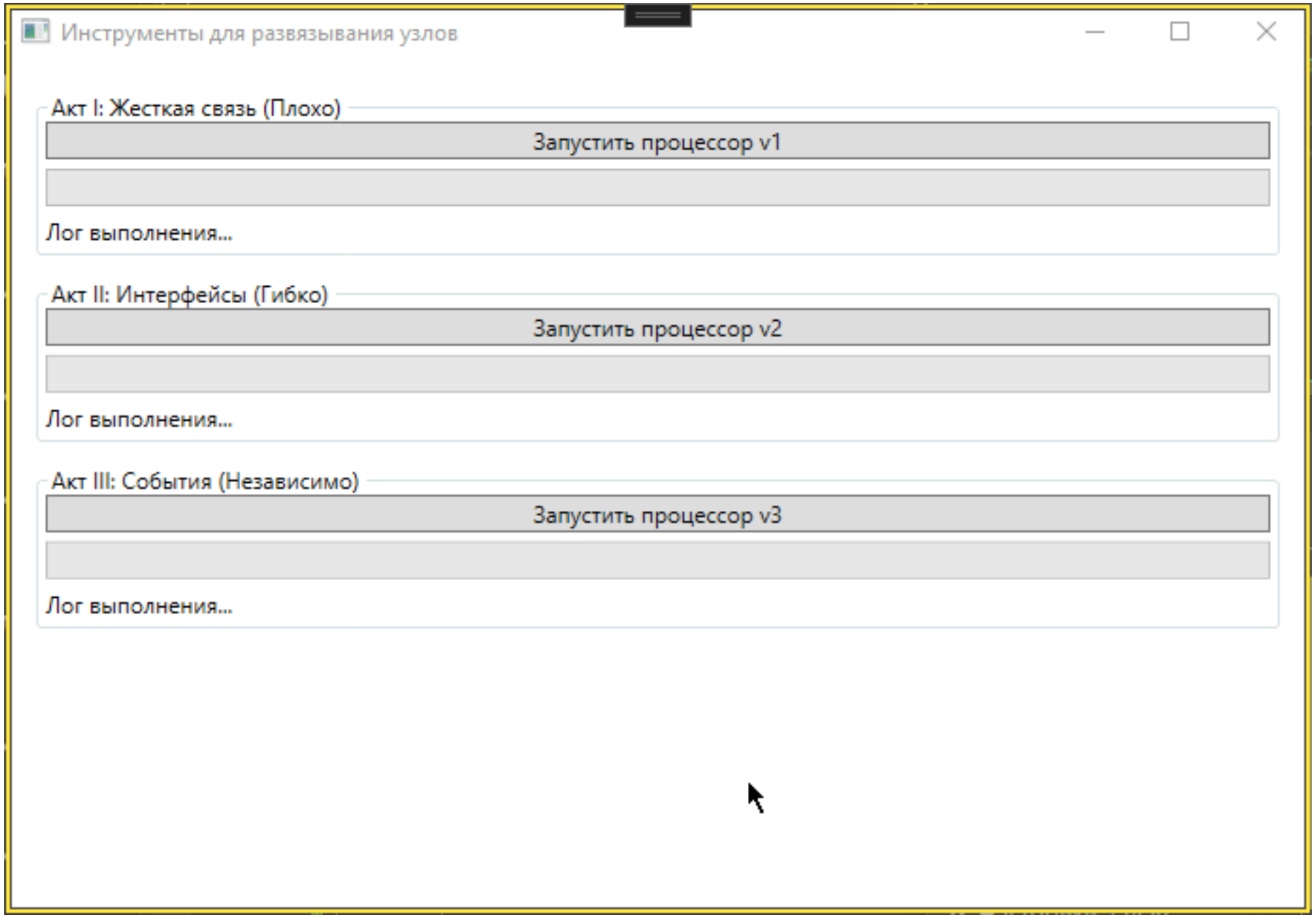
Акт 3

```
public class DataProcessor_V3
{
    // Событие для уведомления о прогрессе работы
    public event Action<object> ProgressChanged;

    // Метод теперь асинхронный и возвращает Task
    // Это позволяет использовать await внутри метода
    // Асинхронный метод не блокирует вызывающий поток (UI)
    // и позволяет UI оставаться отзывчивым во время работы метода
    Ссылок: 1
    public async Task ProcessDataAsync()
    {
        // Сообщаем, что работа началась (можно передать строку)
        OnProgressChanged(data: "Процессор v3 (async) запущен...");
        for (int i = 0; i <= 100; i += 10)
        {
            // Вместо блокирующего Thread.Sleep используем асинхронный Task.Delay
            // Await освобождает поток UI, чтобы он мог вернуться в зал и рисовать
            await Task.Delay(50); // Имитация работы
            OnProgressChanged(i); // Передаем int как прогресс в процентах
        }
        // Сообщаем, что работа завершилась (можно передать строку)
        OnProgressChanged(data: "Процессор v3 (async) завершил работу.");
    }

    // Метод для вызова события ProgressChanged.
    // Можно переопределить в наследниках, если нужно изменить поведение.
    Ссылок: 3
    protected virtual void OnProgressChanged(object data)
    {
        // Вызываем событие, если на него кто-то подписан (?. - безопасный вызов)
        ProgressChanged?.Invoke(data);
    }
}
```

Акт 3

```
// --- АКТ III: Запуск независимого кода ---
Ссылок: 1
private async void StartV3Button_Click(object sender, RoutedEventArgs e)
{
    LogV3.Text = "";
    StartV3Button.IsEnabled = false; // Отключаем кнопку, пока идёт работа

    var processorV3 = new DataProcessor_V3(); // НЕ ЗАВИСИТ ОТ UI ВООБЩЕ!
    // Создаем два разных репортера - для ProgressBar и для TextBlock.
    // Аргументы конструктора - конкретные UI-элементы,
    // но они спрятаны внутри адаптеров.
    var progressBarReporter = new WpfProgressBarReporter(ProgressBarV3);
    var textBlockReporter = new WpfTextBlockReporter(LogV3);

    // Подписываем репортеры на событие ProgressChanged
    // Метод Update будет вызван, когда процессор вызовет событие
    processorV3.ProgressChanged += progressBarReporter.Update;
    // Лямбда-выражение для добавления текста в лог
    processorV3.ProgressChanged += (p:object) => textBlockReporter.Update(data: $"Обработано {p}%...");


    // ЗАПУСКАЕМ РАБОТУ АСИНХРОННО И НЕ ЖДЁМ ЕЁ ЗАВЕРШЕНИЯ ЗДЕСЬ
    // await "ожидает" завершения ProcessDataAsync
    // Если убрать await, то работа запустится,
    // но следующий код выполнится сразу, не дожидаясь завершения работы.
    // Это приведет к тому, что UI сразу станет отзывчивым,
    // прогресс будет отображаться, а кнопка включится сразу.
    await processorV3.ProcessDataAsync();

    // Этот код выполнится только ПОСЛЕ того, как ProcessDataAsync завершится
    StartV3Button.IsEnabled = true; // Включаем кнопку обратно
}
```

Акт 3


Инструменты для развязывания узлов
Акт I: Жесткая связь (Плохо)
Запустить процессор v1
Лог выполнения...
Акт II: Интерфейсы (Гибко)
Запустить процессор v2
Лог выполнения...
Акт III: События (Независимо)
Запустить процессор v3
Лог выполнения...

Акт 3 Vihod

- Наш процессор полностью отвязан от UI. А еще мы решили главную проблему — блокировку потока UI.

Спасибо за внимание

Спасибо за внимание