**Слайд 1 Титульный**

**Слайд 2 План**

**Предыстория, основные и не основные утечки памяти,**

**Слайд 3 Предыстория**

Картинка для привлечения внимания и хотелка заказчика

**Слайд 4 С чем мы имели дело**

Сколько представлений было, сколько протекало, потребление памяти в пике и в нормальном состоянии

**Слайд 5 Основные причины утечек памяти в WPF**

* Incorrect Binding
* Event Handler Leak
* DispatcherTimer
* Links to objects in parent windows/views
* Many instances of resource dictionaries

**Слайд 6Неправильный Binding**

Декларативная реализация паттерна Observer – binding

Если свойство, к которому идет привязка, не является DependencyProperty, либо объект, содержащий его, не реализует INotifyPropertyChanged — механизм байндинга использует событие ValueChanged класса System.ComponentModel.PropertyDescriptor для отслеживания изменений. Проблема здесь в том, что фреймворк держит у себя ссылку на экземпляр PropertyDescriptor, который в свою очередь ссылается на исходный объект, и неясно, когда этот экземпляр можно будет удалить..

Другая возможная проблема при установке байндингов — привязка к коллекциям, которые не реализуют интерфейс INotifyCollectionChanged. Механизм возникновения утечек в этом случае очень похож на предыдущий. Способ борьбы очевиден — нужно либо явно указывать OneTime режим привязки, либо использовать коллекции, реализующие INotifyCollectionChanged — например, ObservableCollection.

INofifyPropertyChanged -> ValueChanged event у PropertyDescriptor класса

Свойство привязки не DependencyProperty

INofifyCollectionChanged

**Слайд 7 Как лечить неправильный Binding?**

INofifyPropertyChanged

OneTime Binding

IDisposable паттерн – сослаться на Кирилла Маурина с его докладом

В случае с OneTime байндингом проблема не актуальна, так как не нужно отслеживать изменения

Коллекции - нужно либо явно указывать OneTime режим привязки, либо использовать коллекции, реализующие INotifyCollectionChanged — например, ObservableCollection.

**Слайд 8 Event Handler leak**

Статические и экземплярные события/обработчики

Про боль от отсутствия отписки от событий говорил и Кирилл Маурин в своем докладе «Масштабирование паттерна Disposable в рамках проекта»

ICommand without WeakReference – будет очень плохо

**Слайд 9 Как лечить Event Handler leak?**

Подробный разбор того, как бороться с утечками памяти при работе с событиями, тянет на отдельный доклад. Я же сосредоточусь на той части, которая явно касается WPF.

С отпиской от события есть проблема – не всегда можно явно определить момент, когда ресурс не нужен.

Реализация WeakEventPattern выходит за рамки доклада. Даже больше – эта тема тянет на полноценный доклад

WeakEventManager используется в WPF по умолчанию

**Слайд 10 Weak event**

Простой пример паттерна Weak Event (Weak Reference) на стороне издателя/источника события

Используется во многих реализациях комманд

Если подписчик не хранит никакой ссылки на делегат, то делегат/обработчик события будет поглощен при первой же сборке мусора

**Слайд 11 DispatcherTimer**

На самом деле нужно четко понимать, когда возникает реальная потребность в использовании DispatcherTimer.

Далеко не весь код нужно выполнять в UI потоке

Если не остановить таймер, то убрать саму ссылку на него будет не так просто – нет прямого доступа к DispatcherTimers

**Слайд 12 Links to objects in parent windows**

CommandBinding – частный случай привязки к свойству другого класса (окна)

Под IDisposable подразумевается отписка от событий и очистка привязки к команде

**Слайд 13 Many instances of ResourceDictionaries**

В крупных приложениях без словарей уж точно придется стрелять себе в ногу и дублировать огромное количество кода

SharedResourceDictionary – кэширование

По умолчанию каждое обращение к содержимому **ResourceDictionary** загружает его копию в память

**Слайд 14 TextBox undo**

Не баг, а фича. Сборщик мусора соберет весь мусор от UndoManager

**Слайд 15 Media effect resource leak**

Плавающая утечка, которая может не воспроизводиться

Мне так и не удалось его воспроизвести. Интернет в данном вопросе разделился на два лагеря – кто-то смог воспроизвести баг, а кто-то нет

**Слайд 16 Как лечить Media effect resource leak?**

Freeze эффект

Как ни странно, перемещение ресурса из словаря в Application на уровень конкретного представления, в котором используется проблемный стиль, решает проблему

**Слайд 17 x:Name**

Очень напрашивается паттерн IDisposable в Code-behind

x:Name можно добавить к любому элементу в XAML, а Name может быть не у всех объектов/типов

Часто упоминается как пример утечки памяти, однако в моем демо такой проблемы не наблюдается

x:Name is a xaml concept, used mainly to reference elements. When you give an element the x:Name xaml attribute, "the specified x:Name becomes the name of a field that is created in the underlying code when xaml is processed, and that field holds a reference to the object." ([MSDN](http://msdn.microsoft.com/en-us/library/cc189028.aspx)) So, it's a designer-generated field, which has internal access by default.

Name is the existing string property of a FrameworkElement, listed as any other wpf element property in the form of a xaml attribute.

As a consequence, this also means x:Name can be used on a wider range of objects. This is a technique to enable anything in xaml to be referenced by a given name.

**Слайд 18 Выводы**

Добавьте реализацию INPC для классов модели, которые планируется отображать на представлении. Как именно это будет сделано – отдельный вопрос. Тут и реактивки, и промежуточные инфраструктурные классы.

Static – очень большое зло. Сведите использование этого модификатора к минимуму.

IDisposable – наш верный друг и помощник. Привыкайте к тому, что во многих частях вашего приложения вам придется руками освобождать/очищать проблемные ресурсы.

Грамотная архитектура спасает от большинства проблем – пример с CommandBinding и явной ссылкой одного представления на другое

Следим за событиями и отписываемся от них, если возможно. Если невозможно – Weak Event pattern вам в помощь

**Слайд 19 Как искать утечки памяти?**

Выбираем профилировщик, который дает подсказки для WPF приложений или же позволяет построить граф зависимостей, чтобы увидеть конкретную утечку. Такую возможность предоставляют многие профайлеры

Запускаем, делаем снимок, следуем рекомендациям профайлера и повторяем до победного конца

**Слайд 20 Чем искать утечки памяти в WPF?**

Существует куда больше инструментов для профилировки памяти, но доклад все же посвящен не сравнению профилировщиков ☺

На боевом приложении тестировались: dotMemory, dotNet Memory Profiler, ANTS Memory Profiler. Конечно же триальные версии

На практике они давали одинаковые результаты – не было ситуации, при которой один профилировщик нашел утечку, а другой нет

**Слайд 21 Самое время для демо!**

**Слайд 22 Контактные данные**

**Слайд 23 Спасибо за внимание**