Сообщения с таймером

При работе со многими программами мы не обращаем внимания на множество вещей в интерфейсе, считая это само собой разумеющимся, или же считаем милой забавной штучкой, на которую потратили от силы полчаса. Такие мелочи в интерфейсе, в конечном счете, складываются в ощущение целостности процесса работы, которые не отвлекают от важной информации, а подчеркивают ее, не заставляют делать лишние телодвижения. Всё кажется логичным, простым и доступным для понимания. Наверно не стоит говорить уже, что такие интерфейсы занимают в своей проработке и реализации уйму времени и сил. Об одной из таких приятных «мелочей» я бы хотел сегодня рассказать.

Задумка

Я думаю, что если подумать, то все вспомнят о статусной строке в приложении, где часто пишется состояние программы, уведомления о завершении каких-либо фоновых задач, другая интересная информация о работе программы. Еще можно вспомнить о программах, которые показывают информационное сообщение пользователю в каком-либо специальном месте на интерфейсе, а через какое-то время (секунд 5) надпись исчезает. Учитывая последние тенденции к тому, чтобы избавлять пользователя от popup-окон, в которых написано что-то в духе: Данная операция не может быть совершена, так как она в процессе выполнения, - и на всплывшем окне только одна кнопка OK. Раздражает такое поведение неимоверно, так как приводит к лишним действиям! В общем, сегодня я покажу, как можно реализовать набор классов для реализации такого поведения и использовать его в дальнейшем без существенных модификаций.

Неискушенный читатель наверно может воскликнуть: «Что за бредятина, какие еще *наборы классов* для того, чтобы сделать два set'а строки? Надо показать сообщение, так присвоил переменной сообщения нужный текст, когда не надо – присвоил пустую строку. Any problem?»

Если кратко, то проблем много! Сейчас попробую перечислить их, как они придут в голову:

- Много инфраструктурного кода получится, ведь надо будет вводить таймеры, ответы и наверняка что-то еще. И это каждый раз при попытке сменить текст.
- Надо запоминать предыдущий текст в сообщении, если он был.
- Легко запутаться что, где и зачем реализовано. Скорее всего, будет много копи-пасты, а следовательно больше мест для ошибок.
- Некрасиво!

Лично для меня хватило только первого пункта из-за моей профессиональной лени.

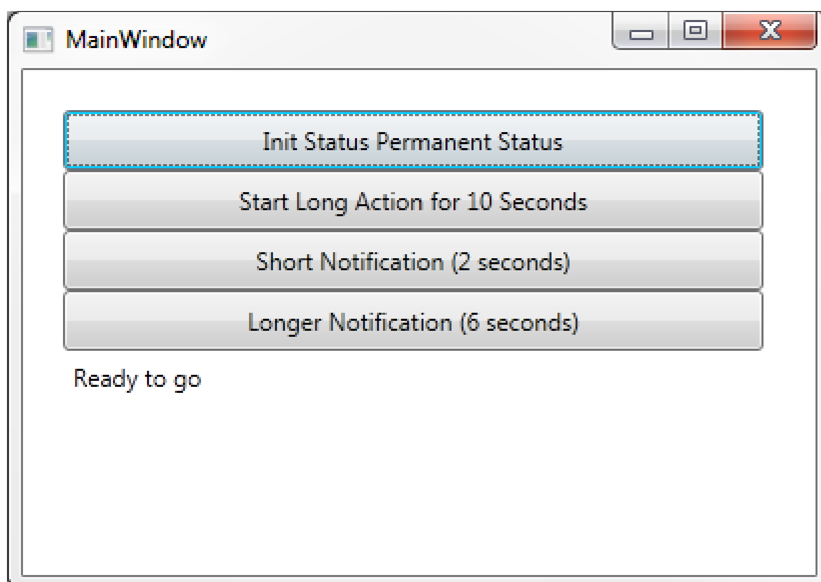
Подготовка

Демонстрационный проект будет на базе WPF с использованием Rx. О да, мне очень понравился Rx и он идеально подходит для нашей цели в силу своей компактности при записи и общем удобстве использования. Честно сказать, я сразу сделал проект на Rx, и хотя была мысль сделать то же самое без использования этого фреймворка, но ужаснувшись дебрям кода который надо будет написать, я решил так не делать. Мне кажется, можно будет и так оценить краткость решения.

Итак, у нас будет настольное приложение на WPF. Для того чтобы не заморачиваться с INotifyPropertyChanged я использую Kind of Magic, так как его использование было описано ранее, то подробно останавливаться на этом я не буду.

Приложение будет эмулировать работу с долгоиграющим сервисом, который в процессе может рапортовать о своем прогрессе. С помощью специальных кнопок мы сможем показывать временные сообщения в той же области экрана.

Визуально, наше приложение будет выглядеть так:



Я думаю, функции приложения ясны из подписей к кнопкам. Xaml интереса не представляет, поэтому его рассматривать не будем.

Из установленных NuGet пакетов:

- Rx-Main
- Rx-WPF (install-package rx-wpf)

Устанавливать можно только последний, так как он автоматом подтянет и поставит основную сборку.

По организации взаимодействия интерфейса и всего остального – используем MVVM.

Основные задачи

Еще раз четко проговорим, какие задачи должны быть решены:

- Возможность установки постоянного информационного сообщения.
- Возможность установки временного информационного сообщения на заданный промежуток времени.
- После завершения показа временного сообщения, автоматически, без нашего участия, показывается постоянное сообщение, если нет других временных сообщений, срок жизни которых еще не истек.
- Любое новое постоянное сообщение не прерывает показ временного сообщения.
- Любое временное сообщение показывается сразу же.

Техническое решение

Исходя из поставленных задач, можно с уверенностью сказать, что нам потребуется стек для хранения временных сообщений.

Далее, следуя традициям ООП, временное сообщение должно самостоятельно отслеживать время собственной жизни. К тому же у меня есть ощущение, что такая организация ответственности будет проще.

Т.е. на данный момент приходим к тому, что должен быть некий публичный класс, который будет знать постоянное сообщение и содержать стек временных сообщений.

```
    public class StatusMessageManager {
        private readonly Stack<TemporaryMessage> timerMessages = new
Stack<TemporaryMessage>();
        private string constMessage;


        private class TemporaryMessage : IDisposable {
            public void Dispose() {
            }
        }
    }
```

Класс сообщений

Остановимся поподробнее на классе сообщения. Он должен знать как минимум:

- Текст
- Время жизни в секундах
- Закончилось ли время жизни

Вообще, один класс можно использовать как для временных, так и для постоянных сообщений. Для постоянных сообщений мы не будем запускать таймер и оно всегда будет «живым», актуальным.

Так же класс должен уметь отсчитывать время и сообщать об истечении этого времени. Для этого используем Timer из пространства System.Timers (другие не очень подойдут)

```
private class TemporaryMessage : IDisposable {
    private readonly Timer timer = new Timer();
    private readonly int lifeTime;
    private Action<TemporaryMessage> expired;

    public string Message { get; private set; }
    public bool IsExpired { get; private set; }

    public void Dispose() {
                expired = null;
                timer.Dispose();
    }
}
```

Сообщение создается один раз и не модифицируется в течение жизни, так что все параметры задаем в конструкторе.

```
public TemporaryMessage(string message, int lifeTime, Action<TemporaryMessage> expired) {
    this.lifeTime = lifeTime;
```

```
        this.expired = expired;

        Message = message;
        IsExpired = false;

        if (lifeTime > 0) {
            timer.Interval = this.lifeTime*1000;
            timer.Elapsed += (sender, args) => Elapsed();
            timer.Start();
        }
    }

    private void Elapsed() {
        timer.Stop();

        IsExpired = true;
        expired(this);
    }
```

В реализации видно, что для временных сообщений мы запускаем таймер, по срабатыванию которого, останавливаем таймер, обозначаем сообщение как «умершее» и уведомляем об этом управляющий класс StatusMessageManager. Думаю тут никаких откровений не случилось.

На данный момент есть класс, который через заданное количество секунд уведомляет управляющего, о том, что надо показать что-то другое. Отлично!

Теперь надо научить управляющий класс создавать временные и постоянные сообщения, и при сигнале находить в стеке следующее «живое» временное сообщение или же показывать последнее постоянное.

Класс управления сообщениями

Управлять публикацией сообщений будем с помощью Rx. Основными методами класса являются:

- Выставление потока сообщений
- Установка новых сообщений

Начнем с выставления потока сообщений. При работе с Rx, важно для себя понять, что это работа с *потоком* данных, а не со статическим набором данных.

```
private readonly BehaviorSubject<string> subject = new BehaviorSubject<string>("");

public IObservable<string> MessageStream() {
      return subject;
}
```

Используем BehaviorSubject, он запоминает только последнее свое значение, чего нам достаточно будет. При использовании, я покажу, как мы будем использовать этот метод для подписки на новые сообщения. Мне кажется, идеологически более верно представить это методом, нежели свойством, в виду «долгосрочности» его работы. Есть такое ощущение.

Создание нового сообщения несложно, но надо внимательно и аккуратно его создавать, так как в стек будем помещать только временные сообщения. Проверки идут буквально по задачам, которые мы себе поставили.

```
public void SetMessage(string message, int freezeTime = 0) {
    if (freezeTime < 0)
        freezeTime = 0;

    if (timerMessages.Count == 0) {
        subject.OnNext(message);
    }

    if (freezeTime == 0) {
        constMessage = message;
    }

    recentMessage = new TemporaryMessage(message, freezeTime, CheckNextMessage);
    if (freezeTime > 0) {
        timerMessages.Push(recentMessage);
        subject.OnNext(message);
    }
}
```

Если нет временных сообщений, то сразу публикуем новое значение в потоке subject. Сохранение последнего постоянного сообщения будет нужно далее.

Важный момент: в качестве метода для отработки по срабатыванию таймера идет проверка наличия следующего сообщения. Сейчас мы этот метод и рассмотрим CheckNextMessage.

```
private void CheckNextMessage(TemporaryMessage expiredMessage) {
    TemporaryMessage message = null;

    if (timerMessages.Count == 0) {
        subject.OnNext(constMessage);

        return;
    }

    if (expiredMessage != timerMessages.Peek()) {
        return;
    }

    while (timerMessages.Count > 0) {
        message = timerMessages.Peek();
        if (!message.IsExpired)
            break;

        timerMessages.Pop();
    }
    subject.OnNext(message == null || message.IsExpired
                       ? constMessage
                       : message.Message);
}
```

В методе описаны возможные ситуации, при которых мы попадаем в этот метод. Чаще всего будет ситуация, когда надо просто вернуться к постоянному сообщению, на этот случай проверяем длину стека и если он пуст, то выдаем в поток последнее постоянное сообщение.

Далее очень важная проверка на то, что мы работает с видимым сообщением, которое сейчас приказало долго жить. Видимое сообщение должно быть на вершине стека. Если это не так, то возвращаем управление из этого метода, потому что умершее сообщение в глубине стека нам не интересно.

Когда оказалось, что видимое в данный момент сообщение закончило свой срок, то начинаем поиск в стеке «живых» сообщений. В итоге публикуем либо найденное живое сообщение, либо последнее постоянное.

Использование

Все использование и красота при желании остается во ViewModel, хотя конечно можно фантазировать. Итак, у нас есть StatusMessageManager, который можно использовать во вьюмодели. Не буду на вьюмодели подробно останавливаться, рассмотрим только самые интересные места.

Итак, объявление и подписка на появление новых сообщений от управляющего класса. Подписываться будем в конструкторе. Обратите внимание, что подписка на поток событий идет для потока GUI.

```
public NotificationViewModel() {
    // some code was skipped

    statusMessageManager = new StatusMessageManager();

    statusMessageManager.MessageStream()
        .ObserveOnDispatcher()
        .Subscribe(SetMessage);
}
private void SetMessage(string message) {
    InfoMessage = message;
}
```

И пара методов для облегчения использования:

```
private void SetStatusMessage(string message) {
    statusMessageManager.SetMessage(message);
}

private void SetStatusMessageFreeze(string message, int freezeTime) {
    statusMessageManager.SetMessage(message, freezeTime);
}
```

Как я уже сказал, при работе с Rx, стоит мыслить о его использовании как о непрерывном потоке данных. С помощью одного метода мы помещаем данные в поток, там они каким-либо образом обрабатываются (задерживаются, агрегируются, суммируются и так далее) и помощью другого метода принимаем результат. Может быть сначала непривычно, но стоит чуть поэкспериментировать и все встает на свои места и становится понятно. Работать с Rx легче чем с событиями (event), компактнее.

И теперь как это использовать, например, при нажатии на кнопки о важных сообщениях на 2 и 6 секунд:

```
private void OnShortNotification() {
    SetStatusMessageFreeze("Attention for 2 sec!", 2);
}

private void OnLongerNotification() {
    SetStatusMessageFreeze("Attention for 6 sec!", 6);
}
```

Ну не красота ли?!

В программе попробуйте запустить долгий процесс и вызвать сообщение на 2 секунды. Или вызвать сообщение на 6 секунд, а потом на 2 и посмотреть, как меняются сообщения, как ведут себя классы.

Я думаю, что использование класса StatusMessageManager предельно простое и цена внедрения в проект низка.

Развитие

Можно развить функционал класса, сделав постоянные сообщения в некотором роде тоже слоями, с метками жизни.

При создании постоянной записи вы получаете GUID записи. Постоянные записи тоже укладываются в стек, и показываются только самые свежие, пока не придет код (GUID) отмены, тогда показывается предыдущее значение. Так до какого-нибудь базового значения в духе «Готово к работе». На мой взгляд, это сильно поможет в плане информативности в программах, которые оперируют большими объемами данных, загружаемыми асинхронно в фоновом режиме. В этом случае возможен следующий сценарий:

- Готово
- Загрузка абонентов…
- Загрузка товаров…
- Загрузка складов… (заканчивается загрузка товаров быстрее и код отмены приходит, затем заканчивается загрузка складов)
- Загрузка абонентов…
- Готово

Или еще можно получать GUID и для временных записей, чтобы продлевать их жизнь, если они до сих пор живы.

Такой сценарий может быть использован для сервисов, которые должны регулярно посылать ответы и по этому ответу продлевать временное сообщение. Например, для индикации работы процесса, в ходе которого нельзя оценить его прогресс и предсказать время окончания работы.

Заключение

Надеюсь, вам понравилось. С Rx можно сделать много интересных вещей. Например, посмотреть, как с помощью Rx публикуется прогресс обработки в этом приложении. Source code.

Hard'n'heavy!

Violet Tape