Эволюция сервисных методов

Давно хотел изложить свои мысли по поводу эволюции сервисных методов, но до последнего момента правильная, на мой взгляд, идея имела плохую реализацию – слишком трудоемкую и развесистую. На данный момент я пришел к некоторому сокращению кода и готов поделиться мыслями по поводу эволюции сервисных методов. Сразу скажу, что речь пойдет о доменных сервисах по обработке данных, так как чаще всего именно они занимаются обработкой данных в течение продолжительного времени. Однако описанные подходы наверно можно применять и к другим классам с «долгоиграющими» методами.

В данном случае под эволюцией подразумевается то, как инфраструктурно выглядит метод, а не его функциональная эволюция (рефакторинг). Описывать, как всегда, я собираюсь свой личный опыт, то, на чем я набил шишки.

Простейшие

Начиная со школы или университета, и в продолжение достаточно долгого времени мы пишем только однопоточные приложения. Все действия в них выполняются последовательно, синхронно. В этом нет ничего плохого в целом, многие методы и библиотеки работают синхронно и это хорошо. Не стоит городить сложность там, где она не нужна. Итак, простейший сервис может выглядеть следующим образом:

```
public class Service {
     public void Method() {
         // do some complicated actions
     }
}
```

Не будем разводить дискуссий по поводу вырожденности примера, мол, большую часть времени есть аргументы и возвращаемое значение – суть от этого не поменяется. Сколько бы раз я не написал вызов этого метода, он будет выполнен последовательно, один за другим.

```
var service = new Service();
service.Method();
service.Method();
service.Method();
...
```

Со временем начинает расти недовольство тем, что интерфейс программы, если он есть, замирает в результате выполнения цепочки последовательных методов и в этот момент ни отменить операции, ни что-то сделать еще. С одной стороны есть «плюс», не надо делать уведомление о том, что операция закончилась, так как экран «отмерзнет», но лучше это делать более цивилизованными способами.

Зато, кстати, не надо было задумываться о том, что этот код может выполняться сразу в нескольких потоках, никаких проверок, что метод уже запущен и работает с данными. Второй раз с интерфейса все равно не запустишь его, если есть такая связь.

Долго, сложно, дорого

После первых вопросов о «размораживани» пользовательского интерфейса, скорее всего вы станете использовать BackgroundWorker. Наверно это самая простая конструкция для освоения, но с достаточно богатым функционалом, out-of-box для выполнения задачи в фоне. При этом класс берет на себя заботу о многих вещах уровня потоков. Если надо было сделать пару операций асинхронными и получить от них информацию, или же просто продолжить работу со специального метода после завершения асинхронной операции – BackgroundWorker подходил хорошо.

```
public class Service {
    private BackgroundWorker worker;

    public void MethodAsync(RunWorkerCompletedEventHandler OnCompleteMethod) {
        worker = new BackgroundWorker {
                                            WorkerSupportsCancellation = true
                                        };

        worker.DoWork += DoMethod;
        worker.RunWorkerCompleted += OnCompleteMethod;
        worker.RunWorkerAsync();
    }

    private void DoMethod(object sender, DoWorkEventArgs e) {
         // do some complicated actions
    }
}
```

По сравнению с использованием потоков, кода кажется не так и много, но по сравнению с исходным вариантом, кода стало заметно больше. При этом в самой простой реализации метод для обратного вызова передается как аргумент, что накладывает определенные ограничения на сигнатуру метода обратного вызова, что может быть неудобно.

При единичном использовании, все еще выглядит хорошо, а если методов таких надо сделать штук 5 или больше? Код сервиса превращается в кашу, которую не так уж приятно становится читать. Я думаю излишне говорить, что писать одну и ту же инфраструктурную обвязку страсть как не хочется.

Дополнительная подписка на события в данной реализации представляется мне несколько запутанной, хотя и простой. Можете представить себе ситуацию, когда выполняется долгоиграющий метод и кто-то еще подписывается на результат выполнения.

На этапе знакомства с BW особенно не задумываешься над многими вещами по работе методов. Проверки, обработка исключений – многие вещи происходят как бы сами собой или же не заостряется на них внимание. Множественный запуск метода, множественная реакция на окончание метода, отписывание событий на все это смотришь сквозь пальцы, так как BW пересоздается каждый раз.

Асинхронно, но неуправляемо

После работы с BW, хочется чего-то более хардкорного, начинаем штудировать MSDN или же припоминаем обрывки лекций или еще как-то натыкаемся на возможность создания потоков и решаем их использовать. Скорее всего, первый вариант вытекает во что-то в таком духе:

```
public class Service {
    private Thread thread;
    private Action callback;

    public void Method(Action callback) {
        var start = new ThreadStart(DoMethod);
        thread = new Thread(start);
        thread.Start();
    }

    private void DoMethod() {
        // do some complicated actions
        callback();
    }
}
```

При этом еще нужно достаточно потрудиться для того, чтобы была возможность отмены метода. Когда методом много или же есть параметры, а они появятся рано или поздно, то все становится чуть грустнее. Либо надо будет использовать анонимные классы, либо же создавать специальные классы/структуры единственная цель которых передать параметры в метод, так как выделять переменный в классе для каждого метода наверно накладно и выглядит некрасиво.

Обратный вызов идет с помощью делегата (в данном случае для простоты используется Action), который идет аргументом. А методы с большим кол-вом аргументов меня, например, не радуют совсем. В стремлении использовать методы с параметрами и передавать их в качестве класса/структуры приводит к еще более плачевным результатам

```
public class Service {
    private Thread thread;

    public void Method(int i, string s, Action callback) {
        var start = new ParameterizedThreadStart(DoMethod);
        thread = new Thread(start);
        thread.Start(new ArgsForMethod (i, s, callback));
    }

    private void DoMethod(object args) {
        var arguments = ((ArgsForMethod) args);
        // do some complicated actions
        arguments.callback();
    }

    private class ArgsForMethod {
        public int i;
        public string s;
        public Action callback;

        public ArgsForMethod(int i, string s, Action callback) {
            this.i = i;
            this.s = s;
            this.callback = callback;
        }
    }
}
```

Конечно, через какое-то время придем к тому, чтобы брать потоки из пула и тогда код станет примерно таким:

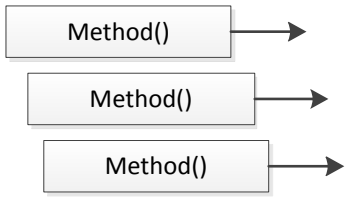
```
public class Service {
    public void Method(int i, string s, Action callback) {
        ThreadPool.QueueUserWorkItem(DoMethod, new ArgsForMethod(i, s, callback));
    }

    private void DoMethod(object args) {
        var arguments = ((ArgsForMethod) args);
        // do some complicated actions
        arguments.callback();
    }

    private class ArgsForMethod {
        public int i;
        public string s;
        public Action callback;

        public ArgsForMethod(int i, string s, Action callback) {
            this.i = i;
            this.s = s;
            this.callback = callback;
        }
    }
}
```

Чуть проще, но совсем не фонтан тоже.



Методы можно запускать почти одновременно, и это ставит перед нами задачу контролирования количества запущенных методов. Можно монопольно захватывать ресурсы, чтобы другие вызовы ждали очереди. Если вызов идет с одними и теми же параметрами, то логичнее было бы просто подписать клиента на ответ. Но об этом чуть позже.

После некоторого времени начинаешь снова задумываться о том, что больше времени проводишь в борьбе с инфраструктурой методов, решением мелких частных проблем синхронизации, согласования контекстов и так далее, а не решением реальных задач приложения.

Off topic: Task для .Net 3.5

Для одного маленького проекта захотелось сделать подобие Task, максимально похожий по своему поведению, и так, чтобы в случае миграции все заработало самой собой с нормальным классом. Вот что из этого получилось:

```
    public class Task<T> : IAsyncResult {
        private readonly ManualResetEvent operationEnds = new ManualResetEvent(false);

        private readonly Func<T> mainAction;
```

```
        private readonly List<Action<Task<T>>> continues = new List<Action<Task<T>>>();
        private Thread thread;

        public bool IsCompleted {
            get { return thread.ThreadState != ThreadState.Running; }
        }


        public WaitHandle AsyncWaitHandle {
            get { return operationEnds; }
        }

        public object AsyncState {
            get { throw new NotImplementedException(); }
        }

        public bool CompletedSynchronously {
            get { throw new NotImplementedException(); }
        }

        public T Result { get; private set; }

        public Task(Func<T> action) {
            mainAction = action;
        }

        public Task<T> Start() {
            Action extendedAction = () => {
                                        Result = mainAction();
                                        operationEnds.Set();
                                        foreach (var action in continues) {
                                            action(this);
                                        }
                                    };
            thread = new Thread(new ThreadStart(extendedAction));
            thread.IsBackground = true;

            thread.Start();
            return this;
        }


        public void ContinueWith(Action<Task<T>> nextAction) {
            continues.Add(nextAction);
        }
    }
```

Оно работает!

Task

С появлением .Net 4.0 стало несколько проще в плане организации работы асинхронных методов, на мой взгляд, благодаря классу Task. Используя этот класс и события можно добиться примерно следующего:

```
public class Service {
    public event Action<Task> OnCompleted;

    public Task Method() {
        var task = new Task(MethodSync);
        task.ContinueWith(OnCompletedMethod);
```

```
        task.Start();

        return task;
    }

    private void MethodSync() {
        // do some complicated actions
    }

    private void OnCompletedMethod(Task task) {
        if(OnCompleted != null)
            OnCompleted(task);
    }
}
```

Данный код так же не блещет компактностью, однако я считаю реализацию на данном классе предпочтительной, так как система сама решает в зависимости от загруженности, создавать ли новые потоки или нет, возможно ли выполнить синхронно или нет. Плюс ко всему конструктор класса позволяет указать достаточное количество подсказок и указаний касающихся выполнения кода. Об этом я рассказывать не буду, а лучше отошлю вас к книге Альбахари “C# 4.0 in the nutshell”.

Некоторое недоумение может вызвать, то почему я в метод **ContinueWith()** передал ссылку на другой метод, а не сразу событие OnCompleted. Это сделано для возможности отложенной подписки на событие. Так как во время начала выполнения задачи, указанные делегаты захватываются и не модифицируются впоследствии.

Естественно, что все несколько усложняется при работе с параметрами:

```
public class Service {
    public event Action<Task> OnCompleted;

    public Task Method(int i, string s) {
        var task = new Task(MethodSync, new {i, s});
        task.ContinueWith(OnCompletedMethod);
        task.Start();

        return task;
    }

    private void MethodSync(dynamic args) {
        var i = (int)args.i;
        var s = (string)args.s;
        // do some complicated actions
    }

    private void OnCompletedMethod(Task task) {
        if(OnCompleted != null)
            OnCompleted(task);
    }
}
```

Работа с динамическими параметрами порой так надоедает, что хочется создавать классы для передачи аргументов в **MethodSync()**.

Да, с классом Task удобнее работать, с выходом .Net 4.5 можно будет не меняя кода сервиса писать:

```
async Method(i, s);
```

при этом можно будет избавиться от контроля подписки на события, и самих событий. Сейчас же вызов в клиентском коде происходит так:

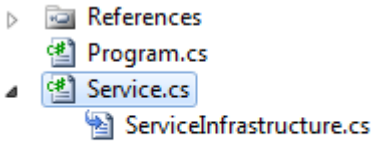
```
private Service service;

public void x() {
    service = new Service();
    service.OnCompleted += OnCompleted;
    service.Method(1, "cool");
}

private void OnCompleted(Task obj) {
    service.OnCompleted -= OnCompleted;
    // do other stuff
}
```

Очень много действий надо делать, что совсем не радует. И в клиентском коде не чисто, и сам сервис раздут инфраструктурным кодом. Я думаю, вы согласитесь, что по подход по сути верный, но реализация заставляет страдать. Можно написать шаблон генерации кода в ReSharper, чтобы писать только нужный код, но это не избавляет от созерцания и поиска важных частей сервиса среди обвязок.

В какой-то момент я подумал о том, чтобы обвязки вынести в отдельный файл, пометив класс как partial, и сгруппировав файлы.



Это могло бы выступить косметическим решением проблемы, но код все равно пришлось бы писать и править в текущем проекте, а его количество неминуемо вело бы к глупым ошибкам из разряда «перепутал методы, так как похожи», «забыл».

Во все примеры выше еще не была включена обработка исключительных ситуаций, что тоже прибавляет инфраструктурного, рутинного кода, в котором так легко ошибиться и так легко сделать ошибку.

Я достаточно долго откладывал этот рассказ, так как остановиться на текущем примере было бы наверно нехорошо. Да, можно было бы рассказать о разных тонкостях класса Task и его применении, как можно расширить обвязки в проверках и так далее, но какова цена использования? Вечная копи-паста?

Make me async

Решение искал и в аспектных подходах, и в кодогенерации T4, но решение пришло в виде своеобразного декоратора для запуска методов. По сути ведь мне надо писать синхронный код в отдельно взятом методе, и асинхронно его запускать, чаще всего асинхронный запуск нужен для того, чтобы не замирал интерфейс пользователя, чтобы можно было параллельно сделать что-то еще, на крайний случай отменить текущую операцию.

В идеале я хочу писать сервисы как в простейшем случае, я не хочу думать об обвязках, анонимных классах, динамических переменных и всем прочем. Хочется написать метод:

```
public class Service {
     public void Method() {
         // do some complicated actions
     }
}
```

И в клиенте сделать что-то в духе:

```
public void x() {
    service = new Service();
    service.Method().Async();
}
```

Хех, однако ясно, что такой способ не пройдет ни при каких обстоятельствах, и **Method()** придется передавать в какой-либо метод и из этого выросло следующее (вдохновение принесла библиотека Rx):

```
public void x() {
    var service = new Service();

    Async.For(service.Method)
        .Subscribe(OnComplete)
        .Run();
}

private void OnComplete(Task task) {

}
```

Ну не красота ли? Компактный исходный сервис, компактный вызов.

Для обработки событий можно использовать метод **OnFault()**.

```
public void x() {
    var service = new Service();

    Async.OnFault(OnFault);

    Async.For(service.Method)
        .Subscribe(OnComplete)
        .Run();
}

private void OnFault(Task obj) {

}
```

При этом он будет использован для всех методов, которые будут вызваны асинхронно. Можете думать об этом как о работе с AppDomain.CurrentDomain.UnhandledException. Для каждого метода есть возможность переопределить поведение в случае возникновения ошибки.

```
   Async.For(service.Method)
        .Subscribe(OnComplete, OnFault)
        .Run();
```

Так же периодически все попадаются на том, что хотят обновить содержимое UI в событии OnComplete. И теперь стало гораздо проще контролировать такие ситуации:

```
Async.For(service.Method)
     .SubscribeOnDispatcher(OnComplete, OnFault)
     .Run();
```

Закономерно наверно возникает вопрос с передачей параметров и с возвращением значений. Это не составляет большого труда, а метод **Run()** контролирует типизацию. Максимальное количество аргументов для функции на данный момент ограничено четырьмя. Это разумный максимум аргументов на мой взглядю

```
var service = new Service();

var info = Async.For<int, string>(service.ProcedureIntStr)
               .Run(4, "s");
```

Метод Run() возвращает специальный класс, который содержит Task, подготовленные результаты, если есть, а так же готовую коллекцию ошибок, если таковые возникли.

Если какой-то метод работает очень долго и запустить его надо как можно раньше, а результаты получить по мере необходимости, но можно указать это при запуске метода. Это необходимо делать явно, так как все подписки обнуляются после завершения работы и передачи управления клиенту.

```
var info = Async.For(service.LongAction1Sec)
    .WithDelayedSubscription()
    .Run();

Async.Subscribe(info, OnCompleted);
```

Попробовать библиотеку можно установив ее с помощью nuGet > install-package MakeMeAsync.

Плюсы:

- Меньше кода – меньше банальных ошибок
- На надо вводить анонимные классы для сервисов
- Легче тестировать сервисы
- Возможность указать единую реакцию на исключения
- Легче читать код
- В целом само отписывается от себя
- Callback возможно выполнить в потоке UI

Минусы:

- Не реализована на данный момент поддержка CancelationTokken

Подробно о реализации библиотеки как-нибудь в другой раз.

У всех свой путь и взгляд на то, как следует писать код, однако надеюсь, что изложенные мысли и куски кода помогут кому-нибудь.

Я же собираюсь развивать библиотеку MakeMeAsync, использовать в реальных проектах и реализовывать только востребованные вещи оставляя библиотеку максимально компактной.

Hard’n’Heavy!

Violet Tape