Тестирование событий с помощью Moq

Вообще название статьи опять очень длинное на самом деле, что-то в духе: Тестирование сложных сценарий с событиями на примере Moq, связыванием StructureMap, построением объектов с помощью NBuilder, и проверки условий с помощью FluentAssertions. Но согласитесь, что это как-то чересчур и надо что-то выбрать одно, а то перечислять все технологии в заголовке плохо. Особенно когда рассматриваешь пример не простого приложения типа Hello World.

При тестировании real-life приложения возникает очень много вопросов по реализации и использовании технологий, которые обычно не рассматриваются в обзорных статьях, которые копипастятся друг у друга блогерами и агрегаторами статей. Надеюсь, данная статья прольет свет на некоторые аспекты использования упомянутых технологий.

Приложение

Поскольку разных QuickStart руководств для вышеозначенных технологий очень много, то не будем рассматривать базовые принципы, благо они отлично расписаны на сайтах и подводных камней в процессе обнаружено не было.

Основная идея приложения классическая, есть список заказчиков, заказов, списков товаров. Для наших целей хватит того, что эти списки можно просматривать. Вот так незамысловато все.

Тестовое приложение несколько необычное, консольное. Для разнообразия - а то все на WPF, да на WPF. Можно ведь и с другим подходом размяться, тем более что общая структура все равно останется такой же.

Общая структура

Одним из грехов современного проектирования ПО является зависание/замораживание интерфейса пользователя. Никто не любит подвисшие приложения, которые не дают ничего сделать и никак не реагируют на команды пользователя, нервируя его и отвлекая. Чтобы этого избежать рекомендуется все действия, которые потенциально могут занять более секунды времени проводить в отдельных потоках. Пользователь благосклоннее отнесется к сообщению о том, что надо подождать, чем к зависшему экрану программы.

Получается, что сервисы в приложении должны выполнять асинхронные операции, так как по постановке задачи идет работа с внешними источниками данных, подключение к которым, и работа с которыми требует времени. По окончании работы вызывается соответствующее событие, подписавшись на которое, можно узнать результаты работы.

Все сервисы зарегистрированы в StructureMap, и их можно получить для работы только по интерфейсу сервиса. Помогает как в разработке, так и в тестировании. Про StructureMap подробно рассказывать уже не буду, ибо есть материал.

Тесты построены на nUnit, с применением NBuilder и FluentAssertion для более легкого написания тестов и их чтения. Запись тестов используется в стиле BDD.

Подключения к реальной базе не будет, все данные будут заранее забиты. Для простоты повествования и программы.

Реализация основных моментов

Сервисы

Начнем рассмотрение программы с сервисов, в том виде как они, скорее всего, будут реализованы в настоящей программе. В нашем примере есть сервисы для получения заказчиков и для получения заказов. Для показа идеи хватит.

Итак, сервис для получения заказчиков называется CustomerLoadService и в нем только один публичный метод, чтобы получить всех наших заказчиков. Метод возвращает указатель на задачу (Task), чтобы по завершении задачи можно было узнать, это наша задача завершилась или нет. Чуть подробнее примеры на это ниже опишу. Так же возвращая тип Task, как результат вызова метода, мы подготавливаем свой код к использованию фреймворка .Net 4.5 и конструкций async\await.

До тех пор пока официального релиза не было, и у вас нет возможности распространить .net 4.5 среди потенциальных пользователей, целесообразно использовать события как способ сообщения о том, что задача завершена.

Как это все может выглядеть в коде:

```
public class CustomerLoadService : ICustomerLoadService {
    private readonly IDataFacade facade;

    public CustomerLoadService() {
        facade = ObjectFactory.GetInstance<IDataFacade>();
    }

    public event Action<Task> CustomersLoaded;

    public Task<List<Customer>> LoadAllCustomersAsync() {
        var task = new Task<List<Customer>>(LoadAllCustomers);

        if(CustomersLoaded != null)
            task.ContinueWith(CustomersLoaded);

        task.Start();

        return task;
    }

    private List<Customer> LoadAllCustomers() {
        return facade.Get<Customer>(i => true);
    }
}
```

Мне кажется, ничего особенно сложного здесь не присутствует. Используется базовая запись для работы с задачами, не используются токкены отмены операции, специальные указания для выполнения и т.д. Просто запускаем задачу, назначаем продолжение задачи в виде вызова события, проверив, что оно не равно null.

Доступ к фасаду данных осуществляется через DataFacade, который получаем в конструкторе с помощью StructureMap.

Расширение задач сервиса развивается по представленному шаблону:

- Событие
- Асинхронный метод
- Реализация основного метода с работой

В данном сервисе не представлено, но целесообразно порой так же проверять, была ли уже запущена задача и если да, то не запускать ее второй раз, а просто подписать на событие получателя. Это может быть использовано, когда данные получать /обрабатывать долго, и они не меняются сильно за короткий промежуток времени.

Остальные сервисы построены с использованием текущего подхода.

DataFacade

Хотел бы немного поподробнее остановиться на данном классе, так как он эмулирует работу базы данных, а так же показывает насколько удобно пользоваться NBuilder для определенных задач.

Так как данный класс эмулирует базу данных, то при создании экземпляра класса необходимо создать все необходимые данные, которые будут приходить как бы из базы данных. Хранить созданные данные удобнее всего наверно в словаре, где ключом будет тип данных. Так как класс не шаблонный, а задавать тип элементов как List<object> не очень мне нравится, то использование dynamic выглядит органично достаточно.

```
public class DataFacade : IDataFacade {
    private readonly Dictionary<Type, dynamic> database = new Dictionary<Type,
dynamic>();

    public DataFacade() {
                                        new Order("3x4", customers[3]),
                                    };

        items.Take(5).ToList().ForEach(i => orders[0].AddItem(i, 2));
        items.Skip(5).Take(2).ToList().ForEach(i => orders[0].AddItem(i, 2));
        items.Skip(7).Take(3).ToList().ForEach(i => orders[0].AddItem(i, 2));

        database[typeof (Item)] = items.ToList();
               var customers = new List<Customer> {
                                                new Customer{Name = "ACME"},
                                                new Customer{Name = "Balto", IsAlive =
false},
                                                new Customer{Name = "Conso"},
                                                new Customer{Name = "Zorga"}
                                            };
        database[typeof(Customer)] = customers;

        var items = Builder<Item>.CreateListOfSize(20).Build();

        var orders = new List<Order> {
                                        new Order("001", customers[0]),
                                        new Order("x05", customers[2]),
database[typeof (Order)] = orders;
        database[typeof (OrderItem)] = orders.SelectMany(i => i.Items).ToList();
    }

    public List<T> Get<T>(Func<T, bool> spec) {
```

```
        return ((List<T>) database[typeof (T)]).Where(spec).ToList();
    }

}
```

Создать заказчиков хотелось уникально, хотя можно было это сделать с помощью NBuilder, и тогда запись была бы:

```
var customers = Builder<Customer>.CreateListOfSize(4)
                .Random(1,0,3)
                .With(x => x.IsAlive = false)
                .Build();
```

Не принципиально, какой клиент будет неактивным. Уже по самой записи, которая использует fluent интерфейс, понятно что и как будет сделано. Это гораздо короче, чем прописывать все свойства самому. При этом вы получите уникальных заказчиков, имена будут у них: Name1, Name2, Name3, Name4 – так же буду проинициализированы все поля.

Если вы не прониклись записью клиентов, то посмотрите, как легко создать список из 20 элементов, которые представляют возможные товары для заказа:

```
var items = Builder<Item>.CreateListOfSize(20).Build();
```

Ну не чудо ли? И элементы будут разные!

Модель

Далее обратим свой взор на модель, которая связывает действия пользователей в последовательность вызова сервисов и отображения результатов работы. Не буду рассматривать всю модель, остановимся на работе с сервисами. Хотя общий подход я уже описывал здесь.

Итак, основные действия:

1. Подписаться на событие завершения работы метода сервиса
2. Вызвать метод
3. При срабатывании подписки проверить, что пришла наша задача
4. Отписаться от события
5. Обработать ответ

Использование Callback метода нехорошо, так как увеличивается сигнатура метода и надо самостоятельно контролировать их работу при нескольких подписках.

Рассмотрим куски кода связанные с загрузкой заказчиков:

```
public void ProcessMainMenu(string obj) {
    switch (obj) {
        case "1":
            Print("Customers Selected and loading...");
            customerLoadService.CustomersLoaded += CustomersLoadLoaded;
            customerLoadedTask = customerLoadService.LoadAllCustomersAsync();
            OnAnswer = SelectCustomer;

            break;
        case "2":
              …
            break;
```

```
        case "3":
            …
            break;
    }
}
```

Видим, что здесь выполнены первые два пункта из списка типовых действий. Далее нужна обработка ответа от сервиса.

```
private void CustomersLoadLoaded(Task obj) {
    if(customerLoadedTask != obj) return;

    customerLoadService.CustomersLoaded -= CustomersLoadLoaded;
    customers = customerLoadedTask.Result;

    for (var i = 0; i < customers.Count; i++) {
        Print(string.Format("{0}. {1}", i, customers[i].Name));
    }
    Print("Select Customer");
}
```

Тут тоже нет специфических кусков кода, все вполне прозаично.

Собственно работу с сервисами в тестах и рассмотрим.

Тестирование

Подошли к основной части повествования, ради чего все и затевалось. Итак, надо протестировать, как работает модель с вызовом асинхронных сервисов.

Начнем с того, что тесты на одну модель делятся на группы файлов с тестами по поведению, это может быть поделено следующим образом:

- WhenModelStarts
- WhenSelectCustomers

И так далее. Так как файлов много, а тестовые данные готовить надо каждый раз, чтобы тесты были независимыми, то надо создать базовый тестовый класс, где определить основные рабочие элементы.

Базовый класс

В настройку тестового класса будет входить создание мокированых сервисов и выделение их в переменные доступные дочерним классам. **Мокировать надо типы конкретных классов.**

```
[TestFixtureSetUp]
public void BaseClassSetup() {
    ObjectFactory.Configure(x => x.For<IDataFacade>().Use<FakeFacade>());

    CustomerService = new Mock<CustomerLoadService>();
    OrderService = new Mock<OrderService>();
}
```

Далее будет идти настройка тестовых методов. Здесь мы уже зададим базовые значения которые будут возвращаться методами сервисов, а так же внесем сервисы в StructureMap, **надо вносить не**

мокированные классы, а отображение конкретных классов, которые можно получить через свойство Object.

```
[SetUp]
public void BaseSetup() {
    Model = new Model();

    CustomerService.As<ICustomerLoadService>()
        .Setup(i => i.LoadAllCustomersAsync())
        .Returns(LoadAllCustomersResults);

    OrderService.As<IOrderService>()
        .Setup(i => i.LoadOrdersByAsync(Model.SelectedCustomer))
        .Returns(LoadOrdersByResult);


    ObjectFactory.Configure(x => {
                    x.For<IOrderService>().Use(OrderService.Object);
                    x.For<ICustomerLoadService>().Use(CustomerService.Object);
                           });


}
```

Для мокированных объектов можно настроить значения, которые будут возвращаться методами при их вызове в тестах. Это можно увидеть сразу после создания модели. Эту настройку следует разобрать детально.

Начнем с того, что мокированные объекты создаются для конкретного класса, это можно видеть в методе BaseClassSetup. В противном случае возникнут сложности с настройкой возбуждения события в тестах. Далее, при настройке поведения переопределяется поведение класса, и для этого нужны либо все виртуальные методы и свойства, либо мокированный объект от интерфейса. Именно поэтому настройка результата для LoadAllCustomersAsync идет после обращения мокированного объекта к интерфейсу.

```
CustomerService.As<ICustomerLoadService>()
        .Setup(i => i.LoadAllCustomersAsync())
        .Returns(LoadAllCustomersResults);
```

Так как метод возвращает задачу, то лучше всего это определить в виде виртуального метода, который можно будет локально переписывать.

```
public virtual Task<List<Customer>> LoadAllCustomersResults() {
    loadAllCustomersResults = new Task<List<Customer>>(CustomerSet);
    loadAllCustomersResults.Start();
    loadAllCustomersResults.Wait();

    return loadAllCustomersResults;
}

public virtual List<Customer> CustomerSet() {
    return new List<Customer>();
}
```

В тестах нам асинхронность не нужна, поэтому после вызова задачи дожидаемся ее выполнения здесь же, и только после этого отдаем Task наружу.

Так же важным моментом является то, что настройки сервисов необходимо проводить до того, как будет вызвано свойство Object.

Конкретный тестовый класс

В конкретных тестовых классах, когда тесты будут на получения данных от сервисов с асинхронными методами, будем вручную поднимать событие в нужный нам момент времени.

Рассмотрим тест на показ всех заказчиков, которые у нас есть.

```
[Test]
public void ShouldShowCustomerList() {
    // Arrange
    var expected = new StringBuilder();
    expected.AppendLine("Customers Selected and loading...");
    expected.AppendLine("0. Name1");
    expected.AppendLine("1. Name2");
    expected.AppendLine("Select Customer");

    Model.InitModel();

    actual.Clear();
    // Act
    Model.ProcessMainMenu("1");

    CustomerService.As<ICustomerLoadService>().Raise(x => x.CustomersLoaded += null,
loadAllCustomersResults);


    // Assert
    actual.ToString().Should().Be(expected.ToString());
}
```

В тесте необходимо самостоятельно вызывать событие мокированного сервиса, так как кроме нас его вызвать некому. Делается это с помощью метода **Raise**, в котором указывается какое событие надо возбудить и какой результат вернуть. Так как событие вне класса может быть только с левой стороны, то подписываем на событие null. Рекомендуемый разработчиками подход.

Опять же, необходимо обязательно добавить интерфейс к мокируемому объекту с помощью конструкции **.As<T>()**, после чего вызывать событие. Если этого не делать, то придется объявить событие виртуальным, что будет выглядеть несколько странно в продуктовом коде.

Далее проверяем результат и для этого используем FluentAssertion. Я уже писал по этому поводу, только у меня была конкретная самописная штука, а тут более общая. И с ней реально удобно, хотя я и дописал в итоге свои методы для проверки точных совпадений.

Скорее всего, почитав повнимательнее про Raise, можно подумать, а чего бы сразу в настройке метода в базовом классе не задавать автоматическое возбуждение события? Если так делать, то переход к методу обработки перейдет до того, как будет инициализирована переменная задачи, по которой будет идти сегрегация задач на свой\чужой. Т.е. такой код не прокатит:

```
    CustomerService.As<ICustomerLoadService>()
        .Setup(i => i.LoadAllCustomersAsync())
        .Returns(LoadAllCustomersResults)
        .Raises(x => x.CustomersLoaded += null, loadAllCustomersResults);
```

Итого

- Мокировать надо настоящие классы.
- Настраивать мокированные классы надо с помощью добавления интерфейса, до того как вызвали свойство Object.
- События автоматически не вызывать.
- Настраивать сервисы необходимо до того, как вызовите свойство Object. По всей видимости обращение к этому свойству закрывает объект для модификации мокируемого класса.
- Настройка события идет тоже через приведение к интерфейсу.

Примеры:

```
[SetUp]
public void BaseSetup() {
    Model = new Model();

    ObjectFactory.Configure(x => {
                    x.For<IOrderService>().Use(OrderService.Object);
                    x.For<ICustomerLoadService>().Use(CustomerService.Object);
                            });

    CustomerService.As<ICustomerLoadService>()
        .Setup(i => i.LoadAllCustomersAsync())
        .Returns(LoadAllCustomersResults)
        .Raises(x => x.CustomersLoaded += null, loadAllCustomersResults);

    OrderService.As<IOrderService>()
        .Setup(i => i.LoadOrdersByAsync(Model.SelectedCustomer))
        .Returns(LoadOrdersByResult);
}
```

Если так написать, то во время выполнения будет ошибка о том, что нельзя добавить интерфейс после вызова свойства Object, класс как бы закрывается.

Далее, если добавить интерфейсы до того, как вызовем волшебное своейство:

```
[SetUp]
public void BaseSetup() {
    Model = new Model();

    CustomerService.As<ICustomerLoadService>();
    OrderService.As<IOrderService>();


    ObjectFactory.Configure(x => {
                    x.For<IOrderService>().Use(OrderService.Object);
                    x.For<ICustomerLoadService>().Use(CustomerService.Object);
                            });

    CustomerService.As<ICustomerLoadService>()
        .Setup(i => i.LoadAllCustomersAsync())
        .Returns(LoadAllCustomersResults)
        .Raises(x => x.CustomersLoaded += null, loadAllCustomersResults);

    OrderService.As<IOrderService>()
        .Setup(i => i.LoadOrdersByAsync(Model.SelectedCustomer))
```

```
        .Returns(LoadOrdersByResult);
}
```

То в тесте вы увидите, что свойство Result у задачи не проинициализировано, что как бы намекает нам на пропуск настройки мокируемого объекта.

Заключение

Статья названа все же верно, так как ключевое внимание направлено на Moq, и я хотел упомянуть о средствах типа NBuilder и FluentAssertions, так как они полезны, но посвящать им статью было бы неправильно, так как по ним отличные примеры на сайтах производителя.

Hard’n’heavy!

Source