DDD & TDD. Часть III

Прочитать статью на сайте

Итак, у нас есть домен с необходимыми классами и сервисами. В целом можно генерировать и обрабатывать данные в любых количествах согласно написанным сервисам. Но есть проблема, даже проблемы:

- Как пользователь будет вводить данные;
- Как данные будут сохраняться/загружаться.

Домен по природе и задумке своей должен быть полностью независимым и работать независимо от того, как построено взаимодействие с пользователем (WPF, WinForms, Web) и как реализован слой данных (MSSQL, MySQL, Oracle и т.д.).

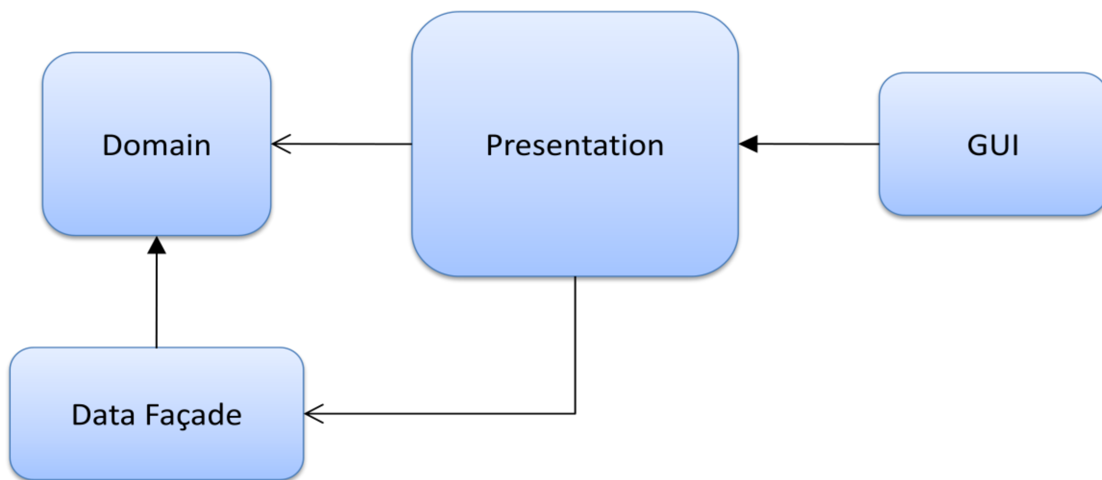
Слоеные пироги

В этой части мы плавно подходим к тому, что приложение должно быть многослойное. Как минимум:

- **Домен – слой бизнес-логики**. В котором описано КАК работать с данными.
- **Слой данных**. В этом слое осуществляется сохранение объектов во внешние ресурсы и восстановление из внешних ресурсов в доменные объекты. Выделением этих операций в отдельный слой мы делаем приложение более гибким, т.к. можно осуществлять работу с несколькими БД, меняя их, а приложение этого даже не заметит.
- **Слой представления**. В этом слое мы говорим приложению ЧТО делать с данными, т.е. вызываем в нужном порядке доменные сервисы и отдаем результат вычислений на прорисовку. Таким образом, этот слой связывает Домен и Front End.
- **Пользовательское отображение (Front End).** Это пользовательский интерфейс, может быть чем угодно: консолью, win forms, web, да хоть сразу в мозг! =) Этот слой должен быть максимально «тупой», тут недолжно быть никаких изменений данных, никакой логики, как и что показывать. Слой просто делает все что ему скажут без капли самодеятельности.

После того, как определились из чего состоит приложение, надо понять как эти слои связаны между собой.

Еще раз скажу, что *домен ни от чего не зависит*, доменная сборка не должна содержать ссылок на другие слои. Слой данных ссылается на домен, т.к. требуется сохранять и восстанавливать данные из внешних источников. Слой представления ссылается на домен и слой данных. Пользовательское отображение ссылается на представление. Более подробно все детали соотношений будут далее по тексту, пока же можно связи представить следующим образом:



Для того чтобы эта схема заработала надо вкратце рассказать о паттерне Inversion Of Control (http://ru.wikipedia.org/wiki/Dependency_Injection)/Dependency Injection(http://ru.wikipedia.org/wiki/Обращение_контроля).

Inversion of Control/Dependency Injection

В двух словах этот паттерн позволяет разорвать жесткую зависимость объектов.

Без применения этого подхода домен был бы вынужден наследоваться от дата фасада для работы сервисов (дозагрузка данных для работы, как пример). Пришлось бы в домене создавать трансляторы или репозитории и работать с ними. Вместо этого в домене создаем интерфейсы для этих классов и работаем с ними.

Было:

В домене код

```
public class DomainService {
        public int CalculationWithDataLayer() {
            var repository = new Repository();
            var list = repository.GetData();
            // вычисления
        }
    }
```

В дата фасаде

```
public class Repository {
        public List<object> GetData() {
            // вычисления
        }
    }
```

Стало:

В домене код

```
public interface IRepository {
        List<object> GetData();
}
```

```
public class DomainService {
        public int CalculationWithDataLayer(IRepository repository) {
            var list = repository.GetData();
            // вычисления
        }
    }
```

В фасаде

```
public class Repository : IRepository {
        public List<object> GetData() {
            // вычисления
        }
    }
```

Получается, что фасад осуществляет реализацию интерфейсов из домена.

Такая же история и с пользовательским интерфейсом. В слое презентации определяются интерфейсы для вьюх (view. По-другому уже не сказать), которые будут реализованы в слое GUI.

Тестирование

Представим что у нас все реализовано без интерфейсов. Придется подготавливать очень много данных, для того чтобы проходили тесты. Это будет адом, да таким что вы перестанете писать тесты вообще. С помощью интерфейсов можно сделать поддельные (Fake) классы.

Например, мы тестируем поведение некоторого сервиса, который вызывает другой сервис. В обоих могут быть достаточно объемные вычисления.

```
    public class ServiceA {
        public int LongCalcA() {
            var result = 0;
            Thread.Sleep(1000000);
            return result;
        }
    }

    public class ServiceB {
        public int LongCalcB(ServiceA serviceA) {
            var a = serviceA.LongCalcA();
            a += 42;
            return a;
        }
    }
```

Тест на сервис B будет выглядеть так:

```
[TestMethod]
public void CalcTest() {
            var a = new ServiceA();
            var b = new ServiceB();
            Assert.AreEqual(42, b.LongCalcB(a));
        }
```

И как вы думаете, сколько будет тест ходить? И будет желание каждый раз его запускать? У меня точно не будет!

Сейчас будем творить самую обыкновенную уличную магию по ускорению прохождения тестов. Для того чтобы тесты забегали быстро-быстро необходимо выделить интерфейсы для сервисов.

```
public interface IServiceB {}

    public interface IServiceA {
        int LongCalcA();
    }

    public class ServiceA : IServiceA {
        public int LongCalcA() {
            var result = 0;
            Thread.Sleep(1000000);
            return result;
        }
    }

    public class ServiceB : IServiceB {
        public int LongCalcB(IServiceA serviceA) {
            var a = serviceA.LongCalcA();
            a += 42;
            return a;
        }
    }
```

После этого в тестовом проекте можно создать фальшивый класс сервиса А. Для удобства так же создадим и свойство, которое будет отвечать за возвращаемый результат.

```
public class ServiceAFake : IServiceA {
        public int LongCalcAResult{get;set;}

        public int LongCalcA() {
            return LongCalcAResult;
        }
    }
```

Теперь тест на сервис В будет выглядеть так:

```
[TestMethod]
public void CalcTest() {
            var a = new ServiceAFake{ LongCalcAResult = 0};

            var b = new ServiceB();
            Assert.AreEqual(42, b.LongCalcB(a));
        }
```

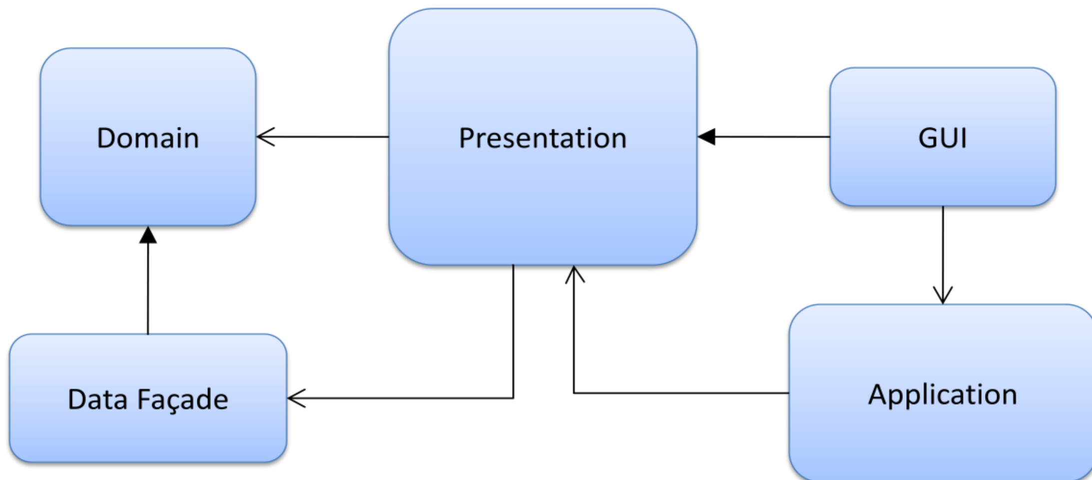
И ходить он будет примерно в 1000000 раз быстрее!!! Это невероятно! ;) Но тесты на сервис А все равно будут долгими, от этого никуда не деться. Но хотя бы использование его в других тестах будет быстрым, а это немало уже. Я надеюсь, что основной принцип тестирования вы уловили.

Запуск приложения

После того как сделаете в порыве творческого запала некоторую работу, захочется проверить первые результаты в деле. И тут возникает вопрос, как все это хозяйство запустить, ведь из сборки с GUI мы не можем создать ни одного класса, который смог бы стать главным пультом управления. Что делать?

Создать еще одну специальную управляющую сборку, которая будет доступна для GUI и будет запускать требуемые сценарии работы программы. Можно сборку назвать что-то в духе

AppRunner (Application). И в этой сборке в самом простом случае может быть достаточно всего одного класса Runner, который будет содержать все возможные варианты сценариев работы из слоя представления и запускать их по запросу с пользовательского интерфейса. Runner может содержать всего один публичный метод, которые принимает параметр указывающий на необходимый сейчас сценарий работы.



Из принципа работы понятно, что Application зависит от типа GUI.

Порядок работы:

1. Запуск приложения. Одновременно запускается Runner с переданным заранее значением, указывающим на сценарий приветствия.
2. Пользователь жмет на пункт меню.
3. В обработчике события вызывается Runner с значением сценария в виде параметра.
4. Сценарий из слоя представления запускается.
5. Все действия не связанные с переходами между сценариями обрабатывает запущенный сценарий.

Да, согласен, в этом месте немного сумбурно, но это потому что тут уже многое зависит от самой структуры программы, от того каким образом запускаются основные сценарии. Причем сборка Application вполне может быть связана и с дата фасадом. Вариантов миллион.

Вопросы, пожелания, замечания приветствуются!