TDD & DDD. Часть II

Прочитать статью на сайте

Диспозиция

У нас появились доменные классы с минимальным набором свойств и методов. Есть некоторое количество тестов на классы. Надо научиться работать с ними – т.е. написать сервисы, которые могут делать что-то более серьезное с доменными классами, например, сортировка, выборка, подсчет учеников в параллели.

Доменные сервисы

Допустим, в условиях к программе сказано, что *вам надо определять количество классов на заданный год обучения (сколько 5ых классов или 9ых) и сколько учеников в параллели*.

Подумаем, какие классы и что у нас знают:

- «Школа» знает сколько всего вообще классов;
- «Класс» знает сколько в нем учеников.

Кажется больше нам и не надо. Как бы я сделал на заре своего обучения? Я бы в класс School добавил метод

```
public int CountStudent(int classNumber) { … }
```

Но это плохой подход по следующим причинам:

- У класса School увеличивается ответственность что приводит к высокой связности классов;
- Его становится труднее поддерживать и изменять;
- Это так же может привести к дальнейшим проблемам тестирования.

Увеличение ответственности классов всегда плохо, потому что тенденцию безконтрольного разрастания (антипаттерн «Суперкласс») все сложнее и сложнее остановить. Если вам страшно менять код программы, относитесь к коду в духе «работет вроде и ладно», значит что-то не так в программе. Скорее всего методы и классы разрослись не на 2-3 экрана, а на 5 и более. Тестировать сильно связанные вещи тяжело по той причине, что придется готовить очень много данных для прохождения теста и очень легко потерять важную часть – собственно что мы хотим протестировать. Возможно вы будете помнить что вы тестировали спустя неделю, но через месяц точно забудете.

Гораздо разумнее сделать отдельный класс, пусть и пока с одним методом который будет принимать номер параллели. Назовем новый класс ClassStatistics.

```
public class ClassStatistics {
        public int CountClassesForYear(School school, int classYear) {
            return -1;
        }
    }
```

Этот класс не хранит никаких данных. Совсем. Все что нужно для работы приходит через параметры. Получается класс который обслуживает другие классы. Вот поэтому-то и называются такие классы «сервисами».

Прошу заметить, что класс возвращает невозможное в реальной жизни значение. Это необходимо для правильного написания тестов. В целом не должно быть ситуации, когда вы пишете тест и он автоматически становится зеленым.

Тестирование доменных сервисов

Итак, начинаем писать тест на этот класс, заодно и посмотрим, насколько удобно мы выбрали название и нотацию. Создаем новый класс, называем его ClassStatisticsTests, и начинаем писать тест. Тут важно правильно его назвать, чтобы если он упадет, вы поняли сразу что он тестировал, что же сломалось у вас.

```
[TestMethod]
public void ShouldCalculateCountClassesForYear() { … }
```

Полезно в тесте сразу написать, что вы ожидаете в итоге увидеть. Т.е. пишите сразу Assert.<условие>.

```
[TestMethod]
public void ShouldCalculateCountClassesForYear() {
     Assert.AreEqual(0, new ClassStatistics().CountClassesForYear(new School(),
0));
     Assert.AreEqual(2, new ClassStatistics().CountClassesForYear(new School(),
1));
}
```

В этом тесте я ожидаю что будет ноль классов для нулевого года обучения, и что четыре класса для первоклашек.

Вот тут же, по написании этих строчек меня разбивает страшнейшая лень, и я не хочу тестировать что будет какой-то дурацкий нулевой класс. Ну и вообще как-то не очень понятно получается. Появляются варианты, что я передам вместо нормального года обучения -4 или 134. Это ведь все придется тестировать, на все писать условия. Представили? Тоже лень стало? ;) Да и про цифры я могу забыть и придется смотреть что означает этот параметр.

Предлагаю изменить сразу код на более удобочитаемый. Создадим энумератор! Пусть вызов метода выглядит так:

```
new ClassStatistics().CountClassesForYear(new School(), EducationYear._01)
```

По-моему намного лучше стало. Тут уж я точно не введу -5 класс или 142ой. Всегда передаются правильные данные, и читать легко. Меняем сигнатуру метода CountClassesForYear, чтобы второй параметр был типа EducationYear.

Вот еще одно доказательство пользы тестов! А ведь в начале создания метода все казалось логично и хорошо. По возможности стоит создавать все новые публичные методы в процессе написания теста.

Теперь тест выглядит так:

```
[TestMethod]
        public void ShouldCalculateCountClassesForYear() {
            var school = new School();

            school.AddClass(new Class{EducationYear = EducationYear._01});
            school.AddClass(new Class{EducationYear = EducationYear._01});
            school.AddClass(new Class{EducationYear = EducationYear._02});
            school.AddClass(new Class{EducationYear = EducationYear._03});

            Assert.AreEqual(2, new ClassStatistics().CountClassesForYear(school,
EducationYear._01));
```

```
        }
```

Запускаем тест, наблюдаем ответ, что количество учебных классов с первоклашками неравно 2. Пишем имплементацию. Самую простую чтобы заработал тест. Не надо думать на несколько ходов вперед. ;)

```
        public int CountClassesForYear(School school, EducationYear classYear)
{
            return 2;
        }
```

Запускаем тест. Работает! Это показывает, что тест мы написали неаккуратно, надо проверять еще на какое-нибудь значение, что метод сервиса действительно считает, а не выдает статическую цифру. Допишем к тесту еще один Assert.

```
            Assert.AreEqual(1, new ClassStatistics().CountClassesForYear(school,
EducationYear._02));
```

Запускаем тест – красный. Теперь придется писать код по-честному!

```
public int CountClassesForYear(School school, EducationYear classYear) {
           return (from c in school.Classes
                       where c.EducationYear == classYear
                       select c).Count();
        }
```

Снова запускаем тест и видим, что он зеленый. Это очень хорошо! Значит считается все верно. После каждого завершенного теста, советую запускать все тесты в тестовом классе, а лучше вообще все тесты в проекте, чтобы выявить возможные поломки на самой ранней стадии.

Теперь нам надо определить кол-во учеников в параллели. Тут можно начинать с тестов сразу. (Вообще психологически тяжело создавать пустой класс и на него писать тесты. Но гораздо легче, когда класс существует, там есть уже какой-то метод и нам надо протестировать несуществующий.)

```
[TestMethod]
public void ShouldCalculateStudentsInClassesForYear() {
    var school = new School();

    var class1A = new Class{EducationYear = EducationYear._01};
    var class1B = new Class{EducationYear = EducationYear._01};
    var class2A = new Class{EducationYear = EducationYear._02};

    school.AddClass(class1A);
    school.AddClass(class1B);
    school.AddClass(class2A);

    var student = new Student("", "");
    class1A.AddStudent(student);
    class1A.AddStudent(student);
    class1B.AddStudent(student);
    class2A.AddStudent(student);

   var statistics = new ClassStatistics();
   Assert.AreEqual(3, statistics.CountStudentsForYear(school,
EducationYear._01));
   Assert.AreEqual(1, statistics.CountStudentsForYear(school,
EducationYear._02));        }
```

Написали имя нового метода, если кажется что все понравилось, то создаем метод с такой же сигнатурой как и предыдущий.

```
public int CountStudentsForYear(School school, EducationYear classYear) {
            return -1;
        }
```

Тест красный, и говорит, что ожидаемые цифры не совпадают с расчетными. Это хорошо, это правильная ошибка. Было бы плохо, если бы говорили о NullReferenceExeptions или еще о чем-нибудь в таком духе.

Пишем имплементацию и снова запускаем тест.

```
public int CountStudentsForYear(School school, EducationYear classYear) {
            return (from c in school.Classes
                        where c.EducationYear == classYear
                        select c).Sum(c=> c.Students.Count);
        }
```

Тест зеленый! Мы молодцы. Если посмотреть на оба метода, то видим, что есть большая основная часть, которую можно вынести. Это называется рефакторингом – улучшение читабельности и устранение copy-paste кода. Выделим общую часть в приватный метод.

```
private IEnumerable<Class> GetClassesForYear(School school, EducationYear
classYear) {
            return from c in school.Classes
                   where c.EducationYear == classYear
                   select c;
        }
```

Методы стали теперь в одну строку и более читабельны. Да тут чистый английский язык!

```
public int CountClassesForYear(School school, EducationYear classYear) {
           return GetClassesForYear(school, classYear).Count();
        }

public int CountStudentsForYear(School school, EducationYear classYear) {
            return GetClassesForYear(school, classYear).Sum(c=>
c.Students.Count);
        }
```

После того как поменяли код, хорошо бы убедится, что мы ничего не испортили. Запускаем снова тесты. Зеленые. Вот это уже полный классический цикл написания тестов. Написали тест, красный, чиним, зеленый, рефакторинг, зеленый. Код принимает стройность и радует глаз, душа спокойна за правильную работу.

Продолжая в том же духе решаем поставленные перед нами задачи.

Итак:

Доменные классы – содержат конкретную информацию о сущностях, которые играют ключевую роль в приложении. Являются представлениями существительных из описания задачи.

Доменные сервисы – обслуживают доменные классы, производят сложные расчеты, занимаются трансоформацией данных, всем-всем-всем. Они могут передавать результаты вычислений друг в друга, идти цепочками длинными, как угодно! Но вся основная логика должна обсчитываться в доменных сервисах.

Еще немного об организации процесса.

В домене полезно создать новую папку (правой кнопкой по проекту, Add > New Folder) которую назвать Services и туда складывать все сервисы. Так же советую поступить в тестовом проекте.

Добавить папку Domain, а в нее Services. Т.к. в тестовом проекте будут и другие тесты, на другие части программы (при относительно небольших размерах приложения). Следуя такому методу будет легко находит нужный файл.

Когда вы замечаете что для тестов внутри одного тестового класса готовятся одинаковые данные, их можно вынести в отдельный метод и пометить его [TestInitialized]. Такой метод будет запускаться автоматически перед каждым тестом в этом классе.

```
[TestInitialize]
public void TestInitialize() {
          //ваша инициализация данных
}
```

В следующих сериях…

Я расскажу, как домен связывается с тем, что видит пользователь, как загружаются данные. Покажу общую картину взаимодействия библиотек/модулей программы и как это взаимодействие тестировать.

Вопросы и пожелания приветствуются))

Hard’n’heavy!