DDD & TDD

Прочитать статью на сайте

Disclaimer

Все ниже написанное специально упрощено и показывает, как с этим начать работать, всем давно практикующим специалистам просьба не быть строгим к словам.

Магические аббревиатуры

DDD – Domain Driven Design, если совсем вкратце, то это способ организации кода приложения. А организуется он таким образом, чтобы в единственной, независимой ни от чего в проекте сборке, лежала сама суть приложения.

TDD – Test Driven Design, проектирование приложения через тесты.

Обе эти практики (хотя наверно это лучше назвать уже философии) обычно идут рука об руку в моих проектах, аки сиамские братья близнецы, и мне, если честно, трудно представить себе их раздельно. Хотя конечно они могут работать и абсолютно раздельно, никто не обязывает использовать их в связке.

По DDD и TDD написано уже куча литературы различных форматов и объемов. Где-то проще, где-то запутаннее. Исторический экскурс благополучно пропустим, кому интересно, могут погуглить книги Эрика Эванса и посетить сайт (http://martinfowler.com/) Мартина Фаулера. Я постараюсь донести свое видение этого предмета максимально просто и доступно. Рассказать, как это помогает, работает, сопровождается и развивается. Надеюсь, у меня все получится. =)

Но это все вкратце, теперь перейдем к более детальному рассмотрению.

Domain Driven Design на примере

Представим, что у вас есть небольшая задача. Задача кажется вам невелика и относительно проста. Пусть это будет что-то в духе учета посещаемости в классе и оценок, эдакий виртуальный классный журнал. Попробую ее расписать, как ее бы сформулировал простой человек.

У нас есть школа и в ней классы с учениками. Классы мы стараемся формировать из расчета от 15 до 25 человек в классе. Ученики ходят в школу 6 дней в неделю, за исключением младшеклассников, у них 5 дневная учебная неделя. В школе у нас 50 педагогов, которые ведут в общей сложности 16 различных предметов. В нашей школе принята 5ти бальная система оценок, по итогам четвертей выставляются оценки по предмету.

Вот наверно и все. Что из этого самое важное? Как организована информация? Попробуем разобраться.

DDD предлагает брать существительные за основы доменных классов, а глаголы за доменные сервисы. Доменные классы должны уметь работать только с собой, они не должны иметь каких либо сложных методов которые позволяют изменять экземпляр класса. Эту работу будут осуществлять сервисы.

Итак:

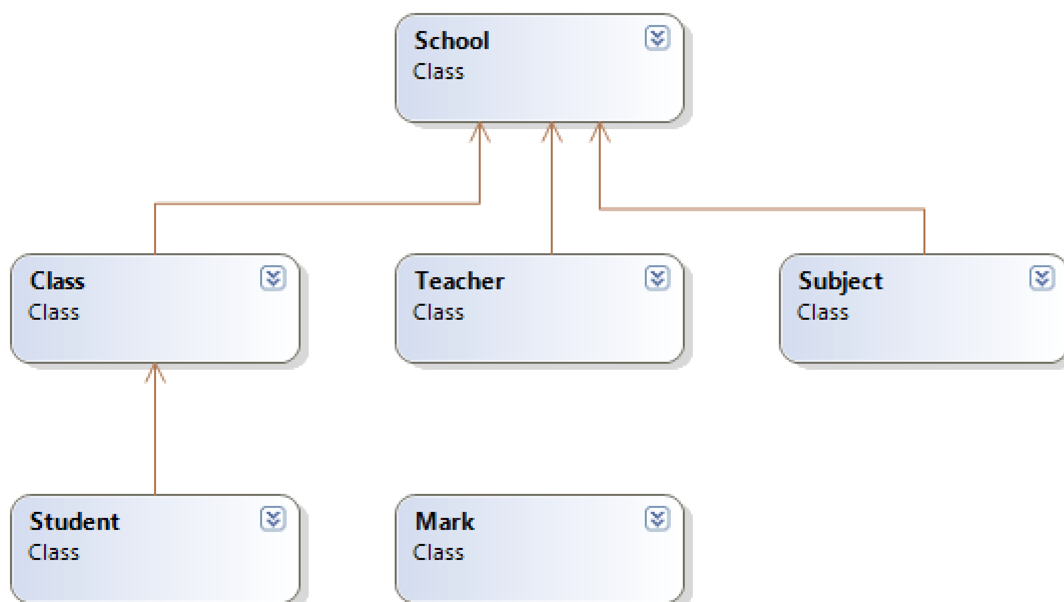
У нас есть **школа** и в ней **классы** с **учениками**. Классы мы стараемся формировать из расчета от 15 до 25 человек в классе. Ученики ходят в школу 6 дней в неделю, за исключением младшеклассников, у них 5 дневная учебная неделя. В школе у нас 50 **педагогов**, которые ведут в общей сложности 16 различных **предметов**. В нашей школе принята **5ти бальная система оценок**, по итогам четвертей выставляются оценки по предмету.

Жирным шрифтов я выделил ключевые существительные и понятия, которые мне кажутся самыми важными, с которыми будет работать ядро программы. У нас получается класс School, Class, Student, Teacher, Subject, Mark.

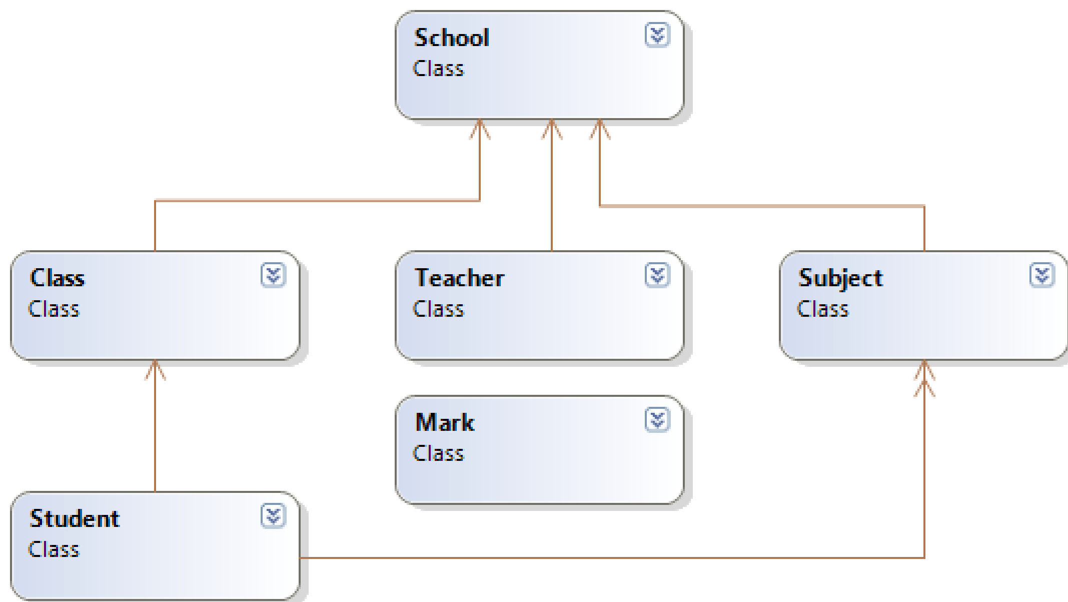
Больше всего сомнений может вызывать класс Mark, т.к. он на первый взгляд будет состоять из одного поля, но скорее всего это не так. В процессе я постараюсь это показать.

Далее интересно как это будет связанно между собой.

Из текста видим, что самый главный агрегат это школа. Школа содержит классы, классы содержат учеников. Школа содержит педагогов. Школа содержит предметы. Педагог может вести несколько предметов. Если все это нарисовать, то получится примерно такая вот схема:



Далее надо определиться как соотносятся ученики и предметы. Видимо понадобится искусственная связь, не отраженная в задаче. Связь будет многие к одному, т.к. ученик посещает многие предметы. Логично? А оценки будут определятся по совокупности ученик+предмет. Т.е. нельзя сказать что ученик содержит оценки, т.к. мы не сможем определить что к какому предмету относится. По той же причине нельзя делать так, что предмет будет являться агрегатом для оценок. С учетом только что сказанного схема приобретает уже следующий вид:



На данный момент схема мне видится более менее адекватной, если есть вопросы то спрашивайте в комментариях, письмах, не стесняйтесь.

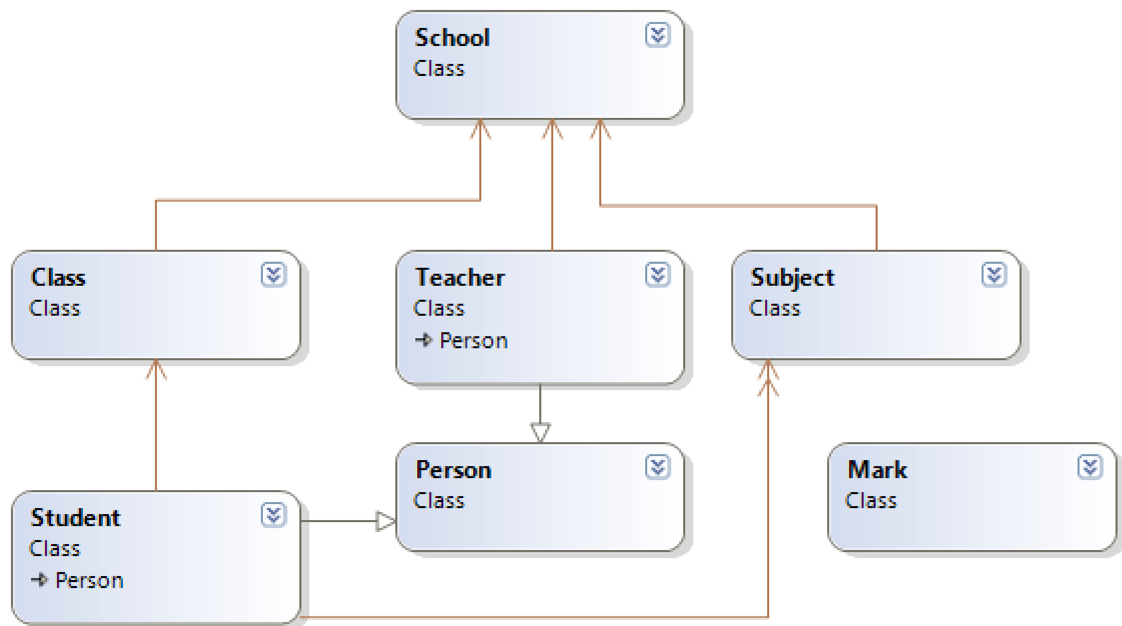
Но схема адекватна до тех пор пока мы видим лишь названия классов. Попробуем их заполнить полями. Я думаю у вас должны мелькнуть мысли о том как улучшить схему в процессе заполнения полей ;). Лень двигатель прогресса, поэтому мне лень заполнять в целом одинаковыми данными классы Student и Teacher, из них можно выделить общий класс Person куда войдет основная информация о человеке.

```
public class Person {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public string MiddleName { get; set; }
    }

public class Student : Person { … }

public class Teacher : Person { … }
```

Итак, после всех преобразований итоговый вариант:



Четко и по делу!

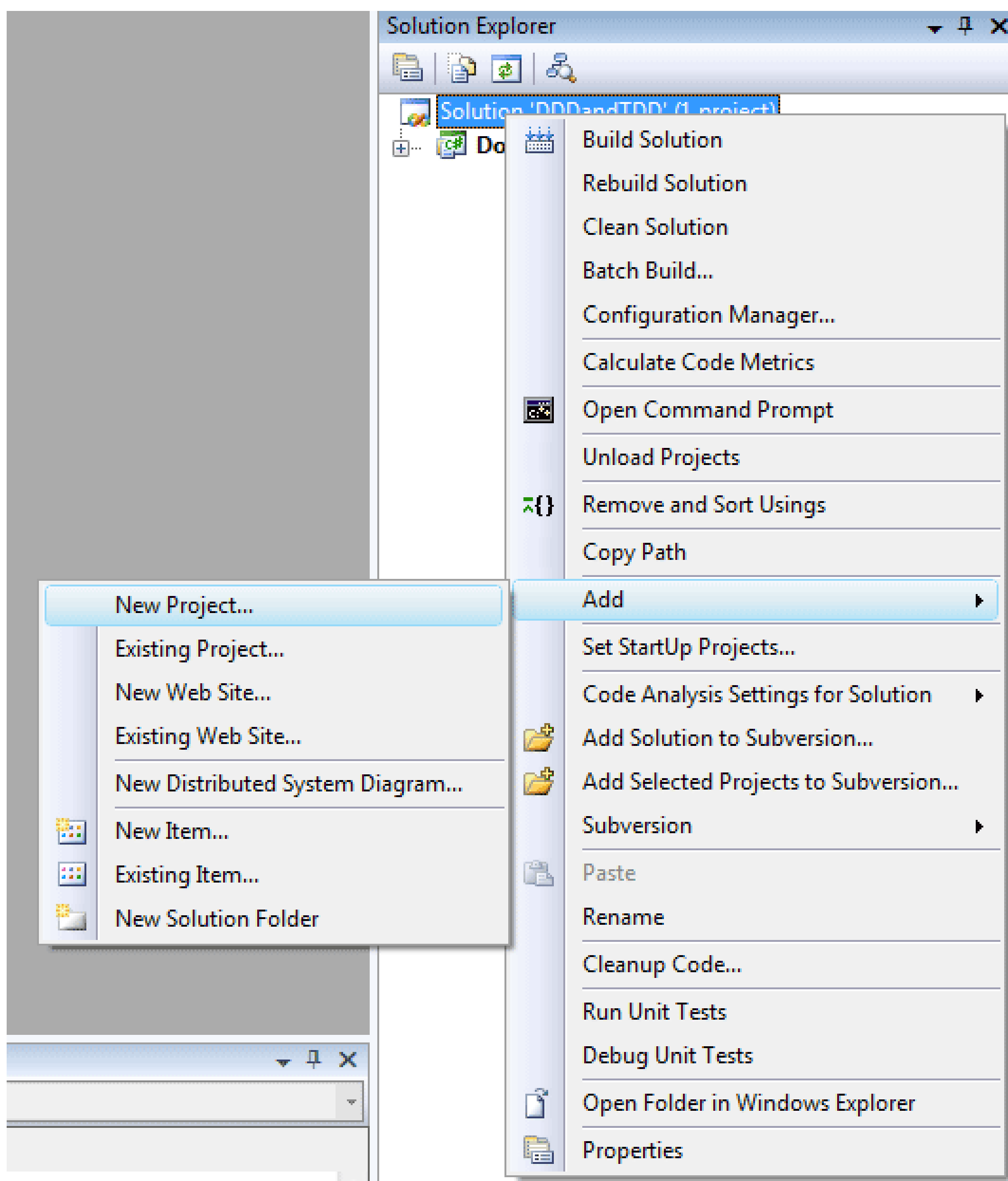
Сейчас у нас только поля и нет механизма создания связей между классами. В этом месте уместно будет начать разговор про TDD. На данном этапе проектирования я примерно представляю как будут выглядеть методы связывания объектов в силу опыта и небольшого размера задачи. Но далее темный лес.

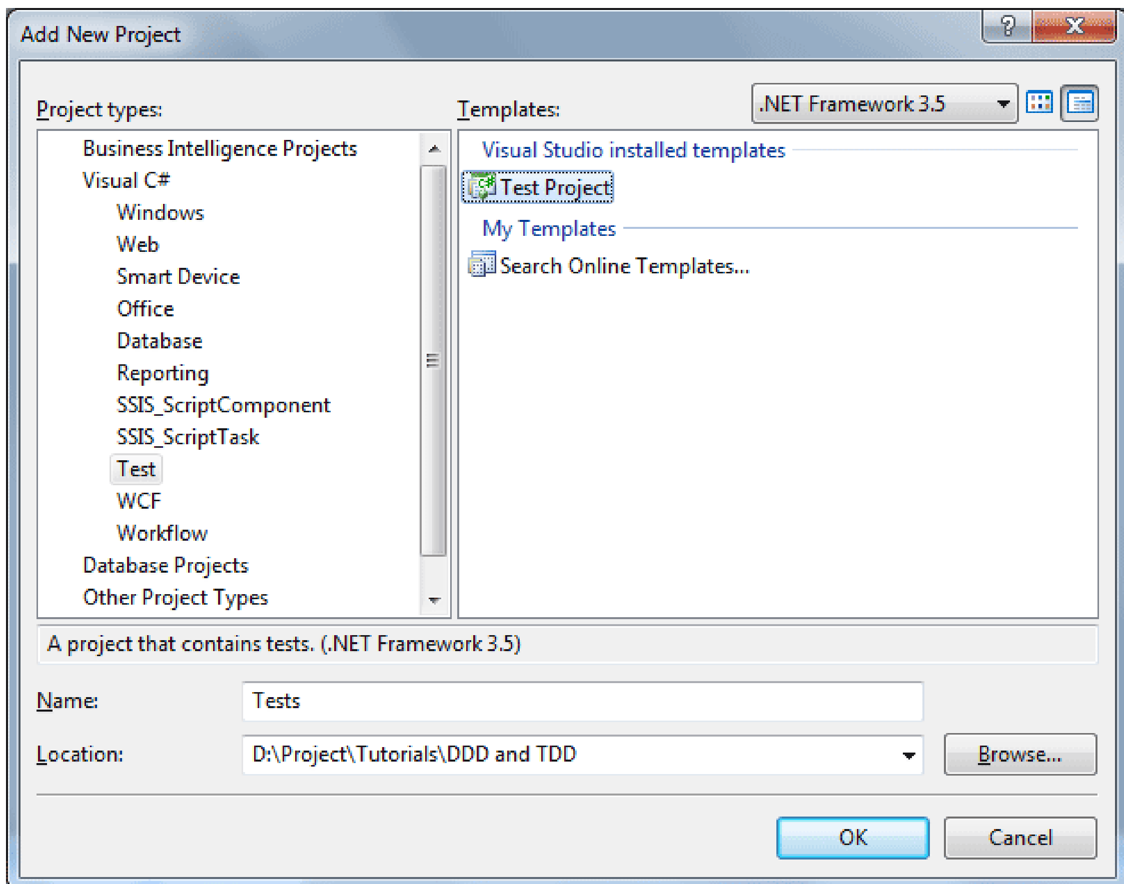
Наполнить классы полями необходимыми я оставляю на вас.

Test Driven Development

Официально эта методика программирования говорит о том, что написание программы делится на маленькие циклы в каждом из которых надо написать маленький тест на какое-то одно простое действие. При этом тест сначала должен быть красным (т.е. не работать), потом заставляем его работать хоть как-нибудь (но без грязных приемов) – тест зеленый, потом улучшаем только что написанный код и в процессе улучшения постоянно запускаем тест, чтобы убедится, что ничего не сломалось. Каждый такой цикл должен длиться до 15 минут. Тесты должны быть быстрыми, все тесты в проекте должны проходить менее чем за минуту, для больших проектов за 5.

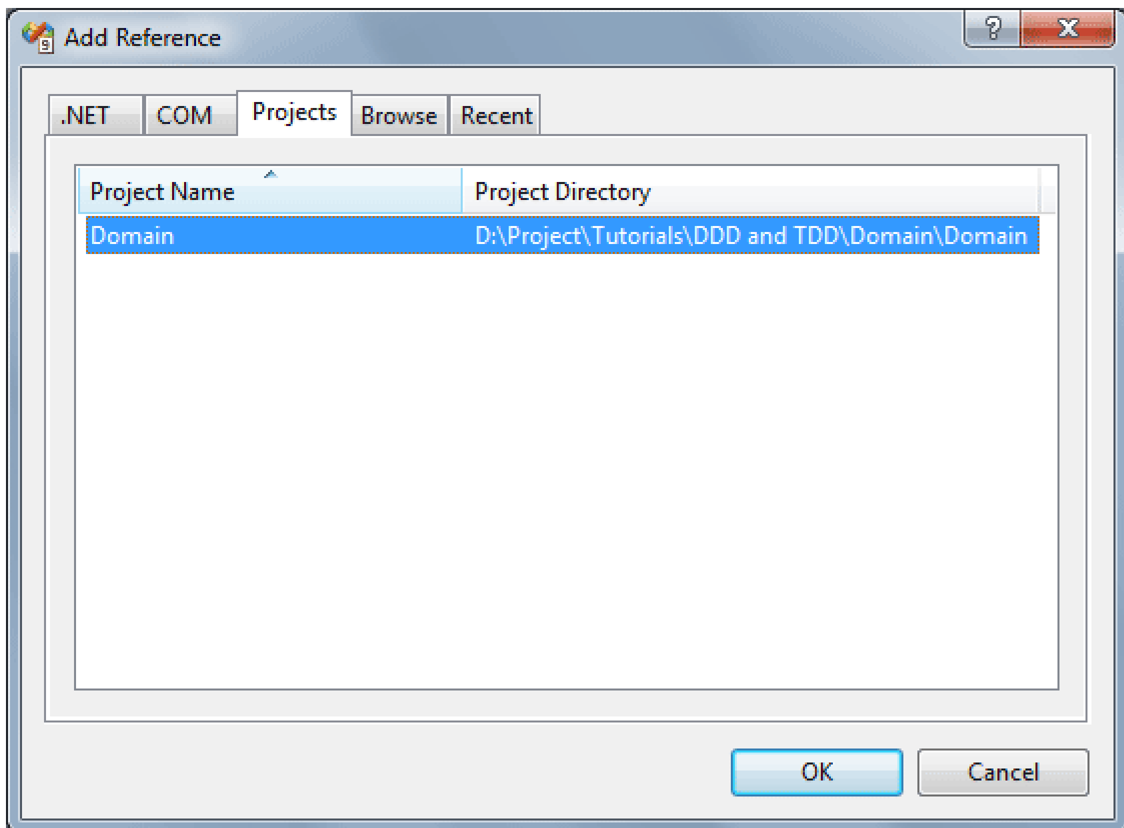
Возникает закономерный вопрос: «Как я буду писать тест на то, чего еще нет?» Сейчас я все расскажу и покажу на примере.





Создаем проект для тестов. Щелкаем правой кнопкой мыши на решении (Solution), выбираем добавить новый проект (Add > New project), в появившемся окне выбираем “Test”, задаем имя тестового проекта и жмем добавить. В результате у нас будет новый проект с тремя файлами внутри. Их надо безжалостно удалить.

Теперь необходимо добавить ссылку (reference) на проект который будем тестировать. Выбираем тестовый проект, жмем правой кнопкой мыши по нему, выбираем “Add Reference…”. В появившемся окне на закладке “Project” и выбираем библиотеку “Domain.dll”. Итак, теперь мы в тестовом проекте можем обращаться к классам из домена. В целом все готово к написанию тестов.



Первый тест напишем на то, чтобы в класс School можно было добавить экземпляры Class. Добавляем новый класс в тестовый проект, называем его SchoolTests и начинаем писать. Исходная позиция такая:

```
namespace Tests {
    public class SchoolTests {

    }
}
```

Для того, чтобы это стало тестом необходимо добавить аттрибуты TestClass, обозначить новый метод, где будет проходить тест и приписать ему TestMethod, вот так:

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace Tests {
    [TestClass]
    public class SchoolTests {
        [TestMethod]
        public void Test() {

        }
    }
}
```

В дальнейшем не забывайте приписывать использование UnitTesting. Далее пишем собственно тест на то, что в класс School можно добавить Class. Переименуем название метода во что-то более осмысленное, передающую главную идею теста.

```
[TestMethod]
        public void ShouldAddClass() {
            var school = new School();
            var testClass = new Class();
```

```
            school.AddClass(testClass);

            Assert.AreEqual(1, school.Classes.Count);
            Assert.AreEqual(testClass, school.Classes[0]);
        }
```

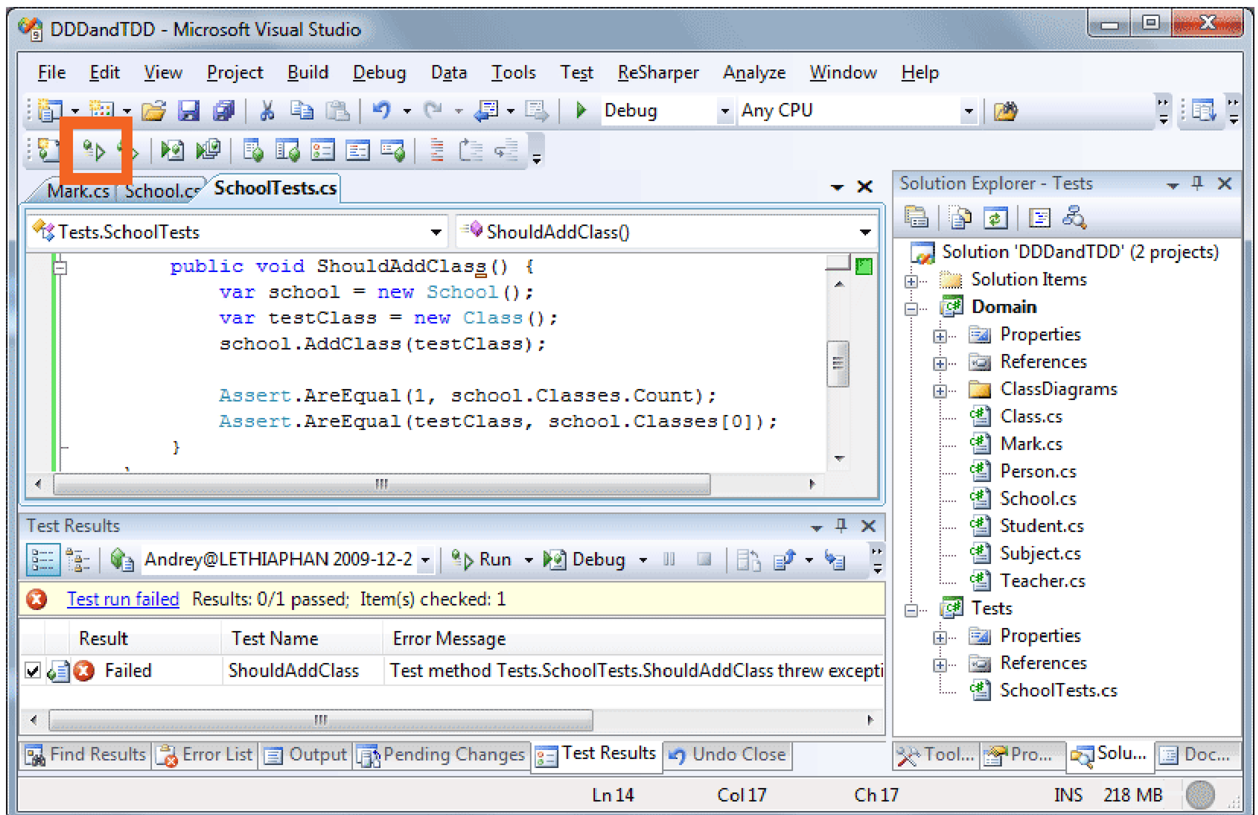
Написав такой код, будет видно что, метод AddClass еще не существует, его надо будет создать, но как пустышку. Т.е. надо делать все, чтобы код стал компилироваться. Строчки начинающиеся с Assert являются самыми важными, они-то и будут сообщать о том, прошел тест или нет, действительно ли произошло то, что мы ожидаем от программы. В данном случаем мы тестируем на то, что в коллекции будет один элемент, и что он будет тот, который мы положили.

Итого:

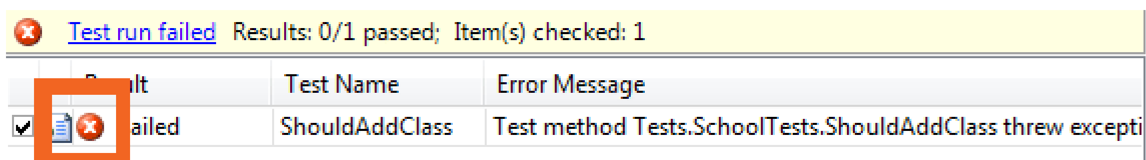
```
public class School {
        public void AddClass(Class clas) {

        }
    }
```

После этого запускаем сборку, смотрим что все скомпилировалось и запускаем тест.



После некоторого времени смотрим вниз и видим:



Тест не прошел (упал). Тут главное определить, что тест упал именно по нашей ожидаемой «правильной» причине. Правильной причиной в нашем случае будет то, что в коллекции 0 элементов, а не один. Дважды нажимаем на строку теста и смотрим причину:

Test method Tests.SchoolTests.ShouldAddClass threw exception: System.NullReferenceException: Object reference not set to an instance of an object..

Это явно говорит о том, что мы что-то не проинициализировали. Отлично! Не совсем то, что ожидали, но тоже повод для радости. Пишем еще один тест, но то что у нас все переменные в классе School инициализированы.

```
[TestMethod]
        public void SchoolInitialized() {
            var school = new School();
            Assert.AreEqual(0, school.Classes.Count);
            Assert.AreEqual(0, school.Teachers.Count);
            Assert.AreEqual(0, school.Subjects.Count);
        }
```

Запускаем его, видим что он не прошел с результатом "Assert.AreEqual failed. Expected:<0>. Actual:<(null)>" – а это правильная ошибка. Идем в класс School и инициализируем эти свойства.

```
public class School {
        private readonly List<Class> classes = new List<Class>();
        private readonly List<Teacher> teachers = new List<Teacher>();
        private readonly List<Subject> subjects = new List<Subject>();

        public ReadOnlyCollection<Class> Classes {
            get { return classes.AsReadOnly(); }
        }

        public ReadOnlyCollection<Teacher> Teachers {
            get { return teachers.AsReadOnly(); }
        }

        public ReadOnlyCollection<Subject> Subjects {
            get { return subjects.AsReadOnly(); }
        }

        public void AddClass(Class @class) {}
    }
```

Снова запускаем тест SchoolInitialized и видим что он зеленый! Теперь запускаем первый тест и видим, что теперь не совпадает кол-во записей в коллекции – что нам и требовалось. Пишем код на добавление

```
public void AddClass(Class @class) {
            classes.Add(@class);
        }
```

Запускаем тест и видим, что он тоже позеленел!

Это вот и является в целом полным циклом написания теста. Написали тест, делаем так чтобы все компилировалось, исправляем и дописываем функционал, чтобы получить ошибку нужного вида. После этого пишем код для работы теста. Тест становится зеленым. После этого код рефакторится (на данном случае рефакторинг не требуется), снова прогоняются тесты, что ничего не сломалось.

В чем прикуп и польза?

Прежде всего в том, что будет уверенность в том, что код работает как ты задумываешь. Нет, это не означает, что он работает правильно =) Тесты показывают, что все идет по твоему плану.

Часто в процессе написания тестов возникают дополнительные мысли, что может пойти не так, или какие еще проверки и условия требуются. Например, в нашем случае, мне пришла мысль, что надо проверять, а не добавим ли мы один и тот же класс в школу дважды. Что надо как-то и классу сказать, что он теперь в этой школе. Все это можно покрыть тестами.

Тесты позволяют писать только тот код, который действительно важен для работы программы. По результатам исследований готовых программ, только 20% кода делают реальную работу, остальные 80% почти никогда не востребованы, но мы можем потратить на них время и силы которые пригодились бы на других фронтах.

Тесты не дают писать неудобный код. Мне лень писать в тесте какие-то сложные функции, переходы и т.к. далее. Я хочу написать тест как можно быстрее (в рамках разумного) и значит за меньшее количество методов. Через тесты я создаю API, которое потом буду с легкостью использовать в реальном коде.

Если у меня есть тесты, у меня нет чувства страха при изменении уже существующего кода. Я могу его улучшать и переписывать, зная что тесты мне подскажут, где я ошибусь в случае чего. Конечно если тесты я писал для себя с пониманием, а не для галочки и для «дяди».

Hard’n’heavy!