# Domain Specific Language для TDD

*Прочитать статью на сайте*

Domain Specific Language (DSL) – это язык специального назначения, который предназначен для решения какой-либо задачи в терминах самой задачи.

Это не какой-то новый язык программирования в общем смысле этого слова. Вы создаете этот язык путем определения домена и разрешенных операций над доменными сущностями. Если вы разрабатываете приложения по философии DDD, у вас в целом получается DSL автоматически (но конечно не такой красивый, как если бы это делать целенаправленно), это такой положительный побочный результат. Вы создаете правила и следуете им, так как они делают решение задачи легче. Вместо кучи строк кода, которые могут менять внутреннее состояние объекта, посылать сообщения, проверять значения и условия – вы пишете один оператор, который имеет значимость для задачи, а весь инфраструктурный код прячется.

DSL должен быть декларативным и, по возможности, «текучим» (fluent interface – как пример, linq). Т.е. цель сообщить ЧТО надо делать, без упоминания КАК.

DSL не зависит от конкретного языка. Можно создавать его на любом языке программирования и конечный результат будет в целом одинаков для любого языка (с поправкой на синтаксис).

Кстати, SQL тоже может рассматриваться как вариант DSL, так как есть вполне определенная предметная область, для которой «язык» создан.

Ладно, но этом вводную часть статьи можно считать законченной. По теории можно шерстить вики и гугл, мы рассмотрим DSL на примере.

## Как DSL может быть применено к TDD

С DSL написание тестов становится реально быстрее и проще! Конечно для этого придется потратить некоторое время, разрабатывая удобный язык, но зато сколько времени и сил экономится в будущем! Для примера, возьмем достаточно простой тест и из второй части стать про DDD и TDD. Помните, там был такой вот тест:

```
 [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));        }
```

Так вот его можно записать в более короткой форме и более понятно. Чтобы он читался как текст, а не как обрывки непонятно чего.

```
[TestMethod]
public void ShouldCalculateStudentsInClassesForYear() {
var student = new Student("", "");

    var school = new School()
                        .WithClassFor(EducationYear._01, c => c
                              .AndStudents(student, student))
                        .WithClassFor(EducationYear._01, c => c
                              .AndStudents(student))
                        .WithClassFor(EducationYear._02, c => c
                              .AndStudents(student));


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

Тест получился короче и, на мой взгляд, читабельнее.  Мы применили fluent  интерфейс, да, то место, где много всего через точку написано. Для того чтобы это заработало, мы будем использовать методы расширения (Extension Methods). Дополнять доменную модель использованными в тесте методами плохая идея, потому что в целом будет много конструкторов, с уже предопределенными значениями класса. Так же много будет перегруженных методов, которые меняют внутренности класса, а это вообще-то работа доменных сервисов в реальном коде.

Итак, как реализовать новые методы, использованные в тесте:

Создаем новый статичный класс в тестовом проекте и называем его SchoolExtension.

```
public static class SchoolExtension { }
```

Первый метод, который мы напишем – будет `WithClassFor`.

Так как мы работаем с методами расширения, то WithClassFor  так же будет статичным, первым параметром пойдет класс, который мы будем расширять + служебное слово this.

```
public static School WithClassFor(this School school, EducationYear year,
Action<Class> action) {
            return school;
        }
```

Последним параметром идет Action, т.к. мы используем лямбда выражения. Внутри этого метода будет создан новый экземпляр Class и добавлен в School. Если требуются дополнительные действия с Class, то они должны быть описаны с помощью лямбда выражения. Получает вот так:

```
public static School WithClassFor(this School school, EducationYear year,
Action<Class> action) {
```

```
            var cls = new Class{EducationYear = year};
            school.AddClass(cls);
            action.Invoke(cls);

            return school;
        }
```

Следующий метод расширения пойдет для класса Class – AddStudents, который принимает массив параметров.

```
public static Class AndStudents (this Class cls, params Student[] students) {
            students.ToList().ForEach(s => cls.AddStudent(s));
            return cls;
        }
```

Правила составления всё те же, а код простецкий. Массив переводим с помощью LINQ в список, и для каждого элемента осуществляем добавление в переменную `cls`. Кстати, это тоже пример fluent интерфейса.

Теперь солюшен должен компилироваться без ошибок. Запускаем тесты и видим что они зеленые!

## Дополнительные улучшения

Можно еще немного улучшить читабельность тестов, оставив только значимую для теста информацию.

```
[TestMethod]
public void ShouldCalculateStudentsInClassesForYear() {
var student = new Student("", "");

    var school = new School()
                        .WithClassFor(EducationYear._01, 2)
                        .WithClassFor(EducationYear._01, 1)
                        .WithClassFor(EducationYear._02, 1);


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

}
```

Можно заметить, что лямбда выражение исчезло и осталось только количество школьников в классе. Это достаточно хорошо, но слишком радикально. Я думаю, что через недели две вы уже забудете, что это за цифры, и потребуется жать ctrl+p чтобы вспомнить что там за параметры, либо лезть в объявление метода. Мы пойдем другим путем.

```
[TestMethod]
        public void ShouldCalculateStudentsInClassesForYear() {
            var school = new School()
                .WithClassFor(c => {
                                    c.EducationYear = EducationYear._01;
                                    c.StudentsInClass = 2;
                                })
                .WithClassFor(c => {
                                    c.EducationYear = EducationYear._01;
                                    c.StudentsInClass = 1;
```

```
                                })
                    .WithClassFor(c => {
                                    c.EducationYear = EducationYear._02;
                                    c.StudentsInClass = 1;
                                });

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

В такой записи уже точно ничего не забудешь, я гарантирую это! =) Тут четко видно, что и как задается. Нет лишней информации, все только по делу: номер класса и количество студентов в классе. Нам не важно, что там за ученики, важно только количество.

Для такого использования метода надо перегрузить метод `WithClassFor` чтобы он принимал Action. Но Action будет параметризованным.

```
public static School WithClassFor(this School school, Action<ClassInit>
action) { … }
```

Правильно мыслите, теперь надо будет создать новый класс `ClassInit`, который будет помогать в инициализации. Когда свойства этого класса будут меняться, надо будет каким-то образом менять значения и в экземпляре Class. Согласно этим соображениям, новый класс должен принимать Class как параметр конструктора. Следуя этой идее и дальше, а так же исходя из использования класса, могу сказать, что в свойствах хватит реализации только Set'а.

ClassInit может выглядеть и так:

```
public class ClassInit {
        private readonly Class cls;
        private readonly Student student = new Student("", "");

        public EducationYear EducationYear {
            set { cls.EducationYear = value; }
        }

        public int StudentsInClass {
            set {
                for (var i = 0; i < value; i++) {
                    cls.AddStudent(student);
                }
            }
        }

        public ClassInit(Class cls) {
            this.cls = cls;
        }
    }
```

Student объявлен как поле класса для сохранения времени объявления и памяти. Так-так, ClassInit появился, но мы ведь работаем с Action<ClassInit>, а не с реальным классом. Как это все организовать в методе `WithClassFor`?

Да почти так же как и до этого!

```
public static School WithClassFor(this School school, Action<ClassInit>
action) {
            var cls = new Class();
            action.Invoke(new ClassInit(cls));
            school.AddClass(cls);

            return school;
        }
```

Похоже на магию. =)

Создаем новый  экземпляр Class и передаем его параметром в ClassInit. **Лямбда выражения не вычисляются до того как результат действительно кто-то попросит**. Так что когда  мы задаем значения полям в тесте ничего не происходит. Ничто никуда не присваивается. Все пойдет по своим местам только по команде Invoke. Можете провести аналогии с пошаговым режимом в играх ;) Думаю, теперь понятно, почему мы не создаем переменную для ClassInit и как заданные значения попадают в реальный экземпляр класса.

Запускаем тест. И он снова зеленый!

Используя описанный подход вы можете сделать свои тесты малыми по размеру и легкочитаемыми. Без всего ненужного информационного шума, что создают все эти new и тд, можно легко увидеть что действительно важно для прохождения теста.

Hard’n’heavy!