Санкт-Петербургский Государственный Университет

Математико-механический факультет

Кафедра системного программирования

Внедрение unit-тестирования в проект на F#

Курсовая работа студента 345 группы

Нишневич Анастасии Юрьевны

Научный руководитель ……. Я.А. Кириленко

Санкт-Петербург

2010

Оглавление

Введение

Unit-тестирование(модульное тестирование) – это метод тестирования, основанный на проверке отдельных модулей кода программы. Тесты обычно пишутся для каждой нетривиальной функции или метода. Это позволяет достаточно быстро проверить, не привело ли очередное изменение кода к появлению ошибок в уже написанных и оттестированных местах программы, а также облегчает обнаружение и устранение таких ошибок.

Основные цели unit-тестирования:

- изолировать отдельные части программы и показать, что по отдельности эти части работоспособны;
- проверить, что написанный код имеет ожидаемое поведение на корректных значениях;
- проверить поведение при неправильных или граничных значениях данных и поставить защиту в таких случаях.

Модульное тестирование стало основой Test-Driven Development(TDD).TDD – процесс разработки программного обеспечения, который предусматривает написание и автоматизацию модульных тестов еще до момента написания соответствующих классов или модулей. Это гарантирует, что все обязанности любого элемента программного обеспечения определяются еще до того, как они будут реализованы.

TDD – распространенная методология. Существует много англоязычных форумов, блогов, статей, посвященных ей. Написано много приложений для unit-тестирования для разных языков и платформ:

- JUnit, TesNG, JavaTESK для Java;
- CUnit , CTESK , cfix для C;
- CPPUnit , Boost Test , Google C++ Testing Framework для C++

и т. д.

Технология .Net активно развивается и вместе с ней развиваются unit-тест frameworku под .Net . Наиболее известные средства тестирования приложений под .Net такие как Nunit, Xunit , MbUnit и встроенный в Visual Studio MsTest поддерживают все .Net языки.

Версия .Net 4.0 включает в себя F#. Более ранние версии требуют интеграции языка в Visual Studio. Таким образом, unit-тестирование применяется и к F# компонентам. Однако,по причине новизны F#, модульное тестирование на нем не было в достаточной мере изучено.

1.Цели и задачи

В рамках данной работы ставится цель изучения возможностей тестирования на F#, а также внедрения системы модульного тестирования в проект YaccConstructor[11]. Основная цель проекта – генерация синтаксического анализатора по входной грамматике. Приложение состоит из front-end'а, преобразователя и генератора. Front-end порождает определенные типы правил, но генератор работает не со всеми из них, поэтому преобразователь раскрывает правила тех типов, с которыми генератор не работает.
Таким образом, основная цель модульного тестирования проверить, что

- Front-end действительно порождает только правильные типы правил;
- преобразователь корректно раскрывает типы правил;
- генератор не принимает ничего лишнего.

2.Framework'и для unit-тестирования на F#

Надо отметить, что изначально было выдвинуто условие тестировать свои приложения прямо с Visual Studio. Но ни Visual Studio 2008, ни Visual Studio 2010 не поддерживают возможность создания тестовых проектов на F#, поэтому использование MsTest или интеграция стороннего framework'а в студию невозможна.

По этой причине в дальнейшем будут рассматриваться Nunit[1], Xunit[2], MbUnit[3].

В большинстве сравнительных статей наибольшее внимание уделяется Nunit и Xunit. MbUnit уступает им по возможностям и имеет меньшее распространение. Код теста с использованием MbUnit от теста с Nunit не отличается.

Приведем пример (Пример 1), проверяющий возведение в квадрат

```
module module1 =
  let inline public Square x = x * x;

  [<TestFixture>]
  type ``Sample Tests`` ()=

    [<Test>] member test.
     ``Test One`` ()=
       Assert.AreEqual (Square 5, 25)
```

Пример 1

Код из примера 1 обработается как MbUnit, так и Nunit.

Xunit самый новый из framework'ов и, как утверждают его авторы, самый современный.
Из существенных отличий его от Nunit'а стоит упомянуть:

1. в Xunit создается новый экземпляр объекта на каждое испытание;

2. в Xunit нет атрибутов тестовых методов [SetUp] и [TearDown] (предварительная настройка/инициализации теста и зачистка его результатов). Их необходимость вопрос спорный. Мнение автора можно узнать по ссылке[4]. Надо отметить, что есть возможность реализовать эти методы;
3. в Xunit нет [ExpectedException]. По причине возможности генерировать ненастоящую ошибку и скрыть реальную ошибку, когда неправильный метод вызова генерирует исключение;
4. в Xunit сокращено число атрибутов. С целью не сильно отклоняться от базовых возможностей языка.

Nunit – самый старый, но при этом активно развивающийся framework. В его последней версии появилось много новых возможностей (с ними можно ознакомиться здесь [5]).
Так же для него есть средство FsUnit[6], которое позволяет писать код теста в функциональном стиле.

С использованием FsUnit код из примера 1 будет выглядеть так (Пример 2):

```
module module1 =
  let inline public Square x = x * x;

  [<TestFixture>]
  type ``Sample Tests`` ()=

    [<Test>] member test.
     ``Test One`` ()=
       Square 5 |> should equal 25
```

Пример 2

В дальнейшем будет рассматриваться Nunit.

3.Реализация

3.1. Подготовительная работа

Для создания тестов необходимо скачать программу для запуска тестов и библиотеки. Все это есть по ссылке [7]

Чтобы использовать Nunit с VisualStudio 2010 нужно проделать следующее:

1. скачать hexeditor (например,[8]);
2. открыть в hexeditor nunit.exe;
3. в открывшейся таблице найти "BSJB" и за ним "v2.0.50727" и заменить на "v4.0.20506";
4. сохранить изменения.

После этого Nunit будет работать корректно.

К тестируемому проекту надо добавить nunit.framework.dll.

Для подключения FsUnit[9]. И добавить к проекту FsUnit.NUnit.dll.

3.2.Создание теста

Напишем тест для функции

```
let run_common path =
   let content = System.IO.File.ReadAllText(path)
   Lexer.currentFileContent := content;
   let reader = new System.IO.StringReader(content) in
   LexBuffer<_>.FromTextReader reader
```

Тест будет выглядеть так:

```
let fpars = ParseFile @"..\..\..\..\Tests\test002.yrd"
[<Test>] member test.``Run_Common test`` () =
let run_common@"..\..\..\..\Tests\test002.yrd"
Assert.AreEqual(rcom.ToString(), "Microsoft.FSharp.Text.Lexing.LexBuffer`1[System.Char]")
```

С использованием Nunit получится так

```
let fpars = ParseFile @"..\..\..\..\Tests\test002.yrd"
[<Test>] member test.``Run_Common test`` () =
let run_common@"..\..\..\..\Tests\test002.yrd"
```

```
rcom.ToString()|> should equal "Microsoft.FSharp.Text.Lexing.LexBuffer`1[System.Char]")
```

Сделаем тестовый модуль компилируемым и соберем проект.

Теперь можно приступать к тестам через Nunit. Откроем программу, загрузим проект и запустим тест.

Результат выполнения теста (положительный или отрицательный) можно увидеть во вкладке «Errors and Failures».

Стоит отметить, что выполнение одного теста не гарантирует корректности операции в целом. А проводить тест на сотне примеров – задача, отнимающая много времени. Решение данной проблемы – библиотека FsCheck.

3.3.Библиотека FsCheck

FsCheck пришла в F# из Haskell (там она называется QuickCheck). Подробнее про FsCheck можно узнать по ссылке[10]. Для использования FsCheck необходимо загрузить ее с сайта(ссылка выше) и добавить в проект FsCheck.dll.

Для тестирования создается свойство, проверяющее ту или иную функциональность. FsCheck проверит это свойство для ста рандомных примеров, которые сгенерируются автоматически.

Ниже приведена функция, которая сопоставляет идентификаторам токенов численный индекс:

```
let tagOfToken (t:token) =
  match t with
  | PATTERN _ -> 0
  | PARAM _ -> 1
  | PREDICATE _ -> 2
  | ACTION _ -> 3
  | STRING _ -> 4
  | LIDENT _ -> 5
  | UIDENT _ -> 6
  | COMMUT  -> 7
  | DLESS  -> 8
  | DGREAT  -> 9
  | RPAREN  -> 10
  | LPAREN  -> 11
  | QUESTION  -> 12
  | PLUS  -> 13
```

```
| STAR  -> 14
| BAR  -> 15
| EQUAL  -> 16
| SEMICOLON  -> 17
| COLON  -> 18
| EOF  -> 19
```

Создадим тест, который будет гарантировать, что при любых изменениях кода численные индексы будут оставаться в промежутке от 0 до 19.

```
[<Test>] member test.``tagOfToken test`` () =
        let tagOfTokenProp (t:token)=  (tagOfToken t >= 0) && (tagOfToken t <= 19)
        quickCheck tagOfTokenProp
```

В Nunit тест с FsCheck окажется в общем списке. Стоит отметить, что во вкладке «Errors and Failures» такой тест всегда будет считаться пройденным, а его результат можно увидеть во вкладке «Text OutPut». Если тест пройден верно результат будет таким:

```
***** Program+module1+Frontend Tests.tagOfToken test
Ok, passed 100 tests.
```

Если в функции изменить

```
| EOF  -> 19
```

на

```
| EOF  -> 20
```

То в тесте получим

```
***** Program+module1+Frontend Tests.tagOfToken test
Falsifiable, after 76 tests (0 shrinks) (StdGen (1777344540,295276107)):
EOF
```

Надо отметить, что FsCheck подставляет в созданные нами свойства значения любой сложности, что играет важную роль в создание тестов.

3.4.Тестирование преобразователя

Для рассматриваемого проекта правильность выходных данных играет большую роль.

Как уже упоминалось выше, важно убедится, что преобразователь раскрывает определенные типы правил.

Библиотека FsCheck оказалась полезной для ее проверки.

Например, необходимо было убедиться, что правила(результаты функции convertToMeta) не принимает значений типов POpt, PSome, PMany. Для проверки этого достаточно создать функцию, которая проверяет тип полученного правила:

```
let ruleIsAfterEBNF (t: Rule.t<Source.t,Source.t>)  =
    match t.body with
    | POpt _ | PSome _ | PMany _   -> false
    | _ -> true
```

Надо отметить, что convertToMeta преобразует одно правило в список, поэтому проверим все элементы списка:

```
let ExpandEBNFTest(t : Rule.t<Source.t,Source.t> ) =  convertToMeta t |> List.forall (fun x -> ruleIsAfterEBNF x)
```

Так же есть возможность создавать условные свойства. Например

```
let isOk t = convertToMeta t |> List.forall (fun x -> ruleIsAfterEBNF x)

let correctForAlExp (t: Rule.t<Source.t,Source.t>)  =
  match t.body with
  | POpt _ | PSome _ | PMany _   -> false
  | _ -> true
```

```
let ExpandAlterProp(t : Rule.t<Source.t,Source.t> ) = (correctForAlExp t.body) ==> lazy ( isOk t)
```

В данном случае, если t.body не будет удовлетворять correctForAlExp, проверка isOk t не будет возможна, так как будет вылетать Exception. Однако, создав условное свойство, мы избегаем получения Exception. Второе условие будет начинать проверяться, только после получения положительно результата в первом. Тест будет считаться пройденным, если для ста t, прошедших первое условие, выполнилось второе.

Более того, необходимо проверять, что все другие случаи обрабатываются с Exception. Для этого создается тест вида:

```
let excProp(i:int) = ((i<0) || (i>20))==> throws<System.Exception,_> (lazy (raise <| System.Exception("tokenTagToTokenId: bad token")))
```

Есть возможность объединять свойства и тестировать их вместе.

```
type ListProperties =
  static member frs xs = excProp i
  static member scn xs = ExpandAlterProp t
```

функция для запуска в этом случае будет немного иной:

quickCheckAll typeof<ListProperties>

3.5.Сложности тестирования

Важным аспектом unit – тестирования является создание mock объектов.

Именно за счет этих объектов модульное тестирование в общем случае ведет к необходимому совершенствованию структуры кода. Ниже дается разъяснение этого высказывания.

Некоторые классы взаимодействуют таким образом, что тестирование отдельного класса распространяется на связанные с ним. Например, класс пользуется базой данных. Тест обращаться к базе не может, так как не должен выходить за границу класса. В результате создается mock объект заменяющий обращение к базе, а связь между элементами системы минимизируется.

Mock объекты – это автоматически генерируемые заглушки, которые позволяют имитировать некоторую функциональность и управлять ею прямо из теста. Моки создаются на базе какого-то интерфейса или класса. В этом случае мок поддерживает все методы эмулируемого интерфейса, плюс методы по настройке поведения мока.

Мнимые объекты или моки (Mock Objects) используются:

- для изоляции тестируемого кода;
- для тестирования делегирования.

При использовании моков для изоляции они заменяют реальные объекты, используемые в тестируемом коде. При этом в тесте появляется возможность контролировать используемые внутри тестируемого кода (делегируемые) объекты, а именно указывать, какой результат они должны возвращать при вызове тех или иных методов с теми или иными параметрами.

При использовании моков для тестирования делегирования мы можем генерировать ожидания тех или иных вызовов в результате работы тестируемого кода. Если эти вызовы не будут совершены, моки сгенерируют ошибки.

Добавим в тестовый класс mock объект, для этого в проект надо добавить nunit.mocks.

Например создадим mock объект, которые будет при вызове функции tagOfToken подставлять на ее место 8.Выглядеть это будет так:

```
open NUnit.Mocks

 [<TestFixture>]
   type ``Sample Tests`` ()=
      val mutable mock: DynamicMock
```

```
val mutable f: ILexer

[<SetUp>]member x.Setup() =
    x.mock <- new DynamicMock ()
    x.f <- x.mock.NewMock<ILexer>()



[<Test>] member test.``Test mock`` () =
   x.mock.ExpectAndReturn ("tagOfToken", 8)
```

Теперь при любом вызове tagOfToken вместо ее результата подставится 8.

Но тут есть довольно серьезная проблема: tagOfToken должна быть методом класса, в конструктор которого нужно передать обновленный интерфейс (т.е. интерфейс, который использует mock объект). Подобная структура свойственна объектно-ориентированным языкам, но в F# таких сложностей хотелось бы избежать.

Иногда это невозможно, но в общем случае необходимости использовать в F# тестах mock объекты нет.

Дело в том, что F# функциональный язык и входные данные функций в нем это передаваемые в нее аргументы. Если есть необходимость передавать в функцию какие-то свои данные, это можно сделать без использования mock объектов.

Результаты и выводы

В процессе работы были:

- проведен обзор инструментов модульного тестирования в .Net;
- подробно изучен и использован инструмент модульного тестирования Nunit;
- изучены и применены инструменты Nunit и FsCheck;
- реализована система модульного тестирования в проекте YaccConstructor, в результате было найдено некоторое количество ошибок и неточностей в реализации функций проекта, и появилась гарантия их отсутствия в дальнейшем.

Проделанная работа еще раз доказала пользу модульного тестирования в целом.

Однако, язык F# еще не получил широкого распространения, что влечет серьезные трудности написания тестов на нем:

- невозможность тестирования непосредственно из Visual Studio;
- небольшое количество доступной информации.

Эти минусы незначительны, но важны при коммерческой разработке, где важную роль играет время. Написание теста на F# становится трудоемкой задачей из-за сложностей с поиском информации. И в ситуации с ограниченным временем проще использовать тесты на C#.

С другой стороны, каждый, кто использовал F#, при написании программ оценил его удобство. Таким образом, тема unit-тестов на F# достаточно актуальна.

И, безусловно, тестирования на F# имеет существенные достоинства:

- при написании тестов сохраняется функциональный стиль (сама структура тестов значительно упрощается);
- с помощью FsUnit тесты становятся читаемыми (такие тесты могут заменить документацию);
- нет необходимости использовать mock объекты (сама структура программы не требует их использования).

При дальнейшем развитии F# модульное тестирование на этом языке несомненно станет удобным средством. Пока же это скорее объект для изучения.

Список литературы

[1]Nunit//Сайт разработчиков Nunit URL: http://nunit.com (дата обращения: 26.05.2010)

[2] Xunit.Net//Сайт разработчиков Xunit URL: http://xunit.codeplex.com (дата обращения: 26.05.2010)

[3] Mbunit generative unit test framework//Сайт разработчиков Mbunit URL: http://www.mbunit.com (дата обращения: 26.05.2010)

[4] Why you should not use SetUp and TearDown in Nunit[Электронный ресурс] Блог James Newkirk URL: http://jamesnewkirk.typepad.com/posts/2007/09/why-you-should-.html (дата обращения: 26.05.2010)

[5] NUnit 2.5: что нового? [Электронный ресурс] gotdotnet.ru : Блог Sergey Popov URL: http://www.gotdotnet.ru/blogs/sergeypopov/4936/ (дата обращения: 26.05.2010)

[6] FsUnit//Сайт разработчиков FsUnit URL: http://fsunit.codeplex.com/ (дата обращения: 26.05.2010)

[7] Nunit downloads [электронный ресурс] Сайт разработчиков Nunit URL: http://www.nunit.org/index.php?p=download (дата обращения: 26.05.2010)

[8] Freeware Hex Editor XVI32 Version 2.51[электронный ресурс] Home page of Christian Maas URL: http://www.chmaas.handshake.de/delphi/freeware/xvi32/xvi32.htm (дата обращения: 26.05.2010)

[9] Fsunit downloads [электронный ресурс] Сайт разработчиков FsUnit URL:http://fsunit.codeplex.com/releases/view/42621 (дата обращения: 26.05.2010)

[10] FsCheck//Сайт разработчиков FsCheck URL: http://fscheck.codeplex.com/ (дата обращения: 26.05.2010)

[11] Recursive-ascent parser generator for F#// google code YaccConstructor
URL: http://code.google.com/p/recursive-ascent/ (дата обращения: 26.05.2010)

[12] Functional Programming Unit Testing – Part 2 /QuickCheck in F#?[Электронный ресурс]Блог Matthew Podwysocki на codebetter.com URL: http://weblogs.asp.net/podwysocki/archive/2008/12/11/functional-programming-unit-testing-part-2.aspx (дата обращения: 26.05.2010)