УДК 681.3.06

МЕТОДИКА ТЕСТИРОВАНИЯ ДИНАМИЧЕСКИХ БИБЛИОТЕК

В. А. Голуб, П. С. Лысачев, Т. М. Потапова, В. А. Москальцов

Воронежский государственный университет

Данная статья посвящена вопросам тестирования динамических библиотек. Особенностью предложенной идеи является комбинация подходов «черного» и «белого» ящиков — теоретических основ тестирования. Практическая реализация метода проведена на примере Microsoft Visual Studio и открытой среды модульного тестирования для языков платформы .NET – NUnit.

ВВЕДЕНИЕ

Разработка эффективных методик тестирования программных продуктов является важной актуальной задачей, решение которой необходимо для оценки работоспособности программ и определения их соответствия предъявляемым требованиям. В соответствие с IEEE Std 829-1983, **тестирование (testing)** — это процесс анализа ПО, направленный на выявление отличий между его реально существующими и требуемыми свойствами (такое отличие мы будем называть дефектом) и на оценку свойств ПО.

1. ТЕОРЕТИЧЕСКИЕ ОСНОВЫ ТЕСТИРОВАНИЯ

Методология тестирования в значительной степени определяется подходом, используемым разработчиками программируемой системы. В настоящее время среди разработчиков ПО широкое распространение получила итеративная модель жизненного цикла программы **RUP — Rational Unified Process** (рис. 1). При использовании этой модели тестирование перестает быть обособленным процессом, который запускается уже после того, как написан весь необходимый программный код. При тестировании для каждой итерации определяется цель тестирования и методы ее достижения. В конце каждой итерации определяется, насколько эта цель достигнута, нужны ли дополнительные испытания, и не нужно ли еще изменить принципы и инструменты проведения тестов.

Методологии тестирования могут базироваться на следующих теоретических подходах.

White-box testing (тестирование «белого ящика»). При таком подходе для конструирования тестов используются внутренняя структура кода и управляющая логика. Недостатком этого метода тестирования является то, что существует вероятность, что код будет проверяться так же, как он был написан, а это не гарантирует корректность логики.

Black-box testing (тестирование **«черного ящика»).** Для конструирования тестов используются требования и спецификации ПО. К недостаткам такого подхода следует отнести во-первых, невозможность выявления взаимно уничтожающихся ошибок, во-вторых, трудность нахождения редко возникающих ошибок, (как правило, ошибок работы с памятью), а следовательно, трудность локализации таких ошибок и отладки программы.

На практике применяется несколько видов тестирования, которые используются совместно на разных этапах разработки продукта или же, напротив, используется только один из них, тот, в пользу использования которого больше аргументов в данном конкретном случае. В большинстве случаев практика тестирования предполагает использование либо **модульного тестирования, либо функционального тестирования, либо их комбинация.**

Модульное тестирование (**unit-тестирование**) подразумевает тестирование отдельных модулей приложения, осуществляемое одновременно с разработкой программного кода.

Функциональное тестирование путем тестирования правильности навигации по объекту, а также ввода, обработки, вывода данных и проверки ряда других функций позволяет убедиться в надлежащем функционировании объекта тестирования в целом.

В случае, когда разрабатывается динамическая библиотека, каждый класс, метод и свойство которой должны соответствовать необходимым требованиям (спецификациям), но нет графического пользовательского интерфейса (GUI),

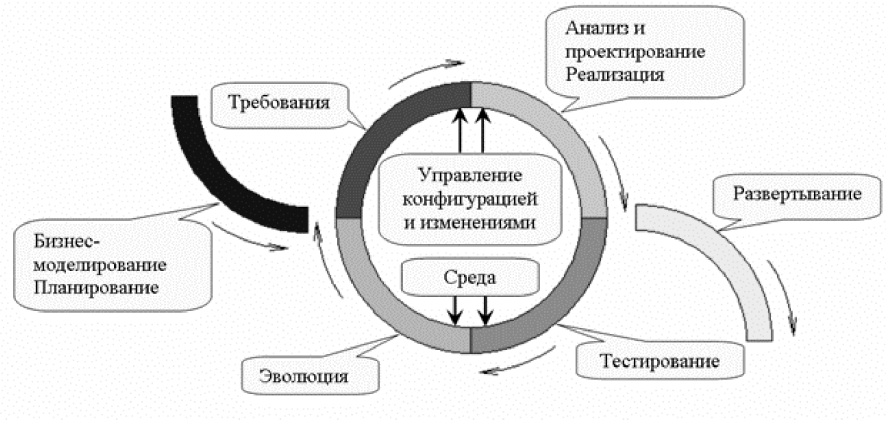

Рис. 1. Жизненный цикл продукта по RUP

функциональное тестирование представляется достаточно проблематичным.

Описанные выше методы тестирования по отдельности не очень эффективны, поэтому необходимо разработать такую методику тестирования, которая предполагала бы использование достоинств каждого из методов и при этом допускала бы достаточно простую практическую реализацию. Иными словами следует разработать методику создания тестовой библиотеки, проверяющей соответствия требованиям и, одновременно тестирующей «критические места» самого кода.

В данной работе предлагается метод тестирования динамических библиотек программных интерфейсов (драйверов, провайдеров данных, прикладных утилит и т.п.), которые не имеют пользовательского интерфейса, отличающийся тем, что совмещает идеи «белого» и «черного» ящиков, а также представляет собой комбинацию соответствующих практических методов тестирования — модульного и функционального.

Практическое применение предлагаемой методики осущесвлялось на примере тестирования динамической библиотеки провайдера данных для ADO.NET 2.0 (3.0) для СУБД ЛИНТЕР — System.Data.LinterClient.dll с помощью открытой среды тестирования NUnit.

2. ТЕХНИЧЕСКАЯ РЕАЛИЗАЦИЯ МЕТОДИКИ ТЕСТИРОВАНИЯ

Предлагаемая методика предполагает поэтапное проведение тестирования.

1 Этап. Параллельно с проектированием и разработкой программного кода проводится модульное тестирование, которое заключается в изолированной проверке каждого отдельного элемента путем запуска тестов в искусственной среде. Оценивая каждый элемент изолированно, и подтверждая корректность его работы, точно установить проблему казывается значительно проще, чем если бы элемент был частью системы. Отталкиваясь от структуры кода (методы, функции, классы, пакеты), фактически при этом реализуется идея «белого ящика».

2 Этап. Для локализации выявленных ошибок пишутся дополнительные тесты. Обычно на этом этапе удается найти еще и скрытые ошибки. Кроме того, можно существенно сократить время, затрачиваемое на отладку программы, так как использование дополнительных тестов является более эффективным, чем продолжительная и трудоемкая отладка критической области кода.

Дополнительные тесты, используемые на втором этапе и предназначенные для локализации ошибок, должны базироваться на следующих принципах.

1. Простота. Из модульного теста необходимо убрать все лишнее так, чтобы ошибка тем не менее была выявлена.

2. Принцип «приращения аргументов». Под приращением будем понимать изменение параметров

- по величине, если параметры — числа;
- по типу, если допустимы параметры различных типов;
- по размеру;
- по кодировке, если параметры строкового типа.

Отдельно необходимо рассмотреть пограничные значения параметров, как то: null, максимально большое число данного типа, пустая строка и т.д.

3. Компромисс «тестер — разработчик», предполагающий тесное взаимодействие разработчика программы и тестера. Кодировать модульные тесты проще всего программисту, который пишет исходный код, но если программист не обладает навыками дизайнера сценариев, то тестирование выполняется тестером, который, в свою очередь, должен хорошо разбираться в испытуемом коде.

3 Этап. После того, как выявленные ошибки локализованы и отлажены, модульные тесты прогоняются еще раз. И только после того, как отдельные вызовы методов не вызывают устойчивых ошибок, можно переходить к следующему этапу. (Под устойчивыми ошибками понимаются все те, которые возникают стабильно каждый раз при вызове метода. Под неустойчивыми — проблемы с памятью. Отлаживать такие ошибки — весьма трудоемкая задача, поэтому часто их оставляют на последний этап отладки.)

4 Этап. Поскольку до этого момента тестирование проводилось с учетом внутренней структуры кода, то, фактически, не был затронут подход «черного ящика», а как отмечалось ранее, при оптимальном тестировании нужно стремиться к тому, чтобы тесты не только проверяли корректность структуры кода, но и его соответствие наложенным системным требованиям. Поэтому на данном этапе необходимо уделить внимание тестированию нескольких взаимосвязанных методов и классов в соответствии с логикой использования программного интерфейса в целом.

В качестве примера практического использования предлагаемой методики тестирования рассмотрим задача тестирования и отладки создаваемого программного интерфейса ADO.NET 2.0 (3.0) провайдера данных для СУБД ЛИНТЕР — System.Data.LinterClient.dll с целью проверки его соответствия спецификациям ADO.NET 2.0(3.0).

Исходя из вышеописанных соображений, методика тестирования программного интерфейса ADO.NET 2.0 (3.0) предполагала тестирование каждого модуля провайдера с помощью прямого вызова основных функций динамической библиотеки, а также возможных прогнозируемых комбинаций этих функций.

Поскольку язык реализации исходных текстов библиотеки провайдера — C#, то для мо-

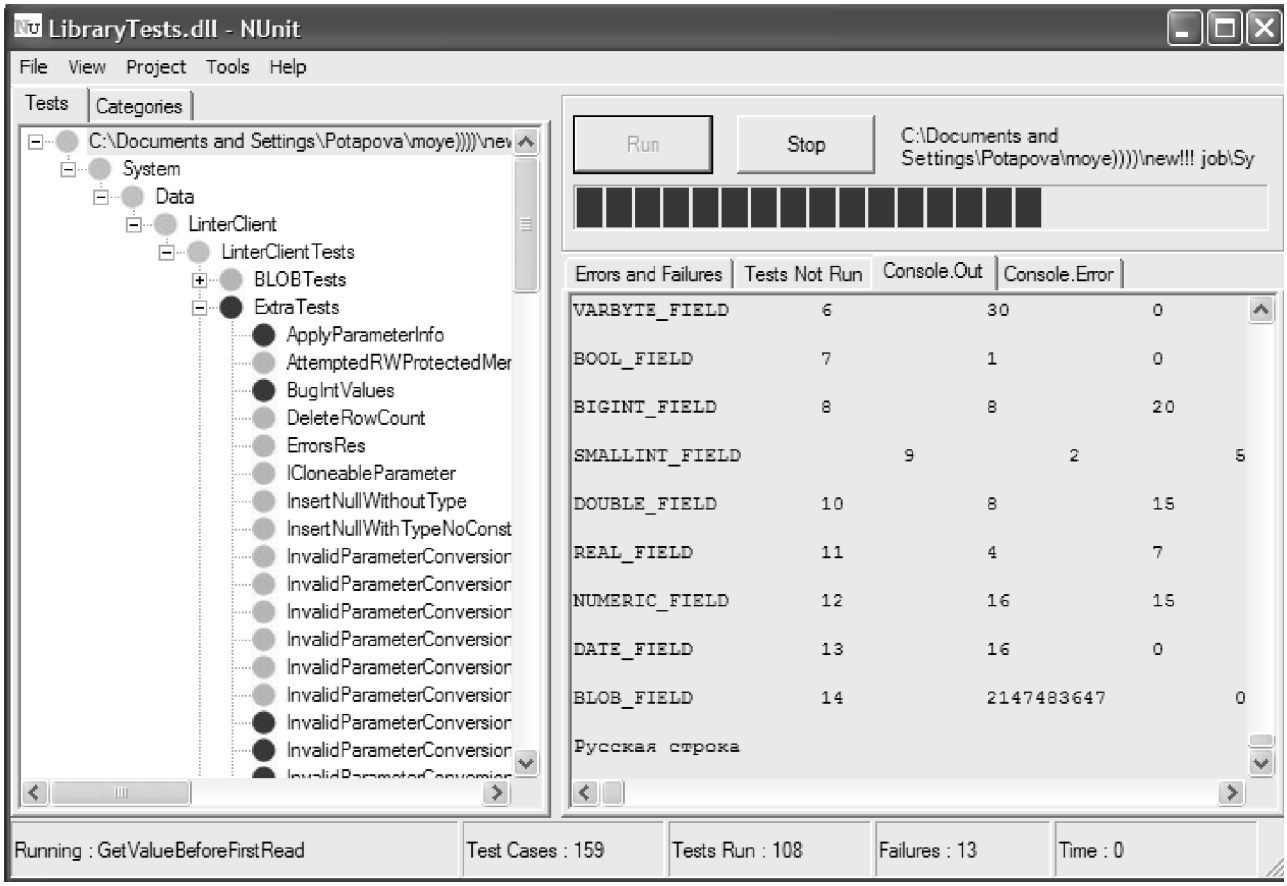

Рис. 2. NUnit в режиме запуска тестовой библиотеки

дульного тестирования была выбрана открытая среда NUnit [4] юнит-тестирования приложений для .NET. Преимуществом данной среды тестирования является то, что возможно запускать тесты выборочно или связанно. На рис. 2 показано окно NUnit в рабочем режиме.

Для тестирования разрабатываемого кода были созданы тесты в соответствии с каждым из вышеприведенных этапов: сначала тесты, вызывающие по отдельности функции, затем — для отладки программы — максимально упрощенные (в соответствии с первым принципом) тесты с измененными параметрами для локализации ошибок (в соответствии со вторым принципом). Существенным моментом является то, что локализация ошибки — это процесс анализа разных тестовых случаев, как пройденных успешно, так и тех, которые вызвали ошибку. В соответствии с третьим принципом эту довольно сложную задачу более целесообразно поручить разработчику кода, нежели тестеру.

Рассмотрим более подробно, как в этом примере реализуется библиотека тестов.

BaseTests — базовый класс для всех модулей тестовой библиотеки. Содержит методы SetUp и TearDown — реализующие подготовку к тестированию (открытие соединения, создание тестовых таблиц, заполнение их данными) и завершение тестирования (закрытие соединения, освобождение памяти) в среде NUnit. Это предварительны этап создания тестовой библиотеки (0-й этап).

На первом этапе были написаны модули тестовой библиотеки LinterClientTests, проверяющие все доступные пользователю методы, вызывая их по очереди. Так, например, был создан тестирующий модуль LDataReaderTest для класса LinterDbdataReader, модуль LAdapterTest для класса LinterDbDataAdapter и так далее для каждого из всех классов тестируемой библиотеки провайдера.

Рассмотрим более подробно тестирующий модуль LAdapterTest, который содержит тесты для класса LinterDbDataAdapter. Следующий пример иллюстрирует NUnit-тест для метода Fill(), который заполняет данными выборки объект типа DataSet:

```
[Test]
        public void Fill()
        {
         LinterDbTransaction transaction = this.Connection.BeginTransaction();
         LinterDbCommand command = new LinterDbCommand("SELECT * FROM TEST", Connection,
            transaction);
            LinterDbDataAdapter adapter = new LinterDbDataAdapter(command);
            adapter.MissingSchemaAction = MissingSchemaAction.AddWithKey;
            LinterDbCommandBuilder builder = new LinterDbCommandBuilder(adapter);
            DataSet ds = new DataSet();
            adapter.Fill(ds, "TEST");
            Console.WriteLine();
            Console.WriteLine("DataAdapter - Fill Method - Test");
            foreach (DataTable table in ds.Tables)
            {
                foreach (DataColumn col in table.Columns)
                {
                    Console.Write(col.ColumnName + "\t\t");
                }
                Console.WriteLine();
                foreach (DataRow row in table.Rows)
                {
                    for (int i = 0; i < table.Columns.Count; i++)
                    {
                        Console.Write(row[i] + "\t\t");
                    }
                    Console.WriteLine("");
                }
            }
            adapter.Dispose();
            builder.Dispose();
            command.Dispose();
            transaction.Commit();
        }
```

Наряду с этим тестом, выполняется тест корректности работы метода Insert(), который модифицирует выборку на отсоединенной стороне (в объекте DataSet), добавляя строку данных:

```
[Test]
        public void Insert()
        {
            LinterDbCommand command = new LinterDbCommand("SELECT * FROM TEST",
            Connection);
            LinterDbDataAdapter adapter = new LinterDbDataAdapter(command);
            adapter.MissingSchemaAction = MissingSchemaAction.AddWithKey;
            LinterDbCommandBuilder builder = new LinterDbCommandBuilder(adapter);
            DataSet ds = new DataSet();
            adapter.Fill(ds, "TEST");
            DataRow newRow = ds.Tables["TEST"].NewRow();
            newRow["INT_FIELD"] = 110010001;
            newRow["CHAR_FIELD"] = "ONE THOUSAND";
            newRow["VARCHAR_FIELD"] = "ONE THOUSAND PLUS SMTH";
            newRow["BIGINT_FIELD"] = 100000;
            newRow["SMALLINT_FIELD"] = 100;
            newRow["DOUBLE_FIELD"] = 100.01;
            newRow["NUMERIC_FIELD"] = 100.01;
            newRow["DATE_FIELD"] = new DateTime(100, 10, 10);
            ds.Tables["TEST"].Rows.Add(newRow);
            adapter.Update(ds, "TEST");
            adapter.Dispose();
            builder.Dispose();
            command.Dispose();
        }
```

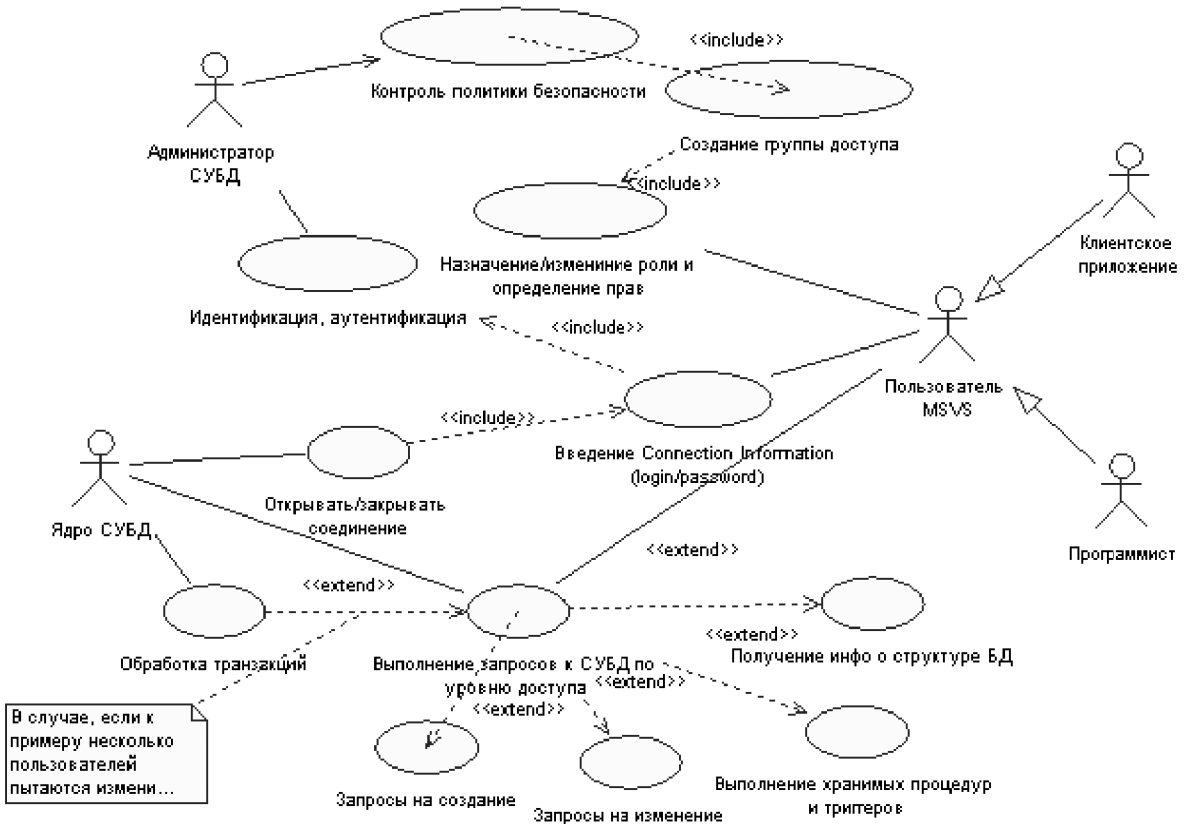

Рис. 3. UML-диаграмма вариантов использования программного продукта

В нашем случае, тест Insert() выявил ошибку. На втором этапе тестирования локализация выявленной ошибки осуществлялась с помощью дополнительных тестов.

После исправления всех найденных ошибок, кроме ошибок, связанных с выделением памяти, встал вопрос о функциональном тестировании (этап номер три) — то есть тестировании на соответствие спецификации. Ошибки выделения памяти, к этому моменту не были локализованы, в силу чего их исправление было отложено до окончательного решения всех других проблем.

На рис. 3 приведена UML-диаграмма вариантов использования программного продукта. На основе этой диаграммы можно написать тесты, реализующие подход «черного ящика» тестирования библиотеки (четвертый этап). Так были разработаны тесты, проверяющие корректность аутентификации и идентификации пользователя, открытия и закрытия соединения, работы с транзакциями, получения данных о структуре базы данных и т.д. Отдельно тестировался вариант использования «Выполнение хранимых процедур и триггеров».

ЗАКЛЮЧЕНИЕ

Предложенный в данной работе метод тестирования при практической реализации показал существенно более высокую эффективность по сравнению с известными методами, определяемую особой тщательностью тестирования программного продукта а также значительным сокращением сроков тестирования.

Полученные результаты позволяют рекомендовать предлагаемую методику для тестирования различных программных интерфейсов.

ЛИТЕРАТУРА

1. Он-лайн документация на СУБД ЛИНТЕР — (http://www.linter.ru/ru/main).

2. *Троелсен Э.* Язык программирования C# 2005 и платформа .NET 2.0 / Э. Троелсен; перевод с англ.: — М. Вильямс. 2007, 1167 с.

3. *Сеппа Д.* Программирование на Microsoft ADO. NET 2.0 / Д. Сеппа. — СПб.: Питер. 2007, 784 с.

4. *Малик С.* ADO.NET 2.0 для профессионалов / С. Малик; перевод с англ.: - М. Вильямс. 2006, 560 с.

5. Свободная энциклопедия Википедия - NUnit (http://ru.wikipedia.org/wiki/Nunit)

Статья принята к опубликованию 25 октября 2007 г.