

УДК 621.398  
К 931

Утверждено учебным управлением МЭИ (ТУ)  
в качестве учебного пособия для студентов

Подготовлено на кафедре прикладной математики

Рецензенты: докт. техн. наук, профессор И.Б. Фоминых,  
докт. техн. наук, профессор А.П. Еремеев

Куриленко И.Е.  
К 931

Модульное тестирование: учебное пособие / И.Е. Куриленко,  
П.Р. Варшавский. — М.: Издательский дом МЭИ, 2011. — 48 с.

ISBN 978-5-383-00632-0 **195-00**

Рассматривается концепция модульного тестирования программного обеспечения и ее реализация для популярных языков программирования – C++, C#, Java. Анализируются преимущества и недостатки модульного тестирования. Рассматривается методика разработки программного обеспечения посредством тестирования.

Пособие предназначено для студентов, обучающихся по специальностям «Прикладная математика и информатика» (010501), «Информационные системы и технологии» (230201) направлений подготовки «Прикладная математика и информатика» (010500), «Информационные системы» (230200), а также по направлению «Информатика и вычислительная техника» (230100) и изучающих курсы «Технологии разработки программных средств», «CASE-технологии разработки программных средств».

Пособие будет полезно аспирантам, научным сотрудникам и специалистам, занимающимся вопросами разработки программных систем.

#### Учебное издание

Куриленко Иван Евгеньевич, Варшавский Павел Романович

#### МОДУЛЬНОЕ ТЕСТИРОВАНИЕ

Учебное пособие по курсам

«CASE-технологии разработки программных средств», «Технологии разработки программных средств» для студентов, обучающихся по направлениям  
«Прикладная математика и информатика»,  
«Информатика и вычислительная техника», «Информационные системы»

Редактор издательства Л.В. Егорова

Темпплан издания МЭИ 2010 г.

Печать офсетная

Тираж 100 экз.

Подписано в печать 20.04.11

Физ. печ. л. 3,0

Заказ № 1471

ЗАО «Издательский дом МЭИ», 111250, Москва, Красноказарменная, д. 14  
Отпечатано в типографии ФКП «НИИ «Геодезия», 141292, Московская обл.,  
г. Красноармейск, просп. Испытателей, д. 14

ISBN 978-5-383-00632-0

© Московский энергетический институт  
(технический университет), 2011

## ВВЕДЕНИЕ

Тестирование — один из важнейших этапов контроля качества разработанного программного обеспечения (ПО). Существует много видов тестирования, которые могут и должны выполняться при разработке ПО. Некоторые из этих видов тестирования требуют широкого вовлечения в процесс конечных пользователей, другие формы требуют привлечения высокооплачиваемых специалистов и иных ресурсов. Ручное тестирование является затратным по времени, трудоемким и часто монотонным процессом. Оно приводит к возникновению проблем, особенно при ограниченных ресурсах и жестких сроках. Для ручного тестирования необходимо разработать программу тестов, подобрать специалистов по качеству ПО, обучить их выполнять необходимую работу, да еще и повторять ее неоднократно до достижения необходимого результата. Если нужно улучшить тестирование приложений для проверки корректности их работы, важно двигаться в сторону автоматизации всех ручных задач тестирования.

Автоматическое тестирование использует тестовые программы для проверки ПО, что помогает упростить процесс тестирования и сократить требуемое на него время. Современные средства разработки позволяют быстро создавать довольно сложные приложения, и их ручное тестирование является очень трудоёмким процессом. Кроме того, появление CASE-средств, позволяющих значительно увеличить производительность труда программистов, сказалось на уровне требований к специалистам по качеству ПО, которые в современных условиях должны обрабатывать большое количество информации за короткий срок.

Недостаток ручного тестирования состоит в том, что результаты выполнения тестов не сохраняются и тесты трудно повторить заново. Присутствует человеческий фактор: человек может устать и не выполнить все необходимые действия, ограничившись частью. Кроме того, специалист в процессе тестирования постепенно обучается типичному (логичному, правильному, ожидаемому) сценарию тестирования и впоследствии повторяет именно его, что уменьшает вероятность обнаружения проблем, возникающих в нестандартных режимах работы программы (возможных, например, при работе начинающего неопытного пользователя). Автоматические тесты позволяют упростить процесс ручного тестирования, сделать его более удобным, точным и повторяемым. Использование автоматических тестов позволяет экономить средства, время специалистов, уменьшает долю рутинного

труда и способствует увеличению границ области охвата тестами и качества тестирования.

Самая первая автоматизация строилась на основе передачи тестовых параметров приложениям через командную строку и анализе получаемых результатов. Однако впоследствии, параллельно с усложнением интерфейса приложений и широким внедрением распределенных технологий, появились более сложные методы тестирования, а затем стали разрабатываться и внедряться соответствующие средства автоматизации. В современных условиях инструментальные средства автоматизации могут записывать последовательность действий при работе пользователей с приложением, а сформированные на этой основе сценарии используются для последующих тестов.

С точки зрения использования знаний о внутреннем устройстве и логике работы программы тестирование подразделяется на:

1) тестирование белого ящика — тестирование приложения или модуля с использованием знаний о его реализации; контроль зависимости выходных результатов и внутреннего состояния от входных данных;

2) тестирование чёрного ящика — тестирование приложения или модуля без использования знаний о его реализации; проверка зависимости выходных результатов от входных данных.

В первом варианте тестирование обычно осуществляется на уровне кода, во втором — часто с помощью специальных инструментов осуществляется имитация действий пользователя.

В конечном счете в любом ПО пользователи обнаруживают ошибки или неисправности. Как правило, такие ошибки исправляются, исправленная программа тестируется, и выпускается обновленная версия программы. Однако, к сожалению, любое изменение в программе, даже исправление одной ошибки, может повлиять на систему непредвиденным образом и привести к более серьезным проблемам по сравнению с исходной.

*Регрессивное тестирование* представляет собой метод повторного тестирования измененной системы в целях проверки отсутствия сбоев ранее нормально работавших функций, вызванных исправлением ошибок или добавлением новых функциональных возможностей. Регрессивное тестирование позволяет полностью гарантировать, что ПО работает именно так, как планировалось. Однако выполнить полное регрессивное тестирование вряд ли возможно. Это вызвано ограничениями по времени и по ресурсам, которые имеются у типовых групп разработчиков. По мере роста и расширения исходного кода

программы становится все сложнее тестировать отдельные ее части. Эта проблема усложняется из-за частоты создания сборок программ. В проектах, в которых поддерживаются частые автоматические сборки, практически невозможно выполнить полное регрессивное тестирование. В таких случаях необходимо тестировать предыдущую функциональность, чтобы обеспечить возможность тестирования новых исправлений и новых функций. При этом остается вероятность выпуска версии ПО с необнаруженными ошибками.

В условиях частого изменения исходного кода, характерного для современных программных проектов, помочь в регрессивном тестировании может технология модульного тестирования, которой посвящено данное учебное пособие. Основная идея модульного тестирования — автоматизация тестирования отдельных компонентов программы. Таким образом, применение модульного тестирования позволяет разрабатывать код, покрытый автоматическими тестами, что практически исключает возможность регрессии.

## МОДУЛЬНОЕ ТЕСТИРОВАНИЕ

Модульное тестирование позволяет проверить на корректность отдельные модули исходного кода программы (рис. 1). Этот тип тестирования обычно выполняется программистами в процессе кодирования. Существует даже методика разработки ПО посредством тестирования, когда перед разработкой какой-либо полезной функциональности создается её модульный тест. Идея состоит в том, чтобы писать тесты для каждой сложной функции. В случае если необходимо разработать модульный тест для класса, обычно для каждого его метода пишется тест. Такой тест проверяет все возможные способы поведения класса (в соответствии со спецификацией) и сообщает об обнаруженных ошибках. Это позволяет быстро проверить, не привело ли очередное изменение кода к регрессии, то есть к появлению ошибок в

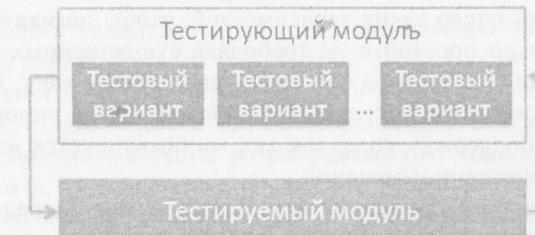


Рис. 1. Пример организации модульного тестирования

уже написанных и протестированных фрагментах кода программы, а также облегчает локализацию и устранение таких ошибок. Цель модульного тестирования — изолировать отдельные части программы и показать, что по отдельности эти части работоспособны.

При внедрении технологии модульного тестирования можно получить следующие преимущества:

- *ускорение и удешевление разработки.* Модульное тестирование позволяет упростить процесс отладки кода. Часто ошибки в низкоуровневом коде приводят к исправлению в высокоуровневом коде. В итоге в процессе исправления не причин, а следствий затрачиваются значительные усилия, усложняется код. Модульное тестирование способствует раннему обнаружению некорректного поведения в низкоуровневых элементах архитектуры ПО;
- *поощрение изменений кода.* Модульное тестирование позволяет проводить рефакторинг, будучи уверенным, что код, покрытый тестами, по-прежнему работает корректно. Это поощряет разработчиков к изменениям кода, поскольку достаточно легко проверить, что код работает и после изменений;
- *упрощение интеграции.* Модульное тестирование помогает устранить сомнения по поводу отдельных модулей и может быть использовано для подхода к тестированию «снизу вверх»: сначала тестируются отдельные части программы, затем программа в целом;
- *документирование кода.* Модульные тесты можно рассматривать как «живой документ» для тестируемого кода. Разработчики, которые не знают, как использовать данный код, могут использовать соответствующий тест в качестве примера;
- *отделение интерфейса от реализации.* Поскольку некоторые классы могут использовать другие классы, тестирование отдельного класса часто распространяется на связанные с ним. Необходимость разработки модульного теста стимулирует уменьшать число таких зависимостей, чтобы логика теста была максимально простой и не требовала существенных усилий по разработке. Это приводит к менее связанному коду, минимизируя зависимости в системе. Что в свою очередь упрощает дальнейшую поддержку кода, так как минимизируется «эффект ряби» при внесении изменений.

В случае использования методики разработки посредством тестирования, также можно получить сокращение времени отладки и сократить длинные циклы тестирования на завершающем этапе проекта.

Модульное тестирование, как правило, может осуществляться на том же языке программирования, на котором ведётся разработка приложения. В настоящее время доступно много библиотек, содержащих реализацию программного окружения для организации модульного тестирования на разных языках программирования. Например, для тестирования программ, разработанных на языке C++, может использоваться библиотека CPPUNIT. Для модульного тестирования программ, разработанных на языке Java, обычно используют JUnit. Для семейства языков, поддерживающих разработку для платформы Microsoft .Net, могут использоваться MSUnit и NUnit. Существуют аналогичные библиотеки и для других языков (Delphi – Dunit, Python – PyUnit и т.д.).

На уровне модульного тестирования проще всего обнаружить дефекты, связанные с алгоритмическими ошибками и ошибками кодирования алгоритмов (ошибки в условных конструкциях, неверная работа с счетчиками циклов, а также неверное использование локальных переменных и ресурсов).

Ошибки, связанные с неверной трактовкой данных, некорректной реализацией интерфейсов, совместимостью, производительностью и т.п., обычно пропускаются на уровне модульного тестирования и выявляются на более поздних стадиях тестирования.

## Контрольные вопросы

1. Что такое модульное тестирование?
2. Какие преимущества дает применение технологии модульного тестирования на практике?
3. Какие ошибки невозможно определить с помощью модульного тестирования?

## МЕТОДИКА РАЗРАБОТКИ ПРОГРАММ ПОСРЕДСТВОМ ТЕСТИРОВАНИЯ

Разработка техники модульного тестирования также привела к появлению новых методик разработки. В частности, среди практик экстремального программирования есть разработка через тестирование (TDD, от англ. test-driven development). В соответствии с этой техникой модульные тесты разрабатываемого кода пишутся до его реализации и, по существу, управляют разработкой. То есть перед реализацией некоторой функциональности разрабатывается код проверки корректности ее работы.

Разработка в стиле TDD состоит из коротких циклов (длительность которых зависит от опыта и стиля работы программиста). Каждый цикл состоит из следующих шагов:

1. Из системы контроля версий исходного кода извлекается код программной системы, находящийся в согласованном состоянии (когда весь набор имеющихся модульных тестов выполняется успешно).
2. Добавляется новый тест. Он может состоять в проверке, реализует ли система некоторое новое поведение или содержит некоторую ошибку, о которой недавно стало известно.
3. Успешно выполняется весь набор тестов, кроме нового теста. Этот шаг необходим для проверки самого теста — включён ли он в общую систему тестирования и правильно ли отражает новое требование к системе, которому она еще не удовлетворяет.
4. Программа изменяется таким образом, чтобы выполнялись все тесты. Нужно реализовать самое простое решение, удовлетворяющее новому тесту, и одновременно с этим не испортить существующие тесты. Большая часть нежелательных побочных и отдалённых эффектов от вносимых в программу изменений отслеживается именно на этом этапе с помощью достаточно полного набора тестов.
5. Проверяется успешно ли выполняется весь набор тестов.
6. Теперь, когда требуемая в этом цикле функциональность достигнута самым простым способом, программа подвергается рефакторингу для улучшения структуры и устранения избыточного, дублированного кода.
7. Контролируется, что весь набор тестов выполняется успешно.
8. Набор изменений, сделанных в этом цикле в тестах и программе, заносится в систему контроля версий, после чего исходный код снова находится в согласованном состоянии и содержит улучшение по сравнению с предыдущим состоянием.

### Контрольные вопросы

1. Сформулируйте основные принципы методики разработки ПО посредством тестирования.
2. Сравните методику разработки посредством тестирования с классическим подходом (тестирование после разработки). Выделите получаемые преимущества и недостатки.

## МОДУЛЬНОЕ ТЕСТИРОВАНИЕ С ПРИМЕНЕНИЕМ CPPUNIT

Рассмотрим автоматизацию тестирования программ, разработанных на языке C++, с помощью открытой библиотеки CPPUNIT, содержащей набор классов для реализации модульных тестов (рис. 2).

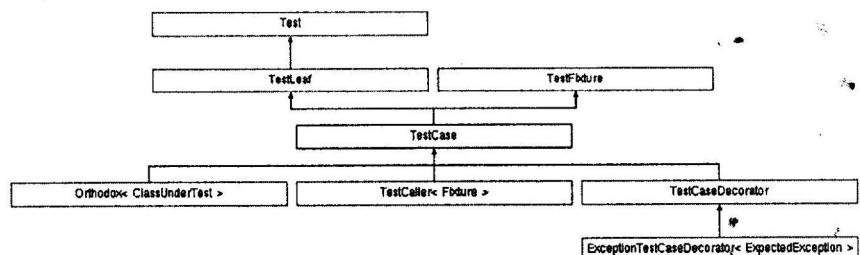


Рис 2. Иерархия классов CPPUNIT

Test является абстрактным базовым классом для всех тестовых классов. Он нужен для того, чтобы можно было полиморфно связывать в иерархии различные тестовые объекты.

TestCase является основным тестовым классом — базовой единицей тестирования. Класс TestCase отражает единичный тестовый вариант. Чтобы его использовать, нужно создать класс-наследник этого класса и переопределить метод runTest(). Однако чаще всего этого не приходится делать, так как для создания объектов типа TestCase используется шаблон TestCaller и методы класса TestFixture. TestFixture содержит методы setUp() и tearDown() для создания и уничтожения тестовой среды и экземпляров объектов, подлежащих тестированию. TestCaller создаёт экземпляры типа TestCase, вызывающие указанную функцию при вызове runTest() и использующие объекты, создающиеся указанным TestFixture.

Класс TestSuite позволяет реализовывать контейнеры множества тестов (рис. 3). TestSuite содержит массив указателей на объекты типа Test, поэтому в него можно помещать как объекты типа TestCase, так и объекты типа TestSuite, тем самым создавая иерархии тестовых случаев любого типа. При этом, при вызове метода run() базового TestSuite-объекта рекурсивно будут выполнены все тесты, находящиеся в иерархии.

TestRunner также является контейнером тестовых объектов (рис. 4). Он управляет всем жиз-

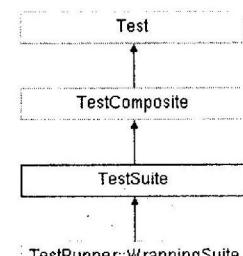


Рис. 3. Класс TestSuite

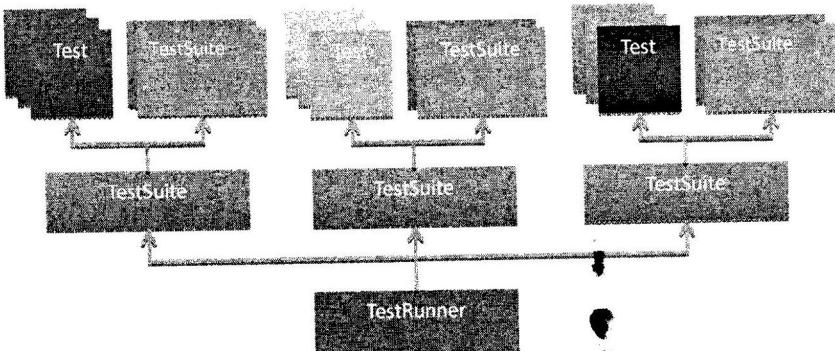


Рис. 4. Класс TestRunner

ненным циклом тестов: от создания тестовых объектов, до их уничтожения и выдачи результатов. TestRunner показывает прогресс (текущее состояние) процесса тестирования и выдает суммарный результат в конце. Возможно выполнение как всех тестов, содержащихся в контейнере типа TestRunner, так и только одного из них.

Рассмотрим пример. Допустим, нужно протестировать класс комплексного числа Complex, чтобы быть уверенным в его работе:

```
class Complex {
    friend bool operator==(const Complex& a, const Complex& b)
    {
        return a.real == b.real && a.imaginary == b.imaginary;
    }
    double real, imaginary;
public:
    Complex(double r, double i = 0)
        : real(r), imaginary(i)
    {
    }
};
```

Для этого нужно создать тестовый класс — наследник класса CppUnit::TestCase и переопределить виртуальную функцию runTest(). Для проверки тестовых условий используется макрос CPPUNIT\_ASSERT(bool). В случае если выражение, заключенное в CPPUNIT\_ASSERT, примет значение false, будет сгенерирована ошибка (т.е. тест не пройдет).

Чтобы проверить оператор равенства для класса комплексных чисел, можно использовать следующий код:

```
class ComplexNumberTest : public CppUnit::TestCase
{
public:
    ComplexNumberTest (const std::string & name) : CppUnit::TestCase (name)
```

```
{
}
void runTest ()
{
    CPPUNIT_ASSERT (Complex (10, 1) == Complex (10, 1));
    CPPUNIT_ASSERT (! (Complex (1, 1) == Complex (2, 2)));
}
}
```

Помимо CPPUNIT\_ASSERT предусмотрены и другие макросы. Например, в приведенном выше примере сравнение комплексных чисел можно было осуществлять с помощью макроса CPPUNIT\_ASSERT\_EQUAL(expected, actual):

```
CPPUNIT_ASSERT_EQUAL (Complex (10, 1), Complex (10, 1));
```

Также предусмотрена специальная версия для сравнения вещественных чисел с необходимым уровнем точности:

```
CPPUNIT_ASSERT_DOUBLES_EQUAL(expected, actual, delta);
```

С помощью макроса CPPUNIT\_ASSERT\_MESSAGE(message, condition) можно также производить проверки, однако в случае, если значение выражения condition будет *false*, будет выведено пользовательское сообщение message. Аналогичные варианты предусмотрены и для макросов CPPUNIT\_ASSERT\_EQUAL и CPPUNIT\_ASSERT\_DOUBLES\_EQUAL — CPPUNIT\_ASSERT\_EQUAL\_MESSAGE(message, expected, actual) и CPPUNIT\_ASSERT\_DOUBLES\_EQUAL\_MESSAGE(message, expected, actual, delta).

С помощью макроса CPPUNIT\_FAIL(message) можно в любой момент прервать выполнение теста с ошибкой, при этом выдается пользовательское диагностическое сообщение message.

Также предусмотрены макросы, позволяющие убедиться, что в процессе теста сгенерировано или не сгенерировано какое-либо исключение. С помощью макроса CPPUNIT\_ASSERT\_THROW(expression, ExceptionType) можно контролировать, генерирует ли выражение expression при тестовом прогоне исключение типа ExceptionType. С помощью макроса CPPUNIT\_ASSERT\_NO\_THROW(expression) можно контролировать, что при вычислении expression никакие исключения не генерировались. Также предусмотрены версии этих макросов с выводом пользовательского сообщения — CPPUNIT\_ASSERT\_THROW\_MESSAGE(message, expression, ExceptionType) и CPPUNIT\_ASSERT\_NO\_THROW\_MESSAGE(message, expression).

Таким образом, с помощью CPPUNIT можно разрабатывать простые тестовые случаи, но на практике чаще применяются контейнеры тестов.

Для запуска нескольких тестов в пакетном режиме в CPPUNIT предусмотрен класс TestSuite. Например, для запуска последовательно двух или более тестов может быть применен следующий код:

```
CppUnit::TestSuite suite;
CppUnit::TestResult result;
suite.addTest(new CppUnit::TestCaller<ComplexNumberTest>(
    "testEquality",
    &ComplexNumberTest::testEquality ) );
suite.addTest(new CppUnit::TestCaller<ComplexNumberTest>(
    "testAddition",
    &ComplexNumberTest::testAddition ) );
suite.run( &result );
```

За счет того что TestSuite может содержать объекты, реализующие интерфейс Test, и управлять ими, в него могут быть включены не только классы-наследники TestCase, но и другие тестовые наборы TestSuite, например:

```
CppUnit::TestSuite suite;
CppUnit::TestResult result;
suite.addTest( ComplexNumberTest::suite() );
suite.addTest( SurrealNumberTest::suite() );
suite.run( &result );
```

CPPUNIT содержит класс TestRunner, используемый для автоматизации запуска многих тестовых наборов (TestSuite) на исполнение и для отображения результатов. Однако его использование предполагает наличие в тестовых классах открытого статического метода suite(). Например, для рассматриваемого выше класса для тестирования комплексного числа такой метод реализуется следующим образом.

```
static CppUnit::Test *suite()
{
    CppUnit::TestSuite *suiteOfTests = new CppUnit::TestSuite(
        "ComplexNumberTest" );
    suiteOfTests->addTest(new CppUnit::TestCaller<ComplexNumberTest>(
        "testEquality",
        &ComplexNumberTest::testEquality ) );
    suiteOfTests->addTest(new CppUnit::TestCaller<ComplexNumberTest>(
        "testAddition",
        &ComplexNumberTest::testAddition ) );
    return suiteOfTests;
}
```

В итоге функция main тестового приложения может быть организована следующим образом:

```
#include <cppunit/ui/text/TestRunner.h>
#include "ExampleTestCase.h"
#include "ComplexNumberTest.h"

int main( int argc, char **argv )
{
    CppUnit::TextUi::TestRunner runner;
    runner.addTest( ExampleTestCase::suite() );
    runner.addTest( ComplexNumberTest::suite() );
    runner.run();
    return 0;
}
```

Таким образом, при запуске приложения будут запущены тесты. Если все они выполняются успешно, то будут выведены соответствующие информационные сообщения. В случае если какой-либо из тестов не выполнится, будут показаны имя теста, который не выполнился, имя исходного файла, содержащего этот тест, номер строки, на которой обнаружена ошибка, а также текст, заданный в параметрах вызова CPPUNIT\_ASSERT, который обнаружил проблему.

На практике для организации тестов и тестовых наборов удобнее использовать макросы:

```
class ComplexTest : public CppUnit::TestCase
{
private:
    Complex *m_10_1, *m_1_1, *m_11_2;
public: // Карта контейнера теста
    CPPUNIT_TEST_SUITE(ComplexTest) // Карта теста
        CPPUNIT_TEST(testEquality); // Регистрация тестовых
        CPPUNIT_TEST(testAddition); // функций
    CPPUNIT_TEST_SUITE_END();
public:
    // Подготовка к тестам
    // (создание и связывание между собой тестируемых объектов)
    void setUp ()
    {
        m_10_1 = new Complex (10, 1);
        m_1_1 = new Complex (1, 1);
        m_11_2 = new Complex (11, 2);
    }
    void tearDown () // Завершение тестов (очистка памяти)
    {
        delete m_10_1;
        delete m_1_1;
        delete m_11_2;
    }
protected: // Функции-тесты
    ...
    void testEquality ()
```

```

    CPPUNIT_ASSERT (*m_10_1 == *m_10_1);
    CPPUNIT_ASSERT (!(*m_10_1 == *m_11_2));
    Complex* pTest = new Complex (10, 1);
    CPPUNIT_ASSERT_EQUAL (*m_10_1, *pTest);

}

void testAddition ()
{
    Complex test_11_2 = *m_10_1 + *m_1_1;
    CPPUNIT_ASSERT (*m_10_1 + *m_1_1 == *m_11_2);
    CPPUNIT_ASSERT_EQUAL (test_11_2, *m_11_2);
}

CPPUNIT_TEST_SUITE_REGISTRATION(ComplexTestCase);

```

Модульные тесты могут быть соединены с процессом сборки приложения. Для этого нужно сделать так, чтобы тестовое приложение возвращало ненулевое значение при ошибке:

```

int main (int argc, char **argv)
{
    CppUnit::TextUi::TestRunner runner;
    runner.addTest (ComplexTestCase::suite ());
    return runner.run ();
}

```

Для запуска теста во время сборки нужно, чтобы приложение выполнялось после компиляции. Например, в Visual C++ это можно сделать в разделе Project Settings/Post-build steps.

### Контрольные вопросы

1. Каким образом реализуется модульное тестирование в C++?
2. Что такое тестовый набор и тестовый случай?
3. Какие макросы CPPUNIT позволяют упростить разработку тестов?
4. Каким образом настраивается проверка автоматических тестов во время компиляции?

### Практическое задание

Разработать модульный тест с применением библиотеки CPPUNIT для программного кода, разработанного на языке C++.

### Рекомендуемый порядок выполнения задания

1. Создать с помощью Microsoft Visual Studio 2008 консольный проект C++ (Console Application).
2. Создать несколько классов, которые будут имитировать тестируемую логику.
3. Подключить к проекту CPPUNIT.

4. Создать в проекте новый контейнерный тестовый класс.
5. Описать в этом классе функции setUp() и tearDown().
6. Разработать не менее пяти тестирующих функций. При разработке этих функций следует активно применять макросы CPPUNIT\_ASSERT.
7. Применить макросы CPPUNIT\_ASSERT\_THROW и CPPUNIT\_ASSERT\_NO\_THROW для контроля генерации исключений.
8. Создать карту-описатель контейнерного теста с помощью макроса CPPUNIT\_TEST\_SUITE.
9. Внести в карту-описатель тестирующие функции с помощью макроса CPPUNIT\_TEST.
10. Добавить в функцию main объект TestRunner.
11. Скомпилировать и запустить проект. Посмотреть на результат теста.
12. Настроить автоматический запуск теста при компиляции.
13. Внести в тестируемые классы изменения, приводящие к ошибкам.
14. Скомпилировать и запустить проект. Посмотреть, пойманы ли ошибки модульным тестом.

## МОДУЛЬНОЕ ТЕСТИРОВАНИЕ С ПРИМЕНЕНИЕМ JUNIT

JUnit 4 в настоящее время самый популярный инструмент модульного тестирования для Java. Он поддерживается многими средами разработки приложений на этом языке (Eclipse, JBuilder и т.д.), которые предоставляют средства для быстрого создания тестов, графические утилиты для их запуска и просмотра результатов. Библиотеки, ориентированные на Java, альтернативные JUnit, такие как TestNG и JTiger, зачастую подобной поддержки со стороны сред разработки не имеют. JUnit часто применяется в проектах, придерживающихся концепции Test Driven Development.

Для иллюстрации возможностей JUnit 4 создадим тестовое приложение Java в среде Eclipse (Java Project, рис. 5). В настройках проекта следует подключить библиотеку JUnit 4. Для этого следует переключиться в появившемся окне на вкладку Libraries (рис. 6) и нажать кнопку Add Library. В появившемся окне (рис. 7) следует выбрать JUnit, далее следует выбрать версию JUnit 4 (рис. 8). По завершении окно настройки параметров нового проекта примет вид, показанный на рис. 9.

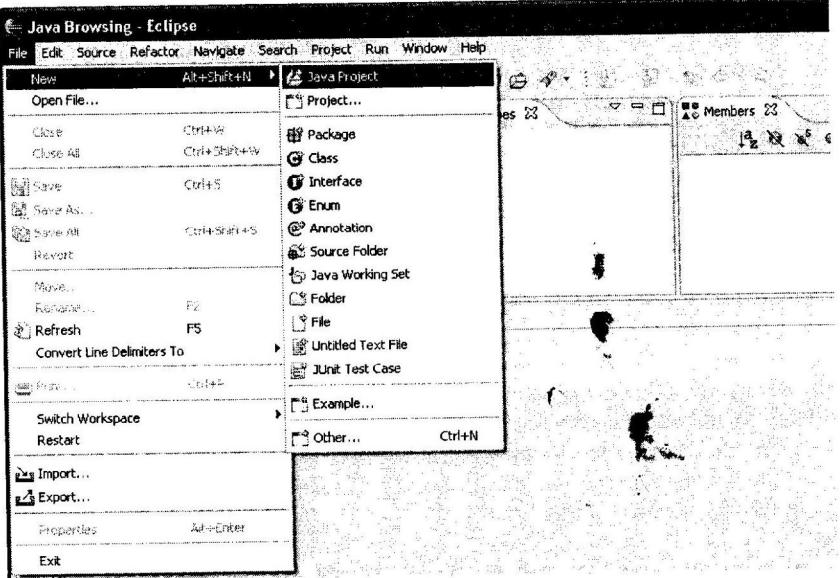


Рис. 5. Создание приложения Java в среде Eclipse

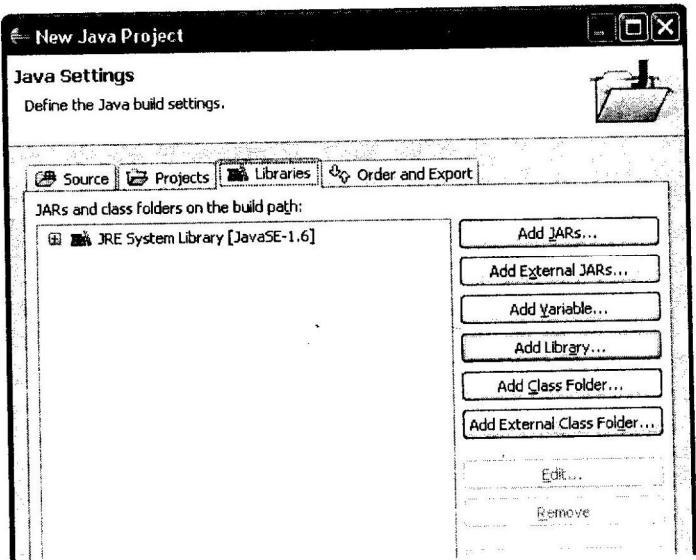


Рис. 6. Настройка подключаемых библиотек приложения Java

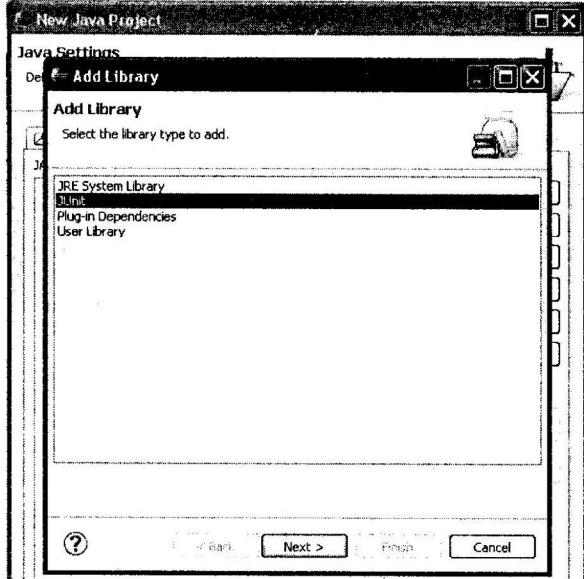


Рис. 7. Подключение библиотеки JUnit в приложении Java

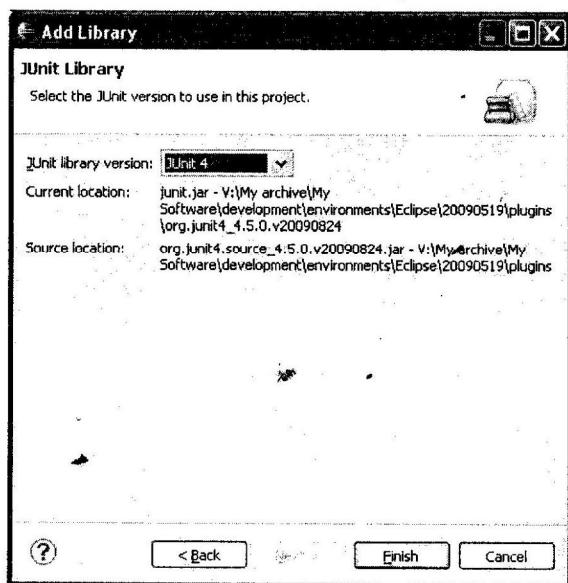


Рис. 8. Выбор версии библиотеки JUnit

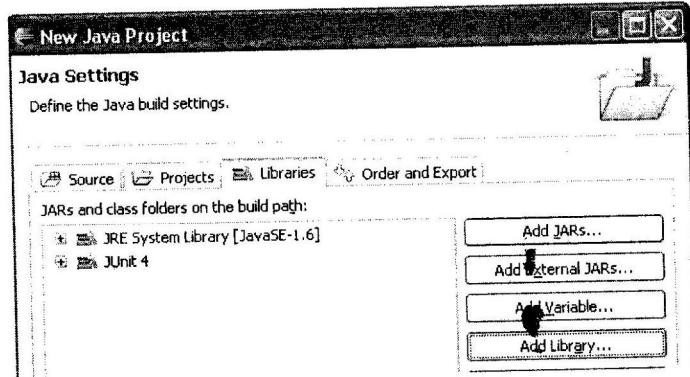


Рис. 9. Подключенная библиотека JUnit 4

```
test.java
import org.junit.Test;
import static org.junit.Assert.*;

public class CalculatorTest
{
    @Test
    public void add()
    {
        ...
        assertEquals(4, calculator.add( 1, 3 ) );
    }
}
```

Рис. 10. Пример теста на JUnit4

На рис. 10 приведен пример типичного теста на JUnit 4. Для идентификации тестовых методов в JUnit 4 широко используются аннотации. В приведенном примере с помощью аннотации `@Test` отмечен тестовый метод `add()`.

Для запуска теста на исполнение следует выбрать класс и в контекстном меню выбрать тип запуска JUnit Test (рис. 11). В результате откроется вкладка JUnit, содержащая результат тестового запуска (рис. 12). В случае если при выполнении теста будут обнаружены проблемы, они будут отображены в специальном окне (рис. 13).

Для выполнения проверок в тестовых функциях в JUnit предусмотрен статический класс `Assert`. Его методы могут быть вызваны напрямую, например `Assert.assertEquals(...)`, однако предпочтительнее использовать импорт статического класса (`import static org.junit.Assert.*;`), как это показано в примере на рис. 1. С помощью `assertEquals` можно сравнивать различные объекты, в том числе числа,

строки и даже массивы. Также предусмотрены варианты `assertTrue(v)` — контроль истинности переменной `v`; `assertFalse(v)` — контроль ложности переменной `v`; `assertNull(obj)` — контроль того, что переменная `obj = null`; `assertNotNull(obj)` — контроль того, что переменная `obj` содержит ссылку на объект, не равную `null`. Метод `fail` может использоваться для прерывания выполнения теста с ошибкой. Все функции, рассмотренные выше, также имеют перегрузку с дополнительным параметром для пользовательского сообщения.

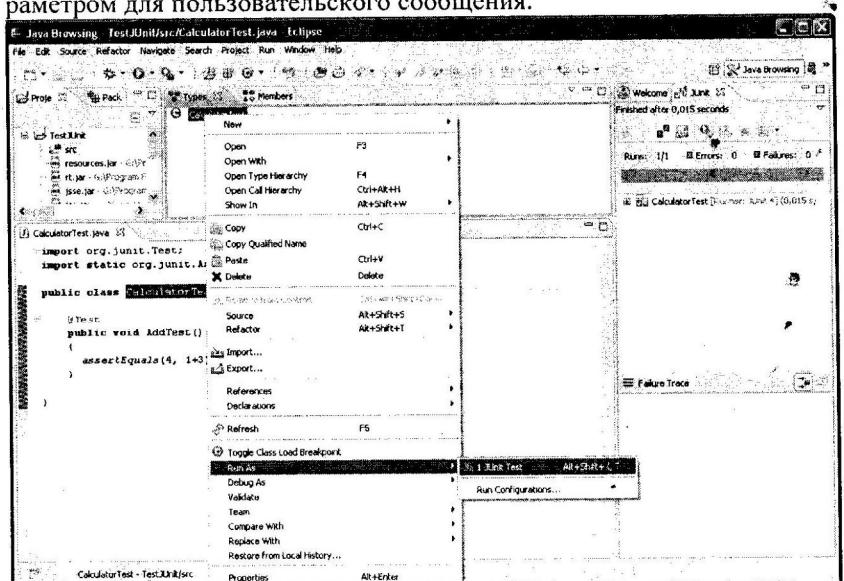


Рис. 11. Запуск теста

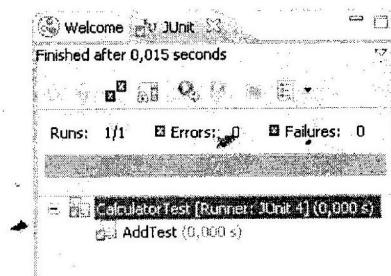


Рис. 12. Результат теста

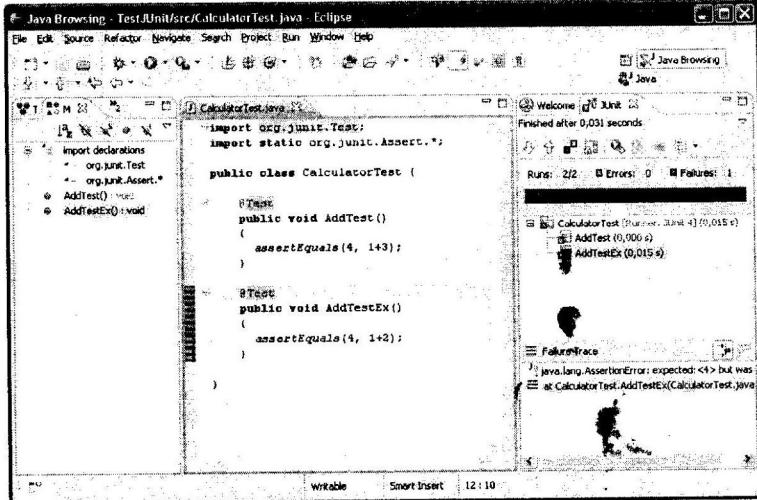


Рис. 13. Результат теста, в котором не все тесты выполнились успешно

С помощью аннотаций `@Before` и `@After` JUnit позволяет создавать методы, выполняемые перед началом и после завершения каждого теста:

```
test.java
import org.junit.Test;
import static org.junit.Assert.*;

public class CalculatorTest
{
    @Before
    public void prepareTestData()
    {
        ...
    }
    @After
    public void cleanupTestData()
    {
        ...
    }
}
```

JUnit также позволяет создавать методы, инициализирующие и деинициализирующие группу тестов, пользуясь аннотациями `@BeforeClass` и `@AfterClass`:

```
test.java
import org.junit.Test;
import static org.junit.Assert.*;

public class CalculatorTest
{
    @BeforeClass
    public static void setupConnection()
    {
        ...
    }
    @AfterClass
    public static void teardownConnection()
    {
        ...
    }
}
```

С помощью аннотаций можно создавать тесты, ожидающие появления определённых исключений. Например, тестовый метод, приведенный ниже, ожидает генерации исключения `ArithmetcException` в процессе выполнения:

```
@Test (expected=ArithmetcException.class)
public void testDivisionByZero ()
{
    new Calculator ().Divide (4, 0);
}
```

Интересной возможностью JUnit 4 является способность ограничивать максимальное время успешного выполнения тестового метода. При превышении указанного в качестве `timeout` лимита времени тест завершается неудачей. Это позволяет помимо правильности вычислений контролировать уровень их производительности:

```
@Test (timeout=5000)
public void testConnect ()
```

```
...
```

При помощи аннотаций `@RunWith` и `@SuiteClasses` все тесты, принадлежащие определённым классам, могут быть объединены в наборы (Suite):

```
@RunWith(value=Suite.class)
@SuiteClasses (value={CalculatorTest.class, AnotherTest.class})
public class AllTests {
```

```
...
```

Аннотация `@RunWith` и класс `Parameterized` позволяют создавать параметризованные тесты:

```
@RunWith(value=Parameterized.class)
public class FactorialTest
```

```
{
    private long expected;
    private int value;
```

```
@Parameters
```

```
public static Collection data () {
    {1, 0}, // expected, value
    {1, 1},
    {2, 2},
    {24, 4},
    {5040, 7} );
}
```

```
public FactorialTest (long expected, int value) {
    this.expected = expected;
    this.value = value;
}
```

```

@Ters
public void factorial () {
    Calculator calculator = new Calculator ();
    assertEquals (expected, calculator.factorial(value));
}

```

В примере класс Parameterized запускает все тесты класса FactorialTest (в примере он только один), используя при этом данные методов, промаркированных аннотацией @Parameters. В данном случае имеется список с пятью элементами. Каждый элемент содержит массив, который будет использован в качестве аргументов конструктора класса FactorialTest.

До появления JUnit 4 игнорирование неудачных или незавершенных тестов представляло определенную проблему. Если было необходимо, чтобы среда проигнорировала определенный тест, нужно было изменить имя этого теста таким образом, чтобы оно не соответствовало системе обозначения тестов. В JUnit 4 предусмотрена специальная аннотация @Ignore, с помощью которой можно промаркировать тестовый метод и тем самым заставить среду выполнения тестов JUnit 4 его проигнорировать. При этом в карте выполнения тестов JUnit 4 проигнорированные тесты отображаются явно (рис. 14).

Поддержка аннотаций, реализованная в JUnit 4, существенно упрощает процесс разработки модульных тестов. Один из полезных побочных эффектов аннотаций заключается в том, что аннотации навсегда документируют все то, что должен делать каждый метод, без необходимости глубокого понимания внутренней модели инфраструктуры тестирования. Что может быть нагляднее, чем маркировка тестового метода аннотацией @Test? Это существенное усовершенствование по сравнению с предшествующими версиями JUnit, для которых требовалось хорошее знание соглашений JUnit, даже если вы просто хотели понять, какой вклад вносит каждый метод в общий тестовый сценарий (test case).

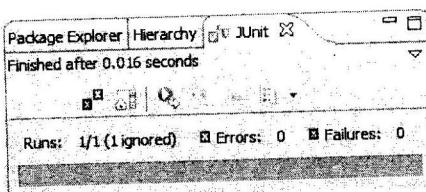


Рис. 14. Статистика выполнения тестов

## Контрольные вопросы

1. Каким образом реализуется модульное тестирование в языке Java?
2. Какие возможности реализованы в классе Assert?
3. Что такое тестовый сценарий?
4. Как реализовать тест, помимо корректности выполнения также проверяющий и скорость?
5. Каким образом настраивается игнорирование некоторых тестов?
6. Сравните JUnit и CPPUnit.

## Практическое задание

Разработать модульный тест с применением библиотеки JUnit 4 для программного кода, разработанного на языке Java.

### Рекомендуемый порядок выполнения задания

1. Создать с помощью Eclipse консольный проект Java (Java Application). Подключить к проекту библиотеку JUnit версии 4.
2. Создать несколько классов, которые будут имитировать тестируемую логику.
3. Создать в проекте новый тестовый класс.
4. Описать в этом классе функции setUp() и tearDown() и отметить их аннотациями @Before, @After.
5. Разработать не менее пяти тестирующих функций, отметить их соответствующими аннотациями. При разработке этих функций следует активно применять функции assertEquals, assertTrue, assertFalse.
6. Добавить в тестовые методы спецификацию ожидаемых исключений.
7. Разработать параметризованный тест.
8. Запустить проверку разработанного тестового класса.
9. Скомпилировать и запустить проект. Посмотреть на результат теста.
10. Внести в тестируемые классы изменения, приводящие к ошибкам.
11. Скомпилировать и запустить проект. Посмотреть, пойманы ли ошибки модульным тестом.

## МОДУЛЬНОЕ ТЕСТИРОВАНИЕ С ПРИМЕНЕНИЕМ NUNIT

NUnit — открытая среда модульного тестирования приложений для .NET. Она была портирована с языка Java (библиотека JUnit). Первые версии NUnit были написаны на J#, но затем весь код был переписан на C#, с использованием атрибутов, что делает современные версии JUnit и NUnit очень близкими по удобству использования и возможностям. Для написания тестов с использованием NUnit необходимо установить ссылку (Reference) на библиотеку NUnit.Framework.dll. Для иллюстрации возможностей NUnit создадим в MS Visual Studio новый проект C# (Console Application) (рис. 15) и добавим в него ссылку на NUnit.Framework.dll (рис. 16).

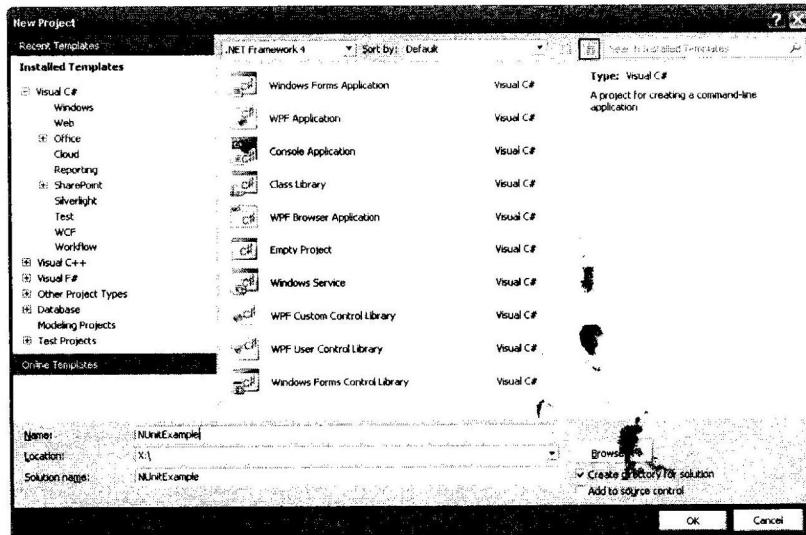


Рис. 15. Создание консольного приложения C# в Visual Studio 2010

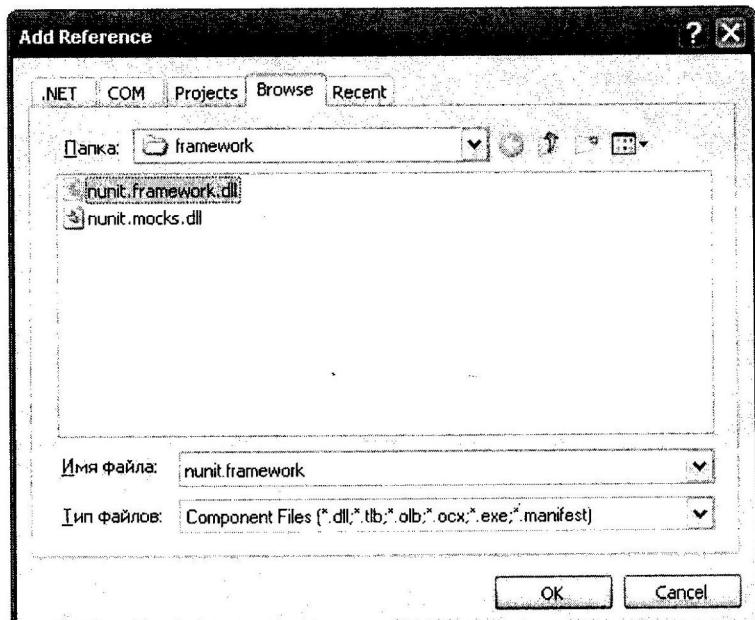


Рис. 16. Установка ссылки (reference) на nunit.framework.dll

Для идентификации тестов в массиве кода используются атрибуты. Для определения тестовых наборов следует использовать классы, отмеченные атрибутом [TestFixture], для единичных тестов – тесты, отмеченные атрибутом [Test]. Обычно для каждой области тестирования создается свой тестовый набор. Таким образом, при разработке тестов с применением NUnit не требуется использовать наследование каких-либо базовых классов этой библиотеки.

Ниже приведен пример теста на NUnit:

```
using NUnit.Framework;
[TestFixture]
public class ExampleTestOfNUnit
{
    [Test]
    public void TestMultiplication()
    {
        Assert.AreEqual(4, 2 * 2, "Умножение");
    }
}
```

Для запуска тестов может быть использована графическая среда NUnit. Для этого следует собрать проект, содержащий тест, запустить графическое приложение NUnit и выбрать в нем (File -> Open Project) сборку с тестом (рис. 17).

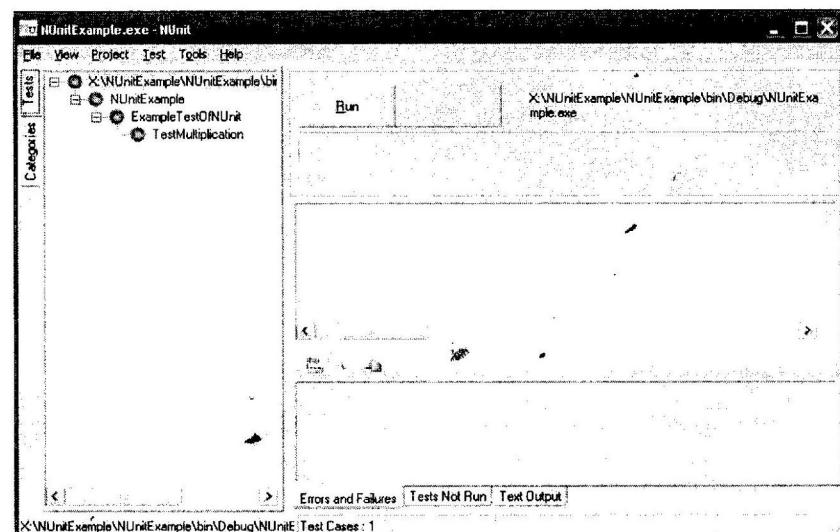


Рис. 17. Интерфейс графической среды NUnit

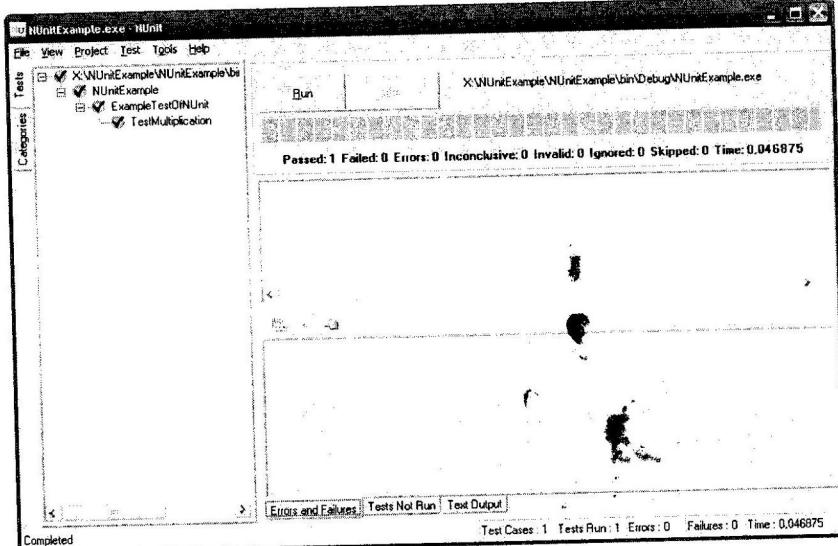


Рис. 18. Запуск теста в NUNIT

В дереве тестов будут отображены все найденные тестовые наборы и тестовые методы. Нажатие на кнопку Run приведет к запуску тестов, в результате будет показан прогресс выполнения и статистика (рис. 18). При наличии нескольких тестов или тестовых наборов также может быть запущен один из них.

Дополним тестовый класс тестом, который сгенерирует ошибку:

```
[Test]
public void TestMultiplicationEx()
{
    Assert.AreEqual(4, 2 * 1, "Осознанно генерируем проблему");
}
```

Если в результате выполнения тестов какой-либо из них не выполнится, будет показана информация о невыполненных тестах, их числе, а также вывод пояснительной информации (рис. 19).

При анализе причин неудачи выполнения тестов NUNIT может быть использован отладчик Visual Studio. Для этого следует перенастроить параметры запуска, как показано на рис. 20. В результате будет запускаться NUNIT. Можно будет выбрать интересующий тест в пользовательском интерфейсе NUNIT, а в коде теста можно будет использовать возможности отладчика (например, точки остановки).

Также предусмотрен режим запуска тестов из консольной оболочки. Для этого следует вызвать nunit-console.exe <assemblyname>, где <assemblyname> — сборка с тестами.

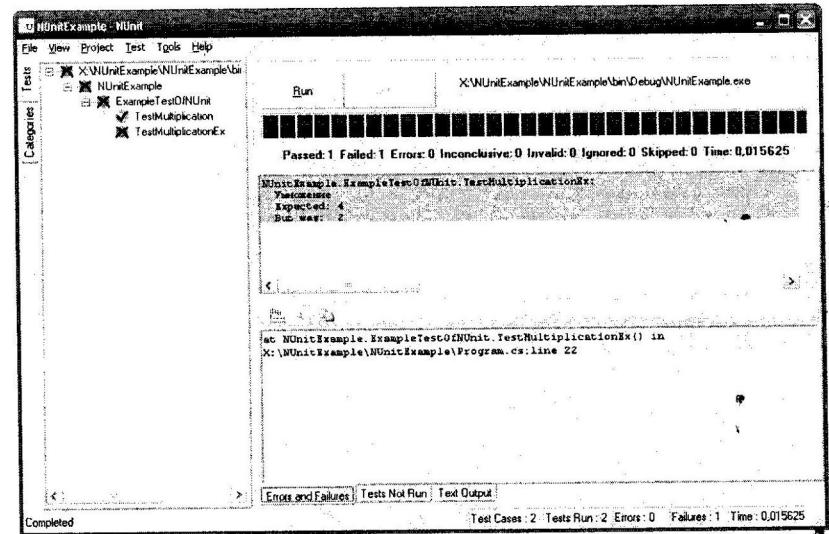


Рис. 19. Отображение информации о невыполнившемся teste

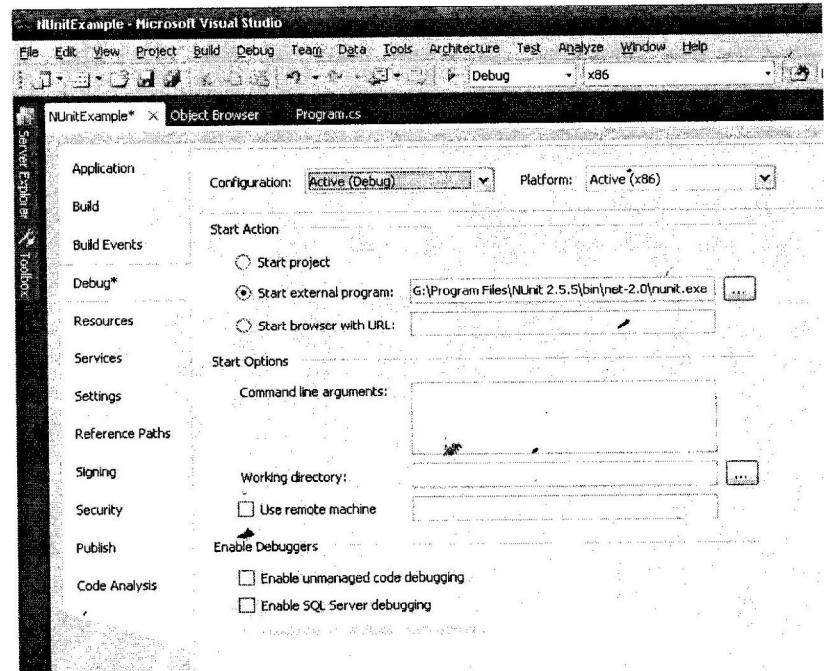


Рис. 20. Настройка отладки тестов NUNIT с помощью Visual Studio

Для организации проверок используются статические методы класса Assert. В частности, предусмотрены методы:

- Assert.AreEqual(expected, actual, [string message]) – позволяет контролировать равенство двух величин;
- Assert.IsNull(object [, string message]) – позволяет контролировать, что переданный параметр не установлен (null);
- Assert.IsNotNull(object [, string message]) – позволяет контролировать, что переданный параметр не равен null;
- Assert.AreSame(expected, actual [, string message]) – позволяет контролировать, что expected и actual являются одним и тем же объектом;
- Assert.IsTrue(bool condition [, string message]) – позволяет контролировать истинность переданного параметра condition;
- Assert.Fail([string message]) – позволяет немедленно прервать выполнение теста с выводом опционального диагностического сообщения.

В NUnit предусмотрен класс для организации тестовых наборов — TestSuite. Для того чтобы его использовать, следует включить в тестовый класс статический метод, помеченный атрибутом [Suite], возвращающий инициализированный экземпляр класса TestSuite. Например:

```
using NUnit.Framework;
using NUnit.Core;
namespace NUnitExample
{
    [TestFixture]
    public class TestSuiteClass
    {
        [Suite]
        public static TestSuite Suite
        {
            get
            {
                TestSuite suite = new TestSuite("Тестовый набор №1");
                suite.Add(new ExampleTestOfNUnit());
                suite.Add(new AnotherExample());
                return suite;
            }
        }
    }
}
```

Тесты в NUnit также могут быть сгруппированы по категориям. Для этого предусмотрен атрибут Category:

```
[TestFixture]
public class Example
{
    [Test]
    [Category("Math")]
    public void TestDivision()
    {
        Assert.AreEqual(4/2, 2, "Деление");
    }

    [Test]
    [Category("Math")]
    public void TestMultiplication()
    {
        Assert.AreEqual(4, 2 * 2, "Умножение");
    }

    [Test]
    [Category("String")]
    public void Teststrcmp()
    {
        Assert.IsTrue(string.Compare("abc", "abc") == 0);
    }
}
```

При этом доступен вариант запуска тестов определенной категории. На рис. 21 показан снимок экрана NUnit в режиме запуска тестов определенных категорий.

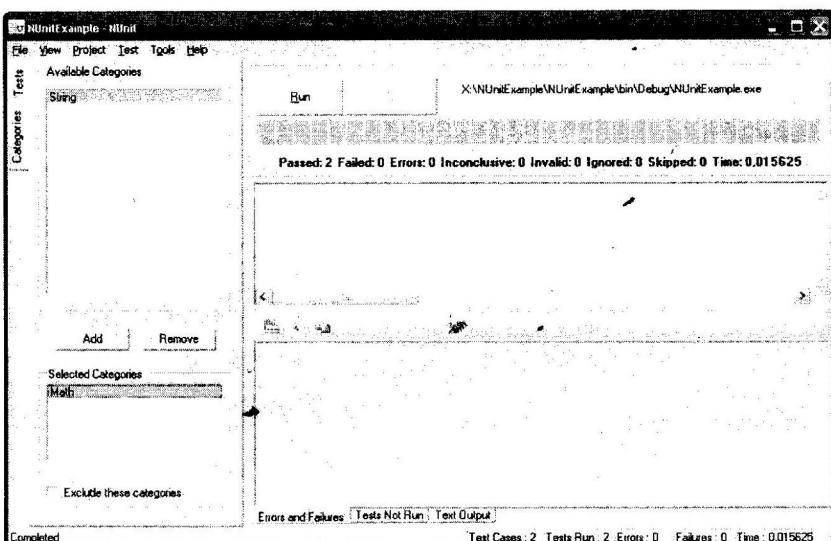


Рис. 21. Запуск тестов по категориям

Каждый тест должен выполняться независимо от других. Чтобы обеспечить это, иногда требуется подготовить тестовое окружение. В NUnit для этого предусмотрены атрибуты [SetUp] и [TearDown]. Например, если внести в тестовый класс такие объявления:

```
[SetUp]
public void Setup()
{
    ...
}

[TearDown]
public void Shutdown()
{
    ...
}
```

то функции `Setup` и `Shutdown` будут выполняться до и соответственно после выполнения каждого тестового метода в этом классе. Также предусмотрены атрибуты `[TestFixtureSetUp]` и `[TestFixtureTearDown]`, позволяющие организовать методы инициализации и деинициализации, действующие один раз для всех методов тестового класса.

NUnit обеспечивает перехват всех исключений, которые генерируются в тестовых методах. При этом информация о них будет выведена в логи, а тесты, в которых они были сгенерированы, будут считаться не пройденными. Также предусмотрена возможность спецификации ожидаемых исключений, что позволяет контролировать поведение тестируемого кода в режиме работы с аномальными входными данными. Например, приведенный ниже фрагмент кода ожидает генерации исключения типа `ArgumentException`:

```
[TestFixture]
public class ExceptionExample
{
    [Test, ExpectedException(typeof(ArgumentException))]
    public void TestException()
    {
        GotoPage(null);
    }
}
```

В случае если исключение не будет сгенерировано, тест будет считаться не пройденным.

Предусмотрена возможность временно отключать тесты. Для этого следует использовать атрибут `[Ignore]`. Например:

```
[Test, Ignore("Временно отключен")]
public void TestDivision()
{
    Assert.AreEqual(4/2, 2, "Деление");
}
```

Благодаря языковой интерабильности платформы .NET тесты с применением NUnit можно писать не только на языке C#, но и на иных языках платформы – Basic, Managed C++ и т.д. NUnit содержит две версии пользовательского приложения – консольную и графическую среду отображения хода выполнения тестов.

Кроме того, существуют известные расширения оригинального пакета NUnit, большая часть из них также с открытым исходным кодом. NUnit.Forms дополняет NUnit средствами тестирования элементов пользовательского интерфейса Windows Forms. NUnit.ASP выполняет ту же задачу для элементов интерфейса в ASP.NET.

## Контрольные вопросы

1. Каким образом реализуется модульное тестирование на языках платформы .NET?
2. Какие возможности реализованы в классе `Assert`?
3. С помощью чего в NUnit реализуются тестовые сценарии?
4. Как реализовать тест, проверяющий помимо корректности выполнения также и скорость?
5. Каким образом настраивается игнорирование некоторых тестов?
6. Каким образом тестируются случаи, в которых тестируемый код должен генерировать исключения?

## Практическое задание

Разработать модульный тест с применением NUnit для программного кода, разработанного на языке C#.

### Рекомендуемый порядок выполнения задания

1. Создать с помощью MS Visual Studio консольный проект C# (Console Application).
2. Подключить к проекту библиотеку `NUnit.Framework.dll`.
3. Создать несколько классов, которые будут имитировать тестируемую логику.
4. Создать в проекте новый тестовый класс, пометить его с помощью атрибута `[TestFixture]`.
5. Описать в этом классе функции `setUp()` и `tearDown()` и отметить их атрибутами `[SetUp]`, `[TearDown]`.
6. Разработать не менее пяти тестирующих функций, отметить их соответствующими атрибутами `[Test]`. При разработке этих функций следует активно применять функции класса `Assert` — `AreEquals`, `IsTrue`, `IsNull`.
7. Разработать тестовые методы со спецификацией ожидаемых исключений.

8. Запустить проверку разработанного тестового класса.
9. Скомпилировать и запустить проект. Посмотреть на результат теста.
10. Внести в тестируемые классы изменения, приводящие к ошибкам.
11. Скомпилировать и запустить проект. Посмотреть, обнаружены ли ошибки модульным тестом.

## МОДУЛЬНЫЕ ТЕСТЫ В VISUAL STUDIO TEAM SYSTEM

В состав Visual Studio Team System входят интегрированные средства создания и выполнения модульных тестов непосредственно из среды разработчика. Модульное тестирование предполагает, что для каждого метода класса пишется определенный тест. Такой тест проверяет все возможные способы поведения класса (в соответствии со спецификацией) и сообщает об обнаруженных ошибках. В Visual Studio реализована поддержка собственной реализации библиотеки модульного тестирования Unit Testing Framework (далее MS Unit). По своим возможностям она очень близка к NUnit, однако гораздо глубже интегрирована со средой разработки и постоянно развивается, что дает существенный стимул к ее использованию.

Для идентификации тестов в MS Unit, так же как и в NUnit, используются атрибуты. С помощью атрибутов [TestClass] и [TestMethod] отмечаются тестовые наборы и тестовые методы. Можно создать методы для инициализации/денициализации тестового набора (для этого используются атрибуты [ClassInitialize()] и [ClassCleanup()]). Также предусмотрены методы инициализации/денициализации для каждого тестового метода, для этого используются атрибуты [TestInitialize()] и [TestCleanup()]. Для выполнения проверок предусмотрен статический класс Assert.

Для иллюстрации возможностей среды Visual Studio 2010 для модульного тестирования создадим консольное приложение C# (рис. 22). В этом приложении создадим класс Calculator, методы которого будут проверяться с помощью модульного теста.

Модульный тест может быть добавлен вручном режиме. Для этого необходимо создать дополнительно проект типа Test Project (рис. 23). После его создания он будет содержать класс UnitTest1 с атрибутом [TestClass], содержащий в себе метод TestMethod1 с атрибутом [TestMethod]:

```
using System;
using System.Text;
using Microsoft.VisualStudio.TestTools.UnitTesting;
namespace Tests
{
```

32

```
[TestClass]
public class UnitTest1
{
    [TestMethod]
    public void TestMethod1()
    {
    }
}
```

Атрибут [TestClass] означает, что этот класс содержит тестовые методы, а [TestMethod] представляет собой конкретный метод.

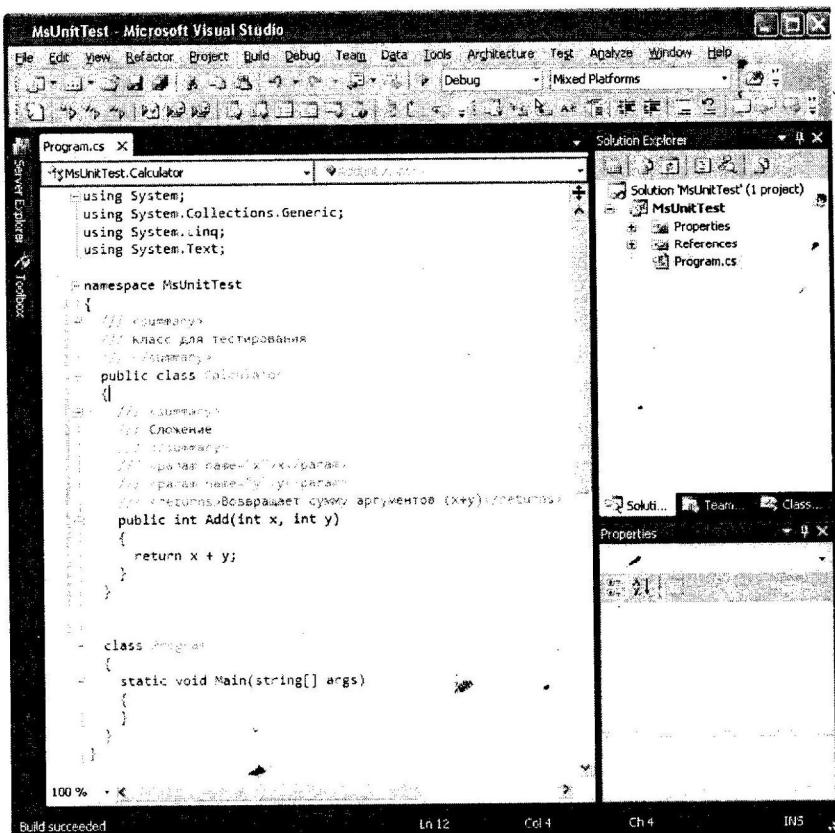


Рис. 22. Пример класса для тестирования

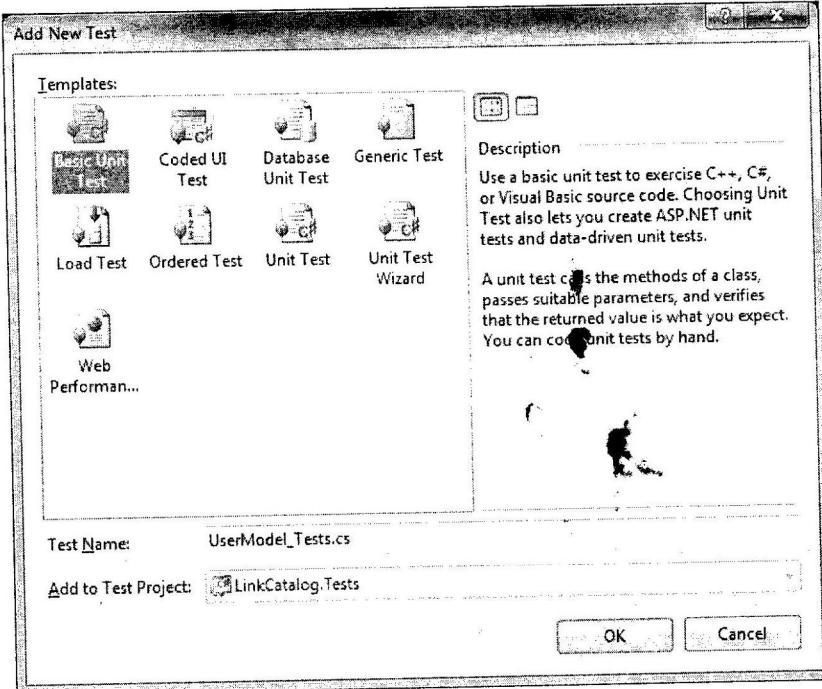


Рис. 23. Создание модульного теста

После чего этот класс может быть скорректирован вручную для организации теста класса Calculator. Для этого следует установить зависимость (Reference) на сборку MSUnitTest, дать тестовому набору и тестовому методу осмысленные имена:

```
[TestMethod]
public void TestCalculatorAddMethod()
{
    Calculator calculator = new Calculator();
    int result = calculator.Add(1, 2);
    Assert.AreEqual(3, result);
}
```

Однако в Visual Studio 2010 предусмотрена удобная возможность генерации теста в автоматическом режиме. Для этого необходимо перейти в Class View (просмотр классов), выбрать класс Calculator и в появившемся контекстном меню выбрать пункт Create Unit Tests (рис. 24). Будет отображено окно, в котором можно будет выбрать для модульного тестирования доступные в проспекте классы и методы (рис. 25).

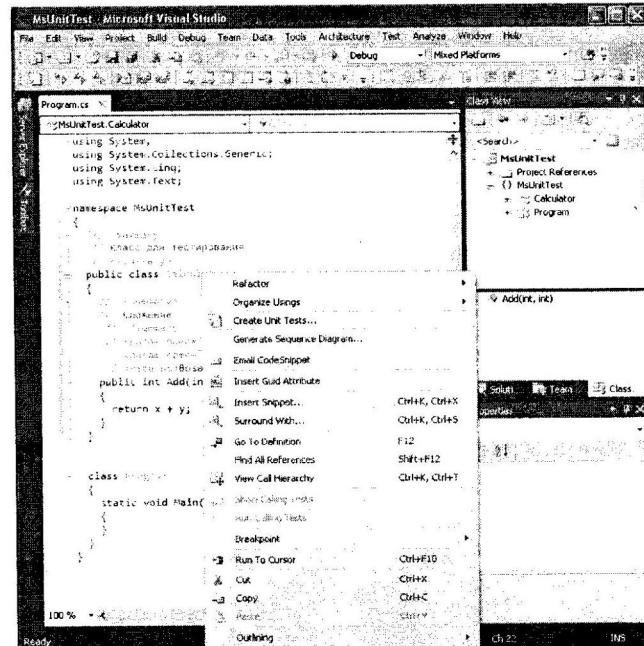


Рис. 24. Вызов встроенного генератора модульных тестов

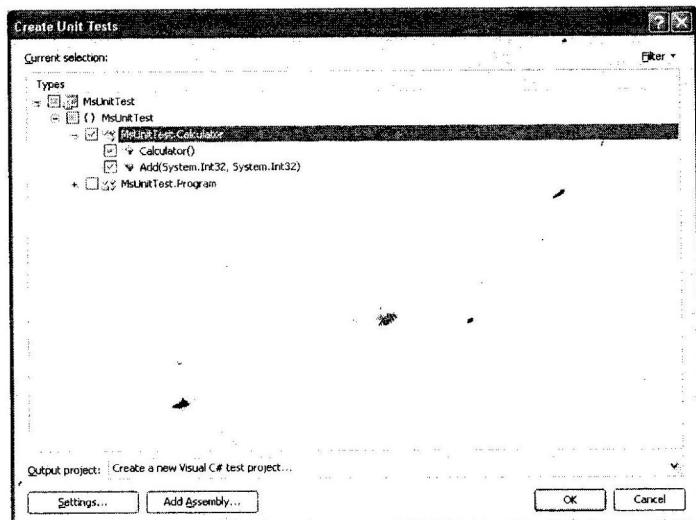


Рис. 25. Выбор методов, которые будут тестируться

В результате будет автоматически сгенерирован тестовый проект и код теста, содержащий тестовый класс и тестовые методы:

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

namespace CalculatorTest
{
    /// <summary>
    /// This is a test class for CalculatorTest and is intended
    /// to contain all CalculatorTest Unit Tests
    /// </summary>
    [TestClass()]
    public class CalculatorTest
    {
        private TestContext testContextInstance;

        /// <summary>
        /// Gets or sets the test context which provides
        /// information about and functionality for the current test run.
        /// </summary>
        public TestContext TestContext
        {
            get
            {
                return testContextInstance;
            }
            set
            {
                testContextInstance = value;
            }
        }

        #region Additional test attributes
        //
        // You can use the following additional attributes as you write your tests:
        //
        // Use ClassInitialize to run code before running the first test in the class
        // [ClassInitialize()]
        // public static void MyClassInitialize(TestContext testContext)
        // {
        // }
        // Use ClassCleanup to run code after all tests in a class have run
        // [ClassCleanup()]
        // public static void MyClassCleanup()
        // {
        // }
        // Use TestInitialize to run code before running each test
        // [TestInitialize()]
        // public void MyTestInitialize()
```

```
///
//>
//Use TestCleanup to run code after each test has run
//#[TestCleanup()]
//public void MyTestCleanup()
//{
//}
//
#endregion

/// <summary>
/// A test for Calculator Constructor
/// </summary>
[TestMethod()]
public void CalculatorConstructorTest()
{
    Calculator target = new Calculator();
    Assert.Inconclusive("TODO: Implement code to verify target");
}

/// <summary>
/// A test for Add
/// </summary>
[TestMethod()]
public void AddTest()
{
    Calculator target = new Calculator(); // TODO: Initialize to an appropriate value
    int x = 0; // TODO: Initialize to an appropriate value
    int y = 0; // TODO: Initialize to an appropriate value
    int expected = 0; // TODO: Initialize to an appropriate value
    int actual;
    actual = target.Add(x, y);
    Assert.AreEqual(expected, actual);
    Assert.Inconclusive("Verify the correctness of this test method.");
}
```

Отметим, что в получаемом коде названия тестовых методов и классов отражают их суть. Кроме того, среда сгенерировала опорный код для реализации теста, сконструировав параметры для соответствующих методов, сгенерировав вызовы и реализовав проверяющий код. Помимо этого в проект добавлены настройки локальных тестов, трассировки и настройки тестовых планов (MSUnitTest) (рис. 26). Из окна настройки списка тестов можно запустить необходимый набор тестов (рис. 27); предусмотрен запуск модульных тестов непосредственно из редактора кода (для этого необходимо в теле тестируемого метода вызвать контекстное меню и выбрать пункт Run Tests).

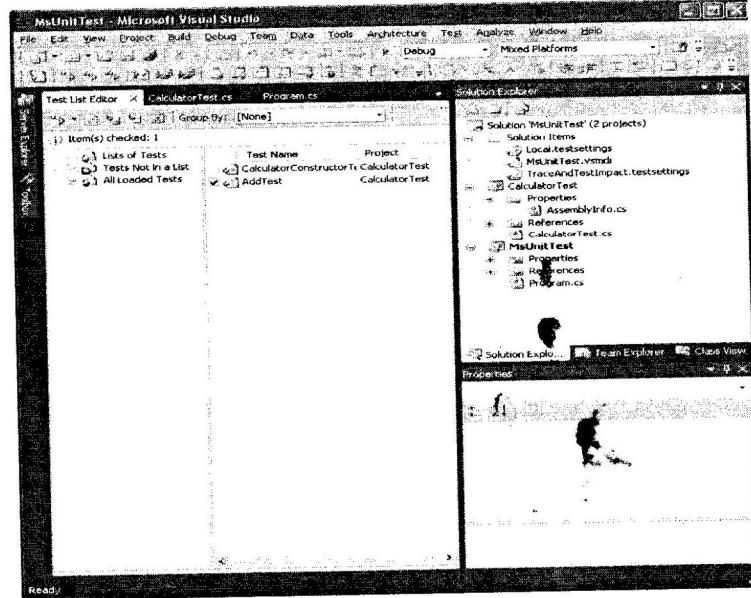


Рис. 26. Тестовый план

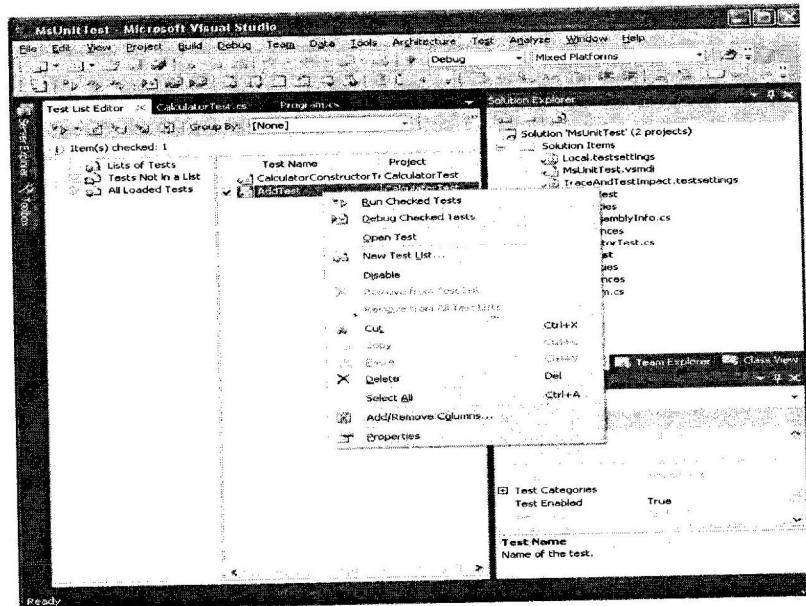


Рис. 27. Запуск тестов

Результат выполнения тестов отображается во вкладке Test Results (рис. 28), также можно посмотреть детали выполнения отдельного тестового метода (рис. 29).

Также MSUnit позволяет применять наследование тестовых классов и организовывать иерархии.

Результаты запуска модульных тестов могут быть сохранены для последующего анализа в виде файлов в XML-формате с расширением .trx (Test Result XML). Благодаря фиксированной схеме этих файлов доступна возможность программной обработки результатов тестиования.

В Visual Studio Team System также реализован режим анализа области покрытия кода тестами. На рис. 30 показана подсветка кода, выполненная Visual Studio в режиме анализа области покрытия кода тестами после прохождения всех модульных тестов. В редакторе исходного кода красным цветом отмечаются фрагменты кода, которые не выполнялись при тестировании.

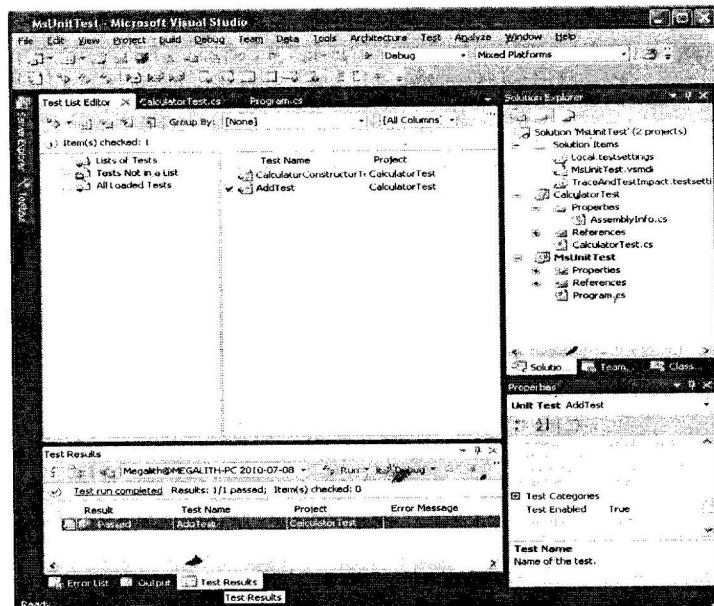


Рис. 28. Результат запуска модульных тестов

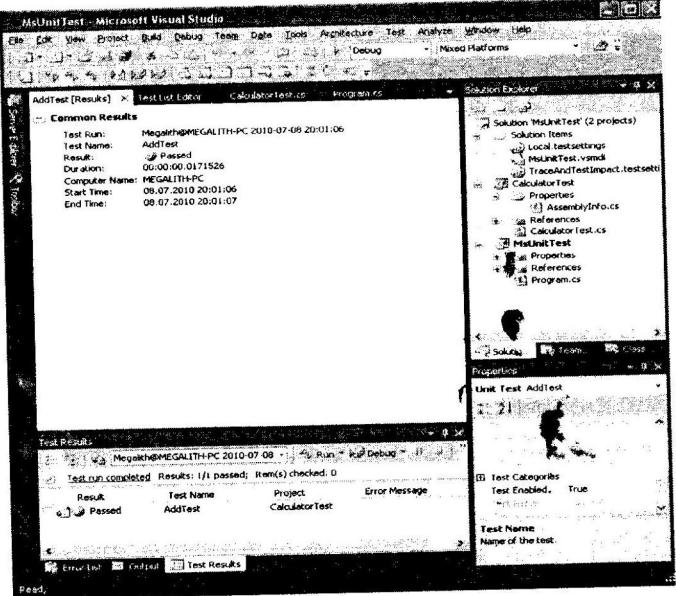


Рис. 29. Детальная информация о пройденном teste

```
Test Explorer LogonInfoTest.cs LogonInfo.cs
VSTS Demo.LogonInfo
ChangePassword(string oldPassword, string newPassword)
public void ChangePassword(
    string oldPassword, string newPassword)
{
    if (oldPassword == Password)
    {
        Password = newPassword;
    }
    else
    {
        throw new ArgumentException(
            "The old password was not correct");
    }
}
```

Рис. 30. Анализ области покрытия кода тестами

### Контрольные вопросы

- Каким образом реализуется модульное тестирование в Visual Studio?
- Какие возможности реализованы в классе Assert?
- С помощью чего в MSUnit реализуются тестовые сценарии?
- Как реализовать тест, проверяющий помимо корректности выполнения также и скорость?
- Каким образом настраивается игнорирование некоторых тестов?

- Каким образом тестируются случаи, в которых тестируемый код должен генерировать исключения?
- Каким образом можно проверить, что весь исходный код покрыт модульными тестами?
- Допускается ли наследование тестовых классов?

### Практическое задание

Разработать модульный тест с применением MsUnit для программного кода, разработанного на языке C#.

### Рекомендуемый порядок выполнения задания

- Создать с помощью MS Visual Studio проект консольного приложения C# (Console Application).
- Создать несколько интерфейсов для классов, которые будут имитировать тестируемую логику. Создать классы-реализации этих интерфейсов.
- Сгенерировать тестовый проект с модульными тестами.
- Реализовать не менее пяти тестирующих функций. При разработке этих функций следует активно применять функции класса Assert.
- Разработать тестовый метод со спецификацией ожидаемых исключений.
- Запустить модульные тесты. Проанализировать результаты запуска.
- Внести в тестируемые классы изменения, приводящие к ошибкам.
- Скомпилировать и запустить проект. Посмотреть, пойманы ли ошибки модульным тестом.

## ТЕСТИРОВАНИЕ БАЗ ДАННЫХ В VISUAL STUDIO TEAM SYSTEM

Техника модульного тестирования также может применяться для тестирования корректности работы приложения с базой данных (БД) и проверки работы хранимых процедур. Обычно подобные модульные тесты работают по следующей схеме. Вначале создается и приводится в необходимое состояние тестируемая БД. Далее выполняются модульные тесты. По завершении тестов тестируемая БД очищается.

Модульный тест БД может проверять корректность таблиц, правильность записей в таблицах, корректность работы хранимых процедур, время выполнения запросов, результаты выполнения запросов и т.д.

Для организации модульных тестов БД в Visual Studio Team System предусмотрен специальный шаблон (рис. 31).

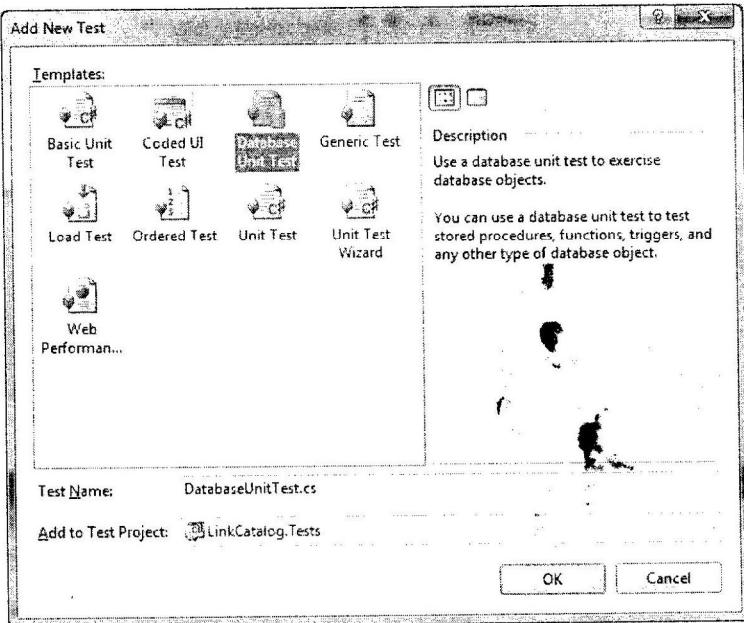


Рис. 31. Детальная информация о пройденном teste

В качестве примера рассмотрим модульный тест базы SQL Server ExampleDB. Эта БД содержит таблицу Users (список пользователей):

```
CREATE TABLE [dbo].[Users](
    [id] [int] IDENTITY(1,1) NOT NULL,
    [login] [nchar](100) NOT NULL,
    [pass] [nchar](50) NOT NULL,
    [level] [int] NULL,
CONSTRAINT [PK_Users] PRIMARY KEY CLUSTERED
    ([id] ASC)
WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON))
```

А также хранимую процедуру, которая может вернуть идентификатор пользователя по его логину и паролю:

```
CREATE PROCEDURE [dbo].[GetUserId]
(
    @login nchar(100),
    @hash nchar(50),
    @id int OUTPUT
)
```

```
AS
SELECT @id = ID FROM Users WHERE Login = @login AND hash = @hash
RETURN
```

Модульные тесты БД создаются в составе проектов типа TestProject. Для создания нового теста БД необходимо добавить в проект Database Unit Test (рис. 31). В процессе генерации теста будет показано окно настройки соединения с БД, в котором в нашем случае необходимо указать подключение к ExampleDB. В этом же окне можно указать параметры автоматической генерации данных для тестов, при этом используется функция Data Generation Plan, которая позволяет заполнить таблицу БД тестовыми значениями, используя шаблоны и даже регулярные выражения (рис. 32). По завершении работы мастера будет открыто окно тестирования БД. В этом окне могут добавляться именованные тесты, вводиться запросы на языке SQL и настраиваться условия проверки корректности возвращаемых результатов. Для реализации теста, проверяющего корректность заполнения таблицы БД, необходимо добавить новый тест RowCountInTable (рис. 33) и ввести в поле запроса: Select count(\*) FROM Users (рис. 34).

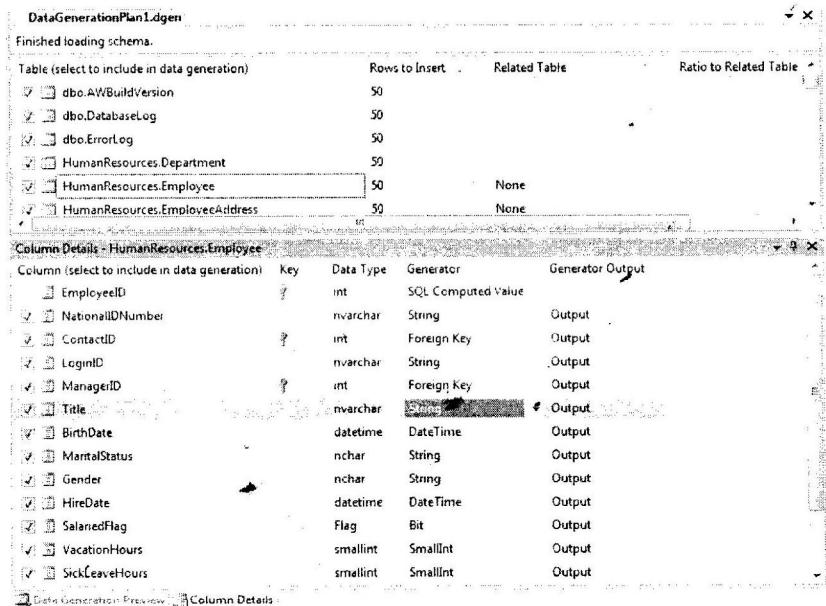


Рис. 32. Внешний вид окна настроек автоматической генерации записей в таблицах тестируемой БД

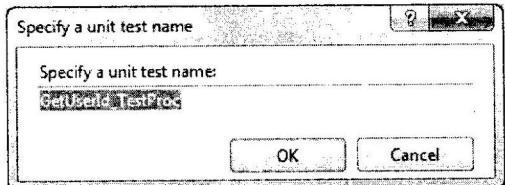


Рис. 33. Ввод имени теста



Рис. 34. Редактирование тестового запроса

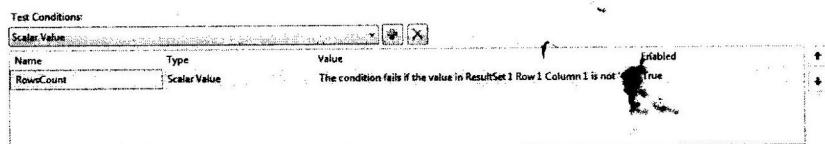


Рис. 35. Настройка условий корректности выполнения запроса

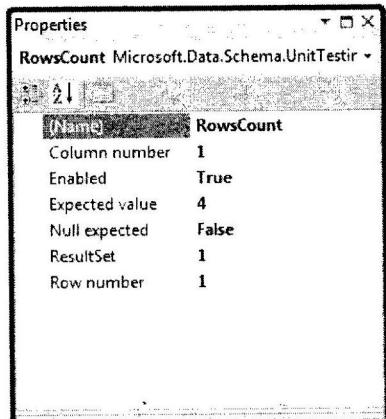


Рис. 36. Настройка параметров условия корректности выполнения запроса

Данный запрос возвращает одиночное значение, поэтому в качестве условия проверки можно выбрать Scalar Value (рис. 35). Параметры условия настраиваются в окне Properties (рис. 36). После настройки условия тест может быть запущен. В результате будет произведено подключение к БД, выполнен запрос, проверено условие и подготовлен отчет о результатах теста (рис. 37).

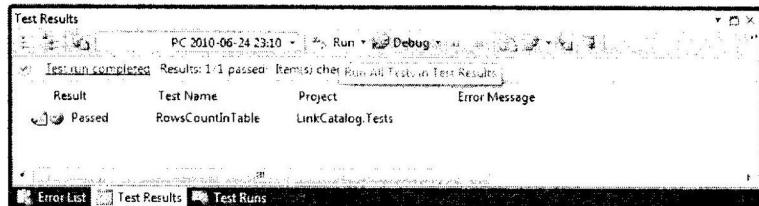


Рис. 37. Результат выполнения теста

Добавим в проект еще один тест, в котором осуществим проверку работы хранимой процедуры. В окне редактирования запроса введем код вызова процедуры:

```
/* Выходное значение */
DECLARE @id INT;
SET @id = -1
```

```
/* Должно установить id = 1 */
EXEC GetUserId N'Test', N'1', @id OUTPUT;
```

```
SELECT * FROM Users WHERE id = @id
```

Этот запрос вернет запись с данными пользователя. В случае если приложение работает корректно, в таблице логинов пользователей должны содержаться уникальные значения. Поэтому запрос должен вернуть одну запись, если пользователь с логином Test зарегистрирован. Настроим соответствующим образом условие корректности выполнения запроса (рис. 38). Если запустить этот тест на исполнение, то будет осуществлено подключение к БД, вызвана хранимая процедура, выполнен запрос к БД и проверено условие корректности. В результате будет сгенерирован отчет о выполнении тестов (рис. 39).

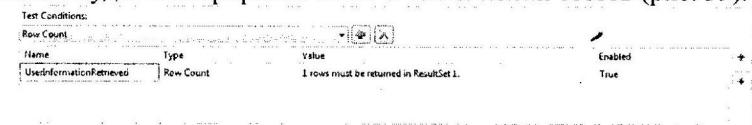


Рис. 38. Настройка условия корректности выполнения запроса

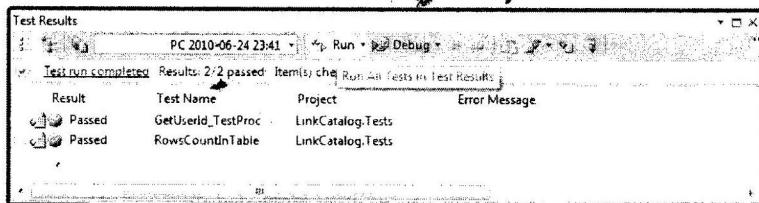


Рис. 39. Результаты выполнения тестов

Таким образом, могут быть разработаны и более сложные модульные тесты БД. Отметим, что среда разработки Visual Studio позволила обойтись при разработке модульных тестов БД без применения высокуюровневых языков программирования типа C# или C++. Тесты были созданы исключительно с применением встроенных оконных мастеров.

### Контрольные вопросы

1. Применима ли техника модульного тестирования к коду работы с базой данных?
2. Возможно ли модульное тестирование с применением языка SQL?
3. Можно ли проверить модульным тестом хранимую процедуру?

### Практическое задание

Разработать модульный тест для базы данных с применением Microsoft Visual Studio.

### Рекомендуемый порядок выполнения задания

1. Создать с помощью MS SQL Server Management Studio новую тестовую БД.
2. Создать в тестовой БД две-три хранимые процедуры.
3. Создать в Visual Studio новый тестовый проект (Test Project).
4. Добавить в тестовый проект модульный тест БД (Database Unit Test).
5. Реализовать не менее пяти тестирующих функций. При разработке этих функций использовать различные выборки и условия корректности выполнения тестов.
6. Запустить модульные тесты. Проанализировать результаты запуска.
7. Внести в одну из хранимых процедур изменения, приводящие к ошибкам.
8. Запустить тесты повторно, убедиться, что обнаружил ошибки модульный тест.

### ОГРАНИЧЕНИЯ МОДУЛЬНОГО ТЕСТИРОВАНИЯ

Как и любая технология тестирования, модульное тестирование не позволяет найти все ошибки программы. Это следует из практической невозможности трассировки всех возможных путей выполнения программы, за исключением простейших случаев. Например, каждое возможное значение булевой переменной потребует двух тестов: один на вариант TRUE, другой — на вариант FALSE. В результате на каждую строку исходного кода потребуется 3—5 строк тестового ко-

да. Кроме того, происходит тестирование каждого из модулей по отдельности. Это означает, что ошибки интеграции, системного уровня, функций, исполняемых в нескольких модулях, не будут определены. Также следует отметить, что данная технология плохо приспособлена для проведения тестов на производительность. Таким образом, модульное тестирование более эффективно при использовании в сочетании с другими методиками тестирования.

Для получения выгоды от модульного тестирования требуется строго следовать современной технологии тестирования во всём процессе разработки ПО. Нужно хранить не только записи обо всех проведённых тестах, но и обо всех изменениях исходного кода во всех модулях. С этой целью следует использовать систему контроля версий ПО. Таким образом, если более поздняя версия ПО не проходит тест, который был успешно пройден ранее, будет несложным отследить изменения в исходном коде и устранить ошибку. Также необходимо убедиться в неизменном отслеживании и анализе неудачных тестов. Игнорирование этого требования может привести к лавинообразному увеличению неудачных тестовых результатов. В связи с этим в некоторых методологиях разработки ПО явно указывается взаимосвязь модульного тестирования и систем управления версиями. Например, команды использующие методологию XP зачастую практикуют правило, что публикация кода в общее хранилище может быть осуществлена только тогда, когда все тесты выполняются правильно.

### Контрольные вопросы

1. Какие существуют ограничения техники модульного тестирования?
2. Что требуется, чтобы достичь выгоды от модульного тестирования?

### БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Котляров В.П., Коликова Т.В. Основы тестирования программного обеспечения М.: Интернет-университет информационных технологий, Бином. Лаборатория знаний, 2006. — 288 с.
2. Автоматизированное тестирование программного обеспечения / Э. Дастин и др. М.: ЛОРИ, 2003. — 592 с.
3. Бек К. Экстремальное программирование. Разработка через тестирование. СПб.: Питер, 2003. — 224 с.
4. Криспин Л., Грегори Д. Гибкое тестирование: практическое руководство для тестировщиков ПО и гибких команд = Agile Testing: A Practical Guide for Testers and Agile Teams. — М.: Вильямс, 2010. — 464 с. — (Addison-Wesley Signature Series).
5. Ошеров Р. The Art of Unit Testing: With Examples in .Net, 2009. — 320 с.

6. **Feathers M. and Lepiller B.** Cppunit cookbook  
[http://cppunit.sourceforge.net/doc/lastest/cppunit\\_cookbook.html](http://cppunit.sourceforge.net/doc/lastest/cppunit_cookbook.html)
7. **Гловер Э.** Переходим на JUnit 4  
<http://www.ibm.com/developerworks/ru/edu/j-dw-java-junit4.html>
8. **Документация** по NUnit, [www.nunit.org](http://www.nunit.org)
9. **Andrew Hunt, David Thomas.** *Pragmatic Unit Testing in C# with NUnit, 2nd Ed.* The Pragmatic Bookshelf, Raleigh 2007, ISBN 0-9776166-7-3
10. **Newkirk J., Vorontsov A.** *Test-Driven Development in Microsoft .NET*. Microsoft Press, Redmond 2004, ISBN 0-7356-1948-4
11. **Bill Hamilton:** *NUnit Pocket Reference*. O'Reilly, Cambridge 2004.
12. **Michaelis M.** A Unit Testing Walkthrough with Visual Studio Team Test  
[http://msdn.microsoft.com/en-us/library/ms379625\(VS.80\).aspx](http://msdn.microsoft.com/en-us/library/ms379625(VS.80).aspx)

## ОГЛАВЛЕНИЕ

Введение .....	3
Модульное тестирование .....	5
Методика разработки программ посредством тестирования .....	7
Модульное тестирование с применением CPPUNIT .....	9
Модульное тестирование с применением JUNIT .....	15
Модульное тестирование с применением NUNIT .....	23
Модульные тесты в Visual Studio Team System .....	32
Тестирование баз данных в Visual Studio Team System .....	41
Ограничения модульного тестирования .....	46
Библиографический список .....	47