Министерство образования и науки, молодежи и спорта Украины

Севастопольский национальный технический университет

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

к лабораторным работам № 5 – 8
по дисциплине
«Технология проектирования и разработки
программно-аппаратных систем»,
для студентов дневной формы обучения
направления 6.050101 – «Компьютерные науки»
Часть 2

Севастополь
2012

УДК 004.415.53
Методические указания к лабораторным работам по дисциплине «**Технология проектирования и разработки программно-аппаратных систем**»
для студентов дневной формы обучения направления 6.050101 – «Компьютерные науки»/**Сост. В. А. Строганов – Севастополь: Изд-во СевНТУ, 2012. – Ч. 2. – 32 с.**

Методические указания призваны обеспечить возможность выполнения студентами лабораторных работ по дисциплине «Технология проектирования и разработки программно-аппаратных систем».

Методические указания составлены в соответствии с требованиями программы дисциплины «Технология проектирования и разработки программно-аппаратных систем» для студентов направления 6.050101 и утверждены на заседании кафедры Информационных систем, протокол № от « » 20 г.

Допущено учебно-методическим центром СевНТУ в качестве методических указаний.

Рецензент: Сергеев Г. Г., канд. техн. наук, доцент кафедры КиВТ.

Содержание

Общие положения.....4
Лабораторная работа №5.....5
Лабораторная работа №6.....11
Лабораторная работа №7.....16
Лабораторная работа №8.....23
Библиографический список.....31

Общие положения

Целью лабораторных работ является получения практических навыков модульного и интеграционного тестирования программного обеспечения, а также профилирования программного кода.

Данный раздел лабораторного практикума представляет собой цикл из четырех лабораторных работ. В лабораторных работах № 5 и № 6 рассматриваются общие принципы модульного и интеграционного тестирования программного обеспечения. Лабораторная работа № 7 позволяет получить практические навыки модульного тестирования с использованием среды NUnit. В лабораторной работе № 8 рассматриваются основные принципы профилирования программного обеспечения на примере профилировщика EQATECProfiler.

В качестве лабораторной установки используется персональный компьютер и программное обеспечение: среда разработки Microsoft Visual Studio, среда тестирования NUnit, профилировщик EQATECProfiler. Порядок работы со средой тестирования и профилировщиком подробно рассмотрен в разделах 2.2 работ № 7 и № 8.

Время на выполнение лабораторных работ распределяется следующим образом: лабораторная работа № 5 – 4 часа, лабораторная работа № 6 – 3 часа, лабораторная работа № 7 – 2 часа, лабораторная работа № 8 – 2 часа. Варианты заданий соответствуют вариантам, выбранным при выполнении лабораторных работ из первой части методических указаний.

Объем работы распределяется следующим образом: в ходе домашней самостоятельной подготовки студенты изучают необходимый теоретический материал. В ходе аудиторных занятий выполняется написание необходимых программных модулей, тестирование (работы № 5 – 7) и профилирование (работа № 8) программ.

Результаты лабораторных работ оформляются студентом в виде отчета, включающего название работы, цель работы, постановку задачи, результаты работы в виде программного кода, графиков и словесного описания, а также выводы по результатам работы.

Лабораторная работа №5

Исследование методов модульного тестирования программного обеспечения

1. Цель работы

Исследовать основные подходы к модульному тестированию программного обеспечения. Приобрести практические навыки составления модульных тестов для объектно-ориентированных программ.

2. Основные положения

2.1. Общий порядок тестирования программного обеспечения

Цель тестирования программных модулей состоит в том, чтобы удостовериться, что каждый модуль соответствует своей спецификации. В процедурно-ориентированном программировании модулем называется процедура или функция, иногда группа процедур. Тестирование модулей обычно представляет собой некоторое сочетание проверок и прогонов тестовых случаев. Можно составить план тестирования модуля, в котором учесть тестовые случаи и построение тестового драйвера.

Тестирование классов аналогично тестированию модулей. Основным элементом объектно-ориентированной программы является класс. Тестирование класса предполагает его проверку на точное соответствие своей спецификации.

Существует два основных подхода к тестированию классов: просмотр (review) программного кода и тестовые прогоны.

Просмотр исходного кода ПО производится с целью обнаружения ошибок и дефектов, возможно, до того, как это ПО заработает. Просмотр кода предназначен для выявления таких ошибок, как неспособность выполнять то или иное требование спецификации или ее неправильное понимание, а также алгоритмических ошибок в реализации.

Тестовый прогон обеспечивает тестирование ПО в процессе выполнения программы. Осуществляя прогон программы, тестировщик стремится определить, способна ли программа вести себя в соответствии со спецификацией. Тестировщик должен выбрать наборы входных данных, определить соответствующие им правильные наборы выходных данных и сопоставить их с реально получаемыми выходными данными.

2.2. Порядок тестирования классов

Рассмотрим тестирование классов в режиме прогона тестовых случаев. После идентификации тестовых случаев для класса нужно реализовать *тестовый драйвер*, обеспечивающий прогон каждого тестового случая, и запротоколировать результаты каждого прогона. При тестировании классов тестовый драйвер создает один или большее число экземпляров тестируемого

класса и осуществляет прогон тестовых случаев. Тестовый драйвер может быть реализован как автономный тестирующий класс.

Тестирование классов выполняют, как правило, их разработчики. В этом случае время на изучение спецификации и реализации сводится к минимуму. Недостатком подхода является то, что если разработчик неправильно понял спецификации, то он для своей неправильной реализации разработает и "ошибочные" тестовые наборы.

В результате тестирования необходимо удостовериться, что программный код класса в точности отвечает требованиям, сформулированным в его спецификации, и что он не делает ничего более.

План тестирования или хотя бы тестовые случаи должны разрабатываться после составления полной спецификации класса. Разработка тестовых случаев по мере реализации класса помогает разработчику лучше понять спецификацию. Тестирование класса должно проводиться до того, как возникнет необходимость использовать этот класс в других компонентах ПО.

Регрессионное тестирование класса должно выполняться всякий раз, когда меняется реализация класса. Регрессионное тестирование позволяет убедиться в том, что разработанные и оттестированные функции продолжают удовлетворять спецификации после выполнения модификации ПО.

В модульном тестировании участвуют компоненты трех типов:

1) Модуль (unit) – наименьший компонент, который можно скомпилировать.

2) Драйверы тестов, которые запускают тестируемый элемент.

3) Программные заглушки – заменяют недостающие компоненты, которые вызываются элементом и выполняют следующие действия:

- возвращаются к элементу, не выполняя никаких других действий;
- отображают трассировочное сообщение и иногда предлагают тестировщику продолжить тестирование;
- возвращают постоянное значение или предлагают тестировщику самому ввести возвращаемое значение;
- осуществляют упрощенную реализацию недостающей компоненты;
- имитируют исключительные или аварийные ситуации.

В большинстве объектно-ориентированных языков члены класса имеют один из трех уровней доступа:

1) Public. Члены с доступом public доступны из любых классов.

2) Private. Члены с доступом private доступны только внутри самого класса, то есть из его методов. Они являются частью внутренней реализации класса и недоступны стороннему разработчику.

3) Protected. Члены с доступом protected доступны из самого класса и из классов, являющихся его потомками, но недоступны извне. Использование этих методов возможно только при создании класса-потомка, расширяющего функциональность базового класса.

Таким образом, необходимость тестирования функциональности класса зависит от того, предоставляется ли им возможность наследования. Если класс является законченным (final) и не предполагает наследования, необходимо тестирование его public части (впрочем, классы final не содержат protected

членов). Если же класс рассчитан на расширение за счет наследования, необходимо тестирование также его protected части.

Кроме того, во многих языках класс может содержать статические (static) члены, которые принадлежат классу в целом, а не его конкретным экземплярам. При наличии public static или protected static членов, кроме тестирования объектов класса, должно отдельно выполняться тестирование статической части класса.

Тестирование классов обычно выполняется путем разработки тестового драйвера, который создает экземпляры классов и окружает эти экземпляры соответствующей средой (тестовым окружением), чтобы стал возможен прогон соответствующего тестового случая. Драйвер посылает сообщения экземпляру класса в соответствии со спецификацией тестового случая, а затем проверяет исход этих сообщений. Тестовый драйвер должен удалять созданные им экземпляры тестируемого класса. Статические элементы данных класса также необходимо тестировать.

Существует несколько способов реализации тестового драйвера:

1) Тестовый драйвер реализуется в виде отдельного класса. Методы этого класса создают объекты тестируемого класса и вызывают их методы, в том числе статические методы класса. Таким способом можно тестировать public часть класса.

2) Тестовый драйвер реализуется в виде класса, наследуемого от тестируемого. В отличие от предыдущего способа, такому тестовому драйверу доступна не только public, но и protected часть.

3) Тестовый драйвер реализуется непосредственно внутри тестируемого класса (в класс добавляются диагностические методы). Такой тестовый драйвер имеет доступ ко всей реализации класса, включая private члены. В этом случае в методы класса включаются вызовы отладочных функций и агенты, отслеживающие некоторые события при тестировании.

В качестве примера рассмотрим тестирование класса `TCommand` следующего вида:

```
public class TCommand {
     public int NameCommand;
     public string GetFullName();
}
```

Спецификация тестового случая должна включать следующие пункты:

1) Название тестируемого класса: `TCommand`.

2) Название тестового случая: `TCommandTest1`.

3) Описание тестового случая – словесное описание логики теста с указанием тестовых данных: Тест проверяет правильность работы метода `GetFullName()` – получения полного названия команды на основе кода команды. В тесте подаются следующие значения кодов команд (входные значения): -1, 1, 2, 4, 6, 20, где -1 – запрещенное значение.

На основе спецификации создадим тестовый драйвер – класс `TCommandTester`, наследующий функциональность абстрактного класса

Tester.

```
    public class Log
    {
     //Создание лог файла
       static private StreamWriter log=new StreamWriter("log.log");
        static public void Add(string msg) //Добавление сообщения в
лог файл
       {
         log.WriteLine(msg);
       }

       static public void Close() //Закрыть лог файл
       {
         log.Close();
       }
    }

    abstract class Tester
    {
      protected void LogMessage(string s)
      //Добавление сообщения в лог-файл
        {
          Log.Add(s);
        }
    }

    class TCommandTester:Tester // Тестовый драйвер
    {
       TCommand OUT;

       public TCommandTester()
       {
         OUT = new TCommand();
         Run();
       }

       private void Run()
       {
         TCommandTest1();
       }

       private void TCommandTest1()
       {
         int[] commands = {-1, 1, 2, 4, 6, 20};
         for(int i=0;i<=5;i++)
          {
            OUT.NameCommand = commands[i];
                    LogMessage(commands[i].ToString()  +  "  :  "  +
OUT.GetFullName());
          }
       }
```

```
        static void Main()
        {
          TCommandTester CommandTester = new TCommandTester();
          Log.Close();
        }
}
```

Класс `TCommandTester` содержит метод `TCommandTest1()`, в котором реализована вся функциональность теста. В данном случае для покрытия спецификации достаточно перебрать следующие значения кодов команд: -1, 1, 2, 4, 6, 20, где -1 - запрещенное значение, и получить соответствующие им полное название команды с помощью метода `GetFullName()`. Пары соответствующих значений заносятся в *log*-файл для последующей проверки на соответствие спецификации.

Таким образом, для тестирования любого метода класса необходимо:

- Определить, какая часть функциональности метода должна быть протестирована, то есть при каких условиях он должен вызываться. Под условиями здесь понимаются параметры вызова методов, значения полей и свойств объектов, наличие и содержимое используемых файлов и т. д.
- Создать тестовое окружение, обеспечивающее требуемые условия.
- Запустить тестовое окружение на выполнение.
- Обеспечить сохранение результатов в файл для их последующей проверки.
- После завершения выполнения сравнить полученные результаты со спецификацией.

3. Порядок выполнения работы

3.1. Выбрать в качестве тестируемого один из классов, спроектированных в лабораторных работах №№ 1 – 4.

3.2. Составить спецификацию тестового случая для одного из методов выбранного класса, как показано в разделе 2.2.

3.3. Реализовать тестируемый класс и необходимое тестовое окружение на языке C#.

3.4. Выполнить тестирование с выводом результатов на экран и сохранением в *log*-файл.

3.5. Проанализировать результаты тестирования, сделать выводы.

4. Содержание отчета

4.1. Цель работы.

4.2. Постановка задачи, включающая описание обязанностей тестируемого класса.

4.3. Спецификация тестового случая.

4.4. Текст программы.

4.5. Выводы по работе.

5. Контрольные вопросы

5.1. Для чего применяется тестирование программного обеспечения?
5.2. Какие существуют разновидности тестирования?
5.3. Что такое модульное тестирование?
5.4. Что понимается под модулем при модельном тестировании?
5.5. Каков порядок мольного тестирования классов?
5.6. Какие разделы включает спецификация тестового случая?

Лабораторная работа № 6

Интеграционное тестирование программного обеспечения

1. Цель работы

Исследовать основные принципы интеграционного тестирования программного обеспечения. Приобрести практические навыки организации интеграционных тестов для объектно-ориентированных программ.

2. Основные положения

Основное назначение тестирования взаимодействий состоит в том, чтобы убедиться, что происходит правильный обмен сообщениями между объектами, классы которых уже прошли тестирование в автономном режиме (на модульном уровне тестирования).

Тестирование взаимодействия, или интеграционное тестирование, представляет собой тестирование собранных вместе, взаимодействующих модулей (объектов). В интеграционном тестировании можно объединять разное количество объектов – от двух до всех объектов тестируемой системы. При интеграционном тестировании используется подход «белого ящика». Целью интеграционного тестирования является только проверка правильности взаимодействия объектов, а не проверка правильности функционирования системы в целом.

2.1. Идентификация взаимодействий

Взаимодействие объектов представляет собой просто запрос одного объекта (отправителя) на выполнение другим объектом (получателем) одной из операций получателя и всех видов обработки, необходимых для завершения этого запроса.

В ситуациях, когда в качестве основы тестирования взаимодействий объектов выбраны только спецификации общедоступных операций, тестирование намного проще, чем когда такой основой служит реализация. В данной лабораторной работе мы ограничимся тестированием общедоступного интерфейса. Такой подход вполне оправдан, поскольку мы полагаем, что классы уже успешно прошли модульное тестирование. Тем не менее, выбор такого подхода отнюдь не означает, что не нужно возвращаться к спецификациям классов, дабы убедиться в том, что тот или иной метод выполнил все необходимые вычисления. Это обусловливает необходимость проверки значений атрибутов внутреннего состояния получателя, в том числе любых агрегированных атрибутов, т.е. атрибутов, которые сами являются объектами. Основное внимание уделяется отбору тестов на основе спецификации каждой операции из общедоступного интерфейса класса.

Взаимодействия неявно предполагаются в спецификации класса, в которой

установлены ссылки на другие объекты. Выявить такие взаимодействующие классы можно, используя отношения ассоциации, агрегирования и композиции, представленные на диаграмме классов. Связи такого рода преобразуются в интерфейсы класса, а тот или иной класс взаимодействует с другими классами посредством одного или нескольких способов:

1) Общедоступная операция имеет один или большее число формальных параметров объектного типа. Сообщение устанавливает ассоциацию между получателем и параметром, которая позволяет получателю взаимодействовать с этим параметрическим объектом.

2) Общедоступная операция возвращает значения объектного типа. На класс может быть возложена задача создания возвращаемого объекта, либо он может возвращать модифицированный параметр.

3) Метод одного класса создает экземпляр другого класса как часть своей реализации.

4) Метод одного класса ссылается на глобальный экземпляр некоторого другого класса. Разумеется, принципы хорошего тона в проектировании рекомендуют минимальное использование глобальных объектов. Если реализация какого-либо класса ссылается на некоторый глобальный объект, рассматривайте его как неявный параметр в методах, которые на него ссылаются.

2.2. Выбор тестовых случаев

Исчерпывающее тестирование, другими словами, прогон каждого возможного тестового случая, покрывающего каждое сочетание значений – это, вне всяких сомнений, надежный подход к тестированию. Однако во многих ситуациях количество тестовых случаев достигает таких больших значений, что обычными методами с ними справиться попросту невозможно. Если имеется принципиальная возможность построения такого большого количества тестовых случаев, на построение и выполнение которых не хватит никакого времени, должен быть разработан систематический метод определения, какими из тестовых случаев следует воспользоваться. Если есть выбор, то мы отдаем предпочтение таким тестовым случаям, которые позволяют найти ошибки, в обнаружении которых мы заинтересованы больше всего.

Существуют различные способы определения, какое подмножество из множества всех возможных тестовых случаев следует выбирать. При любом подходе мы заинтересованы в том, чтобы систематически повышать уровень покрытия.

2.3. Подробное описание тестового случая

Продемонстрируем тестирование взаимодействий на примере класса `TCommandQueue`:

```
public class TCommandQueue : System.Windows.Forms.ListBox
{
```

```
     public TCommandQueue()
     // Добавляет команду в очередь команд на указанную позицию
      public void AddCommand(int NameCommand, // Код команды
                                int Position)                  //
Позиция в очереди
      // Удаляет команду из очереди
     public void DeleteCommand(int Position)
      // Выполняет первую команду в очереди
     public void ProcessCommand()
    }
```

Класс реализует очередь FIFO объектов типа `TCommand`. Наследуется от `System.Windows.Forms.ListBox` библиотеки .NET. Количество команд в очереди не ограничено.

Операции:

- Конструктор `TCommandQueue()`.
- Операция `AddCommand(...)` создает объект типа `TCommand`, присваивает ему переданные параметры и добавляет в очередь команд на указанную позицию. Значение позиции в очереди = -1 означает, что команда будет добавлена в конец очереди.
- Операция `DeleteCommand(...)` удаляет команду из очереди на указанной позиции.
- Операция `ProcessCommand()` при наличии команд в очереди посылает первую команду из очереди на выполнение.

С объектом `TCommand` осуществляется взаимодействие третьего типа, т. е. `TCommandQueue` создает объекты класса `TCommand` как часть своей внутренней реализации.

Для тестирования взаимодействия класса `TCommandQueue` и класса `TCommand`, так же, как и при модульном тестировании, разработаем спецификацию тестового случая:

1) **Названия взаимодействующих классов**: `TCommandQueue`, `TCommand`.

2) **Название теста**: `TCommandQueueTest1`.

3) **Описание теста**: тест проверяет возможность создания объекта типа `TCommand` и добавления его в очередь при вызове метода `AddCommand()`.

4) **Начальные условия**: очередь команд пуста.

5) **Ожидаемый результат**: в очередь будет добавлена одна команда.

На основе этой спецификации был разработан тестовый драйвер – класс `TCommandQueueTester`, который наследуется от класса `Tester`. Этот класс содержит:

- Метод `Init()`, в котором создается объект класса `TCommandQueue`. Этот метод необходимо вызывать в начале каждого теста, чтобы тестируемые объекты создавались вновь:

```
    private void Init()
    {
     CommandQueue=new TCommandQueue();
```

```
}
```

- Методы, реализующие тесты. Каждый тест реализован в отдельном методе.

- Метод `Run()`, в котором вызываются методы тестов.

- Метод `dump()`, который сохраняет в *log*-файле теста информацию обо всех командах, находящихся в очереди в формате: «номер позиции в очереди: полное название команды».

- Точку входа в программу – метод `Main()`, в котором происходит создание экземпляра класса `TCommandQueueTester` и запуск метода `Run()`.

Сначала создадим тест, который проверяет, создается ли объект типа `TCommand`, и добавляется ли команда в конец очереди.

```
private void TCommandQueueTest1()
{
 Init();
 LogMessage("/////////// TCommandQueue Test1 /////////////");
 LogMessage("Проверяем, создается ли объект типа TCommand");
 // В очереди нет команд
 dump();
 // Добавляем команду
 // параметр = -1 означает, что команда должна быть добавлена
 //в конец очереди
 CommandQueue.AddCommand(1, -1);
 LogMessage("Command added");
 // В очереди одна команда
 dump();
}
```

Для выполнения этого теста в методе `Run()` необходимо вызвать метод `TCommandQueueTest1()` и запустить программу на выполнение:

```
private void Run()
{
  TCommandQueueTest1();
}
```

После завершения теста следует просмотреть текстовый журнал теста, чтобы сравнить полученные результаты с ожидаемыми результатами, заданными в спецификации тестового случая `TCommandQueueTest1`.

3. Порядок выполнения работы

3.1. Выбрать в качестве тестируемого взаимодействие двух или более классов, спроектированных в лабораторных работах №№1 – 4.

3.2. Составить спецификацию тестового случая.

3.3. Реализовать тестируемые классы и необходимое тестовое окружение на языке C#.

3.4. Выполнить тестирование с выводом результатов на экран и

сохранением в *log*-файл.

3.5. Проанализировать результаты тестирования, сделать выводы.

4. Содержание отчета

4.1. Цель работы.

4.2. Постановка задачи, включающая описание взаимодействия тестируемых классов.

4.3. Спецификация тестового случая.

4.4. Текст программы.

4.5. Выводы по работе.

5. Контрольные вопросы

5.1. Для чего применяется интеграционное тестирование программного обеспечения?

5.2. Какие существуют типы взаимодействий объектов?

5.3. Исходя из каких соображений выполняется выбор тестовых случаев при интеграционном тестировании?

5.4. Какие разделы включает спецификация тестового случая для интеграционного тестирования?

Лабораторная работа №7

Модульное тестирование программного обеспечения в среде NUnit

1. Цель работы

Исследовать эффективность использования методологии TDD при разработке программного обеспечения. Получить практические навыки использования фреймворка NUnit для модульного тестирования программного обеспечения.

2. Общие положения

2.1. Разработка через тестирование

Разработка через тестирование (Test-driven development, TDD) представляет собой одну из современных «гибких» (agile) методологий разработки программного обеспечения. Эта методология предполагает использование модульного тестирования для контроля разрабатываемого программного кода. Основная идея состоит в том, что модульные тесты разрабатываются до разработки программного кода, который они будут тестировать. Разработка таких тестов заставляет программиста подробно разобраться в требованиях к разрабатываемому модулю до начала разработки, четко определиться с конечными целями разработки; успешное выполнение тестов является критерием прекращения разработки. Разработку через тестирование можно представить в виде последовательности следующих основных этапов:

1) получение программы в непротиворечивом состоянии и набора успешно выполняемых модульных тестов;
2) разработка нового модульного теста;
3) максимально быстрая разработка минимального программного кода, позволяющего успешно выполнить весь набор тестов;
4) рефакторинг разработанного программного кода с целью улучшения его структуры и устранения избыточности;
5) контроль переработанного программного кода с помощью полного набора тестов.

В результате выполнения этих шагов в программу будет добавлена новая функциональность, а работоспособность модифицированной программы будет гарантироваться выполнением полного набора модульных тестов. Такой подход позволяет избежать ситуации, когда добавление новой функции нарушает работоспособность ранее разработанного кода.

Такой подход дает следующие преимущества:

- предотвращается возникновение ошибок во вновь разработанном коде;
- появляется возможность вносить изменения в существующий

программный код без риска нарушить его работоспособность, т. к. возникающие ошибки будут сразу же обнаружены модульными тестами;

- модульные тесты могут быть использованы в качестве документации к программному коду, показывать способы обращения к соответствующим программным модулям (объектам, методам и т.п.);
- улучшается дизайн кода: тесты заставляют создавать более «легкие» и независимые компоненты, которые проще поддерживать и модифицировать;
- повышается квалификация разработчиков, т. к. разработка качественных модульных тестов требует глубоких знаний ООП и паттернов проектирования.

Таким образом, применение методологии TDD позволяет создавать более качественный программный код, снижает вероятность возникновения ошибок и ускоряет процесс разработки.

К недостаткам разработки через тестирование можно отнести, во-первых, высокие требования к квалификации разработчиков. Еще одной проблемой является тестовое покрытие программного кода. В идеале набор модульных тестов должен обеспечивать покрытие 100% программного кода. Однако на практике этого достичь не удается, особенно в тех случаях, когда TDD применяется для модификации уже существующего программного обеспечения. Поэтому необходимо обеспечить покрытие тестами наиболее критических модулей, а также компонентов, которые будут подвергаться рефакторингу.

2.2. Тестирование с помощью NUnit

NUnit представляет собой открытый фреймворк для тестирования приложений под Microsoft .NET Framework. NUnit включает библиотеки для написания тестов, а также ПО для выполнения этих тестов.

Рассмотрим порядок создания и выполнения модульных тестов с помощью NUnit.

Для примера создадим dll-библиотеку, содержащую класс, описывающий простой калькулятор с возможностью выполнения основных арифметических операций. Для разработки будем использовать среду Microsoft Visual Studio.

Создадим новый проект: меню File/New Project/ Class Library.

Созданный проект содержит один пустой файл Class1.cs. Переименуем его в ICalc.cs и опишем в этом файле интерфейс `ICalc`, который включает основные арифметические операции:

```
namespace CalcLib
{
    public interface ICalc
    {
        double Add(double a, double b);
        double Subtract(double a, double b);
        double Multiply(double a, double b);
        double Divide(double a, double b);
    }

}
```

Теперь добавим к проекту файл Calc.cs, в котором опишем класс `Calc` (пункт меню Project/Add Class или сочетание клавиш Shift + Alt + C):

```
using System;

namespace CalcLib
{
    public class Calc : ICalc
    {
        public double Add(double a, double b)
        {
            throw new NotImplementedException();
        }

        public double Subtract(double a, double b)
        {
            throw new NotImplementedException();
        }

        public double Multiply(double a, double b)
        {
            throw new NotImplementedException();
        }

        public double Divide(double a, double b)
        {
            throw new NotImplementedException();
        }
    }
}
```

Видно, что класс `Calc` реализует интерфейс `ICalc`. Методы класса пока не реализованы, т. к. в соответствии с методологией TDD сначала должны быть реализованы тесты, а потом – код, который будет ими тестироваться.

Для того чтобы можно было использовать возможности NUnit при написании тестов, необходимо добавить в проект ссылку на сборку `nunit.framework` (меню Project/Add Reference):

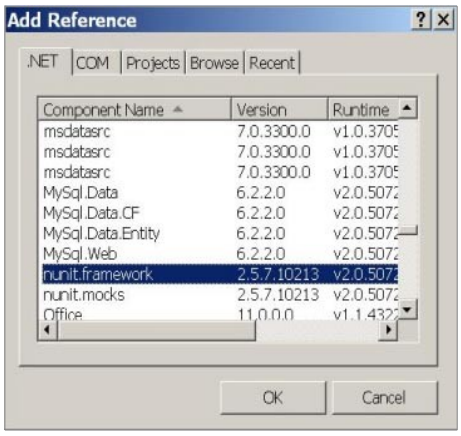

Рисунок 2.1 – Добавление ссылки на сборку

Добавим в проект класс CalcTest, который будет содержать модульные тесты для класса Calc:

```
using System;
using NUnit.Framework;

namespace CalcLib
{
    [TestFixture]
    public class CalcTest
    {
        public void AddTest()
        {
            ICalc calculator = new Calc();
            double actualVal = calculator.Add(2, 2);
            double expected = 2 + 2;
            Assert.AreEqual(expected, actualVal);
       }
    }
}
```

Класс CalcTest имеет атрибут [TestFixture], который указывает среде выполнения NUnit на то, что данный класс будет содержать модульные тесты. Он должен быть объявлен с модификатором public и иметь конструктор по умолчанию.

Каждый отдельный тест должен иметь атрибут [Test]. Метод, который описывает тест, обязан возвращать void, быть доступным извне и не иметь входных параметров.

В нашем случае в методе AddTest() для проверки правильности работы метода Add() класса Calc создается экземпляр калькулятора, выполняется операция сложения и полученный результат сравнивается с ожидаемым (результатом выполнения одноименной операции в языке C#). Класс Assert, использованный в этом примере, содержит набор методов, позволяющих выполнять различные проверки данных. Если данные неверны, метод оповещает среду выполнения NUnit, что тест не пройден. Ниже приведены некоторые проверки, доступные в классе Assert:

1. Assert.AreEqual – проверяет равенство входных параметров;
2. Assert.AreNotEqual – проверяет то, что входные параметры неравны;
3. Assert.AreSame – проверка на то, что входные параметры ссылаются на один и тот же объект;
4. Assert.AreNotSame – входные параметры не ссылаются на один и тот же объект;
5. Assert.Contains – метод получает на входе объект и коллекцию и проверяет, что данный объект содержится в это коллекции;
6. Assert.IsNull – входной параметр – null;
7. Assert.IsEmpty – входной параметр – пустая коллекция.
8. Assert.Fail – прерывает выполнение теста и среде NUnit, что тест не пройден. Эта функция может быть использована, когда нужно организовать

более сложные типы проверок.

Добавим в класс `CalcTest` модульные тесты для остальных методов класса `Calc`:

```
        [Test]
        public void SubtractTest()
        {
            ICalc calculator = new Calc();
            double actual = calculator.Subtract(2, 2);
            double expected = 2 - 2;
            Assert.AreEqual(expected, actual);
        }

        [Test]
        public void MultiplyTest()
        {
            ICalc calculator = new Calc();
            double actual = calculator.Multiply(2, 2);
            double expected = 2 * 2;
            Assert.AreEqual(expected, actual);
        }

        [Test]
        public void DivideTest()
        {
            ICalc calculator = new Calc();
            double actual = calculator.Divide(2, 2);
            double expected = 2 / 2;
            Assert.AreEqual(expected, actual);
        }
```

После того как написаны модульные тесты и проект скомпилирован (меню Build/Build Solution или F6), можно переходить к тестированию.

Для запуска тестов используется среда NUnit. После запуска среды NUnit необходимо загрузить тестируемый проект (File/Open Project или сочетание клавиш Ctrl + O). В нашем случае это будет библиотека CalcLib.dll.

В левой части программного окна отображается древовидная структура, включающая проект (CalcLib), наборы тестов TextFicture (CalcTest) и собственно отдельные модульные тесты (AddTest, DivideTest, MultiplyTest, SubtractTest). После выполнения тестов в правой части окна отображаются результаты тестирования. В том случае, если тесты выполнились неудачно, есть возможность просмотреть сообщения об ошибке и исходный код неудачного теста и тестируемого им модуля.

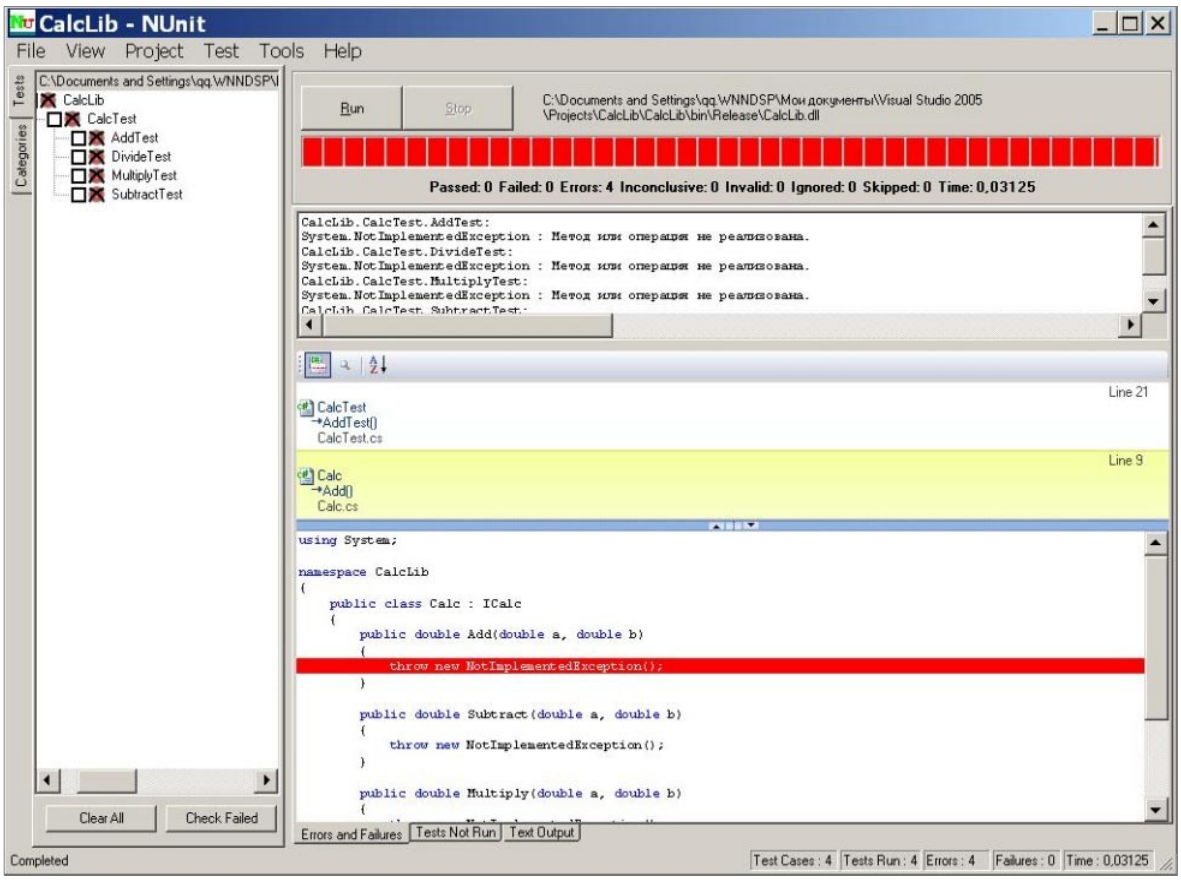

Рисунок 2.2 – Выполнение тестов в NUnit

При написании тестов можно использовать еще ряд атрибутов. Атрибут `[SetUp]` помечает метод, который будет выполнен перед исполнением каждого теста из набора. Например, следующий метод позволяет вывести время начала каждого теста:

```
[SetUp]
public void TestSetup()
{
    Trace.WriteLine("Test started at " + DateTime.Now);
}
```

(Для использования класса `Trace` необходимо добавить директиву `using System.Diagnostics;`)

Метод, помеченный атрибутом `[TestFixtureSetUp]`, выполняется один раз, перед запуском всего набора тестов:

```
[TestFixtureSetUp]
public void TestKitSetup()
{
    Trace.WriteLine("Initialized at " + DateTime.Now);
}
```

Аналогично, атрибуты `[TearDown]` и `[TestFixtureTearDown]` помечают методы, выполняющиеся после каждого теста и после всего набора тестов соответственно. Например:

```
[TearDown]
public void TestDispose()
```

```
    {
        Trace.WriteLine("Test finished at " + DateTime.Now);
    }

    [TestFixtureTearDown]
    public void TestKitDispose()
    {
        Trace.WriteLine("Completed at " + DateTime.Now);
    }
```

Если тест помечен атрибутом `[Ignore]`, то он игнорируется.

3. Порядок выполнения работы

3.1. Реализовать на языке C# один из классов, спроектированных в лабораторной работе № 1. Методы класса при этом не реализовывать.

3.2. Разработать для созданного класса набор модульных тестов, включающий тесты для каждого метода.

3.3. Запустить набор тестов, проанализировать и сохранить результаты.

3.4. Поочередно реализовать методы класса, выполняя тестирование при каждом изменении программного кода.

3.5. После того, как весь набор тестов будет выполняться успешно, реализацию классов можно считать завершенной.

4. Содержание отчета

4.1. Цель работы.
4.2. Постановка задачи.
4.3. Описание обязанностей тестируемого класса.
4.4. Программный код тестируемого класса и тестов.
4.5. Результаты тестирования, полученные в процессе разработки.
4.5. Выводы по работе.

5. Контрольные вопросы

5.1. В чем заключаются основные принципы методологии TDD (Разработка через тестирование)?
5.2. Какие можно выделить достоинства и недостатки подхода TDD?
5.3. В чем состоит общий порядок модульного тестирования с помощью NUnit.
5.4. Каково назначение основных атрибутов NUnit?
5.5. Какие типы проверок реализованы в классе Assert фреймворка NUnit?

Лабораторная работа № 8

Исследование способов профилирования программного обеспечения

1. Цель работы

Исследовать критические по времени выполнения участки программного кода и возможности их устранения. Приобрести практические навыки анализа программ с помощью профайлера EQATECProfiler.

2. Основные положения

2.1. Профилирование программного обеспечения

После разработки программного кода, удовлетворяющего спецификации и модульным тестам, возникает задача оптимизации программы по таким параметрам, как производительность, расход памяти и т. п. Процесс анализа характеристик системы называется *профилированием* и выполняется с помощью специальных программных инструментов – так называемых *профилировщиков*, или *профайлеров* (profiler). Программа-профайлер выполняет запуск программы и анализирует характеристики выполнения. Процесс профилирования сводится к декомпозиции программы на отдельные выполняющиеся элементы и измерению времени, затрачиваемого на выполнение каждого элемента, и расходования памяти. В результате профилирования может быть построен *граф вызовов*, показывающий последовательность вызовов функций, а также определено время выполнения каждой функции и ее вклад в общее время выполнения программы. Анализ этой информации позволяет выявить *критические участки программного кода* (хот-споты, hot-spot), т. е. «узкие места», на которые следует в первую очередь обратить внимание при оптимизации программы.

2.2. Порядок работы с профайлером EQATECProfiler

EQATECProfiler представляет собой свободно распространяемый профайлер для приложений под .NET. Данный инструмент позволяет выполнять анализ производительности программы, строить *граф вызовов методов* (method-call graph), анализировать время выполнения программы и относительный вклад времени выполнения вызываемых функций.

Рассмотрим работу профайлера на следующем примере. Разработаем программу, вычисляющую площадь и периметр прямоугольника с заданными сторонами. Для этого воспользуемся библиотекой `CalcLib`, разработанной в предыдущей работе.

На базе класса `Calc` реализуем класс `Rect`, описывающий прямоугольник и предоставляющий функциональность для вычисления его площади и периметра. Для этого добавим в проект новый класс (пункт меню Project/Add

Class), назовем файл *Rect.cs* и поместим в него следующее описание класса:

```
using System;
using System.Text;

namespace CalcLib
{
    class Rect
    {
        private double rectWidth;
        private double rectHeight;

        public double Width
        {
            get { return rectWidth; }
            set { rectWidth = value; }
        }

        public double Height
        {
            get { return rectHeight; }
            set { rectHeight = value; }
        }

        public Rect(double aWidth, double aHeight)
        {
            rectWidth = aWidth;
            rectHeight = aHeight;
        }

        public double getArea()
        {
            ICalc calc = new Calc();
            double areaValue = calc.Multiply(rectWidth,
rectHeight);
            return areaValue;
        }

        public double getPerimeter()
        {
            ICalc calc = new Calc();
            double perimeterValue =
calc.Add(calc.Multiply(rectWidth, 2), calc.Multiply(rectHeight,
2));
            return perimeterValue;
        }
    }
    }
```

Рассмотрим текст класса подробней. Класс содержит два приватных поля `rectWidth` и `rectHeight`. Принцип инкапсуляции предполагает, что к внутренним переменным объекта нельзя получить доступ непосредственно.

Единственный способ изменить состояние объекта – это использовать его интерфейс. Поэтому для изменения и получения значений полей класса созданы *свойства* (property):

```
public double Width
{
    get { return rectWidth; }
    set { rectWidth = value; }
}

public double Height
{
    get { return rectHeight; }
    set { rectHeight = value; }
}
```

Свойства позволяют ужесточить контроль доступа к внутреннему состоянию объекта, организовать проверку корректности присваиваемых данных, отложить вычисление значений полей до тех пор, пока они не будут запрошены клиентом и т. п. Для клиента же свойство выглядит как обычное поле со свободным доступом.

После того, как реализованы необходимые классы, перейдем к реализации приложения. Создадим класс `MainClass`, который будет содержать статический метод `Main()`. Этот метод будем использовать в качестве *входной точки* (entry point) приложения, т.е. функции, которая запускается первой при запуске программы (аналогично функции `main()` в языке C++):

```
using System;
using System.Collections.Generic;
using System.Text;

namespace CalcLib
{
    class MainClass
    {
        public static void Main()
        {
            Rect rect = new Rect(5, 8);

            double rectArea = rect.getArea();
            double rectPerimeter = rect.getPerimeter();

            Console.WriteLine("Rectangle " + rect.Width + " x " +
rect.Height + " has area " + rectArea + " and perimeter " +
rectPerimeter);
        }
    }
}
```

Преобразуем библиотеку CalcLib в запускаемое приложение. Для этого изменим свойства проекта (пункт меню Project/ CalcLib Properties): свойство

Outout Type установим в «Console Application» и в качестве Startup Object укажем CalcLib.MainClass.

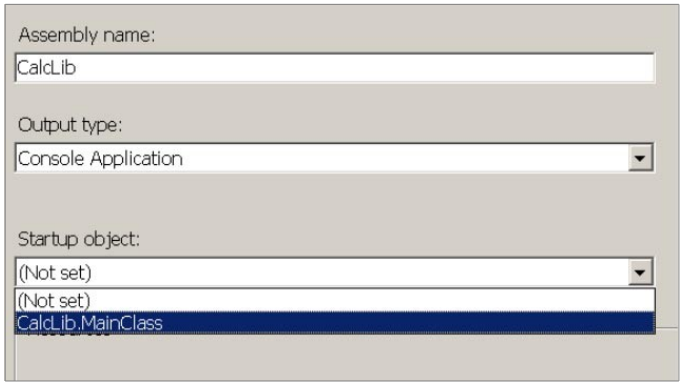

Рисунок 2.1 – Задание параметров проекта

После сохранения настроек приложение можно скомпилировать (клавиша F6) и запустить (клавиша F5).

Для наглядности профилирования изменим методы класса `Calc`, искусственно добавив в них задержку:

```
using System;
using System.Threading;

namespace CalcLib
{
    public class Calc : ICalc
    {
        public double Add(double a, double b)
        {
            Thread.Sleep(50);
            return a + b;
        }

        public double Subtract(double a, double b)
        {
            Thread.Sleep(50);
            return a - b;
        }

        public double Multiply(double a, double b)
        {
            Thread.Sleep(100);
            return a * b;
        }

        public double Divide(double a, double b)
        {
            Thread.Sleep(100);
            return a / b;
        }
```

```
    }
}
```

Перейдем к собственно профилированию разработанной программы. Во вкладке Build программы EQATECProfiler необходимо выбрать папку, в которой расположена скомпилированная программа (.exe-файл). После загрузки следует выбрать те модули, которые будем профилировать.

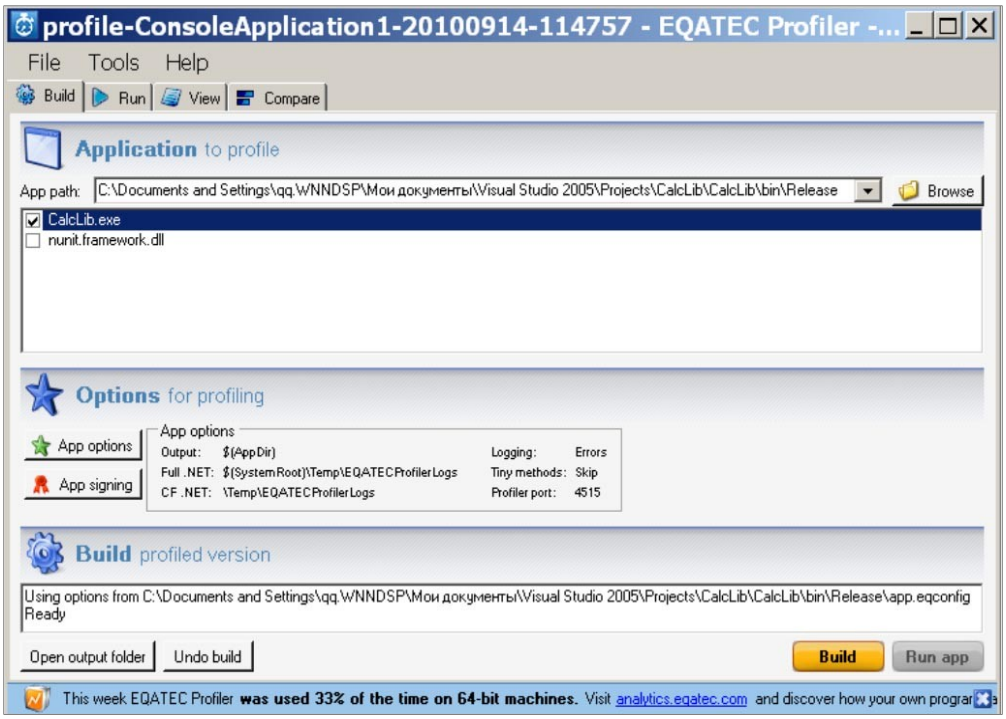

Рисунок 2.2 – Выбор программы для профилирования

Затем нажимаем кнопку «Build». В результате профайлер добавит в нашу программу дополнительные служебные инструкции. После этого нажатие кнопки «Run app» запускает процесс профилирования. По его окончании создается отчет. Список отчетов показан во вкладке «Run». Выбрав нужный (последний по порядку) отчет, переходим на вкладку «View». Здесь в таблице представлены параметры выполнения функций программы. Ниже построен граф вызовов для выбранной функции (на рисунке 2.3 – для метода `MainClass.Main()`). В вершинах графа, соответствующих каждой из вызываемых функций, показано время выполнения в миллисекундах, а также относительный вклад (в процентах) этой функции в общее время выполнения вызывающей функции. Анализ этих данных позволяет выявить «узкие места» программы. Например, в нашем случае видно, что наибольшее время выполнения имеет метод `Rect.getPerimeter()`, а в этом методе, в свою очередь, узким местом является метод `Calc.Multiply()`.

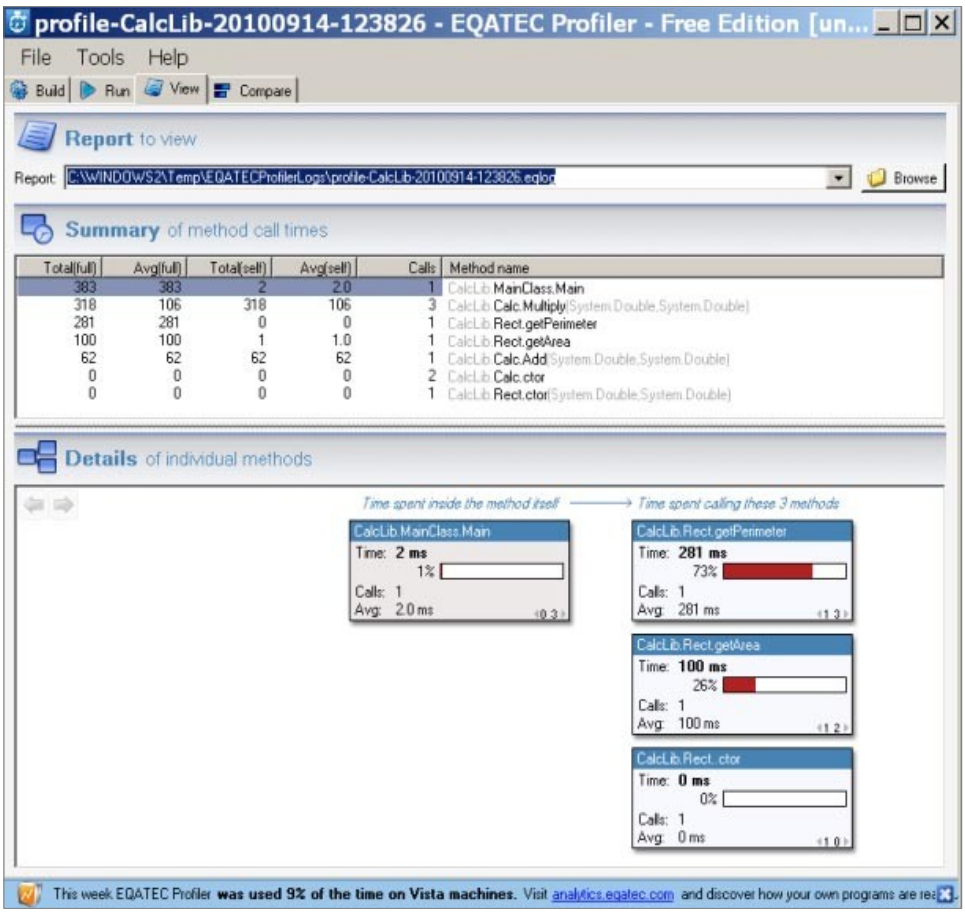

Рисунок 2.3 – Просмотр результатов профилирования

С целью оптимизации времени выполнения программы перепишем метод `Rect.getPerimeter()`, заменив в нем операции умножения операциями сложения:

```
public double getPerimeter()
{
    ICalc calc = new Calc();
    double perimeterValue = calc.Add(calc.Add(rectWidth,
rectWidth), calc.Add(rectHeight, rectHeight));
    return perimeterValue;
}
```

Скомпилировав приложение, повторив в EQATECProfiler операции «Build», «Run app» и просматривая результаты, видим, что такая модификация кода привела к сокращению общего времени выполнения программы. При этом уменьшился вклад функции `Rect.getPerimeter()` в общее время выполнения.

Для сравнения профилей программы до и после оптимизации выбираем на вкладке Run два отчета и нажимаем кнопку «Compare». В результате строится таблица, в которой приводятся как абсолютные показатели старого и нового варианта программы (время выполнения Old Avg и New Avg, количество вызовов Old Calls и New Calls), так и изменения нового варианта по сравнения со старым (относительное изменение времени выполнения функций Speedup,

абсолютное изменение времени выполнения Diff Avg, изменение количества вызовов функций Diff Calls).

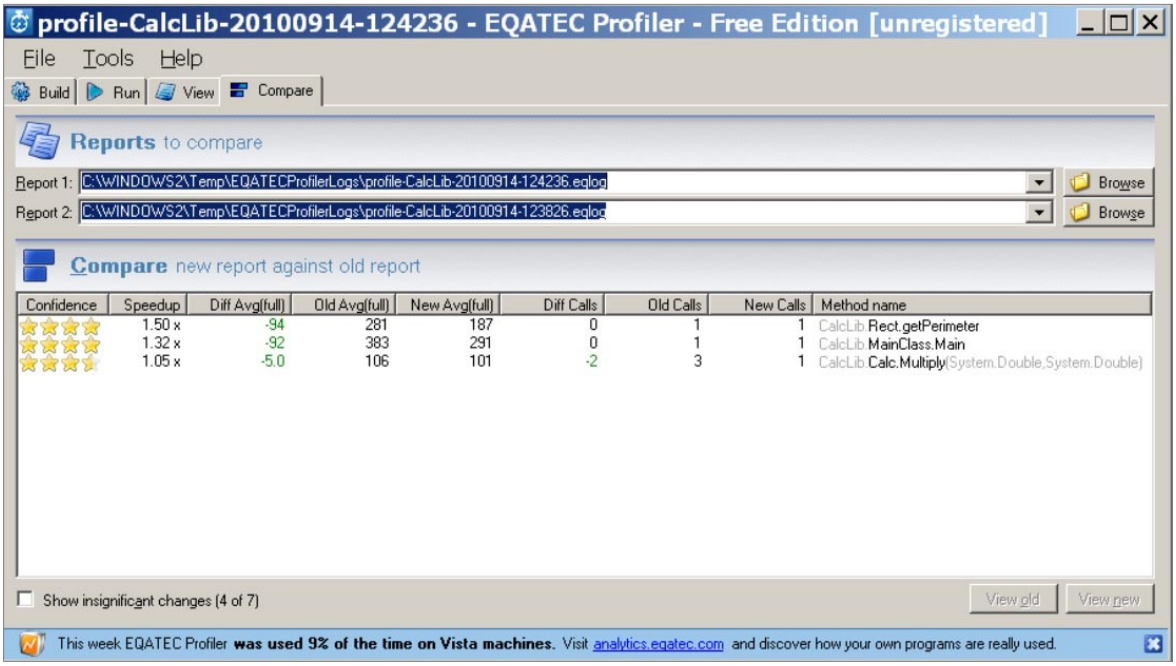

Рисунок 2.4 – Сравнение профилей программы

3. Порядок выполнения работы

3.1. Разработать программу на основе библиотеки классов, реализованной и протестированной в предыдущей работе. Программа должна как можно более полно использовать функциональность класса. При необходимости для наглядности профилирования в методы класса следует искусственно внести задержку выполнения.

3.2. Выполнить профилирование разработанной программы, выявить функции, на выполнение которых тратится наибольшее время.

3.3. Модифицировать программу с целью оптимизации времени выполнения.

3.4. Выполнить повторное профилирование программы, сравнить новые результаты и полученные ранее, сделать выводы.

4. Содержание отчета

4.1. Цель работы.

4.2. Постановка задачи.

4.3. Текст первоначального варианта программы.

4.4. Результаты профилирования первоначального варианта программы с подробным анализом «узких мест».

4.5. Текст модифицированного варианта программы со словесным описанием и обоснованием внесенных изменений.

4.6. Результаты профилирования модифицированного варианта программы, их сравнение с результатами профилирования первоначального варианта программы.

4.7. Выводы по работе.

5. Контрольные вопросы

5.1. В чем заключается процесс профилирования программного обеспечения?

5.2. Какие разновидности «узких мест» программы определяются с помощью профилирования?

5.3. Что такое граф вызовов программы, для чего он строится?

5.4. В чем заключается общий порядок профилирования программ с помощью EQATECProfiler?

Библиографический список

1. Котляров В. П. Основы тестирования программного обеспечения: Учебное пособие / В. П. Котляров, Т. В. Коликова.– М.: Интернет-Университет Информационных технологий; БИНОМ. Лаборатория знаний, 2006.– 285 с.

Заказ №_____ от « » _____ 2012 г. Тираж _____ экз.

Изд-во СевНТУ