Лабораторная работа 3 – Применение модели грид-заданий для внедрения в грид существующего приложения

Введение 2
Цель лабораторной работы 2
Упражнение 1 – Знакомство с компилятором Microsoft Visual C# 3
Настройка переменных окружения 3
Параметры входных и выходных файлов 4
Параметры ресурсов 4
Параметры генерирования кода и вывода предупреждений 5
Параметры языка программирования 5
Упражнение 2 – Запуск компиляции на локальной машине 5
Задание 1 – Открытие проекта RemoteCompiler 5
Задание 2 – Реализация обработки грид-заданий на локальной машине 6
Задание 3 – Тестирование обработки грид-заданий на локальной машине 9
Задание 4 – Реализация обработки грид-заданий на удаленной машине 10
Дополнительные упражнения ..... **Ошибка! Закладка не определена.**
Заключение ..... **Ошибка! Закладка не определена.**
Литература 12

1. Введение

В первых двух лабораторных работах была подробно освящена основная модель программирования инструментария Alchemi – модель *грид-потоков*. Мы рассмотрели основные принципы разработки *новых* программ для данного инструментария, а также ряд вопросов, касающихся внедрения в грид *существующих* приложений[1].

Как мы выяснили, процесс внедрения существующих приложений в грид с помощью инструментария Alchemi является довольно простым, прозрачным и экономным. Напомним, что основная идея процесса внедрения приложений в грид состоит в том, чтобы распределить обработку большого набора входных данных на несколько компьютеров сети. Однако не трудно заметить, что для запуска существующей программы в грид (а в качестве такой программы выступал пакет трассировки лучей POV-Ray) нам пришлось реализовать некоторые *общие* сервисные операции, связанные с копированием входных файлов на удаленную машину, передачей параметров, получением выходных файлов и некоторые другие. Очевидно, что подобные сервисные операции необходимы для внедрения в грид *любого* приложения, поэтому реализовывать их всякий раз представляется неразумным. Естественным выходом из этой ситуации является вынесение данных сервисных операций в стандартную функциональность Alchemi API, что позволит разработчику полностью сконцентрироваться на основной логике приложения и значительно сэкономить время. Именно таким образом поступили разработчики инструментария Alchemi, определив такую сущность, как *грид-задание* (grid job). По-существу, грид-задание представляем собой *специализированный* грид-поток, который инкапсулирует все необходимые сервисные операции для запуска приложений на удаленных узлах. Данная лабораторная работа посвящена применению модели грид-заданий для внедрения в грид существующих приложений.

В качестве программы для внедрения выберем компилятор для языка программирования Microsoft Visual C#, который входит в стандартную поставку платформы Microsoft .NET Framework. К сожалению, процесс компиляции в общем случае трудно распределить на несколько вычислительных узлов, поэтому компиляция будет целиком осуществляться на одной удаленной машине. Данный подход оказывается полезным при компиляции больших объемов исходных текстов, обработка которых на обычной машине пользователя занимает слишком много времени (вплоть до нескольких часов). Для этой цели можно выделить специальный производительный вычислительный ресурс, доступ к которому будет осуществляться с помощью разработанной программы.

Заметим, что читатель может предложить свою программу для внедрения в грид, и проделать для нее те же самые шаги, что описаны в данной лабораторной работе. Подобный вариант будет даже более предпочтительным по сравнению с предложенным, если выбранная программа допускает распределение вычислений на несколько компьютеров сети.

2. Цель лабораторной работы

Целью лабораторной работы является внедрение существующего приложения в грид, построенную на базе инструментария Alchemi, с использованием модели грид-заданий. В качестве такого приложения выбирается компилятор для языка программирования Microsoft Visual C#. Разработанное приложение должно запускать процесс компиляции на удаленной машине и загружать результаты его работы.

- *Упражнение 1*.

  Знакомство с компилятором Microsoft Visual C#.

- *Упражнение 2*.

  Разработка программы для удаленной компиляции.

Примерное время выполнения лабораторной работы: **180 минут**.

При выполнении лабораторной работы предполагаются знания раздела “Инструментарий Alchemi” учебного курса по технологиям грид, а также базовые знания языка программирования Visual C# и навыки работы в среде Microsoft Visual Studio. Весьма полезным было бы знакомство со второй лабораторной работой, однако возможно и отдельное использование данной лабораторной работы, поскольку необходимые основные положения будут изложены повторно.

---

[1] Под внедрением в грид некоторого приложения здесь подразумевается организация распределенной обработки данных на вычислительных ресурсах грид с помощью данного приложения. После завершения обработки данных пользователь должен получить итоговый результат как единое целое. Таким образом, для пользователя распределенные вычислительные ресурсы выступают в роли *единого виртуального* вычислительного ресурса. Заметим, что в частном случае обработка данных может осуществляться на единственном грид-узле (например, на удаленном кластере или суперкомпьютере).

3. Упражнение 1 – Знакомство с компилятором Microsoft Visual C#

В данном упражнении описывается процесс сборки приложений с помощью компилятора командной строки для языка программирования Microsoft Visual C#. Данный компилятор входит в стандартную поставку платформы Microsoft .NET Framework и называется **csc.exe**. Мы рассмотрим ряд возможностей компилятора, которые позволят с легкостью собирать проекты, состоящие из одного или нескольких файлов, без использования интегрированной среды разработки.

Компилятор Microsoft Visual C# поддерживает большое количество различных параметров, которые влияют на процесс сборки приложения. На высоком уровне данные параметры можно разбить на следующие категории:

- *Выходные файлы.*

  Определяют тип выходных бинарных файлов, позволяют сгенерировать опциональный файл документации в формате XML.

- *Входные файлы*.

  Позволяют указать входные файлы с исходными текстами, а также ссылки на другие сборки, необходимые во время компиляции.

- *Ресурсы*.

  Используются для добавления к сборке любых необходимых ресурсов, таких как пиктограммы и таблицы строк.

- *Генерирование кода.*

  Позволяют добавить вывод отладочной информации и включить оптимизацию кода.

- *Ошибки и предупреждения.*

  Определяют, каким образом компилятор обрабатывает ошибки и предупреждения во время компиляции.

- *Язык*.

  Позволяют включить или отключить различные возможности языка программирования Microsoft Visual C#.

- *Дополнительно.*

  Некритичные для большинства случаев вспомогательные опции.

Общий формат использования компилятора Microsoft Visual C# имеет следующий вид:

```
csc /Параметр:Значение /Параметр:Значение /Параметр ... FirstClass.cs ... SecondClass.cs
```

Некоторые параметры могут не иметь значений, и в этом случае символ двоеточия опускается. Список компилируемых исходных файлов обычно перечисляется в конце команды. Далее мы рассмотрим некоторые важные опции большинства перечисленных категорий, которые потребуется использовать при выполнении лабораторной работы.

3.1. Настройка переменных окружения

Для того чтобы использовать инструменты, входящие в состав платформы Microsoft .NET Framework (в том числе компилятор Microsoft Visual C#), из любой командной строки, необходимо вручную модифицировать переменную окружения **PATH**, добавив в нее путь к соответствующим исполняемым файлам. Для этого щелкните правой кнопкой мышки на значке **My Computer** (**Мой Компьютер)** и выберите пункт меню **Properties** (**Свойства**). Далее в появившемся диалоговом окне перейдите на вкладку **Advanced** (**Дополнительно**) и щелкните на кнопке **Environment Variables** (**Переменные среды**). Затем в списке **System variables** (**Системные переменные**) выберите переменную окружения **PATH** и добавьте к ней путь к установочному каталогу платформы Microsoft .NET Framework, который обычно имеет вид **C:\Windows\Microsoft.NET\Framework\v2.0.xxxxx** (последние пять цифр соответствуют используемой версии инструментария). После обновления переменной **PATH** следует закрыть все диалоговые окна и открытые командные строки, для того чтобы изменения вступили в силу. Для тестирования установок достаточно запустить командную строку и выполнить одну из двух команд:

```
csc /help
ildasm /help
```

Если в результате выполнения данных команд на экране появилось много различной информации, то изменения вступили в силу.

3.2. Параметры входных и выходных файлов

Перейдем к описанию параметров компилятора Microsoft Visual C#. Начнем с описания параметров, определяющих входные и выходные файлы.

- *Входные файлы*
  - **`/recurse`** – включить компилирование файлов во всех дочерних каталогах проекта
  - **`/reference`** – задать ссылки на другие сборки, необходимые для компиляции
- *Выходные файлы*
  - **`/out`** – задать имя генерируемого выходного файла
  - **`/target`** – задать тип выходного файла
  - **`/doc`** – включить генерирование опционального файла документации

Наиболее важной опцией из перечисленных выше является опция **`/target`**, которая позволяет специфицировать тип выходного файла. Данная опция может принимать следующие значения:

- *Тип сборки*
  - **`exe`** – создать консольное приложение (предполагается по умолчанию)
  - **`winexe`** – создать графическое приложение на основе библиотеки Windows Forms
  - **`library`** – создать динамическую библиотеку
  - **`module`** – создать модуль, который может быть добавлен в другую сборку

Для демонстрации перечисленных опций скомпилируем воображаемый файл **TestType.cs** в динамическую библиотеку. Предположим, что в данном файле используется библиотека **TestLib.dll**, а для кода необходимо сгенерировать файл документации. Перейдем в каталог с компилируемым файлом и затем воспользуемся следующей командой:

```
csc /target:library /reference:TestLib.dll /out:Lib.dll /doc:Doc.xml TestType.cs
```

В результате в рабочем каталоге появится сгенерированная динамическая библиотека **Lib.dll** и файл документации **Doc.xml**.

Обычно при разработке приложений создается некоторая логическая структура каталогов проекта. Предположим, что исходные файлы некоторого проекта располагаются в директориях **FirstDir** и **SecondDir** и для работы приложения требуется библиотека **TestLib.dll**. Будем считать, что оба каталога и библиотека располагаются в некотором корневом каталоге. Тогда для компиляции проекта в консольное приложение необходимо перейти в данный корневой каталог и воспользоваться командой:

```
csc /target:exe /out:App.exe /reference:TestLib.dll /recurse:FirstDir /recurse:SecondDir *.cs
```

Обратите внимание, что в данной команде мы не указывали конкретные файлы для компиляции, а воспользовались символом обобщения “*”. В результате будут скомпилированы все исходные файлы в корневом и указанных дочерних каталогах.

3.3. Параметры ресурсов

Рассмотрим теперь параметры компилятора, позволяющие использовать различные виды ресурсов (такие как изображения и таблицы строк) в приложениях.

- *Ресурсы*
  - **`/win32res`** – задать файл ресурсов Win32 (в формате **.res**)
  - **`/win32icon`** – использовать заданный значок для вывода
  - **`/resource`** – внедрить указанный ресурс
  - **`/linkresource`** – компоновать указанный ресурс вместе с этой сборкой

Предположим, что некоторый файл **TestType.cs** требуется скомпилировать как графическое приложение для операционной системы Microsoft Windows, причем результирующему бинарному файлу необходимо присвоить пиктограмму **TestIcon.ico**. Для этого пиктограмму следует поместить в каталог с компилируемым файлом и выполнить команду:

```
csc /target:winexe /out:App.exe /win32icon:TestIcon.ico TestType.cs
```

В результате этих действий исполняемый файл получит заданную пиктограмму.

Очень часто при создании приложений дополнительные данные, такие как таблицы строк, изображения, пиктограммы и даже аудио-файлы, помещаются в файлы ресурсов (с расширением

.resources). Для того чтобы внедрить в сборку файл ресурсов с именем **TestRes.resources**, его необходимо поместить в каталог проекта и выполнить команду:

```
csc /target:winexe /out:App.exe /resource:TestRes.resources *.cs
```

В результате этой команды приложение получит доступ к необходимым ресурсам во время работы.

3.4. Параметры генерирования кода и вывода предупреждений

Приведем параметры компилятора, относящиеся к генерированию кода и выводу предупреждений.

- *Генерирование кода*
  - **`/debug`** – выдать отладочную информацию
  - **`/optimize`** – включить оптимизацию
- *Ошибки и предупреждения*
  - **`/warnaserror`** – обрабатывать все предупреждения как ошибки
  - **`/warn`** – установить порог предупреждений (0-4)
  - **`/nowarn`** – позволяет отключить некоторые предупреждения компилятора

Предположим, что нам требуется скомпилировать файл **TestType.cs** в консольное приложение, причем необходимо выдавать всю отладочную информацию и оптимизировать код. Для этого следует перейти в каталог, в котором содержится данный файл, и выполнить команду:

```
csc /target:exe /out:App.exe /debug /optimize TestType.cs
```

В результате выполнения данной команды сборка будет модифицирована таким образом, что она может быть присоединена к отладчику непосредственно во время исполнения.

3.5. Параметры языка программирования

Рассмотрим некоторые параметры компилятора, относящиеся к языку программирования.

- *Язык*
  - **`/checked`** – сгенерировать проверки переполнений
  - **`/unsafe`** – допускать “небезопасный” код

Для того чтобы разрешить программе **TestType.cs** на языке Visual C# использовать небезопасный код (в первую очередь, это относится С-подобным указателям), следует воспользоваться следующей командой:

```
csc /target:exe /out:App.exe /unsafe TestType.cs
```

Мы рассмотрели лишь базовые параметры компилятора Microsoft Visual C#, но этого будет вполне достаточно для выполнения данной лабораторной работы. Получить более полное представление об использовании компилятора можно в документации Microsoft Developer Network Library (MSDN).

4. Упражнение 2 – Запуск компиляции на локальной машине

Начальный вариант будущей программы представлен в проекте **RemoteCompiler**, который содержит некоторую часть исходного кода. В ходе выполнения дальнейших упражнений необходимо дополнить имеющийся вариант программы операциями ввода исходных данных, их обработки на локальной и удаленной машине и проверки правильности результатов работы программы.

4.1. Задание 1 – Открытие проекта RemoteCompiler

Для открытия проекта **RemoteCompiler** выполните следующие шаги:

- Запустите среду Microsoft Visual Studio 2005, если она еще не запущена,
- В меню **File** выполните команду **Open | Project/Solution**,
- В диалоговом окне **Open Project** выберите папку **RemoteCompiler**,
- Выберите файл **RemoteCompiler.sln** и выполните команду **Open**.

После выполнения описанных шагов в окне **Solution Explorer** (рис. 1) будет отображена структура проекта **RemoteCompiler**. В его состав входит единственное приложение **RemoteCompiler**, подлежащее дальнейшей доработке.

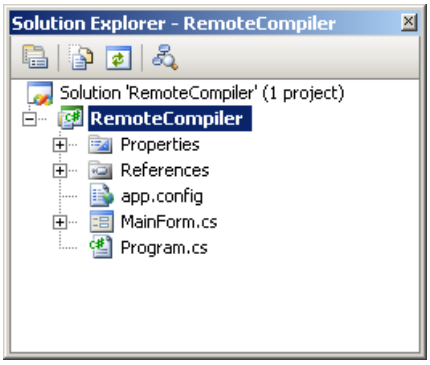

Рис. 1. Состав проекта **RemoteCompiler**

Проект **RemoteCompiler** представляет собой обычное приложение для операционной системы Microsoft Windows. Данное приложение позволяет задать все входные параметры, необходимые для запуска компилятора, такие как список входных файлов с исходными текстами, список ссылок на другие сборки, имя и тип выходного файла, а также ряд вспомогательных параметров. На основе введенных данных формируется грид-задание для запуска компилятора и отправляется на удаленную машину для исполнения средствами инструментария Alchemi. Кроме того, возможен также запуск грид-задания на локальной машине пользователя. В дальнейших заданиях мы подробно рассмотрим реализацию приложения **RemoteCompiler**.

4.2. Задание 2 – Реализация обработки грид-заданий на локальной машине

Для того чтобы открыть заготовку кода, предназначенную для дальнейшей доработки, выберите проект **RemoteCompiler** и щелкните правой кнопкой мышки на файле **MainForm.cs**. В появившемся контекстном меню выберите команду **View Code**. В результате этих действий на экране появится исходный код главной формы приложения.

В данной лабораторной работе мы не рассматриваем вопросы проектирования интерфейса приложения с использованием библиотеки Windows Forms. Однако поскольку в дальнейших фрагментах кода участвуют различные элементы управления, рекомендуется внимательно ознакомиться с главной формой программы.

В данном задании мы подробно рассмотрим реализацию метода **`void CompileLocal()`**, в котором осуществляется формирование и запуск грид-задания на локальной машине пользователя. Основные действия метода состоят в создании нового грид-задания, спецификации его входных и выходных файлов, формировании команды запуска компилятора и запуске задания на локальной машине. Опишем основные этапы реализации данного метода.

В первую очередь, необходимо создать новый экземпляр грид-задания. Соответствующий фрагмент кода имеет вид:

```
// Проверяем, выбраны ли файлы для компиляции
if (listViewSources.Items.Count == 0)
{
    // Выводим сообщение об ошибке
    MessageBox.Show("Please select files to compile.");

    // Выходим из процедуры
    return;
}

// Создаем новое задание для компиляции
GJob compileJob = new GJob();
```

Обратите внимание, что перед созданием нового экземпляра грид-задания проверяется, выбраны ли файлы для компиляции. Ситуация, когда ни один файл не выбран, считается ошибочной.

Далее следует задать входные и выходные файлы грид-задания. В нашем случае к входным файлам относятся файлы с исходными текстами, а также файлы вспомогательных сборок, необходимых для компиляции. Единственным выходным файлом является скомпилированный бинарный файл сборки. Соответствующий фрагмент кода показан на приведенном ниже листинге.

```
// Задаем входные файлы с исходными текстами
foreach (ListViewItem item in listViewSources.Items)
{
    compileJob.InputFiles.Add(new EmbeddedFileDependency(item.Text,
                                                         item.SubItems[1].Text + item.Text));
```

```
}

// Задаем ссылки на другие сборки
foreach (ListViewItem item in listViewReferences.Items)
{
    compileJob.InputFiles.Add(new EmbeddedFileDependency(item.Text,
                                                         item.SubItems[1].Text + item.Text));
}

// Задаем выходные файлы
compileJob.OutputFiles.Add(new EmbeddedFileDependency(FormatOutputName()));
```

Обратите внимание, что для добавления нового файла к грид-заданию используется специальный класс **EmbeddedFileDependency**, который является производным от *абстрактного* класса **FileDependency**. Абстрактный класс **FileDependency** определяет *базовую* функциональность для реализации так называемых *файловых зависимостей*, которые представляют собой файлы, пересылаемые на удаленную машину и необходимые для работы грид-приложения. В роли таких файлов, как правило, выступают вспомогательные динамические библиотеки или входные файлы программы. Класс **EmbeddedFileDependency** является реализацией абстрактного класса **FileDependency** и содержит конкретные методы для *пакетирования* (pack) и *распаковки* (unpack) произвольных файлов на основе алгоритма кодирования Base64. Данные методы необходимы для передачи файлов по сети. Реализация данных классов весьма проста, и для более подробного ее изучения можно обратиться к исходным кодам инструментария Alchemi. Следует заметить, что для создания нового экземпляра класса **EmbeddedFileDependency** используется несколько перегруженных конструкторов, предназначенных для различных сценариев работы. Рассмотрим данные конструкторы класса подробнее.

Конструктор **EmbeddedFileDependency(string name)** используется для создания экземпляра файловой зависимости с заданным *именем*, при этом никакой конкретный файл *не загружается*. Данный конструктор обычно используется для указания *выходных* файлов грид-задания, которые отсутствуют на момент спецификации всех параметров и будут получены на удаленной машине как *результат* исполнения грид-задания. Если вновь обратиться к приведенному выше листингу, то можно заметить, что именно этот конструктор используется для спецификации выходного файла грид-задания.

Конструктор **EmbeddedFileDependency(string name, string fileLocation)** используется для создания экземпляра файловой зависимости и принимает в качестве параметров *имя* и *путь* к файлу. При этом заданный файл *загружается* и кодируется на основе алгоритма Base64. Закодированный таким образом файл можно легко передать по сети на удаленную машину и распаковать. Данный конструктор обычно используется для задания *входных* файлов грид-задания, которые будут использованы при запуске “внешнего” приложения. Как видно из приведенного выше фрагмента кода, именно этот конструктор был использован для спецификации входных файлов с исходными текстами и вспомогательных сборок. Вернемся к доработке метода.

После того, как специфицированы входные и выходные файлы, необходимо задать команду для запуска “внешнего” приложения, в качестве которого в нашем учебном примере выступает компилятор Microsoft Visual C#. Кроме того, при запуске грид-задания на *локальном* компьютере необходимо указать рабочий каталог, который будет использован для хранения всех входных и выходных файлов. При запуске грид-задания на *удаленной* машине с помощью класса грид-приложения рабочий каталог указывается автоматически и данное действие не требуется. Соответствующий фрагмент кода показан ниже.

```
// Задаем команду для запуска компиляции
compileJob.RunCommand = string.Format(FormatCommand());

// Устанавливаем рабочий каталог для копирования и обработки файлов
compileJob.SetWorkingDirectory(Path.GetTempPath());
```

Обратите внимание, что для формирования команды запуска компилятора используется метод **string FormatCommand()**. Мы не рассматриваем реализацию данного метода, поскольку она не имеет непосредственного отношения к теме лабораторной работы. Заметим лишь, что в данном методе считываются введенные пользователем настройки и входные файлы с элементов управления, после чего формируются конкретные параметры компилятора, которые затем объединяются в команду запуска. Для более подробного изучения метода можно обратиться к исходному тексту. Отметим также, что в качестве рабочего каталога передается текущий системный каталог для хранения временных файлов, который можно получить с помощью статического метода **string GetTempPath()** стандартного класса **Path**.

Дальнейшие действия состоят в запуске грид-задания и получении результатов его работы, которые в нашем случае представлены единственным выходным файлом сборки. Соответствующий фрагмент кода имеет вид:

```
// Запускаем задание на выполнение
compileJob.Start();

// Обрабатываем завершенное задание
JobFinished(compileJob);
```

На этом разработку метода **`void CompileLocal()`** можно считать законченной. Заметим, что для обработки завершенного задания вызывается метод **`void JobFinished(GThread thread)`**, причем в качестве параметра передается выполненное грид-задание. Перейдем к рассмотрению данного метода.

Действия метода **`void JobFinished(GThread thread)`** сводятся к получению успешно завершившегося грид-задания, созданию каталога с именем выходного файла сборки и записью в данный каталог выходных файлов грид-задания. В нашем случае генерируется три выходных файла: скомпилированный файл сборки, а также текстовые файлы стандартного потока вывода и стандартного потока ошибок. Последние два файла пересылаются автоматически средствами библиотеки Alchemi. Возможная реализация метода показана на приведенном ниже листинге.

```
/// <summary>
/// Обработчик успешно завершившихся грид-заданий.
/// </summary>
/// <param name="thread">успешно завершившееся грид-задание</param>
public void JobFinished(GThread thread)
{
    // Получаем успешно завершившееся грид-задание
    GJob compileJob = (GJob)thread;

    // Задаем имя каталога для сохранения выходных файлов
    string directoryName = Path.Combine(outputDirectory,
                                        Path.GetFileNameWithoutExtension(textBoxOut.Text));

    // Создаем каталог для сохранения выходных файлов
    Directory.CreateDirectory(directoryName);

    // Сохраняем выходные файлы
    foreach (FileDependency fileDependency in compileJob.OutputFiles)
    {
        // Сохраняем выходной файл
        fileDependency.UnPack(Path.Combine(directoryName, fileDependency.FileName));
    }
}
```

Из приведенного выше фрагмента кода видно, что каталоги с выходными файлами грид-задания создаются в каталоге, путь к которому определяется переменной **`outputDirectory`**. Значение данной переменной хранится в конфигурационном файле приложения и загружается в конструкторе главной формы с помощью следующей команды:

```
// Устанавливаем каталог для записи выходных файлов
outputDirectory = ConfigurationSettings.AppSettings["Output Directory"];
```

Конфигурационный файл приложения хранится в каталоге с исполняемым файлом и называется в нашем случае **RemoteCompiler.exe.config**. На приведенном ниже листинге показано содержимое данного файла:

```
<configuration>
    <appSettings>
        <add key="Output Directory" value="C:/Temp" />
    </appSettings>
</configuration>
```

Для изменения каталога для хранения результатов достаточно просто отредактировать конфигурационный файл. Подобные файлы часто используются разработчиками приложений, поскольку позволяют изменять некоторые настройки без необходимости перекомпиляции. Более подробно конфигурационные файлы обсуждаются во второй лабораторной работе, посвященной применению модели грид-потоков для внедрения в грид существующего приложения. В следующем задании мы протестируем обработку грид-заданий на локальной машине.

4.3. Задание 3 – Тестирование обработки грид-заданий на локальной машине

Теперь приложение можно запустить и проверить работоспособность разработанных методов. Для этого выполните команду **Debug | Start Debugging** или нажмите клавишу **F5**. Если в процессе выполнения предыдущих заданий не было допущено синтаксических ошибок, программа успешно откомпилируется и запустится. На рис. 2 показано главное окно приложения.

В первую очередь, необходимо выбрать исходные файлы для компиляции, а также при необходимости указать ссылки на другие сборки. Для того чтобы добавить новый исходный файл, воспользуйтесь кнопкой **Add Source**, для добавления вспомогательной сборки – кнопкой **Add Reference**. Выбранные файлы будут отображены в соответствующем графическом списке. В верхней части окна располагается список исходных файлов, в середине окна – список ссылок на другие сборки. Для того чтобы исключить ненужные исходные файлы или ссылки на другие сборки воспользуйтесь кнопками **Remove Source** и **Remove Reference** соответственно.

Далее следует выбрать тип генерируемой сборки (консольное приложение, графическое приложение или динамическая библиотека) и задать ее имя. Затем можно включить вывод отладочной информации или оптимизацию кода. Для запуска компилятора на локальной машине щелкните на кнопке **Compile on Local**.

Для тестирования работоспособности программы можно скомпилировать простейший пример использования языка программирования Visual C# – программу “Hello, World!”:

```
class Program
{
    static void Main(string[] args)
    {
        Console.Out.WriteLine("Hello, World!");
    }
}
```

Данная программа является консольным приложением для операционной системы Microsoft Windows и не требует дополнительных сборок для компиляции.

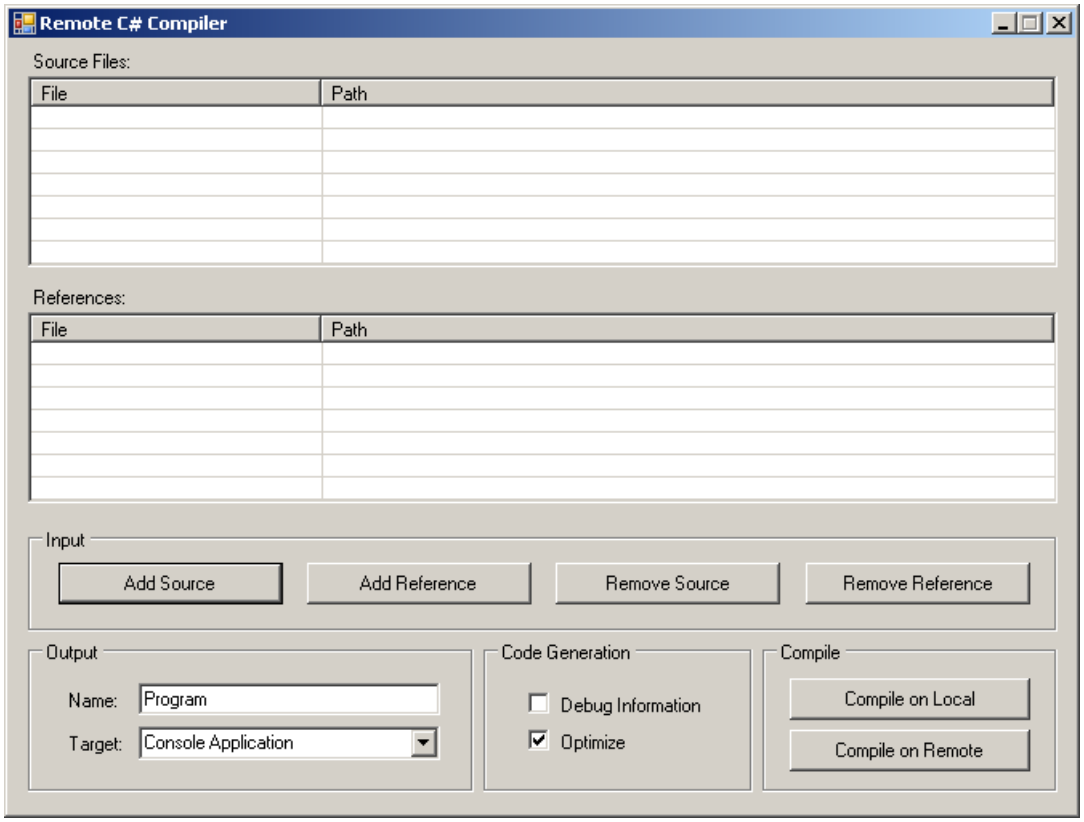

Рис. 2. Главное окно приложения после первого запуска

В результате указанных действий на локальной машине пользователя запустится компилятор Microsoft Visual C#, по завершении работы которого скомпилированная сборка будет помещена в заданный каталог для выходных файлов (указывается в конфигурационном файле приложения).

В следующем задании мы модифицируем код таким образом, чтобы грид-задание запускалось на удаленной машине.

4.4. Задание 4 – Реализация обработки грид-заданий на удаленной машине

Напомним, что наша основная цель состоит в том, реализовать обработку грид-задания для компиляции на удаленной машине. Заметим, что для достижения данной цели потребуется лишь незначительно модифицировать предшествующий исходный код. В первую очередь, необходимо объявить дополнительные переменные для взаимодействия с вычислительной сетью Alchemi.

```
/// <summary>Грид-приложение для обработки задания на удаленной машине.</summary>
GApplication gridApplication;

/// <summary>Отвечает за инициализацию грид-приложения.</summary>
private bool gridInit;
```

Как обычно, необходимо объявить экземпляр грид-приложения, посредством которого грид-задание будет отправляться на исполнение в вычислительную сеть (в нашем случае – на удаленную машину), а также логическую переменную, необходимую для однократной регистрации.

Перейдем к доработке метода **`void CompileRemote()`**, в котором осуществляется формирование грид-задания и его запуск на удаленной машине. При первом запуске грид-задания необходимо осуществить подключение к удаленному узлу, на котором установлен Менеджер Alchemi. Данный код полностью повторяет соответствующий код в предыдущих лабораторных работах, посвященных инструментарию Alchemi, поэтому мы не будем на этом подробно останавливаться.

```
// Проверяем, требуется ли инициализация грид-приложения
if (!gridInit)
{
    // Создаем новый экземпляр грид-приложения
    gridApplication = new GApplication(true);

    // Создаем диалог установки соединения с грид
    GConnectionDialog connectionDialog = new GConnectionDialog();

    // Отображаем диалог установки соединения с грид
    if (connectionDialog.ShowDialog() == DialogResult.OK)
    {
        // Устанавливаем соединение с грид
        gridApplication.Connection = connectionDialog.Connection;

        // Устанавливаем имя грид-приложения
        gridApplication.ApplicationName = "Remote C# Compiler - Alchemi sample";

        // Добавляем обработчик успешно завершившихся грид-заданий
        gridApplication.ThreadFinish += new GThreadFinish(JobFinished);

        // Добавляем обработчик неудачно завершившихся грид-заданий
        gridApplication.ThreadFailed += new GThreadFailed(JobFailed);

        // Инициализация завершена
        gridInit = true;
    }
}
```

Прокомментируем лишь основные действия данного фрагмента кода. В первую очередь проверяется, проинициализировано ли грид-приложение. Если грид-приложение уже проинициализировано, то никаких действий не требуется. В противном случае необходимо произвести инициализацию, то есть создать новый экземпляр грид-приложения, установить его название, назначить обработчики событий, возникающих при успешном и неудачном завершении грид-заданий, а также установить соединение с грид. Для установки параметров соединения разработчики инструментария Alchemi предусмотрели специальный диалог **`GConnectionDialog`**, с помощью которого можно задать имя пользователя и пароль, а также адрес и порт Менеджера.

Далее следует создать новое грид-задание для компиляции файлов и установить его параметры (входные файлы, выходные файлы и команда запуска). Единственное отличие от ранее рассмотренного кода состоит в том, что на этот раз грид-задание не запускается непосредственно после установки всех необходимых параметров, а добавляется к грид-приложению. На приведенном ниже листинге показан соответствующий фрагмент кода.

```
// Проверяем, выбраны ли файлы для компиляции
if (listViewSources.Items.Count == 0)
{
    // Выводим сообщение об оишбке
    MessageBox.Show("Please select files to compile.");

    // Выходим из процедуры
    return;
}

// Создаем новое задание для компиляции
GJob compileJob = new GJob();

// Задаем входные файлы с исходными текстами
foreach (ListViewItem item in listViewSources.Items)
{
    compileJob.InputFiles.Add(new EmbeddedFileDependency(item.Text,
                                                         item.SubItems[1].Text + item.Text));
}

// Задаем ссылки на другие сборки
foreach (ListViewItem item in listViewReferences.Items)
{
    compileJob.InputFiles.Add(new EmbeddedFileDependency(item.Text,
                                                         item.SubItems[1].Text + item.Text));
}

// Задаем выходные файлы
compileJob.OutputFiles.Add(new EmbeddedFileDependency(FormatOutputName()));

// Добавляем грид-задание к экземпляру грид-приложения
gridApplication.Threads.Add(compileJob);
```

Обратите также внимание, что на этот раз мы не устанавливали рабочий каталог грид-задания, поскольку при добавлении грид-задания к грид-приложению рабочий каталог будет присвоен автоматически (в роли данного каталога выступает один из временных каталогов пользователя).

Далее следует проверить, запущено ли в данный момент грид-приложение. Если это так, то грид-приложение необходимо корректно остановить. Следует иметь в виду, что данное действие может генерировать ряд исключений, поэтому его следует заключить в блок **`try-catch`**.

```
// Проверяем, запущено ли грид-приложение
if (gridApplication.Running)
{
    try
    {
        // Останавливаем грид-приложение
        gridApplication.Stop();

        // Выходим из процедуры
        return;
    }
    catch (Exception e)
    {
        // Выводим сообщение об ошибке
        MessageBox.Show("Can not stop already running grid application: " + e.ToString());
    }
}
```

Наконец, грид-приложение необходимо запустить. Данное действие также может генерировать ряд исключений, поэтому его следует заключить в блок **`try-catch`**. Соответствующий фрагмент кода показан на приведенном ниже листинге.

```
try
{
    // Запускаем грид-приложение
    gridApplication.Start();
}
catch (Exception e)
{
    // Выводим сообщение об ошибке
    MessageBox.Show("Can not start grid application: " + e.ToString());
}
```

На этом доработку метода можно считать завершенной. В заключение рассмотрим реализацию обработчика события **`ThreadFailed`**, которое возникает при неудачном завершении грид-потока (или

грид-задания). В нашем учебном примере мы ограничимся выводом сообщения об ошибке. Соответствующий фрагмент кода имеет вид:

```
/// <summary>
/// Обработчик неудачно завершившихся грид-заданий.
/// </summary>
/// <param name="thread">неудачно завершившееся грид-задание</param>
public void JobFailed(GThread thread, Exception e)
{
    // Выводим сообщение об ошибке
    MessageBox.Show("Job with ID = " + thread.Id.ToString() + " failed: " + e.ToString());
}
```

Заметим, что при разработке практических приложений такого поведения программы явно недостаточно. Правильнее было бы проанализировать случившуюся ошибку и перезапустить грид-задание. Очевидно, что обработка неудачно завершившихся грид-заданий специфична для каждого конкретного случая, поэтому очень сложно дать общие рекомендации. Основная цель данного замечания – обратить внимание читателя на данную проблему и учитывать ее при разработке реальных программ.

4.5. Задание 5 – Тестирование обработки грид-заданий на удаленной машине

Наконец, приложение можно запустить и проверить работоспособность разработанных методов. Для этого выполните команду **Debug | Start Debugging** или нажмите клавишу **F5**. Если в процессе выполнения предыдущих заданий не было допущено синтаксических ошибок, программа успешно откомпилируется и запустится. На рис. 2 показано главное окно приложения.

Все параметры приложения устанавливаются точно также, как и при запуске на локальной машине пользователя. Сначала необходимо выбрать компилируемые файлы и указать необходимые ссылки на другие сборки. Затем следует выбрать тип генерируемой сборки (консольное приложение, графическое приложение или динамическая библиотека) и задать ее имя. При необходимости можно также включить вывод отладочной информации или оптимизацию кода. Для запуска компилятора на удаленной машине щелкните на кнопке **Compile on Remote**.

В результате указанных действий на удаленной машине запустится компилятор Microsoft Visual C#, по завершении работы которого скомпилированная сборка будет помещена в заданный каталог для выходных файлов (указывается в конфигурационном файле приложения).

5. Заключение

В данной лабораторной работе был рассмотрен процесс внедрения существующего приложения в грид с помощью модели грид-заданий. В качестве такого приложения выступал компилятор для языка программирования Visual C#.

Как уже было сказано, грид-задание представляет собой специализированный грид-поток, специально предназначенный для запуска приложений на удаленных ресурсах. Грид-задание позволяет разработчику специфицировать входные файлы, команду запуска и выходные файлы некоторого приложения. Таким образом, разработчик может сконцентрироваться на основной логике программы, не отвлекаясь на программирование различных сервисных операций (которые подробно рассматривались в предыдущей лабораторной работе). Как следствие, значительно упрощается и ускоряется процесс внедрения некоторого приложения в грид. Заметим, однако, что в ряде случаев все же необходимо использовать более низкоуровневую модель грид-потоков для запуска приложения на удаленных ресурсах. Подобная необходимость возникает, например, когда стандартных действий по копированию входных файлов и запуску некоторого приложения недостаточно.

Подводя итоги, приведем некоторые достоинства и недостатки рассмотренного подхода. К достоинствам можно отнести:

- *Простота реализации и компактность кода.*

  Использование модели грид-заданий позволяет еще более упростить процесс внедрения некоторого приложения в грид (по сравнению с подходом, обсуждавшимся в предыдущей лабораторной работе). С помощью модели грид-заданий можно легко сделать приложение, предназначенное для запуска любой программы в грид (в самом деле, различные приложения отличаются лишь списком входных и выходных файлов, а также командой запуска).

- *Возможность быстро и удобно проектировать пользовательский интерфейс.*

  Инструментарий Alchemi использует все возможности библиотеки пользовательского интерфейса Windows Forms.

- *Легкость в отладке и использовании приложений.*

  Для тестирования правильности всех параметров грид-заданий (рабочий каталог приложения, команда запуска, входные файлы и т.п.) можно произвести их запуск сначала на машине разработчика и лишь затем осуществлять их запуск в грид.

- *Возможность запуска разработанных приложений на локальных компьютерах.*

  Разработанные приложения можно запускать на локальных компьютерах без установленных компонент инструментария Alchemi. Любая программа для инструментария Alchemi может использоваться как обычное приложение для операционной системы Microsoft Windows с установленной платформой Microsoft .NET Framework (при этом, конечно, приложение не получит доступ к ресурсам грид). Таким образом, разработанное приложение может использоваться, например, в качестве удобной графической надстройки над устаревшей консольной программой.

Однако при использовании инструментария Alchemi можно обнаружить некоторые недостатки. Отметим на наш взгляд наиболее критичные из них:

- *Отсутствие переносимости кода и ориентация на единственную платформу.*

  К сожалению, на сегодняшний день отсутствует полноценная замена платформы Microsoft .NET Framework для других операционных систем. Данный факт делает невозможным запуск программ, написанных с использованием инструментария Alchemi, под другими операционными системами. Стоит заметить, что прогресс в этом направлении не стоит на месте, и межплатформенная разработка Mono[2] уже сейчас предлагает неплохой уровень совместимости с платформой Microsoft .NET Framework.

- *Слабая масштабируемость инструментария.*

  Инструментарий Alchemi лучше всего подходит для объединения в вычислительную сеть нескольких десятков компьютеров. При подключении большего числа компьютеров могут появиться проблемы со стабильностью системы (в частности, со стабильностью работы Менеджера). Улучшить масштабируемость позволяет организация иерархической структуры из отдельных Менеджеров. Опять же, это скорее особенность, которую следует учитывать, а не недостаток инструментария Alchemi.

6. Вопросы

1. В чем на ваш взгляд состоит проблема наполнения грид приложениями?
2. Каким требованиям должен обладать инструментарий для успешного решения данной проблемы?
3. В чем состоит процесс внедрения в грид некоторого приложения с помощью модели грид-заданий?
4. В чем состоят преимущества и недостатки данной модели по сравнению с моделью грид-потоков?
5. Каким требованиям должно отвечать приложение для внедрения в грид?
6. Назовите основные этапы процесса внедрения приложения в грид с помощью модели грид-заданий.

7. Упражнения

В качестве дополнительных упражнений можно предложить следующее:

1. Реализовать корректную обработку неудачно завершившихся грид-заданий.
2. Проделать описанные в данной лабораторной работе шаги для другого приложения.

8. Литература

1. Akshay Luther, Rajkumar Buyya, Rajiv Ranjan, and Srikumar Venugopal. Peer-to-Peer Grid Computing and a .NET-based Alchemi Framework. http://www.gridbus.org/~alchemi/files/alchemi_bookchapter.pdf

[2] Главной задачей проекта Mono (http://www.mono-project.com) является создание полноценной реализации среды разработки Microsoft .NET Framework для UNIX-подобных операционных систем. Инициативу Mono возглавляет Мигель де Иказа, один из участников проекта Gnome. Спонсирует же группу разработчиков Mono корпорация Novell.

2. Akshay Luther, Rajkumar Buyya, Rajiv Ranjan, and Srikumar Venugopal. Alchemi: A .NET-based Enterprise Grid Computing System.
http://www.gridbus.org/%7Eraj/papers/alchemi_icomp05.pdf
3. Akshay Luther, Rajkumar Buyya, Rajiv Ranjan, and Srikumar Venugopal. Alchemi: A .NET-based Grid Computing Framework and its Integration into Global Grids.
http://www.gridbus.org/~alchemi/files/alchemi_techreport.pdf
4. http://www.alchemi.net – официальный сайт проекта Alchemi