Основы платформы Microsoft .NET

Тема:

Понятия сборок и компоновок

Понятие сборки (assembly)

Основной задачей программиста, который непосредственно занимается кодированием (написанием исходного кода) программы, является написание исходного текста программы на одном из языков программирования. Хороший программист, разрабатывая новую программу, не пишет весь код заново. Он старается использовать уже готовые (написанные ранее) программные коды (библиотеки), написанные как им самим, так и другими разработчиками. Если рассматривать эти библиотеки, как строительные блоки, то программист из них, как из кирпичей строит здание – новую программу. Такой подход к программированию называется *технологией повторного использования кода*. Что обычно из себя представляют подобные строительные блоки? Это программный код и необходимые ресурсы (например, файлы данных или рисунков). Стоит отметить, что по мере развития библиотек, появляются их новые версии. Но при этом старые не исчезают, а продолжают использоваться уже написанными ранее программами.

В процессе развития технологии программирования было несколько вариантов реализации подхода к повторному использованию кода. На сегодняшний день – это широко используемые динамически подгружаемые библиотеки (DLL-библиотеки). DLL-файлы – это обычные *PE-файлы (файлы в формате PE – portable executable)*. Это значит, что компьютер, работающий под управлением 32- или 64-разрядной версии Windows, способен загрузить этот файл и выполнить код, содержащийся в нем. Данный подход используется уже несколько лет и кроме достоинств в нем есть ряд недостатков (наиболее

известный получил звучное название "Ад DLL") от которых настало время избавляться. И в качестве решения проблем DLL-библиотек в .NET предложен новый подход, в соответствии с которым на замену DLL-библиотекам пришло понятие *сборок*. *Сборка* – это единица повторного использования кода, в которой поддерживается система управления версиями и заложена система управления безопасности программного обеспечения. Стоит обратить внимание на одну из революционных особенностей сборок по отношению к DLL-библиотекам – сборки *самодостаточны*, они содержат *метаданные* (metadata), которые несут в себе информацию о версии, зависимостях, типах, атрибутах и многое другое.

О том, как это реализовано в .NET и как воспользоваться всеми преимуществами нового подхода к повторному использованию кода, пойдет речь в этом разделе.

Структура сборки

Как отмечалось ранее, сборка наряду с программным кодом содержит метаданные и данные (ресурсы), необходимые при исполнении сборки. В общем виде структура сборки представлена на рис. 2.1.

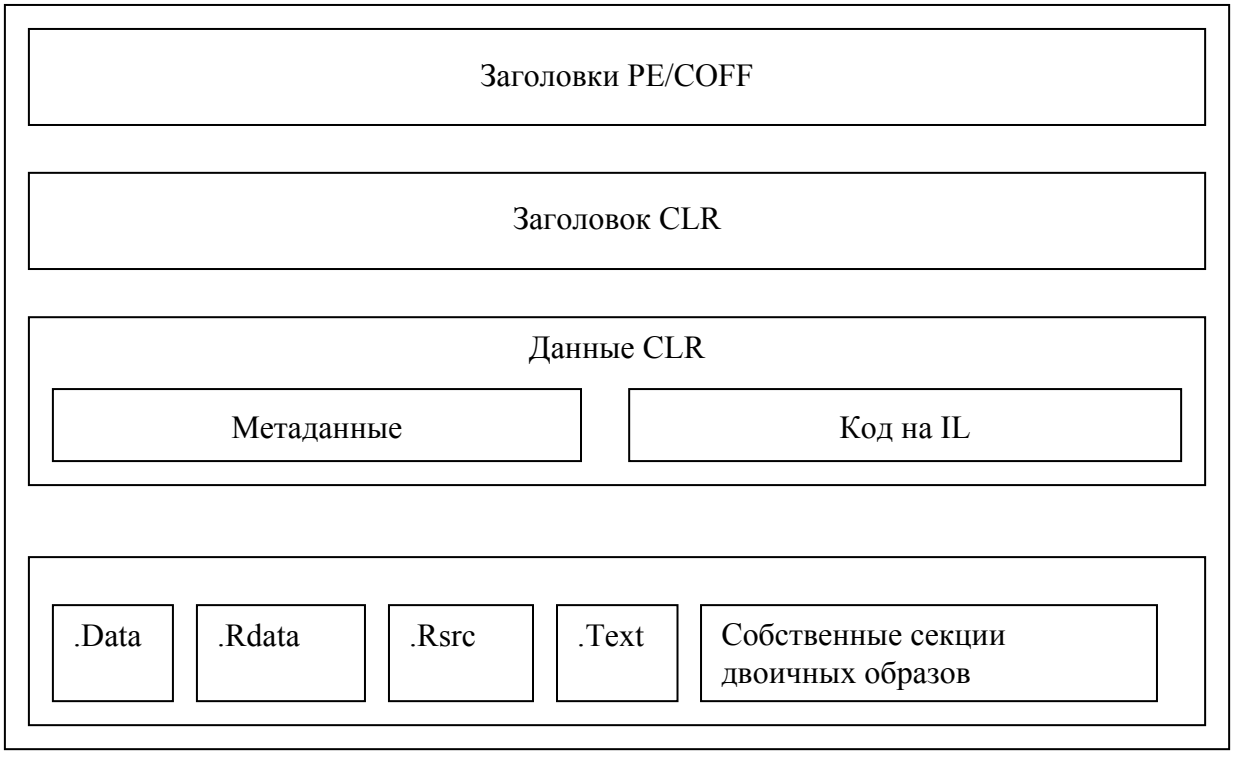

Рис 2.1. Формат PE-файла, генерируемый средой .NET

Для поддержки новых возможностей среды CLR Microsoft немного изменила формат PE-файла, генерируемого средой .NET. А именно, были добавлены несколько новых секций:

- Заголовок CLR – содержит информацию, указывающую, что PE-файл является исполняемым файлом .NET,

- Данные CLR – определяют, как будет выполняться программа.

Понятие манифеста

Служебная информация (метаданные), которая включается в сборку дополнительно к программному коду, получила название *манифест*. В состав манифеста входят:

- Данные о версии сборки,
- Список файлов, входящих в сборку. Следует отметить, что для каждого файла хранится контрольная сумма, вычисляемая во время создания сборки при помощи криптографических хэш-функций[1]. В момент исполнения приложения данные файлы проверяются по контрольным суммам, чтобы удостовериться в целостности файлов сборки, а так же в том, что файлы не были подменны другими с такими же именами (в том числе новыми версиями фалов),
- Имена и версии сборок, которые используются данной сборкой (зависимости от других сборок). Во время выполнения версии сборок строго сверяются, чтобы удостовериться в том, что загружена именно нужная сборка,
- Параметры, определяющие правила использования сборки. Различают два уровня защиты:
  - Права на запуск данной сборки (автор сборки может указать, какой минимальный уровень привилегий пользователь должен иметь, для запуска данной сборки),
  - Права на возможность использования сборки. Управление правами данного типа для сборок, являющимися коммерческими продуктами, обеспечивают возможность задания наборов функциональности в зависимости от типа приобретенной лицензии (например, набор функций приложения при установленной версии лицензии для "Домашнего использования" может отличаться от "Профессиональной" версии, хотя в обоих случаях бинарный код самого приложения один и тот же). Имеется возможность совсем запретить исполнение сборки, если не установлена ни одна из лицензий.

Просмотр метаданных

Для демонстрации примера вида и структуры метаданных используем учебную программу на C# из раздела "Введение в технологию Microsoft.NET":

[1] Хеш-функция – это функция, которая выдает в соответствие с определенным алгоритмом каждому входному объекту, например, файлу, некоторый индекс фиксированной длины.

```
using System;
class MainApp
{
  public static void Main()
  {
    Console.WriteLine("C# Hello, World!");
  }
}
```

Компиляция примера производится при помощи следующей команды:

csc HelloWorld.cs

Для этого необходимо запустить приложение "**Командная строка**"[1], перейти в каталог, где сохранен исходный текст примера и набрать приведенную выше команду.

В результате должен получиться исполняемый файл HelloWorld.exe

Для просмотра метаданных получившейся сборки необходимо воспользоваться утилитой **ildasm.exe** (Intermediate Language Disassembler - дизассемблер промежуточного языка), входящей в Microsoft .NET SDK. Для этого выполните следующую команду:

ildasm.exe HelloWorld.exe

Рис. 2.2. Работа утилиты ildasm.exe

Для демонстрации манифеста необходимо выполнить двойной щелчок левой кнопкой мыши на разделе M A N I F E S T. Представление метаданных манифеста имеет следующий вид:

[1] В большинстве случаев, для этого необходимо нажать кнопку **Пуск,** выбрать пункт **Все программы,** затем подпункт **Стандартные**.

```
// Это ссылка на основную библиотеку классов .NET
.assembly extern mscorlib
{
// Это хеш публичного ключа данной сборки
// он нужен для подтверждения валидности сборки
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89) // .z\V.4..
// Версия сборки которая использовалась при создании
// приложения
  .ver 1:0:5000:0
}
//Описание нашей сборки
.assembly HelloWorld
{
  // --- The following custom attribute is added
  // automatically, do not uncomment -------
  // .custom instance void [mscorlib]
  // System.Diagnostics.DebuggableAttribute::.ctor(bool,
  // bool) = ( 01 00 00 01 00 00 )
// Алгоритм по которому считается хэш
  .hash algorithm 0x00008004
// Версия нашей сборки
  .ver 0:0:0:0
}
// Название запускаемого файла
.module HelloWorld.exe
// MVID: {D48BD0D0-AE3E-4E46-AF65-B6E7886D36DA}
// Предпочтительный адрес для загрузки сборки
.imagebase 0x00400000
// Подсистема (консоль, оконное приложение, приложение времени загрузки)
.subsystem 0x00000003
// Выравнивание секций
.file alignment 512
// Зарезервированный флаг
.corflags 0x00000001
// Image base: 0x02ca0000
```

Как можно увидеть, манифест содержит полную системную информацию о сборке. Эта информация наиболее активно используется при разработке сложных динамически загружаемых приложений - для начального же знакомства дадим краткое предварительное описание содержимого манифеста сборки:

- В начале манифеста находится секция *.assembly extern mscorlib,* в которой описываются зависимости от внешних сборок, используемых в данной программе. Здесь для каждой сборки указывается версия и контрольная сумма. Эти данные берутся из сборок при компиляции программы, что гарантирует во время работы приложения использование именно тех сборок, которые использовались при компиляции и тестировании,
- Далее следует секция *.assembly*, но уже без модификатора *extern*. С этой директивы и начинается описание сборки. Как можно заметить, *.ver* описывает версию сборки, секция *.hash algorithm* определяет функцию, по которой будет вычисляться хэш-код, затем в манифесте располагаются описания имени самого модуля, подсистемы исполнения, информация о выравнивании секций и еще некоторые данные. Полная

документация по данным, хранящимся в манифесте находится в Microsoft Developer Network (MSDN) Library [1].

Многомодульные сборки

Компоновка исполняемого кода в модули

Как уже отмечалось ранее, сборка может состоять из нескольких файлов. В простейшем случае – это файлы ресурсов, например .gif- или .jpg-файлы, используемые во время работы программы. Следует обратить внимание, что когда идет речь о том, что сборка состоит из нескольких файлов, то это не файлы исходного текста программы (а их может быть очень много), а файлы, полученные после компиляции приложения. Зачем это может понадобиться? Допустим, что при построении приложения используется несколько библиотек. При этом исполняемый код, который используются чаще всего, можно разместить в одном файле, а используемый реже – в другом. Допустим, что разрабатываемое приложение развертывается путем загрузки через Интернет, тогда клиенту файл с редко используемым кодом может совсем не понадобиться во время работы и соответственно не будет загружен.

Процесс распределения исполняемого кода по нескольким файлам называется *компоновкой*. Файлы с исполняемым кодом, полученные в результате *компоновки*, называются *модулями*. И удачно скомпоновать модули – это профессиональное качество программиста, которое приходит с опытом разработок.

Рассмотрим несколько преимуществ многофайловых сборок. Данный подход позволяет следующее:

- Разделять код по разным файлам – как результат можно избирательно загружать функциональные части приложения из Интернет во время его работы,
- Добавлять к сборке файлы с ресурсами и данными. В информации о сборке будут храниться данные об используемых файлах, которые могут находиться как в том же каталоге, где и остальные файлы сборки, так и в Интернете (будут загружаться во время работы по мере необходимости),
- Создавать сборки, состоящие из кода, написанного на разных языках программирования. При компиляции исходного текста на C# компилятор создает один модуль, а при компиляции исходного текста на Visual Basic – другой. Затем все эти модули можно объединить в одну сборку. Использующий такую сборку разработчик будет видеть лишь общий набор библиотек, даже не задумываясь, что часть кода была написана на одном языке программирования, а часть на другом.

Объединение модулей для создания сборки

Чтобы скомпоновать сборку, нужно выбрать один из PE-файлов, который будет хранителем манифеста. Можно также создать отдельный PE-файл, в котором ничего, кроме манифеста не будет.

Манифест позволяет потребителям сборки не заботиться о деталях распределения файлов, входящих в нее и делает сборку самодостаточной. Стоит заметить, что файл, содержащий манифест, "знает", какие файлы составляют сборку, но отдельные файлы "не знают", что они включены в сборку.

Компилятор C# генерирует сборку, если в командной строке указан один из переключателей: **/t[arget]:exe, /t[arget]:winexe или /t[arger]:library**. При использовании любого из переключателей генерируется одиночный PE-файл с таблицами метаданных.

Если же при компиляции указать переключатель **/t[arget]:module**, то будет создан PE-файл без таблиц метаданных. При использовании этого переключателя всегда получается DLL-файл в формате PE, которому по умолчанию присваивается расширение .netmodule.

Добавить модуль к сборке можно при помощи переключателя командной строки **/addmodule**. Чтобы понять, как создаются многофайловые сборки, рассмотрим пример. Пусть у нас имеется два файла:

- RUT.cs, содержащий редко используемый код,
- FUT.cs, содержащий часто используемый код.

Скомпилируем файл с редко используемым кодом в отдельный модуль, чтобы пользователи могли отказаться от его развертывания, если содержащаяся в нем функциональность не понадобятся во время исполнения приложения:

csc /t:module RUT.cs

В результате будет создан файл RUT.netmodule. Этот файл представляет собой библиотеку DLL, но CLR не сможет ее загрузить, т.к. в ней нет манифеста.

Теперь нам нужно скомпилировать в отдельный модуль часто используемый код и сделать его хранителем манифеста сборки. Фактически этот модуль будет представлять из себя целую сборку, поэтому давайте дадим ей имя MultyModule.dll:

csc /out:MultyModule.dll /t:library /addmodule:RUT.netmodule FUT.cs

Переключатель **/t:library** сообщает компилятору, что необходимо сгенерировать библиотеку DLL с именем файла MultyModule.dll, в которой, как было отмечено выше, будет размещен манифест сборки. Переключатель **/addmodule:RUT.netmodule** делает библиотеку, хранящуюся в файле RUT.netmodule, частью сборки.

Любой клиентский код, использующий функциональность сборки MultyModule.dll, должен быть скомпилирован с ключем **/r[eference]:MultyModule.dll**. Данный переключатель заставляет компилятор загрузить сборку MultyModule.dll и все файла, используемые в этой сборке (указанные в манифесте). Таким образом, в момент компиляции необходимо, чтобы все эти файлы были доступны.

Во время исполнения клиентский код вызывает разные методы. Если код вызывает функции, реализованные в библиотеке MultyModule.dll, то срабатывают внутренние механизмы CLR, которые делают эти функции доступными, и, если это необходимо, загружают модули в память. Это означает, что во время работы приложения, доступность всех модулей *не является необходимой*.

Управление версиями

Практически каждый пользователь, работающий в операционной системе Windows, сталкивался с проблемой, когда при установке нового программного обеспечения перестают работать другие программы. Это может произойти из-за не совсем удачного подхода к реализации идеи повторного использования кода, корни которого находятся в далеком (по меркам IT) прошлом. В те времена для решения данной проблемы были созданы динамически подгружаемые библиотеки (Dynamic-Link Load Library, DLL). Они позволяли "выносить" часто используемый код в отдельные библиотеки, которые могли использовать любые приложения. Проблема была в том, что DLL изначально не предоставляли никаких средств управлениям версиями, тогда об этом просто никто не задумывался.

Представим себе следующую ситуацию. Программист пользуется библиотекой, в которой в одной из функций имеется ошибка. Т.к. у данного разработчика нет возможности исправить ошибку в чужой библиотеке (в подавляющем большинстве случаев пользователь библиотеки не имеет ее исходных кодов), то ему приходится при разработке своей программы учитывать "побочный" эффект от ошибки. Разработанная программа передается пользователю, у которого она замечательно работает (используя библиотеку с учетом ошибки). Позднее автор библиотеки обнаруживает ошибку и исправляет ее. Пользователь устанавливает совершенно другое программное обеспечение, использующее ту же библиотеку, и вместе с ним ее новую версию (без ошибки). И вот тут программа, написанная с учетом ошибки в библиотеке, может перестать работать, т.к. ошибки уже нет!!! И что в данной ситуации делать "бедному" пользователю? Пользоваться старой версией библиотеки, с которой работает первая программа, но не

работает вторая, или установить новую версию, отказавшись от работы с первой программой?

Какой же способ решения данной проблемы предлагается в .NET? А решения очень простое – нужно хранить все необходимые версии библиотеки на компьютере пользователя. И при работе разных программ будет использоваться именно та версия библиотеки, с которой данная программа разрабатывалась и тестировалась. Может возникнуть резонный вопрос – а почему нельзя было раньше так делать? Дело в том, что когда разрабатывалась технология DLL-библиотек о проблеме существования и использования одновременно нескольких разных версий одних и тех же библиотек не задумывались. А до эпохи .NET библиотеки идентифицировались только по имени файла. Таким образом, получается, что разные версии одной и той же библиотеки хранятся в файлах с одним и тем же именем. А, как известно, в одном каталоге (где хранятся библиотеки) не может находиться два файла с одним и тем же именем. Давайте посмотрим, как эта проблема решается в Microsoft .NET.

Локальные сборки

Локальные (приватные) сборки (библиотеки) поставляются с самим приложением, используются только им и храниться в его каталоге. Конечно, можно подумать, что с каждым приложением поставлять одни и те же сборки нецелесообразно, можно ведь сделать сборку совместной, сэкономив тем самым немного места. Но когда приходится выбирать между устойчивостью работы приложения и небольшим увеличением эффективности использования ресурсов компьютера в виде сохранения дискового пространства, целесообразней остановиться на первом варианте.

Локальные сборки видны только самому приложению и никому более, т.е. приложение изолируется от внешнего воздействия, как других программ, так и самой операционной системы. Соответственно, локальные сборки лишены многих проблем, связанных с совместными сборками. К примеру, такой, как уникальность имен: так как сборка локальная, нет необходимости заботиться об уникальности имен во всем глобальном пространстве имен. Концепция приватных сборок сильно упрощает развёртывание (инсталляцию) ваших приложений, так как больше не придется делать записей в реестре. Теперь нужно будет просто копировать сборки в директорию приложения или в поддиректории. Общая среда исполнения (CLR) при запуске приложения прочитает его манифест и определит, какие сборки необходимы. Затем будет произведён процесс зондирования (probing) рабочего каталога приложения на предмет

нужной сборки. Необходимая сборка определяется по имени файла, определенного в манифесте.

Совместные сборки

Среда исполнения .NET поддерживает также *совместные сборки*. Это сборки, которые могут быть использованы сразу несколькими приложениями и которые, соответственно, "видны" всем. Правда, к таким сборкам предъявляются более строгие правила, чем к приватным сборкам. Например, необходима уникальность имен сборки: имена внутри сборки не должны конфликтовать с уже существующими в глобальном пространстве имен, предоставляемом средой исполнения по умолчанию.

Действия системы управления версиями при поиске необходимых сборок могут быть изменены при помощи политики версий, которую сможет изменять администратор или автор приложения. Данная политика позволит принудительно изменить версию сборки, запрашиваемой приложением, а также поведение среды исполнения при поиске и загрузке сборок. Таким образом, имеется возможность "заставить" приложение использовать сборку другой версии, даже если оно на это не рассчитано. Данная политика настраивается при помощи файла конфигурации, который помещается в каталог приложения и имеет то же имя, что и у приложения, только с расширением .config.

Совместные сборки хранятся в *глобальном кэше сборок* (global assembly cache - GAC). Сборки, хранящиеся там, используются многими приложениями. К ним также имеет доступ администратор, который при необходимости сможет ставить исправления ("заплатки") на совместно используемые сборки, которые, соответственно, окажут влияние на все приложения, которые используют данные сборки. Для примера это может быть заплатка на общие библиотеки среды – исправление в каком-нибудь классе. Хранилище сборок располагается в каталоге %SystemRoot%\assembly. Вид содержимого этого каталога в проводнике показан на рис. 2.3.

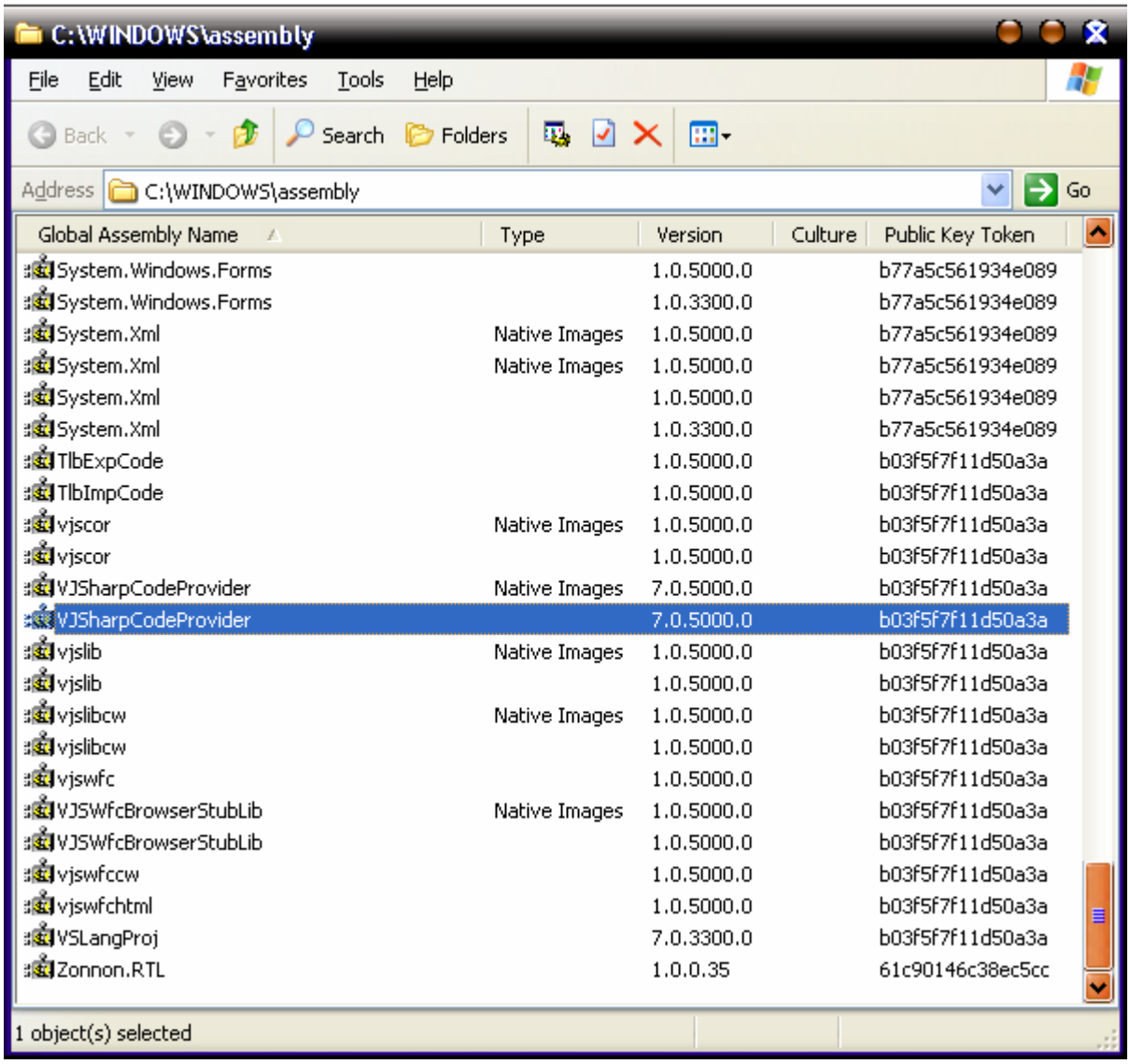

Рис. 2.3. Список совместных сборок

Обратите внимание, что вид содержимого каталога %SystemRoot%\assembly отличается от вида обычных каталогов в проводнике. Когда открывается этот каталог, активизируется специальное расширение оболочки Windows и показываются сборки, которые сейчас хранятся в GAC. Содержимое хранилища, расположенного каталоге %SystemRoot%\assembly\GAC на компьютере, где проводились описанные эксперименты, показано на рис. 2.4:

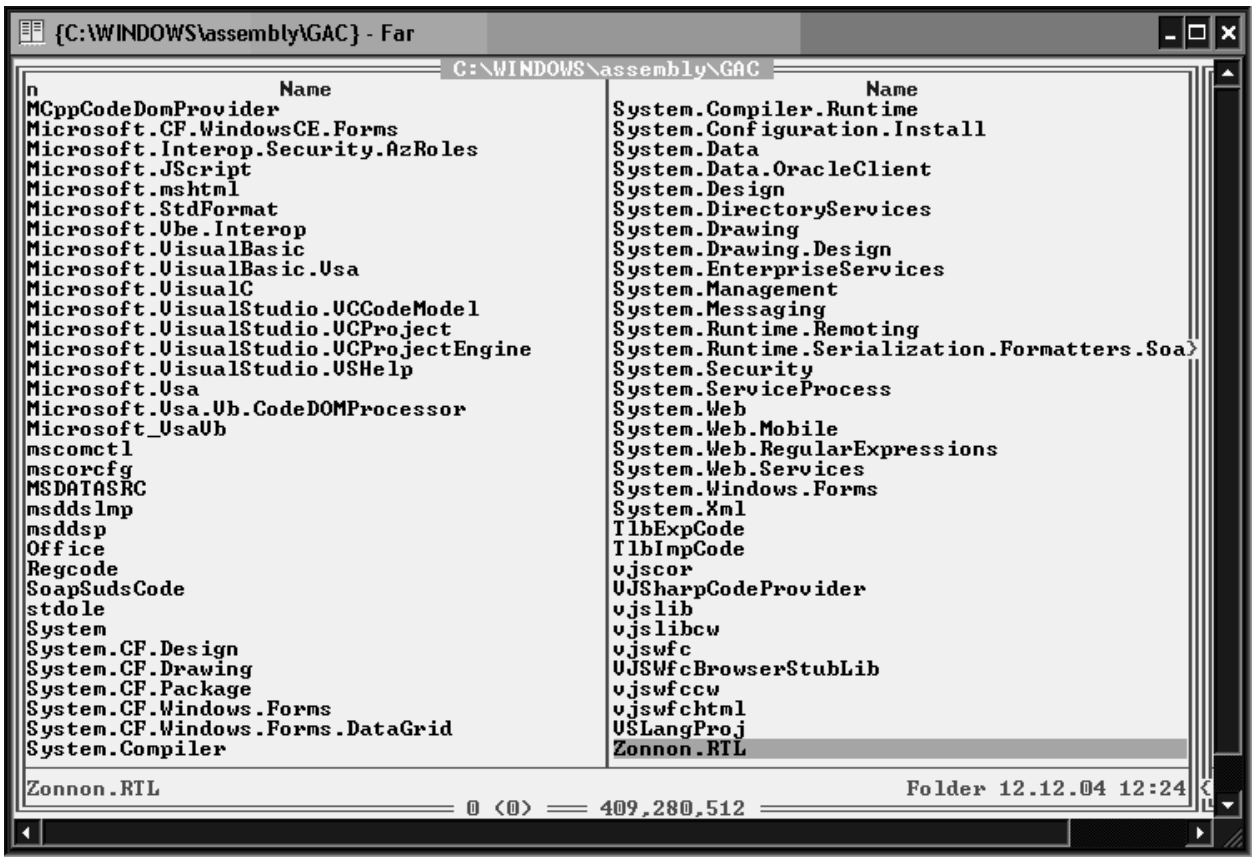

Рис. 2.4. Хранение совместных сборок

В каталоге GAC есть подкаталоги, представляющие каждую сборку, в которых хранятся директории, разбивающие данную сборку по версиям. Таким образом, на компьютере может храниться любое количество версий одной и той же сборки. При таком подходе каждое приложение сможет использовать именно ту совместную сборку, на которую оно рассчитано. То есть при загрузке приложения для него будет выбираться сборка именно той версии, которую оно запросит, хотя может существовать и гораздо более новая версия этой же сборки.

Сведения о версии сборки

Номер версии состоит из четырёх чисел: Major.Minor.Build.Revision (например, 1.2.10.6), где пара Major.Minor – это *основная часть*, а Build.Revision – это *дополнительная часть*:

- Major - основная версия,
- Minor - подверсия приложения,
- Build - количество построений (полных компиляций) для данной версии,
- Revision - номер ревизии для текущего построения.

При поиске нужной сборки, основная часть версии должна строго совпадать. Если будет найдено несколько сборок с одинаковыми основными частями, то будет выбрана

сборка с наибольшей дополнительной частью. Версию сборки можно задать при помощи параметра командной строки при компиляции приложения, либо при помощи атрибута *System.Reflection.AssemblyVersionAttribute*. Можно задавать версию не полностью, а только ее часть. Например, вот так: "1.*", "1.5.*", "1.5.2.*". При отсутствии каких либо частей, компилятор допишет их сам по следующим правилам:

- Minor - приравнивается к нулю,
- Build - приравнивается количеству дней прошедших с первого января 2000 года,
- Revision - приравнивается количеству секунд, прошедших с полуночи, деленных на два.

Данная схема позволяет гарантировать уникальность версии для каждого построения приложения.

Общие выводы

Благодаря технологии сборок, появившейся в Microsoft .NET, удалось решить следующие существующие проблемы разработки и использования программного обеспечения:

- Благодаря технологии самодостаточности сборок, приложение само "знает" какие файлы/ресурсы нужны для его работы и где они находятся (например, в Интернете на сайте разработка программы),
- Благодаря возможности компоновки сборок в отдельные модули, загружаемые по мере их необходимости, появляется возможность экономии трафика Интернет и времени загрузки приложения,
- Система управления версиями сборок позволяет решить проблему, когда приложение может перестать работать при появлении на компьютере более новой DLL-библиотеки, с которой оно не разрабатывалось и не тестировалось.

Дополнительная информация о сборках может быть получена, например, в [2-4].

Литература

1. Microsoft Developer Network (MSDN) (http://msdn.microsoft.com)
2. Пономарев В. Программирование на C++/C# в Visual Studio .NET 2003. – СПб.: БХВ-Петербург, 2004.
3. Рихтер Д. Программирование на платформе Microsoft .NET Framework. – М.: Издательско-торговый дом "Русская Редакция", 2002.

4. Чакраборти А., Кранти Ю., Сандху Р. Дж. Microsoft .NET Framework: разработка профессиональных проектов. – СПб.: БХВ-Петербург, 2003.