электронный журнал
МОЛОДЕЖНЫЙ НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК
Издатель ФГБОУ ВПО "МГТУ им. Н.Э. Баумана". Эл No. ФС77-51038.

УДК 004.43

Обзор технологии генерации исходного кода T4 и оболочки PowerShell

Баранкова И.А., *студент*
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана,
кафедра «Системы обработки информации и управления»

Минакова С.В., *студент*
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана,
кафедра «Системы обработки информации и управления»

Научный руководитель: Гапанюк Ю.Е., к.т.н., доцент,
Россия, 105005, г. Москва, МГТУ им. Н.Э. Баумана
gapyu@bmstu.ru

Стремительное развитие информационных технологий всё значительно расширяет круг задач, которые можно решить с их помощью. Конечно, среди них преобладают прикладные (пользовательские) задачи, причем работе пользователя с приложением или продуктом уделяется основное внимание. Но отойдём немного от пользователя, как потребителя окончательного варианта работы команды программистов, и обратимся к программистской практике, её сложностям. Очевидно, что при усложнении решаемых задач объём работы программиста увеличивается: логическое решение задачи и его реализация усложняются. Если при этом используются недостаточно эффективные средства разработки, то будни разработчика становятся каторжными. Чтобы этого избежать, программисты вынуждены искать возможные средства, максимально упрощающие их труд и облегчающие реализацию их идей.

Рассмотрим трудности, возникающие при программировании приложений в вебе.

При решении каких-либо сложных задач возникает ситуация, когда нужно выполнить большой объем рутинной работы и избавиться от неё обычными средствами ООП не получается.

В современных проектах и абстракция, и наследование принимают вид многоуровневых. Наследование само по себе подразумевает многоуровневость: от общих классов к специфическим их сложность растёт. В итоге, программист, который работает с конечным классом, использует только 10-20 % его возможностей в каждом конкретном случае (для примера, в ASP.NET GridView имеет сотни свойств, многие из которых вообще в проекте не понадобятся). В таком наборе свойств и методов, с одной стороны,

сложно ориентироваться программисту, с другой, они занимают оперативную память и процессорное время. В многоуровневой абстракции на верхних уровнях находятся объекты реального мира, чем ниже уровень, тем больше сущностей исполняющей среды. Этот подход хорош, но тем не менее тоже содержит в себе определённую сложность. Образуется очень много классов, и программист должен держать в голове много информации при построении и сопровождении системы. Кроме того, методы, связывающие абстрактные классы, не всегда являются оптимальными. Также, в ASP.NET нет generic типов, что усугубляет ситуацию.

В качестве примера рутинной работы можно привести программирование веб-приложения, работающего с базой данных, в частности, разработку для каждой таблицы схожих форм создания, удаления и так далее. Другой пример — создание однотипных страниц таких, что каждая следующая отличается от предыдущей лишь источником или представлением данных. Это наиболее простые примеры из возможных на практике.

Найденным средством автоматизации рутинной работы и преодоления сложности, связанной с использованием многоуровневого наследования и абстракции является *кодогенерация*. Её можно рассматривать как новый этап или переход на новый уровень работы. Если сказать точнее, термин «кодогенерация» уже используется для обозначения процесса создания компилятором исполнимых команд, поэтому для раскрытия сути найденного средства автоматизации будем использовать термин «*генерация исходного кода*» программного обеспечения.

Хотя термин «генерация исходного кода» не на слуху, с самой технологией уже можно столкнуться; она уже применяется, например, в таких областях и продуктах, как:

1. Интернет – World Wide Web.

В настоящее время стандартом предоставления информации для Web-сайтов является HTML. Но, создавая сайт только из статических HTML-страниц, мы не получим динамического сайта, отвечающего современным требованиям. Поэтому разработчики сайтов пишут ПО (движок сайта), целью которого является генерация HTML-кода для браузера пользователя. Примеры систем, которые помогают программистам в этом: ASP.NET, Java, PHP, Perl, RubyOnRails и другие.

Также HTML код генерируют и клиентские скрипты (в частности JavaScript).

2. Java, .NET и схожие платформы.

Такие платформы, как .NET и Java генерируют на основе высокоуровневого кода низкоуровневый. В .NET эта возможность активно используется для многоязычности

(многие языки высокого уровня генерируют одинаковый низкоуровневый код). Для WWW складывается подобная ситуация, многие языки генерируют HTML-страницы.

3. Визуальные редакторы средств разработки.

Начиная с Microsoft Visual Basic, в MS включены графические редакторы Windows-форм. Стоит программисту перенести кнопку на форму, как генератор уже позаботился: записал в специальном файле свойства кнопки и её местоположение. Изначально код генерировался на специальных языках разметки (Delphi, Visual Basic), но потом его стали генерировать на том же языке, на котором пишет программист. Позже этот же подход стал применяться и в программировании Web-приложений. Сейчас сложно представить среду программирования без визуальных редакторов, которые являются генераторами исходного кода.

4. MVC (Model-Veiw-Controller) фреймворки.

Их неотъемлемыми компонентами стали скрипты-генераторы, с помощью которых на основе xsd-файла со схемой базы данных создаются типизированные наборы данных (typed datasets), Модель может быть сгенерирована из базы, можно создать представления (View) и контроллеры (Controller) на основе модели для работы с базой данных. Как правило, сгенерированные контроллеры содержат стандартные методы записи, удаления, редактирования и так далее. Скорее всего, сгенерированные таким образом представления и контроллеры необходимо будет как-то менять и добавлять к ним дополнительные, то есть, такая генерация не является исчерпывающей. Она осуществляется с помощью стандартных подписей на английском языке (edit, delete, create), но можно обеспечить её русификацию.

5. LINQ

В третьей версии C# появился LINQ (Language Integrated Query). В общем подходе к объектам (LinqToObject) используется генерация анонимных методов и классов. LinqToObject позволяет работать с коллекциями объектов, делать выборки из них, каким-либо образом обрабатывать выборки или объекты, причем очень эффективно и в простой и понятной для программиста форме. В подходе LinqToSql используется генерация SQL кода для Microsoft SQL Server. Готовится реализация LINQ для Oracle и других источников данных. Отличная идея — из одного высокоуровневого языка генерируются реализации под конкретные типы данных.

Практически на любом языке можно написать генератор исходного кода, который будет помогать программисту. Встречаются примеры генераторов на Php, Java, Pyton, генерация с помощью шаблонов XSLT, C# (через ASP.NET MVC). Рассмотрение всех

перечисленных генераторов и программ, в которых уже используется генерация исходного кода, в рамках данной статьи невозможно, поэтому рассмотрим одну из них – технологию генерации кода ASP.NET MVC.

Технология генерации кода на основе шаблона в ASP.NET MVC – это t4 шаблонизация. Мы будем рассматривать шаблоны, написанные на языке C#.

В Microsoft Visual Studio T4 шаблоны – это файлы, содержащие текстовые блоки и управляющую логику, которая записывается в виде программного кода на языке C# или Visual Basic (имеют расширение .tt). На их основе можно создать текстовый файл: веб-страницу, файл ресурсов или файл исходного кода.

Существует 2 типа текстовых шаблонов T4:

1. Шаблон времени выполнения (препроцессный) - Run time T4 text template.

Такой шаблон сначала компилируется: происходит анализ инструкций по обработке текста и блоков кода, T4 создает конкретный класс *TextTransformation* и компилирует его в сборку. Затем она выполняется: T4 создает экземпляр *GeneratedText-Transformation* класса, вызывает его *TransformText* метод и хранит строку, которую он возвращает в выходном файле:

```
string webResponseText = new MyTemplate().TransformText();
```

Шаблон можно отлаживать, как обычный код, расставлять в нем точки останова и так далее (для этого необходимо установить атрибут директивы *template debug*=”true”).

Написанный программистом код содержит как некоторые повторяющиеся части, так и уникальные, уникальный код выделяется в операторных блоках <# #>, а повторяющийся напрямую транслируется в выходной файл. Вычисляемые выражения, результат которых представляется в виде строки и транслируется в выходной файл, помещаются между <#= #>. Кроме того, предусмотрены блоки функций класса <#+ #>, которые используются для добавления методов, полей, свойств или вложенных классов в текст шаблона. Они также могут содержать методы или поля, доступные из других шаблонов при «наследовании» (пример взаимодействия шаблонов будет рассмотрен ниже).

Рассмотрим пример. Сгенерируем на основе абстрактного класса Shapes классы-наследники, имена которых объявим в enum ClassesNames: Rectangle, Circle, Square, Triangle, Romb. Первая часть кода – использование директив, они выделяются в <#@ #>:
template – директива, определяющая, как шаблон должен быть обработан (атрибут *language* задаёт язык, на котором будет сгенерирован код, *debug* – отладка шаблона, *hostspecific* – при значении “true” свойство с именем Host будет добавлено в класс,

созданный с помощью текстового шаблона. Host – это ссылка на узел. *this.Host* можно привести к типу *IServiceProvider* для доступа к функциям Visual Studio);

output – задаёт формат и кодировку выходного сгенерированного файла (атрибут *extension* – расширение выходного файла, возможная кодировка us-ascii, utf-16BE, utf-16, utf-8, utf-7, utf-32);

assembly – служит для подключения сборок (атрибут *name* – имя сборки);

import – служит для импорта пространства имен (атрибут *namespace* – имя пространства имён).

Следующая часть – объявление пространства имён MyNameSpace и класса Shapes: его свойства сгенерируются из соответствующих enum (StringPoperties, IntProperties), с помощью двух циклов foreach, далее – объявление наследников (их имена в enum ClassesNames). Весь код, не выделенный в управляющие блоки (<# #>, <#= #>, <#@ #>, <#+ #>), напрямую транслируется в выходной файл.

```
<#@ template language="C#" debug="true" hostspecific="true" #>
<#@ output extension="cs" #>
<#@ assembly name="System.Core" #>
<#@ assembly name="System.Windows.Forms" #>
<#@ import namespace="System" #>
<#@ import namespace="System.IO" #>
<#@ import namespace="System.Diagnostics" #>
<#@ import namespace="System.Linq" #>
<#@ import namespace="System.Collections" #>
<#@ import namespace="System.Collections.Generic" #>

namespace MyNameSpace
{       public class Shapes
        { //генерация всех свойств внутри класса слов - подстановка названий из перечислений
         <#  foreach(var Property in Enum.GetNames(typeof( StringProperties))) {#>
                    public string <#= Property #> {get; set;}
        <# } #>
         <#  foreach(var Property in Enum.GetNames(typeof( IntProperties))) {#>
                    public int <#= Property #> {get; set;}
          <# } #>
          //генерация классов-наследников
         <#  foreach(string ClassName in Enum.GetNames(typeof( ClassesNames)))  { #>
                    public class <#= ClassName #> : Shapes
```

```
                    {   //код или методы внутри класса
                        //конструкторы
                        <#= ClassName #>();
                        <#= ClassName #>(string Name, string Color, string Type, int Heigh, int Weigh)
                    { name = Name;  color = Color;  type = Type;  heigh=Heigh; weigh=Weigh; }
                    //виртуальный метод, вычисляющий площадь фигуры  для каждого класса
                        public override virtual int AreaCount()
                        { return 0; }
                    }
            <# } #>
        //код или методы внутри класса непосредственно скопируются в код
        //конструкторы
        Shapes();
        Shapes(string Name,string Color,string Type, int Heigh, int Weigh)
        { name = Name;  color = Color;  type = Type;  heigh=Heigh;  weigh=Weigh; }
        //виртуальный метод, вычисляющий площадь фигуры
        public virtual int AreaCount()
        {    return 0; }
    }
}
<#+  enum ClassesNames
   {      Rectangle, Circle,  Square,  Triangle,  Romb }
   // Общие свойства фигур
      enum IntProperties
   {      heigh, weigh, area }
      enum StringProperties
   {    color,  name, type  }
#>
```

Сгенерированный текст одного класса Shapes и Rectangle:

```
namespace MyNameSpace
{       public class Shapes
        {    //генерация всех свойств внутри класса слов - подстановка названий из перечислений
                    public string color {get; set;}
                    public string name {get; set;}
                    public string type {get; set;}
                    public int heigh {get; set;}
                    public int weigh {get; set;}
                    public int area {get; set;}
```

```
            //генерация классов-наследников
            public class Rectangle : Shapes
            {  //код или методы внутри класса
               //конструкторы
               Rectangle();
               Rectangle(string Name,string Color,string Type, int Heigh, int Weigh)
         {     name = Name;  color = Color;  type = Type; heigh=Heigh;  weigh=Weigh; }
        //виртуальный метод, вычисляющий площадь фигуры - для каждого класса
              public override virtual int AreaCount()
              {   return 0;  }
             }
//конструкторы
        Shapes();
        Shapes(string Name,string Color,string Type, int Heigh, int Weigh)
        {  name = Name;  color = Color;  type = Type;  heigh=Heigh;  weigh=Weigh; }
        //виртуальный метод, вычисляющий площадь фигуры
        public virtual int AreaCount()
        {  return 0; }
     }
}
```

Шаблоны могут взаимодействовать между собой, например, вызывать методы тех шаблонов, чьими «потомками» являются. Понятия наследования для шаблонов нет, но связь подчинённого шаблона с главным устанавливается с помощью атрибута *inherits* (наследует) директивы *template*, в нём указывается имя шаблона-«предка». Базовый класс шаблона может быть абстрактным. «Наследником» у шаблона времени выполнения может быть только шаблон времени выполнения.

Рассмотри пример. Шаблон MyTextTemplate1 «наследуется» от SharedFragments и вызывает общий метод, записанный в шаблоне-«предке» (SharedText(2) – выводит на экран Shared Text и значение, переданное в функцию).

SharedFragments.tt:

```
<#@ template language="C#" #>
<#+  protected void SharedText(int n)
{ #>
  Shared Text <#= n #>
<#+ } #>
```

MyTextTemplate1.tt:

```
<#@ template language="C#" inherits="SharedFragments" #>
begin 1
```

```
  <# SharedText(2); #>
end 1
```

MyProgram.cs:

```
…
MyTextTemplate1 t1  = new MyTextTemplate1();
string result = t1.TransformText();
Console.WriteLine(result);
```

Результат на экране:

```
begin 1
   Shared Text 2
end 1
```

2. Шаблон времени разработки - Design-time T4 text template

Функционально он также служит для генерации части исходного кода и других ресурсов приложения. Этот шаблон можно написать таким образом, чтобы переменная часть кода зависела от данных модели, что целесообразно. При таком методе, регенерация во много раз надёжнее, чем исправление кода вручную.

Под моделью, в этом случае, понимается файл базы данных, XML-файл. T4 работает с *UML*-моделью и *Domain-specific Language* моделью (*DSL*). Как и предыдущий тип, они имеют текстовые блоки, программные (переменные части) <# #>, блоки выражений<#= #>, к нему также можно подключать файлы, сборки, пространства имён.

Для вывода сообщения об ошибках и предупреждений предусмотрены специальные методы: Error("An error message"); Warning("A warning message"); .Свойство *Custom tool* для такого файла-шаблона принимает значение *TextTemplatingFileGenerator.*

Стоит выделить для этих типов шаблонов общие черты и различия.

Общим для них является состав компонентов. Шаблоны могу включать:

- *текстовые блоки*, то есть такие блоки текста, которые не являются программными и транслируются без изменений в выходной файл;

- *директивы*, <#@ #>, к ним относятся *include, assembly, import, output, template*. Их довольно много, но эти являются основными;

- *управляющие блоки*, которые выполняются в процессе генерации. Все типы управляющих блоков уже встретились в примерах: это блок выражений - <#= #>, блоки класса<#+ #>, заключающие в себе методы, которые можно вызывать из шаблона, свойства или поля конкретного класса, и стандартный блок <# #>, который генерирует некоторую переменную часть кода.

Шаблоны можно *отлаживать*, *использовать* в них *базовые методы* типа WriteLine() и так далее.

Различия же таковы:

- шаблоны времени выполнения могут выполняться без VS.

- к шаблонам времени выполнения может быть применён атрибут директивы template inherits (наследуется). Если этот параметр не указан, будет создан класс без базового класс, который будет содержать реализацию всех стандартных методов класса *TextTransformation.* Если этот атрибут установлен, то шаблон будет создан с указанным базовым классом без реализаций стандартных методов класса *TextTransformation.*

Если рассмотреть пример практического применения технологии T4 для генерации контроллеров для базы данных, причем они содержат помимо стандартных некоторые необходимые разработчику методы, то можно выделить особенности генерации (код довольно громоздкий, поэтому его не приводим) и прийти к следующим выводам:

- в отличие от ранее рассмотренного примера генерации классов-наследников, в этом случае приходится писать много кода, поэтому, такая генерация оправдывается в случае многократного использования шаблона (большого количества однотипных таблиц), что показывает, что не во всех случаях удаётся обойтись несколькими строками для получения большого объема полезного кода. Причем и методы для таблиц должны быть однотипными, то есть возможна, в основном, генерация стандартных методов – get, edit, delete, create.

- при попытке создать некоторый «общий» шаблон, мы получим много лишнего в конкретном сгенерированном файле. Проблемой любого универсального метода является работа с деталями.

- после изменения шаблона, файлы, полученные с его помощью, необходимо перегенерировать. Регенерация исправленного кода более надёжна, чем исправление кода вручную. При необходимости можно настроить автоматическую регенерацию. Но если логика нашей базы изменилась кардинально, то возможно, придётся переписывать шаблоны.

- принцип работы T4 заключается в трансляции во входной файл большого количество текста, введённого нами, и «проработке» только некоторых его фрагментов. Получается, что то, что в нём написал программист, то и получил на выходе, тогда полезная работа генератора сводится только к проработке переменных частей, что составляет лишь малую часть файла.

Таким образом, можно сделать вывод о том, что хотя шаблонизация T4 является на сегодняшний день одним из лучших промышленных решений по проблеме генерации

исходного кода, ей необходима дальнейшая проработка и, возможно, определение новых подходов к решению проблемы генерации исходного кода.

Следующей технологией, которая позволяет облегчить жизнь программисту и связана с T4, является PowerShell.

Windows PowerShell — это разработанная Microsoft командная оболочка Windows, которая включает в себя интерактивную командную строку и сопутствующий собственный язык для написания сценариев – скриптов. Компоненты PowerShell (командную оболочку и язык сценариев) можно использовать совместно и по отдельности.

Командная оболочка используется преимущественно для администрирования системы. Она схожа с командной строкой в её традиционном понимании, но команды являются *командлетами* – некоторыми специализированными классами .NET. Они наследуются от базового класса Cmdlet или от PSCmdlet (если необходимо взаимодействовать с исполняемой частью PowerShell – PowerShell runtime).

Командлеты могут принимать в качестве входных данных объекты, типы которых являются основными типами данных .NET и определяют его структуру, или их коллекции и параметры. Внутри командлетов могут использоваться встроенные команды PowerShell.

Синтаксис командлета:

Имя_командлета –параметр1 –параметр2 аргумент1 аргумент2

–параметр1: параметр без аргументов (переключатель)

Например, переключатель *–Recurse* распространяет действия на все подкаталоги.

–параметр2: имя параметра имеющего значение аргумент1

Например*, –Filter *.txt* означает, что и действия будут производиться только с текстовыми файлами.

аргумент2 - безымянный аргумент или параметр.

Реализации командлетов могут быть написаны на любом языке .NET и могут вызывать любые API, доступные .NET. PowerShell также предоставляет некоторые дополнительные API. Командлеты могут использовать API для доступа к данным напрямую.

Объекты PS могут иметь статические методы, их можно группировать, сортировать, фильтровать, выделять их свойства, агрегировать или производить произвольные действия (с помощью команлета Foreach-Objects { выполняемые действия }). Возможно создание .NET объектов. Оно используется, когда в PS нет подходящего объекта, но в .NET есть объект, обладающий нужной функциональностью.

Кроме того, каждый объект содержит информацию о самом себе (*самодокументация*), что облегчает программисту освоение возможностей работы как с конкретным объектом, так и со средой в целом.

Для повышения эффективности использования объектов PS поддерживает механизм *композиции команд или конвейеризации* - последовательного перенаправления выходного потока одной команды во входной поток другой команды, что позволяет передавать объекты или текстовую информацию между разными процессами.

Данные между командлетами передаются в виде полноценных объектов соответствующих типов (или их коллекций), а не потоком байтов, поэтому содержащиеся в объектах элементы сохраняют свою структуру и типы при передаче между командлетами (командами), и нет необходимости использования каких-либо алгоритмов сериализации. Информация об объекте извлекается из его свойств. Передаваемый объект также может содержать некоторые методы (функции), предназначенные для работы с данными. Они также становятся доступными для получающего их командлета.

Конвейер не зависит от числа передаваемых элементов, так как обработчик конвейера вызывается для каждого объекта отдельно. Также, благодаря последовательной передаче элементов снижается потребление ресурсов для сложных команд. Над командами конвейера (так же называемыми его элементами) можно производить операции фильтрации, группировки и сортировки по определенному критерию. Также можно изменять структуру элементов.

Выходной поток командлета можно форматировать с помощью специальных командлетов (Format-Table, Format-List, Format-Custom, Format-Wide). Например, можно представить выходной поток в качестве таблицы, столбцы которой содержат свойства объекта либо списка свойств, либо в виде пользовательского представления, либо выводить только одно свойство каждого объекта.

Для простого *перенаправления выводимой информации* используются символы < (сохранение с замещением имеющейся информации в файле) и << (добавление новой информации в файле к имеющейся).

В языке написания скриптов PowerShell, как и любом другом, определены *переменные* (пример: $conn). Пользовательская переменная имеет имя и тип, который явно указан или который определяется типом последнего присвоенного ей значения, если такого указания нет.

Ядро PS составляет модель типа .NET, но может возникнуть необходимость общаться с другими объектными моделями, поддерживаемыми ОС (COM, WMI, ADO,

ADSI) В этом случае PS не обращается к объектам напрямую, а использует объект промежуточного уровня типа [psobject], что позволяет обращаться одинаково к объектам всех типов.

Работа с массивами осуществляется привычным образом, но у неё есть особенность: в PS возможны отрицательные индексы элементов (в таком случае отсчитываются элементы с хвоста массива, последний элемент имеет индекс -1). Массивы могут быть полиморфными (содержать в себе объекты разных типов) или определённого типа, если он явно не указан. Помимо обычных массивов в PS используются хеш-таблицы или ассоциативные массивы. При использовании оператора присваивания, будет использоваться ссылочная адресация, как в обычных массивах.

В PS определены *стандартные операторы* (присваивания, сложения, умножения, инкремента, декремента, получения остатка от деления, сравнения, логические). Поскольку PS поддерживает полиморфизм, то поведение операторов для основных типов данных реализуется непосредственно интерпретатором, а не конкретным методом для данного типа. При попытке сложить две переменных, имеющих различные типы, переменная, находящаяся справа будет приведена к типу переменной, находящейся слева, если это возможно. Иначе будет выдана ошибка. Например, операция a*3 будет выполнена как «aaa» и результат будет иметь строковый тип. При попытке сравнить переменные разных типов, они приводятся к типу переменной, стоящей слева.

Среди операторов стоит выделить *операторы соответствия шаблону*. В качестве шаблона могут выступать *выражения с подстановочными символами* и *регулярные выражения*. Первые могут быть представлены одним произвольным символом, диапазоном символов, любым символом из диапазона, любым количеством произвольных символов, следующих за определённым. Операторы осуществляют сравнение и выявляют совпадение или несовпадение с учетом подстановочного символа текста. Если возникает необходимость проверки более сложных условий, чем в шаблонах с подстановочными символами, используются регулярные выражения.

В PS определены стандартный *оператор условия* (if-elseif-else), *циклы* while, do-while, for, foreach, *метки*, *break, continue*, *инструкция switch* (кроме обычных операторов сравнения можно использовать операторы проверки на соответствие шаблону).

Также в PS присутствуют *функции,* которые могут возвращать и один объект, и массив. Обрабатывать входные аргументы они могут помощью переменной $Args или путем задания формальных параметров. $Args содержит массив параметров, указанный при запуске последней функции. Функции в PS полиморфны, то есть способны

обрабатывать значения различных типов. Если при вызове функции задать больше переменных, чем она принимает, они будут сохранены в массиве $Args. По умолчанию функция определяет типы своих параметров и свой возвращаемый тип автоматически, но его можно задать и самостоятельно.

Особенность *обработки ошибок* в PS заключается в том, что здесь ошибки делятся на критические (прерывающие выполнение команды) и некритические (при их возникновении выполнение команды продолжается). При возникновении некритических ошибок информация о них помещается в специальную переменную типа ErrorRecord, который записывается в специальный поток ошибок. Для перехвата некритических ошибок определенной команды без перенаправления их в выходной поток существует параметр – Errorvariable.

Мы можем использовать все языковые возможности PS в *сценариях* (файлы с расширением .ps1): различные конструкции PowerShell, утилиты командной строки и обращения к обычным классам .NET, объектам WMI или COM.

Разбор и обработка сценариев и функций одинаковы. Формальные параметры в сценарии задаются с помощью специальной инструкции Param. Если формальные параметры не указаны явно при вызове функции, используются их значения по умолчанию. Выход из сценариев осуществляется либо после выполнения в нем последней инструкции, либо по команде Exit.

При работе с PS можно использовать различные *хранилища данных*, такие как файловая система или реестр Windows, которые предоставляются PS посредством поставщиков (англ. *providers*). Их задачей является обеспечение возможности работы с другими хранилищами, как с локальной файловой системой и регулирование доступа к данным. Хранилища данных представляются через буквы дисков и иерархическую структуру внутри них (директории). PS поддерживает собственные *виртуальные диски*, доступные только из оболочки PS и связанные с хранилищами данных разных типов.

В заключение рассмотрим пример работы с PS рассмотрим сценарий выгрузки данных из таблицы SQL server :

```
# загружаем библиотеку ADO.NET для работы с данными
[System.Reflection.Assembly]::LoadWithPartialName("System.Data")
# создаём соединение с БД SQL Server и открываем его
$conn = New-Object "System.Data.SqlClient.SqlConnection" -ArgumentList "server=mysrv;
database=mydbname; integrated security=SSPI"
$conn.Open()
# создаём sql запрос и указать текст запроса
```

```
$cmd = $conn.CreateCommand()
$cmd.CommandText = "select * from [dbo].[myTable]"
# создаём адаптер, таблицу и заполняем ее данными
$adapter = New-Object "System.Data.SqlClient.SqlDataAdapter" -ArgumentList $cmd
$table = New-Object "System.Data.DataTable" -ArgumentList "Table1"
$adapter.Fill($table)
# делаем экспорт, импорт и проверяем наличие данных
$table | Export-Csv "c:\data.csv" -Encoding "Unicode" -Delimiter "`t"
Import-Csv "c:\data.csv" -Delimiter "`t" | Out-GridView
```

Список литературы

1. Попов А. Введение в PowerShell. СПб: БХВ-Петербург, 2009. 464 с.
2. Кох Ф. Windows PowerShell. Бёрн: Microsoft Switzerald, 2007. 44 с.
3. Finke D. Windows PowerShell for Developers. Sebastopol: O'Really, 2012. 210 p.
4. Официальный ресурс MSDN для разработчика. Режим доступа: http://msdn.microsoft.com/en-us/library/bb126445.aspx (дата обращения 15.04.2014).