Научно-технический вестник Санкт-Петербургского государственного университета информационных технологий, механики и оптики. Вып. 53. Автоматное программирование. 2008, с. 230–237.

УДК 004.4’242

ДЕКЛАРАТИВНЫЙ ПОДХОД К ВЛОЖЕНИЮ И НАСЛЕДОВАНИЮ АВТОМАТНЫХ КЛАССОВ ПРИ ИСПОЛЬЗОВАНИИ ИМПЕРАТИВНЫХ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ

А. А. Астафуров

В работе предлагается декларативный подход к реализации автоматных объектов при использовании объектно-ориентированных императивных языков программирования со статической проверкой типов. Отличительной особенностью предлагаемого подхода является возможность применения наследования и вложения макросостояний.

Введение

При реализации автоматов с использованием объектно-ориентированных языков применяются различные подходы. Их можно разделить на императивные и декларативные.

Наиболее распространенным императивным подходом является использование паттерна проектирования *State* [1, 2], основными достоинствами применения которого являются распределение специфического поведения между состояниями и выполнение переходов в явном виде непосредственно в коде.

Недостатком применения этого паттерна является необходимость создания сложной иерархии при наличии большого числа состояний у автомата, которая может быть решена при использовании шаблона *Decorator* [2, 3].

Основным недостатком императивного подхода в целом при реализации автоматных программ является то, что в коде автоматного объекта присутствуют такие лишние детали, как явное делегирование вызова из контекста конкретному состоянию или вызов вложенного автомата.

В данной работе предлагается подход, являющийся компромиссом между декларативным и императивным подходами. От декларативного подхода используется возможность описания структуры автомата, а от императивного – описание логики переходов, так как она является частью автоматного кода и императивна по своей природе.

Рассматриваемый подход к реализации автоматных классов является декларативным, но ориентирован на использование объектно-ориентированных императивных языков со статической проверкой типов. Подход иллюстрируется примерами программ на языке C#.

Описание подхода

При задании поведения автоматов будем использовать модифицированную нотацию диаграмм *Statecharts* [4]. Основное отличие между применением диаграмм *Statecharts* и *SWITCH*-технологией [5] состоит в том, что в *SWITCH*-технологии явно водится понятие автомата. Семантика используемых в *SWITCH*-технологии графов переходов близка к семантике диаграмм *Statecharts*, но не эквивалентна ей. В *SWITCH*-

технологии вводится понятия *«автомат»* и *«вложение автоматов»*, и отсутствуют понятия *«вложенное»* и *«ортогональное»* состояния. Вложение и ортогонализация состояний в *SWITCH*-технологии реализуется при помощи вложения автоматов и введения понятия *«система автоматов»*. *SWTICH*-технологию удобнее применять в документации: при использовании вложения автоматов не возникает проблем с нотацией, как это происходит при использовании вложенных ортогональных состояний в диаграммах *Statecharts*. Так, например, при использовании диаграмм *Statecharts* возникают трудности с расположением названий макросостояний, содержащих вложенные ортогональные состояния [4].

В предлагаемом подходе автомат с вложенными в него состояниями рассматриваются как набор *макросостояний*, так как у состояния нет никакой специфики по отношению к автомату. Макросостояние может включать в себя другие макросостояния с их состояниями. При реализации библиотеки для поддержки автоматного программирования, между состоянием и автоматом также нет никакой разницы. В рамках библиотеки, разрабатываемой в настоящей работе, в системе будет создан класс, определяющий состояния, который помимо стандартных свойств, таких как текущее состояние, вложенные макросостояния и т. д., реализует интерфейс, описывающий допустимые входные воздействия.

При этом отметим, что в диаграмме *State Machine* в языке *UML 2* (диаграмме *State Chart* в языке *UML*) [6] понятие автомата также отсутствует: существуют только состояния, которые могут включать другие состояния. Таким образом, решается вопрос вложения автоматов.

Вложение автоматных объектов

Благодаря отождествлению понятий состояния и автомата и введения термина «макросостояние», определим вложение автоматных объектов следующим образом: при получении события состоянием оно сначала передается всем его вложенным состояниям, а затем выполняется действие, связанное с этим событием. Таким образом, в данной работе будем считать, что событие сначала передается всем вложенным состояниям, а затем выполняется действие внутри состояния.

Наследование автоматных объектов

Рассмотрим вопрос о наследовании автоматов. Разрабатываемый подход должен позволить наследовать автоматы, переопределяя состояния и правила перехода между ними. Наследование автоматных объектов основано на переопределении состояний базового автоматного объекта: производный объект должен переопределить поведение базового объекта как минимум в одном из его состояний. В производный объект могут быть добавлены новые состояния и переходы между ними [7].

Декларативный подход при использовании императивных языков

Декларативный подход и императивный язык. Несмотря на императивность многих современных языков программирования, при их использовании тоже можно применять декларативный подход. Для этого необходимо инкапсулировать все недекларативные детали внутри библиотеки или модели, которая будет управляться декларативной мета-информацией, позволяя таким образом использовать декларативные конструкции. В зависимости от языка и платформы это может быть достигнуто разными способами. Например, в языке *Java* и платформе *Microsoft .NET* существует механизм отражения (*reflection*), позволяющий получать доступ к компонентам класса во время

исполнения. Это дает возможность декларативно описывать поведение классов, а затем, во время исполнения, добавлять императивное поведение.

Декларативный подход на основе атрибутов. Для более детального рассмотрения реализации декларативного подхода при применении императивных языков в качестве примера используем язык *C#* и платформу *Microsoft .NET*. Наличие такой конструкции, как *атрибут* (*attribute*), позволяет применить декларативный подход для этого языка. Атрибуты в языке *C#* – это конструкции, позволяющие добавлять мета-информацию, как к членам класса, так и к самим классам. В дальнейшем, информация об атрибутах может быть получена во время исполнения библиотекой, инкапсулирующей недекларативные составляющие программы при помощи механизма отражения.

Рассмотрим подходы к выполнению декларативной программы в императивной среде.

Механизм *Context Bound Objects*. *Context Bound Object* (объект, привязанный к контексту) – это специальный объект, позволяющий контролировать все вызовы, поступающие к нему из контекста.

Контекст (в терминах *Microsoft CLR*) – это окружение, устанавливающее правила, которым будут следовать объекты, находящиеся в нем [8]. Таким образом, если объект привязан к контексту, он также привязан и к его правилам. Контекст создается при активации объекта. Общение объекта с внешним миром происходит при помощи сообщений, что не имеет отношения к автоматам. Среда *Microsoft CLR* предоставляет механизмы, позволяющие встраиваться в цепочку обработки сообщений. Это необходимо для их перехвата, а также для генерации или модификации сообщений, проходящих через границу контекста.

Отметим, что использование механизма *Context Bound Object* оправдано в тех случаях, когда производительность системы не критична. Этот механизм выгодно использовать при создании прототипа будущей декларативной системы.

Инструментализация сборок. Данный подход эффективен для продуктов, критичных к скорости, так как все вызовы выполняются напрямую, без дополнительных затрат на управление очередью сообщений, как это происходит в *Context Bound Object*. Явным недостатком рассматриваемого подхода являются трудности при отладке приложения: из-за того, что байт-код сборки модифицируется, код, полученный в результате этой модификации, перестает соответствовать отладочной информации, сгенерированной на этапе компиляции.

Реализация на платформе *.NET*

Описание библиотеки *DOME*. В рамках данной работы для реализации декларативного подхода при использовании императивных языков *C#* и *Visual Basic.NET* была разработана библиотека классов *DOME* (*Declarative Object Machines Extension*). На рис. 1 приведена диаграмма основных классов этой библиотеки.

На рисунке `State` – базовый класс для состояний, а свойство `Container` – экземпляр класса *State*, которому принадлежит рассматриваемое состояние. Это свойство необходимо для доступа к объемлющему состоянию при вложении состояний. Свойство `CurrentState` – текущее вложенное состояние. Оно используется при вложении состояний, а также непосредственно в коде автоматного объекта. Метод `SetState(Type state)` изменяет текущее состояние. В качестве параметра ему необходимо передать тип (класс) состояния, в которое требуется перейти. `StateAttribute` – атрибут, применяемый к классам-наследникам `State` для описания возможных состояний для данного макросостояния. `Type` – *CLR*-тип (класс) состояния. `Name` – имя состояния. *Конструктор* – конструктор с одним параметром, принимает тип (класс) состояния. Перегруженный конструктор с двумя параметрами предоставляет

механизм переопределения состояний. Для этого необходимо в качестве значения параметра `Type` указать тип (класс) нового состояния, а в качестве значения параметра *Overrides* – тип (класс) состояния, которое будет переопределено. В этом случае в ходе создания экземпляра атрибута будет также произведена проверка того, что переопределяемое состояние присутствует выше в иерархии.

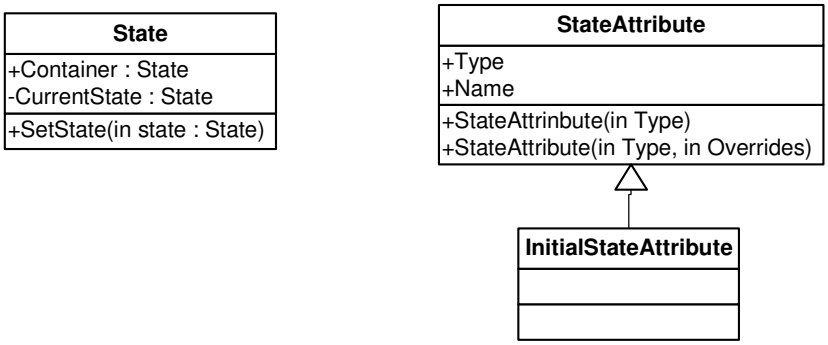

Рис. 1. Основные классы библиотеки

`InitialStateAttribute` имеет ту же семантику, что и атрибут `StateAttribute`, и применяется для обозначения начального состояния автомата или начального вложенного состояния.

Реализация автоматных объектов

В качестве первого примера рассмотрим простейший автомат, состоящий из двух состояний: *ON* и *OFF* (рис.2).

Определим три интерфейса:`IOn` – интерфейс для состояния ON, содержащий только один метод `E0` – единственное событие, которое обрабатывается в этом состоянии; `IOff` – интерфейс для состояния OFF, содержащий метод `E1` – событие, обрабатываемое в этом состоянии; `ISwitch` – интерфейс, описывающий автомат в целом и реализующий интерфейсы `IOn` и `IOff`.

Эти интерфейсы объявляются следующим образом:

```
public interface IOn
{
 void E0();
}

public interface IOff
{
 void E1();
}

public interface ISwitch : IOn,IOff
{

}
```

После этого необходимо реализовать классы состояний ON и OFF:

```
class On : State, IOn
```

```
{
 public void E1()
 {
  Container.SetState(typeof(Off));
 }
}
class Off : State, IOff
{
 public void E1()
 {
  Container.SetState(typeof(On));
 }
}
```

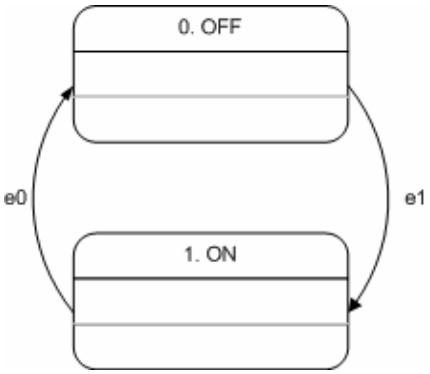

Рис. 2. Автомат выключателя

Поясним приведенный фрагмент кода. Поле `Container` является ссылкой на автомат, которому принадлежит данное состояние. Метод `SetState(Type)` изменяет текущее состояние автомата.

После этого реализуем автомат, который, как отмечено выше, рассматривается как набор макросостояний и поэтому наследуется от класса `State`, реализуя при этом интерфейс `ISwitch`:

```
[State(typeof(On)), State(typeof(Off))]
class Switch : State, ISwitch
{
 public void E0() { }
 public void E1() { }
}
```

Обратим внимание на атрибуты класса `Switch`. Они указывают на то, что автомат, описанный этим классом, будет содержать два состояния `On` и `Off`, описанные, как показано выше.

Во время выполнения программы все вызовы методов класса `Switch` будут перехватываться и передаваться методу текущего состояния с набором параметров, идентичным вызываемому методу. После этого выполняется метод автомата. В случае отсутствия у текущего состояния метода с набором параметров, идентичным вызываемому методу, вызов перейдет непосредственно к методу автомата.

Реализация вложенных состояний

Для описания вложенных состояний рассмотрим более сложный пример (рис. 3).

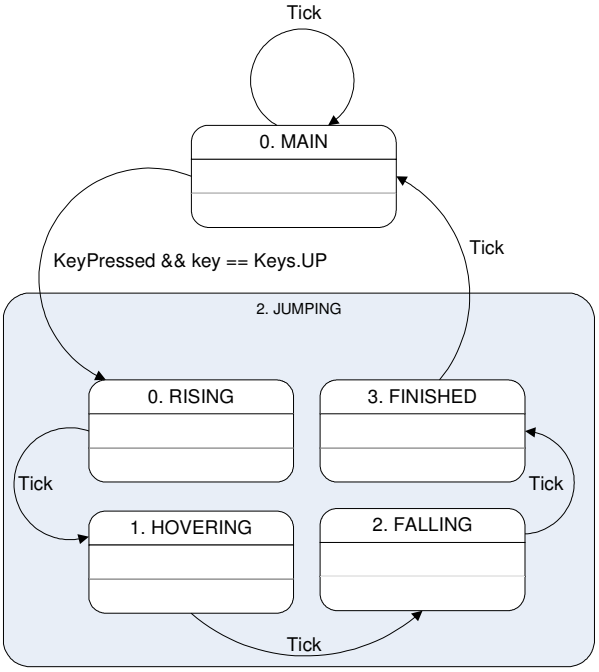

Рис. 3. Диаграмма состояний бойца с детализацией состояния «Прыжок»

Пусть имеется диаграмма состояний героя-бойца (*Fighter*) из компьютерной игры, в котором детализировано состояние «Прыжок».

Рассмотрим, как реализуются вложенные состояния в исходном коде:

```
    [InitialState(typeof(Jumping.Rising)),
State(typeof(Jumping.Falling)),
    State(typeof(Jumping.Hovering)),
    State(typeof(Jumping.Finished))]
    public class Jumping : State, IFighter
    {
     public void Tick()
     {
      if (CurrentState is Finished)
       Container.SetState(
       typeof(Fighter.Main));
     }

     public void ButtonPressed(Keys key)
     {}
     public bool InAir()
     {
      return true;
     }

     public class Rising : State, ITickable
     {
      public void Tick()
```

```
    {
     Console.WriteLine("rising");
Container.SetState(typeof(Hovering));
    }
   }

   public class Hovering : State, ITickable
   {
   public virtual void Tick()
   {
    Console.WriteLine("hovering");
    Container.SetState(
    typeof(Falling));
   }
}

public class Falling : State, ITickable
{
 public void Tick()
 {
  Console.WriteLine("falling");
  Container.SetState(
  typeof(Finished));
 }
}

public class Finished : State, ITickable
  {
   public void Tick()
   {
   }
  }
}
```

Рассмотрим состояния, вложенные в состояние `Jumping`. В этом случае при реализации вложенных состояний `Raising`, `Hovering`, `Falling`, `Finished` событие в состоянии `Jumping` сначала будет передано текущему вложенному состоянию, а затем обработано соответствующим методом состояния `Jumping`. Как и в предыдущем примере, вызов передается при помощи перехвата сообщения с использованием механизма *Context Bound Object*, описанного в разделе с тем же названием.

Реализация наследования

При использовании декларативного подхода достаточно просто и интуитивно понятно удается описывать наследование автоматов. Для этого достаточно создать класс-наследник базового класса «автомат». При этом для переопределения состояний необходимо использовать атрибут `State`. Таким же образом можно наследовать и базовые состояния, переопределяя их вложенные состояния или добавляя новые.

В качестве примера, рассмотрим задачу переопределения одного из состояний в автомате `Fighter` из предыдущего примера. Пусть необходимо переопределить состояние `Hovering`, вложенное в состояние `Jumping`. Для этого создадим класс

EasternFighter, являющийся наследником класса Fighter, и с помощью атрибутов опишем переопределение состояния Fighter.Jumping. Для этого заменим это состояние на класс Fighter.EasternJumping, который будет содержать переопределенное вложенное состояние Hovering.

Программно это выражается следующим образом:

```
[State(typeof(EasternJumping), typeof(Fighter.Jumping))]
public class EasternFighter : Fighter
{
}
```

Теперь необходимо создать наследника класса-состояния Jumping, чтобы переопределить его вложенное состояние Hovering новым состоянием EasternHovering:

```
[State(typeof(EasternHovering),
typeof(Fighter.Jumping.Hovering))]
public class EasternJumping : Fighter.Jumping
{
}
```

Наконец, необходимо реализовать само состояние EasternHovering, создав наследника от класса Jumping.Hovering:

```
public class EasternHovering : Fighter.Jumping.Hovering
{
 public override void Tick()
 {
  Console.WriteLine("Eastern Hovering");
  Container.SetState(
  typeof(Fighter.Jumping.Falling));
 }
}
```

Таким образом, когда новый автомат будет вызван, то при переходе в состояние Jumping будет использован новый класс EasternJumping, который, в свою очередь, использует переопределенное состояние EasternHovering, о чем он сообщает при протоколировании на консоль.

Заключение

В данной работе предложен декларативный подход к реализации автоматных объектов. При этом описаны способы реализации такого подхода при использовании императивных языков, а также предложена библиотека *DOME* (*Declarative Object Machines Extension*), позволяющая реализовать этот подход на платформе *Microsoft .NET*. Использование библиотеки проиллюстрировано примерами реализации наследования и вложения. Эта библиотека будет опубликована на сайте http://is.ifmo.ru/..

Данный подход к реализации автоматов позволяет совместно использовать декларативный и императивный стили программирования, позволяя применять декларативный объектно-ориентированный подход к реализации автоматов с помощью императивных языков. Благодаря тому, что данная концепция описана на уровне парадигмы

декларативного программирования без привязки к конкретному языку программирования, ее можно применять и для других императивных языков, например, для языка *Java*.

В дальнейшем планируется более детально рассмотреть вопрос создания библиотеки подобной библиотеке *DOME* для языка *Java*, а также изучить вопрос о виртуальных и не виртуальных состояниях и их переопределении.

Литература

1. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб.: Питер, 2001.
2. Adamczyk P**.** The Anthology of the Finite State Machine Design Patterns / The 10th Conference on Pattern Languages of Programs, 2003.
3. Odrowski J., Sogaard P. Pattern Integration – Variations of State / Proceedings of PLoP96.
4. Harel D. Statecharts: A visual formalism for complex systems // Sci.Comput. Program. 1987. Vol.8, pp. 231–274.
5. Шалыто А.А. SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998. http://is.ifmo.ru/books/switch/1
6. Шопырин Д.Г. Методы объектно-ориентированного проектирования и реализации программного обеспечения реактивных систем. Диссертация на соискание ученой степени кандидата технических наук. СПбГУ ИТМО, 2005. http://is.ifmo.ru/disser/shopyrin_disser.pdf
7. Буч Г., Рамбо Д., Джекобсон А. UML. Руководство пользователя. М.: ДМК. 2000.
8. Box D., Sells C. Essential .NET Vol. 1, The Common Language Runtime. NJ: Addison-Wesley, 2002.