УДК 681.3.07

ОБЪЕКТНО-ОРИЕНТИРОВАННАЯ РЕАЛИЗАЦИЯ КОНЕЧНЫХ АВТОМАТОВ НА ОСНОВЕ ВИРТУАЛЬНЫХ МЕТОДОВ

Д. Г. Шопырин,
старший программист
ЗАО «Транзас технологии»

В работе рассматривается проблема совместного использования объектно-ориентированной и автоматно-ориентированной технологий программирования. Подробно обсуждаются вопросы интеграции автомата в объектно-ориентированную систему и реализации автомата в объектно-ориентированном стиле.

The paper addresses the problem of merging object-oriented and automaton-based programming technologies. There are two major questions how to integrate an automaton into an object-oriented system and how to implement an automaton in the object-oriented fashion.

Введение

Проблема совместного использования объектно-ориентированной и автоматно-ориентированной технологий программирования рассмотрена в работах [1–6].

В традиционной объектно-ориентированной системе объекты взаимодействуют между собой через интерфейсы. Каждый интерфейс – это контракт между объектом и его клиентами. Интерфейс определяет один или более методов. Каждый метод может рассматриваться как событие (сообщение) с параметрами и, возможно, возвращаемым значением.

Система может содержать *объекты с выделенным состоянием*. Поведение таких объектов зависит от *состояния*, которое может быть представлено в виде скалярного значения. Текущее состояние объекта выделено. Одним из способов описания такого поведения являются *конечные автоматы*.

Объект с выделенным состоянием может быть реализован посредством следующей трехуровневой структуры:

1) традиционный объектно-ориентированный интерфейс;

2) промежуточный уровень, конвертирующий методы в события;

3) контекст, реализующий логику с выделенным состоянием.

Предлагаемый в данной работе подход удовлетворяет основным принципам объектно-ориентированной парадигмы.

1. *Инкапсуляция.* Факт наличия логики с выделенным состоянием скрыт от клиентов объекта.

2. *Полиморфизм.* Если имеется несколько объектов с разным поведением, но с одинаковым интерфейсом, то клиент может взаимодействовать с ними универсальным способом.

3. *Наследование.* Поведение объекта с выделенным состоянием может быть расширено с помощью стандартного механизма наследования.

Объектно-ориентированный интерфейс

На интерфейс объекта с выделенным состоянием не накладывается никаких дополнительных ограничений. Интерфейс может содержать любое количество методов. Метод может принимать произвольное число параметров и иметь возвращаемое значение. Некоторые методы могут быть объявлены константными.

Например, интерфейс, предназначенный для управления доступом к файлу, реализуется следующим образом:

```
struct IFileAccess {
  virtual void Open( string const& _mode ) = 0;
  virtual void Close() = 0;
  virtual bool CanRead() const = 0;
  virtual bool CanWrite() const = 0;
};
```

Промежуточный уровень

Промежуточный уровень представлен классом, реализующим все методы интерфейса. Этот класс не содержит какой-либо логики, связанной с пове-

дением объекта, и является посредником между объектно-ориентированной системой и автоматом, реализующим логику объекта.

При вызове метода интерфейса выполняется следующая последовательность действий:

создается объект-событие, содержащий информацию о вызванном методе и переданных параметрах;

объект-событие обрабатывается базовым конечным автоматом;

клиенту передается возвращаемое значение и выходные параметры вызванного метода.

Промежуточный класс `FileAccessBase` должен быть унаследован от интерфейса и служебного класса `StateBased`, как показано на рис. 1.

Класс `StateBased` обеспечивает базовую функциональность, необходимую для применения данного подхода.

Рассмотрим пример реализации промежуточного уровня. Во-первых, для каждого метода интерфейса должен быть определен специализированный тип объекта-события. Эти типы должны быть унаследованы от класса `StateBased::Event`. Других требований к объектам-событиям не предъявляется. Эти типы могут быть вложены в класс `FileAccessBase`.

```
   class FileAccessBase : public virtual
IFileAccess,
   protected virtual StateBased {
   protected:
     struct EOpen : public Event {
       EOpen( string const& _mode ) : mode( _mode )
{}
   string const& GetMode() const {
         return mode;
       }
     private:
       EOpen& operator=( EOpen const& );
       string const mode;
     };

     struct EClose : public Event {};
     struct BoolEvent : public Event {
       BoolEvent() : result( false ) {}
       bool GetResult() {
         return result;
       }
       bool SetResult( bool _result ) {
         result = _result;
         return true;
       }
     private:
       bool result;
     };
     struct ECanRead : public BoolEvent {};
     struct ECanWrite : public BoolEvent {};
   /*...*/
   };
```

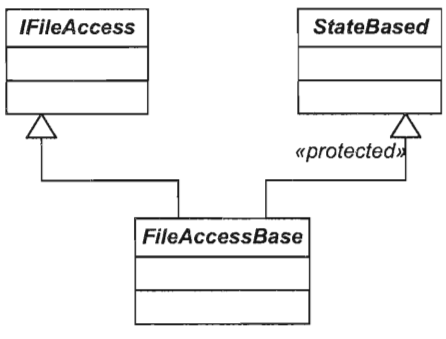


Рис. 1. *Схема наследования промежуточного класса (уровня)*

Во-вторых, должны быть реализованы все методы интерфейса. Их реализации имеют одинаковую структуру – на стеке создается соответствующий объект-событие и передается методу `StateBased::Execute()`.

```
   class FileAccessBase : public virtual
IFileAccess,
   protected virtual StateBased {
     /*...*/
   public:
     virtual void Open( string const& _mode ) {
       EOpen event( _mode );
       Execute( event );
     }
     virtual void Close() {
       EClose event;
       Execute( event );
     }
     virtual bool CanRead() const {
       ECanRead event;
       Execute( event );
       return event.GetResult();
     }
     virtual bool CanWrite() const {
       ECanWrite event;
       Execute( event );
       return event.GetResult();
     }
   };
```

Контекст и логика с выделенным состоянием

В данной работе используется такое средство описания конечных автоматов, как граф переходов. Разновидности графов переходов применяются в `Statecharts` [7], `SyncCharts` [8], *SWITCH*-технологии [9] и т. д.

Каждое состояние автомата на графе переходов изображается в виде прямоугольника с закругленными углами, каждый переход – в виде дуги. Дуги помечаются в формате <условие>/[<действие>]. Квадратные скобки означают, что действие необязательно.

Пример графа переходов показан на рис. 2. Этот автомат позволяет иметь два режима доступа к файлу – `reading` и `writing`.

В качестве основных средств реализации графов переходов используются:

таблицы функций переходов и выходов;

оператор `switch` с вложенными условными операторами if (этот метод применяется в *SWITCH*-технологии).

В настоящей работе предлагается новый метод, основанный на встроенном механизме виртуальных функций. Пространство состояний автомата отображается на множество виртуальных методов. Каждый такой метод называется *методом состояния*. В дальнейшем, если не указано иное, то вместо введенного термина используется слово метод.

Введенные методы имеют одинаковую сигнатуру. Каждое состояние автомата соответствует одному и только одному методу. Экземпляр объекта с выделенным состоянием хранит текущее состояние в виде ссылки на метод, соответствующий этому состоянию. Существует специальный метод main, соответствующий начальному состоянию.

Каждый метод возвращает значение true, если переход был осуществлен, и значение false – в противном случае. Если в процессе обработки события не было совершено ни одного перехода, то класс `StateBased` обеспечивает генерацию исключения `StateBase::UnexpectedOperation`.

Учитывая данные предположения, автомат, изображенный на рис. 2, может быть реализован следующим образом:

```
class FileAccess : public FileAccessBase {
protected:
  virtual bool main( Event& _event ) {
    if ( EOpen* e = EventCast< EOpen >( _event )
) {
      if ( e->GetMode() == "r" )
        return NextState( *this,
&FileAccess::reading );
      else if ( e->GetMode() == "w" )
        return NextState( *this,
&FileAccess::writing );
    }
    return false;
  }
  virtual bool reading( Event& _event ) {
    if ( ECanRead* e = EventCast< ECanRead >(
_event ) )
      return e->SetResult( true );
    else if ( ECanWrite* e = EventCast<
ECanWrite >( _event ) )
      return e->SetResult( false );
    else if ( EClose* e = EventCast< EClose >(
_event ) )
      return NextState( *this, &FileAccess::main );
    return false;
  }
  virtual bool writing( Event& _event ) {
    if ( ECanRead* e = EventCast< ECanRead >(
_event ) )
      return e->SetResult( false );
    else if ( ECanWrite* e = EventCast<
ECanWrite >( _event ) )
      return e->SetResult( true );
    else if ( EClose* e = EventCast< EClose >(
_event ) )
      return NextState( *this, &FileAccess::main );
    return false;
  }
};
```

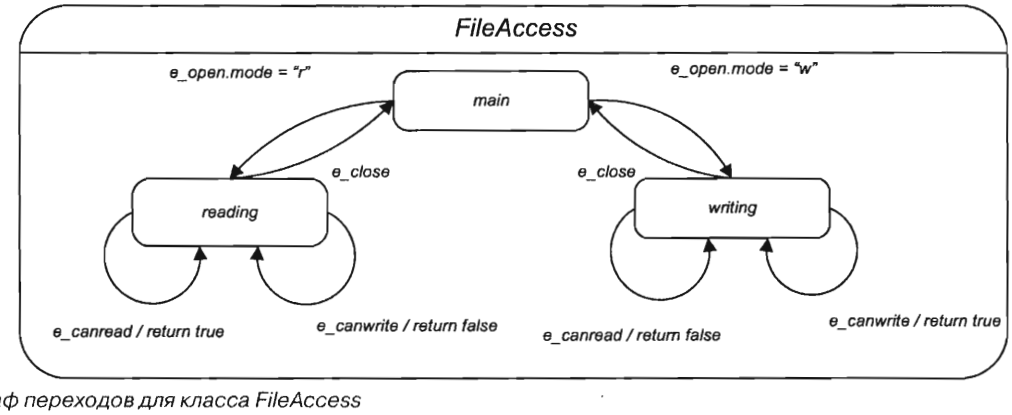
Расширение логики посредством наследования

Основное достоинство предлагаемого метода – возможность наследования. Поведение объекта в каком-то конкретном состоянии может быть изменено или расширено. В автомат могут быть добавлены новые состояния.

Например, в рассмотренный выше класс `FileAccess` может быть добавлен смешанный режим запись/чтение. Граф переходов автомата `RWFileAccess`, расширенного введением состояния `readwriting`, показан на рис. 3.

Звездочка над названием состояния main означает, что данное состояние расширено – все переходы из состояния main в другие состояния сохранены в новом автомате. Приведем реализацию расширенной версии автомата.

```
class RWFileAccess : public FileAccess {
protected:
  virtual bool main( Event& _event ) {
    if ( FileAccess::main( _event ) )
      return true;
    if ( EOpen* e = EventCast< EOpen >( _event ) )
      if ( e->GetMode() == "rw" )
        return NextState( *this,
&RWFileAccess::readwriting );
```



■ ***Рис. 2.*** *Граф переходов для класса FileAccess*

```
      return false;
      }
      virtual bool readwriting( Event& _event ) {
        if ( ECanRead* e = EventCast< ECanRead >(
_event ) )
          return e->SetResult( true );
        else if ( ECanWrite* e = EventCast<
ECanWrite >( _event ) )
          return e->SetResult( true );
        else if ( EClose* e = EventCast< EClose >(
_event ) )
          return NextState( *this,
&RWFileAccess::main );
        return false;
      }
    };
```

При таком подходе возможно применение множественного наследования, что позволяет объединять логику нескольких автоматов. Для этого вводятся дополнительные требования:

интерфейс объекта и служебный класс `StateBased` должны наследоваться виртуально;

в классе автомата, являющегося потомком нескольких автоматов, должен быть переопределен метод `main`.

Последнее необходимо для устранения неоднозначности вызова метода main результирующего автомата.

Например, пусть задан класс `AppendFileAccess`, граф переходов которого изображен на рис. 4.

Реализуем этот класс:

```
  class AppendFileAccess : public FileAccessBase {
  protected:
    virtual bool main( Event& _event ) {
      if ( EOpen* e = EventCast< EOpen >( _event )
)
        if ( e->GetMode() == "a" )
          return NextState( *this,
&AppendFileAccess::appending );
      return false;
      }
      virtual bool appending( Event& _event ) {
        if ( ECanRead* e = EventCast< ECanRead >(
_event ) )
          return e->SetResult( false );
        else if ( ECanWrite* e = EventCast<
ECanWrite >( _event ) )
          return e->SetResult( true );
        else if ( EClose* e = EventCast< EClose >(
_event ) )
          return NextState( *this,
&AppendFileAccess::main );
        return false;
      }
  };
```

Пусть требуется объединить логику классов `AppendFileAccess` и `RWFileAccess` в одном объекте, который может работать в четырех режимах – `reading`, `writing`, `readwriting` и `appending`. Граф переходов соответствующего автомата изображен на рис. 5.

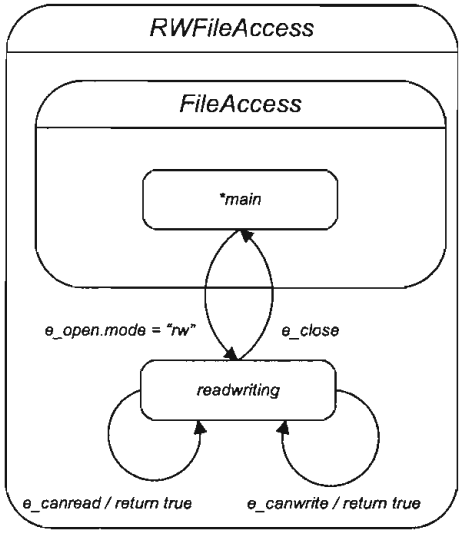
Данный автомат может быть реализован следующим образом:

```
  class RWAFileAccess : public RWFileAccess,
public AppendFileAccess {
  protected:
    virtual bool main( Event& _event ) {
      if ( RWFileAccess::main( _event ) )
        return true;
      if ( AppendFileAccess::main( _event ) )
        return true;
      return false;
    }
  };
```

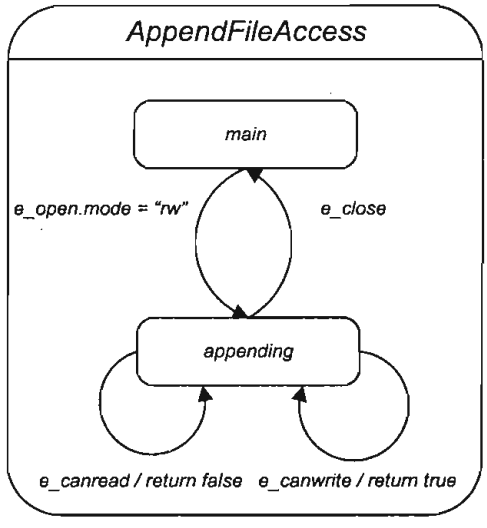
Детали реализации

Все объекты с выделенным состоянием должны наследоваться от служебного класса `StateBased`. Этот класс предоставляет:

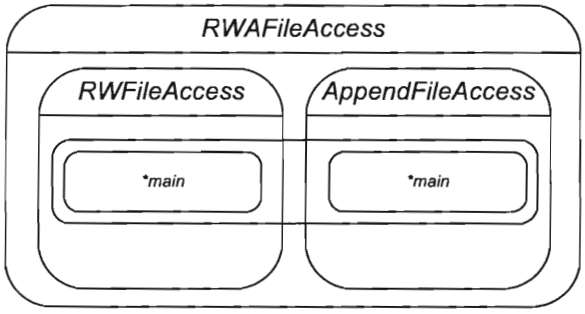
базовый для объектов-событий класс `StateBased::Event`;



■ ***Рис. 3.*** *Граф переходов для класса RWFileAccess*



■ ***Рис. 4.*** *Граф переходов для класса AppendFileAccess*



■ ***Рис. 5.*** *Граф переходов для класса RWAFileAccess*

константную и неконстантную версии метода `Execute()`, который используется на промежуточном уровне для обращения к контексту;

метод `NextState()`, применяемый для изменения состояния объекта;

шаблонный метод `EventCast()`, используемый для выяснения конкретного типа объекта-события.

Реализация константных методов интерфейса. К сожалению, механизм автоматического контроля константности объекта во время выполнения метода состояния не доступен. Это объясняется тем, что в предлагаемом подходе в процессе конструирования объекта с выделенным состоянием запоминается неконстантная ссылка на текущий экземпляр. В процессе вызова константного метода интерфейса используется неконстантная ссылка, что не позволяет применять встроенный механизм контроля константности объекта.

Для устранения этого недостатка класс StateBased автоматически контролирует константность состояния объекта во время выполнения. Если вызван константный метод интерфейса и он пытается вызвать метод `NextState()`, то классом `StateBased` гарантируется генерация исключения `StateBased::ReadOnlyViolation`.

Константность вспомогательных данных должна контролироваться вручную. Если метод состояния пытается получить неконстантный доступ к данным и метод `StateBased::IsImmutable()` возвращает `true`, то программистом должно быть сгенерировано указанное выше исключение.

Заключение

В данной работе описан основанный на виртуальных методах способ объектно-ориентированной реализации объектов, в которых текущее состояние выделено. Его основным достоинством является возможность расширения логики автомата с помощью наследования.

Предлагаемый подход применим при выполнении следующих условий:

существует множество объектов с одинаковым интерфейсом, но с разным поведением;

логика объектов поддается структуризации в виде иерархии;

в объектах мало вспомогательных (кроме состояния) данных.

Предложенный подход может быть усовершенствован, в частности, за счет упрощения преобразования методов в события.

Изложенный подход впервые был опубликован на английском языке на сайте http://www. codeproject. com в июле 2004 года [10].

Автор выражает благодарность профессору А. А. Шалыто за помощь при написании статьи.

Литература

1. **Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.** Приемы объектно-ориентированного проектирования. Паттерны проектирования. – СПб.: Питер, 2001. – 368 с.
2. **Faison T.** Object-oriented state machines // Software Development Magazine. – 1993. – September (http://www.faisoncomputing.com/publications/articles/OOStateMachines.pdf)
3. **Henney K.** State Government // C/C++ Users Journal. – 2002. – June. (http://www.cuj.com/documents/s=7982/cujcexp2006henney/).
4. **Adamczyk P.** The anthology of the finite state machine design patterns // The 10th Conference on Pattern Languages of Programs. – 2003. (http://jerry.cs.uiuc.edu/~plop/plop2003/Papers/Adamczyk-State-Machine.pdf).
5. **Шопырин Д. Г., Шалыто А. А.** Объектно-ориентированный подход к автоматному программированию // Информационно-управляющие системы. – 2003. – № 5. – С. 29–39
6. **Шамгунов Н. Н., Корнеев Г. А., Шалыто А. А.** State Machine – новый паттерн объектно-ориентированного проектирования // Информационно-управляющие системы. – 2004. – № 5. – С. 13–25.
7. **Harel D.** Statecharts: A visual formalism for complex systems // Sci. Comput. Program. – 1987. – N 8. – P. 231–274.
8. **André C.** Representation and Analysis of Reactive Behaviors: A Synchronous Approach // CESA'96, Lille, France, IEEE-SMC, 1996. – P. 19–29. http://wwwi3s.unice.fr/~andre/CA%20Publis/Cesa96/SyncCharts_Cesa96.pdf
9. **Шалыто А. А.** SWITCH-технология. Алгоритмизация и программирование задач логического управления. – СПб.: Наука, 1998. – 628 с.
10. **Shopyrin D.** Object-oriented implementation of state-based logic (http://www.codeproject.com/cpp/statebased.asp), 2004.