Статья опубликована в журнале «Информационные технологии моделирования и управления», 2005, № 1 (19), с. 87 – 96.

**Д.Г. Шопырин**

# МЕТОД ПРОЕКТИРОВАНИЯ И РЕАЛИЗАЦИИ КОНЕЧНЫХ АВТОМАТОВ НА ОСНОВЕ ВИРТУАЛЬНЫХ ВЛОЖЕННЫХ КЛАССОВ

Санкт-Петербургский государственный университет информационных технологий, механики и оптики

*В работе рассматривается проблема совместного использования объектно-ориентированной и автоматно-ориентированной технологий программирования. Предлагается расширение известного паттерна проектирования* `State`*, которое позволяет расширять логику конечного автомата, используя механизм наследования.*

## 1. Введение

Объектно-ориентированная система может содержать объекты с *выделенными состояниями* [1] или объекты с *выделенными режимами работы* [2]. Термин *состояние* в данном контексте неоднозначен, так как объект может содержать данные, которые формально являются его «состоянием», но при этом незначительно влияют на его поведение. Поэтому далее будет использоваться термин *режим работы* (*mode*). Этот термин заимствован из англоязычной литературы, в то время как в работе [3] состояния объекта разделены, как и в машине Тьюринга, на две группы – управляющие и вычислительные, причем режимам работы соответствует управляющие состояния.

Множество возможных режимов вместе с правилами перехода между ними назовем *макрологикой*. *Макрологика* описывает поведение объекта. Одним из способов формализации макрологики являются конечные автоматы [4].

Совместное использование объектно-ориентированной и автоматно-ориентированной технологий программирования связано с решением следующих задач:

- интеграция конечного автомата в объектно-ориентированную систему;
- реализация конечного автомата в объектно-ориентированном стиле.

Подробный обзор существующих подходов к решению указанных задач приведен в работах [5; 6]. В настоящей работе предлагается новый подход, являющийся комбинацией паттерна проектирования `State` [7] и подхода, основанного на виртуальных методах [8].

Достоинством паттерна `State` является декомпозиция режимов работы объекта в отдельные классы, а недостатком – отсутствие возможности расширять макрологику за счет наследования.

Преимуществом подхода, основанного на виртуальных методах, является возможность расширять макрологику, используя механизм наследования, а его недостатками:

- трудоемкость преобразования методов в события;
- практическая невозможность иметь *внутренние (вычислительные) состояния* в конкретных режимах работы.

Предлагаемый подход, основанный на виртуальных вложенных классах, сохраняет возможность расширения логики с помощью наследования и избавляет от указанных недостатков.

Механизм виртуальных вложенных классов позволяет перегружать вложенный класс базового класса в его потомках. Например, если класс `Vector` содержит вложенный виртуальный класс `Item`, то класс `IntVector` может перегрузить его вложенным классом `IntItem`. Подобная возможность предоставляется, например, в языке программирования Python [9]. Многие статически типизируемые объектно-ориентированные языки программирования не поддерживают виртуальные вложенные классы, хотя в большинстве из них существует возможность эмулировать подобную функциональность. Ниже будет использоваться язык программирования C++, который не является единственным подходящим языком программирования.

## 2. Пример

Рассмотрим в качестве примера семейство объектов доступа к файлу. Пусть существуют следующие разновидности объектов доступа:

- доступ на чтение;
- доступ на запись;
- доступ на чтение, запись и чтение/запись.

Клиент должен быть изолирован от конкретного типа объекта, и взаимодействовать с ним через обобщенный интерфейс, описанный ниже.

```
class IFile
{
public:
  virtual void Open(string fname, string mode) = 0;
  virtual void Close() = 0;
  virtual bool IsOpened() const = 0;

  virtual void Write(int value) = 0;
  virtual int Read() = 0;
```

```
  class InvalidOperation : public exception {};
};
```

Рассматриваемые объекты имеют автоматную природу (с режимами работы «закрыт», «открыт на чтение» и т.д.). Они образуют иерархию (рис. 1).

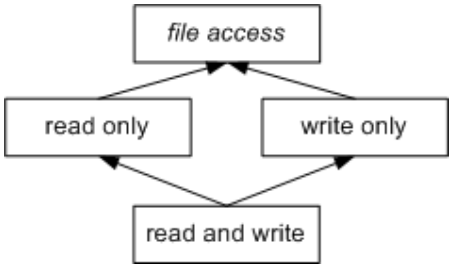


**Рис. 1 Иерархия объектов доступа к файлу**

Приведем пример использования описанных объектов доступа к файлу. Функция `ReadingClient()` ожидает в качестве параметра ссылку на интерфейс `IFile`. Эта функция открывает файл *на чтение*, читает целое число и закрывает файл. Функция main() создает два различных объекта доступа к файлу (`ReadFile` и `ReadWriteFile`), с которыми клиент (`ReadingClient`) работает универсально (полиморфно). В целях повышения читабельности, в приведенном коде не обеспечена, например, безопасная обработка исключений.

```
void ReadingClient(IFile& file) {
  file.Open("data.txt", "r");
  int value = file.Read();
  file.Close();
}

int main() {
  ReadFile rfile;
  ReadWriteFile rwfile;

  ReadingClient(rfile);
  ReadingClient(rwfile);

  return 0;
}
```

# 3. Описание подхода

## *3.1. Термины и определения*

Для упрощения изложения введем следующие термины.

**Состояние** (**State**) – совокупность значений всех атрибутов объекта.

**Режим Работы** (**Mode**) – формально выделенное множество состояний, характеризуемое специфическим поведением объекта.

**Интерфейс** (**Interface**) – общий интерфейс семейства автоматных объектов.

**Класс Режима** (**Mode Class**) – класс, реализующий **Интерфейс** и представляющий поведение объекта в конкретном **Режиме Работы**.

**Прототип Режима** (**Mode Prototype**) – класс, обобщающий поведение объекта, неизменное в каком-то подмножестве **Режимов Работы**.

**Фабрика Режима** (**Mode Factory**) – метод **Контекста**, используемый для создания экземпляра **Класса Режима**. Пространство возможных режимов работы объекта отображается на множество фабрик режимов. Принято, что **Режимы Работы** имеют такие же имена, как и **Фабрики Режимов**.

**Посредник** (**Proxy**) – класс, реализующий **Интерфейс** и переадресовывающий все вызовы текущему экземпляру **Класса Режима**.

**Контекст** (**Context**) – класс, являющийся потомком (возможно непрямым) **Посредника**, предоставляющий набор **Фабрик Режимов**.

## *3.2. Распределение ролей в примере*

На рис. 2 показана иерархия внешних классов, появляющаяся в соответствии с ролями, определенными в разд. 3.1.

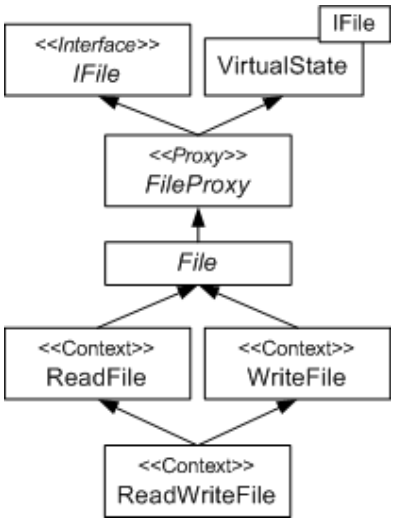


**Рис. 2 Иерархия классов доступа к файлу**

Класс `IFile` играет роль **Интерфейса**, класс `FileProxy` – **Посредника**. На этом рисунке изображены три различных **Контекста**: `ReadFile`, `WriteFile` и `ReadWriteFile`, причем последний является потомком двух предыдущих. Промежуточный класс `File` используется для хранения **Прототипов Режима**.

Обратим внимание на Каркасный класс `VirtualState.` Он является основой предлагаемого подхода и все **Посредники** должны от него наследоваться.

### *3.3. Диаграмма макрологики*

Рассмотрим диаграмму макрологики, показанную на рис. 3.

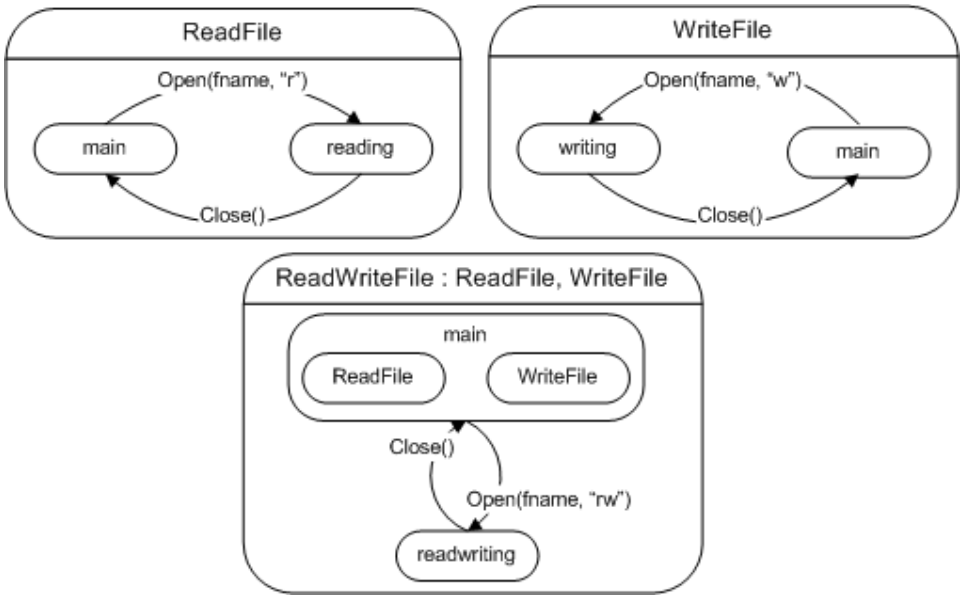


**Рис. 3 Диаграмма макрологики**

Данная диаграмма является расширением графов переходов [10, 11]. Автоматные объекты изображены в виде *рамок*, режимы работы объекта – как закругленные прямоугольники внутри соответствующей рамки, а переходы между режимами – в виде направленных дуг.

Дуги помечены условиями в терминах **Интерфейса**. Например, для обозначения перехода при вызове метода `Open()` с параметром `mode` равным `“r”` – дуга помечена строкой `Open(fname, “r”)`.

Режим работы `ReadWriteFile::main` расширяет логику режимов `ReadFile::main` и `WriteFile::main`. Переходы с метками `Open(fname, “r”)` и `Open(fname, “w”)`, существующие в базовых режимах, в этом режиме представлены неявно.

### 3.4. Отношения и взаимодействия

**Контекст** – класс, предоставляющий множество **Фабрик Режимов**. Пространство **Режимов Работы** автоматного объекта отображается на пространство **Фабрик Режимов**.

**Фабрика Режима** является *виртуальным конструктором* соответствующего **Класса Режима** и может иметь произвольный набор параметров. Виртуальная **Фабрика Режима** может быть перегружена в потомках, что позволяет изменять поведение объекта в соответствующем состоянии. Предопределенная абстрактная **Фабрика Режима** `main()` используется для создания начального режима работы. Каждый конкретный **Контекст** должен перегрузить эту фабрику.

**Посредник** хранит экземпляр текущего **Класса Режима** и переадресует к нему все вызовы методов **Интерфейса**. Экземпляр **Класса Режима** отвечает за изменение **Режима Работы** объекта, когда это необходимо. Другими словами, макрологика распределена между **Режимами Работы**.

Для переключения между режимами используются указатели на методы. Текущий экземпляр **Класса Режима** передает указатель на **Фабрику Режима** и все необходимые этой фабрике параметры методу `NextState()` каркаса `VirtualState`. Таким образом, в соответствии с механизмом вызова виртуальных методов по указателю [12], если данная **Фабрика Режима** была перегружена, то будет вызвана наиболее подходящая в текущем контексте версия.

Поясним изложенное на примере. Пусть контекст `ACtx` имеет две фабрики режимов: `foo()` и `goo()`. Причем фабрика `goo()` – виртуальная. Контекст `ACtx` содержит два класса режимов – `FooMode` и `GooMode`. Класс `FooMode` переключается в режим `goo()` используя указатель на метод `ACtx::goo()`. Контекст `BCtx` является потомком контекста `ACtx` и перегружает фабрику режима `goo()`. Фабрика `BCtx::goo()` создает экземпляр режима `ExtendedGooMode`. Таким образом, если будет создан экземпляр класса `BCtx` и из класса `ACtx::FooMode` будет запрошено переключение в режим `goo`, то будет вызван метод `BCtx::goo()` и создан экземпляр режима `ExtendedGooMode`.

### 3.5. Каркас `VirtualState`

Рассмотрим каркас, представленный шаблонным классом `VirtualState` (рис. 2), который можно рассматривать в качестве библиотечного. Единственным параметром этого класса является **Интерфейс**.

Класс `VirtualState` предоставляет два основных семейства методов: `CurrentMode()` и `NextMode()`. Константная и неконстантная версии метода `CurrentMode()`возвращают текущий экземпляр **Класса Режима**.

Основная функциональность класса `VirtualState` сосредоточена в семействе шаблонных методов `NextMode()`. Каждый из этих методов получает указатель на **Фабрику Режима** и ее параметры. Метод `NextMode()` создает экземпляр соответствующего класса `ModeCtorX`, где `X` – число параметров вызываемой фабрики режима. Все классы `ModeCtorX` реализуют интерфейс `IModeCtor`. Этот интерфейс содержит единственный метод `Create()`, возвращающий экземпляр **Класса Режима**.

Созданный экземпляр класса `ModeCtorX` запоминается классом `VirtualState` в виде указателя на `IModeCtor`. Когда текущая транзакция обращения к объекту завершается, класс `VirtualState` проверяет присутствие экземпляра `IModeCtor`. Если указатель ненулевой, то с помощью экземпляра `IModeCtor` создается новый экземпляр **Класса Режима**. Уничтожение текущего экземпляра **Класса Режима** является отложенным процессом, поэтому экземпляр не будет разрушен, пока не завершится текущий вызов.

## 4. Реализация

Далее на примере, описанном в разд. 2, приведено краткое описание подробностей использования предлагаемого подхода.

### *4.1. Реализация посредника*

Реализация класса `FileProxy` (рис. 2) состоит только в переадресации всех методов **Интерфейса** текущему экземпляру **Класса Режима**. Класс `FileProxy` является потомком классов `IFile` и `VirtualState<IFile>`.

```
class FileProxy
  : public IFile
  , public VirtualState<IFile>
{
public:
  virtual bool Open(string fname, string mode) {
    return CurrentMode()->Open(fname, mode);
  }

  virtual void Close() {
```

```
    CurrentMode()->Close();
  }

  /*...и так далее...*/
};
```

### 4.2. Реализация прототипов режимов

Рассмотрим класс `File` (рис. 2), предназначенный для хранения **Прототипов**. В рассматриваемом примере использованы два **Прототипа Режимов**: `ClosedMode` и `OpenedMode`.

Эти прототипы обобщают поведение объекта, когда файл открыт или закрыт соответственно. В частности, они перегружают метод `IsOpened()`, возвращающий значение `false` в классе `CloseMode` и значение `true` в классе `OpenedMode`. Реализация класса `ClosedMode` приведена ниже.

```
class File : public FileProxy {
protected:
  class ClosedMode : public Mode
  {
  public:
    /*...конструктор и деструктор...*/

    virtual void Close() {
      //because already closed file cannot be closed again
      throw IFile::InvalidOperation();
    }

    virtual bool IsOpened() const {
      return false;
    }

    /*...и так далее...*/
  };

  /*...реализация класса OpenedMode...*/
};
```

Класс `ClosedMode` является потомком класса `Mode`, вложенного в каркасный класс `VirtualState`.

### *4.3. Реализация контекста*

**Контекст** (рис. 2) предназначен для предоставления **Фабрик Режимов**. **Класс Режима** объявляется внутри **Контекста**. Существуют следующие альтернативы:

- объявить **Класс Режима** в виде вложенного класса **Контекста**;
- объявить **Класс Режима** в виде локального класса **Фабрики Режима**.

Класс `ReadFile`, код которого приведен ниже, иллюстрирует эти альтернативы. Класс `MainMode` объявлен как вложенный класса контекста `ReadFile`, а класс `ReadingMode` – как локальный класс фабрики режима `reading()`.

```
class ReadFile : public virtual File {
protected:

  /* вложенный класс режима */
  class MainMode : public virtual ClosedMode
  {
  public:
    /*...конструктор и деструктор...*/

    virtual bool Open(string fname, string mode);
  };

  virtual IFile* main();
  virtual IFile* reading(string fname);
};



IFile* ReadFile::main() {
  return new MainMode( *this );
}

IFile* ReadFile::reading(string fname) {
  /* локальный класс режима */
  class ReadingMode : public OpenedMode
  {
  public:
    ReadingMode(string fname, TVirtualState& _outer)
      : OpenedMode( _outer )
      , file( fname.c_str() ) {}
```

```
    virtual void Write(int value) {
      throw IFile::InvalidOperation();
    }

    /*...и так далее...*/
  private:
    ifstream file;
  };

  return new ReadingMode(fname, *this);
}
```

Достоинство *вложенного* **Класса Режима** – возможность наследоваться от него в потомках контекста `ReadFile`. *Локальные* же **Классы Режимов** не позволяют наследоваться, но лучше инкапсулированы. Пример наследования от **Класса Режима** проиллюстрирован в контексте `ReadWriteFile`. **Класс Режима** `ReadWriteFile::MainMode` является потомком классов `ReadFile::MainMode` и `WriteFile::MainMode`. Поведение класса `ReadWriteFile::MainMode` является расширением поведения его предков.

```
class ReadWriteFile
  : public virtual ReadFile
  , public virtual WriteFile
{
protected:
  class MainMode
    : public virtual ReadFile::MainMode
    , public virtual WriteFile::MainMode
  {
  public:
    /*...конструктор и деструктор...*/

    virtual bool Open(string fname, string mode);
  };

  virtual IFile* main();
  virtual IFile* readwriting(string fname);
};
```

```
bool ReadWriteFile::MainMode::Open(string fname, string mode) {
  if ( mode == "rw" ) {
    NextMode( &ReadWriteFile::readwriting, fname );
    return true;
  } if ( ReadFile::MainMode::Open(fname, mode) )
    return true;
  else
    return WriteFile::MainMode::Open(fname, mode);
}
```

Приведенная реализация иллюстрирует возможность наследования и расширения макрологики. Кроме того, отсутствует трудоемкое преобразование методов интерфейса в события, и классы режимов могут иметь вычислительные состояния.

## Заключение

Предложенный метод реализации конечных автоматов на основе вложенных виртуальных классов предоставляет возможность реализовывать объекты с выделенными состояниями в объектно-ориентированном стиле. В предлагаемом подходе отсутствует *ручная* инициализация логики автоматного объекта.

Подход полностью удовлетворяет основным принципам объектно-ориентированного программирования.

**Инкапсуляция.** Факт того, что данный объект является автоматным, скрыт от окружения. Экземпляры классов режима могут иметь вычислительные состояния.

**Полиморфизм.** Если существует несколько автоматных объектов с различным поведением, но с одинаковым интерфейсом, то клиент может взаимодействовать с ними универсальным образом.

**Наследование.** Поведение автоматного объекта может быть расширено с помощью обычного механизма наследования. Кроме того, похожее поведение в разных режимах может быть обобщено в отдельном классе, и быть повторно использовано.

Отметим, что предложенный подход отличается не только от прототипа – классического паттерна `State`, но и от предложенного в работе [11] паттерна `State Machine`. В отличие от последнего, предлагаемый подход позволяет расширять макрологику объекта без дублирования исходного кода.

## Благодарности

Автор выражает благодарность Анатолию Абрамовичу Шалыто и Александру Витальевичу Евдокимову за их поддержку и помощь.

## Литература


1. **Шалыто А.А., Туккель Н.И.** Программирование с явным выделением состояний //Мир ПК. 2001. № 8, с.116–121; № 9, с.132–138, 2001. http://is.ifmo.ru/works/mirk/
2. **Henney K**. Methods for states. A pattern for realizing object lifecycles // In VikingPLoP 2002, Proceedings of the First Nordic Conference on Pattern Languages of Programs, 2002, September.
3. **Шалыто А.А., Туккель Н.И.** От тьюрингова программирования к автоматному // Мир ПК. 2002. № 2, с. 144–149. http://is.ifmo.ru/works/turing/
4. **Automata Studies /** Shannon C.E., McCarthy J. Princeton University Press, 1956.
5. **Adamczyk P.** The Anthology of the Finite State Machine Design Patterns // The 10th Conference on Pattern Languages of Programs, 2003.
6. **Шалыто А.А., Наумов Л.А.** Методы объектно-ориентированной реализации реактивных агентов на основе конечных автоматов // Искусственный интеллект. 2004. № 4, с. 756–762. http://is.ifmo.ru/works/_aut_oop.pdf
7. **Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.** Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб.: Питер, 2001. – 368 с.
8. **Shopyrin D.** Object-oriented implementation of state-based logic. http://www.codeproject.com/cpp/statebased.asp
9. **Лутц М.** Программирование на Python. СПб.: Символ-Плюс, 2002. – 1136 с.
10. **Harel D**. Statecharts: A visual formalism for complex systems // Sci. Comput. Program., 1987, № 8.
11. **Шалыто А. А.** SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998. – 628 с.
12. **Страуструп Б**. Язык программирования C++. 3-е издание. СПб.: Невский диалект, 1999. – 991 с.
13. **Шамгунов Н.Н., Корнеев Г.А., Шалыто А.А.** State Machine - новый паттерн объектно-ориентированного проектирования //Информационно-управляющие системы. 2004. № 5, с. 13–25