Статья опубликована «Информационно-управляющие системы». 2004. № 5. с. 13– 25.

УДК 681.3.06

State Machine — новый паттерн объектно-ориентированного проектирования

Н. Н. Шамгунов, Г. А. Корнеев, А. А. Шалыто

В статье предлагается новый паттерн объектно-ориентированного проектирования, названный State Machine. *Этот паттерн расширяет возможности паттерна* State, *предназначенного для реализации объектов, поведение которых зависит от их состояния. В статье предложено использовать события для уведомления об изменении состояния. Это позволяет проектировать объекты такого рода из независимых друг от друга классов. Приведенный паттерн по сравнению с паттерном* State *лучше приспособлен для повторного использования входящих в него классов.*

This paper presents a new pattern for object-oriented design — State Machine. *This pattern extends capabilities of* State *design pattern. These patterns allow an object to alter its behavior when its internal state changes. Introduced event-driven approach allows to decrease coupling. Thus automaton could be constructed from independent state classes. The classes designed with* State Machine *pattern are more reusable than* State *pattern.*

Введение

Начало формальному изучению систем с конечным числом состояний положено в работе [1].

Различные модели детерминированных конечных автоматов, которые обычно называют «конечными автоматами» или просто «автоматами», были разработаны в середине 50-х годов прошлого века [2-5]. При этом если первые две из этих работ были посвящены синтезу схем (аппаратуры), то остальные носили более абстрактный характер. Считается, что итог этапу становления теории автоматов был подведен выпуском сборника статей [6], который, в частности, содержал работы [3,5]. Интересно отметить, что перевод этого сборника на русский язык появился в СССР в том же году. Недетерминированные автоматы и их эквивалентность детерминированным автоматам были рассмотрены в работе [7].

В дальнейшем применительно к построению аппаратуры теория автоматов «распалась» две взаимосвязанных теории: абстрактную и структурную [8], для первой из которых характерно последовательная обработка информации, а для второй — параллельная.

В программировании автоматы начали применяться после появления работы [9], в которой были введены регулярные выражения и приведено доказательство их эквивалентности конечным автоматам. При этом абстрактная теория автоматов используется в основном для обработки текстов [10-12], а структурная — для программного управления [13].

Еще одна область применения конечных автоматов — объектно-ориентированные программирование, в котором они используются для описания логики объектов, поведение которых зависит от состояния. При этом у объекта выделяются состояния, влияющие на его поведение (так называемые управляющие состояния). Заметим, что в этой области могут применяться конечные автоматы, существенно отличающиеся от абстрактных и структурных автоматов, например, тем, что в них используются понятия объектно-ориентированного программирования. В частности, объекты проектируются в терминах интерфейсов и методов (понятиях, отсутствующих в классических автоматах), а не в терминах входных и выходных воздействий. В данной статье рассматриваются вопросы реализации именно таких объектов.

В объектно-ориентированном программировании под поведением объекта обычно понимается функциональность его методов (действий). Однако во многих приложениях такого определения понятия «поведение объекта» недостаточно, так как необходимо учитывать и его внутренне состояние.

Наиболее известной реализацией объекта, изменяющего поведение в зависимости от состояния, является паттерн ***State*** [14]**.** В указанной работе данный паттерн недостаточно полно специфицирован, поэтому в разных источниках, например в работах [15, 16], он реализуется по-разному (в частности, громоздко и малопонятно). Поэтому многие программисты считают, что этот паттерн не предназначен для реализации автоматов. Другой недостаток паттерна ***State*** состоит в том, что разделение реализации состояний по различным классам приводит и к распределению логики переходов по ним, что усложняет понимание программы. При этом не обеспечивается независимость классов, реализующих состояния, друг от друга. Таким образом, создание иерархии классов состояний и их повторное использование затруднено. Несмотря на эти недостатки, паттерн ***State*** достаточно широко применяется в программировании, например, при реализации синхронизации с источником данных в библиотеке ***Java Data Objects*** (***JDO***) [17].

Заметим, что проблемы с паттерном ***State*** возникают не только при его описании, но даже в определении, в том числе, из-за неправильного перевода. Так в работе [18] приведено следующее определение: «шаблон ***State*** — состояние объекта контролирует его поведение», в то время как более правильным было бы следующее: «шаблон ***State*** — состояния объекта управляют его поведением». Эти определения имеют разный смысл, так как в английском языке слово ***control*** переводится как *управление*, а в русском языке управление — это действие на объект, а контроль — мониторинг и проверка выполняемых действий на соответствие заданному поведению. Не случайно часто говорят «система управления и контроля».

Кроме работы [14], паттерн ***State***, как отмечалось выше, описывается в работах [15, 16]. В них авторы рассматривают реализацию графов переходов автоматов с помощью этого паттерна. В работе [15] код реализации графа переходов с двумя вершинами занимает более десяти страниц текста. Такая же странная ситуация и в работе [16]. Автор этой книги легко справился с описанием и примерами к сорока шести паттернам, и только для паттерна ***State*** ему понадобилась помощь рецензента, который предоставил ему пример. Однако приведенный в этом примере граф переходов с четырьмя вершинами реализован таким образом, будто бы автор недостаточно разобрался в паттерне.

В настоящей работе, предлагается новый паттерн, объединяющий достоинства реализации автоматов в ***SWITCH-технологии*** [13] (централизация логики переходов) и паттерна ***State*** (локализация кода, зависящего от состояния в отдельных классах).

Новый паттерн назван ***State Machine***. Обратим внимание, что в работе [19] уже был предложен паттерн с аналогичным названием, предназначенный для программирования параллельных систем реального времени на языке ***Ada 95***. Тем не менее, авторы выбрали именно это название, как наиболее точно отражающее суть предлагаемого паттерна.

Для обеспечения повторного использования классов состояний, входящих в паттерн, предложено применять механизм *событий*, которые используются состояниями для уведомления автомата о необходимости смены состояния. Это позволяет централизовать логику переходов автомата и устранить «осведомленность» классов состояний друг о друге. При этом реализация логики переходов может осуществляться различными способами, например с использованием таблицы переходов или оператора выбора (оператор `switch` в языке *C++*).

Более двадцати методов реализации объектов, изменяющих поведение в зависимости от состояния, рассмотрены в работе [20]. Паттерн ***State Machine*** мог бы продолжить этот список. Наиболее близким к новому паттерну является объединение паттернов ***State*** и ***Observer***, рассмотренное в работе [21]. Однако, предложенный в этой работе подход достаточно сложен, так как добавляет новый уровень абстракции — класс `ChangeManager`. В паттерне ***State Machine*** используется более простая модель событий, не привлекающая относительно тяжелую реализацию паттерна ***Observer***. В работе [22] предложена реализация паттерна ***State***, позволяющая создавать иерархии классов состояний. Зависимость между классами состояний снижается за счет того, что переход в новое состояние осуществляется по его имени. Такая реализация, тем не менее, не снимает семантической зависимости между классами состояний.

1. Описание паттерна

В названиях разделов будем придерживаться соглашений, введенных в работе [14].

1.1. Назначение

Паттерн ***State Machine*** предназначен для создания объектов, поведение которых варьируется в зависимости от состояния. При этом для клиента создается впечатление, что изменился класс объекта. Таким образом, назначение предлагаемого паттерна фактически совпадает с таковым для паттерна ***State*** [14], однако, как будет показано ниже, область применимости последнего уже.

Отметим, что в этом определении имеются в виду так называемые *управляющие*, а не *вычислительные* состояния [23]. Их различие может быть проиллюстрировано на следующем примере. При создании вычислительной системы для банка имеет смысл выделить режимы нормальной работы и банкротства в разные управляющие состояния, так как в этих режимах поведение банка может существенно отличаться. В то же время, конкретные суммы денег и другие характеристики в банковском балансе будут представлять вычислительное состояние.

1.2. Мотивация

Предположим, что требуется спроектировать класс `Connection`, представляющий сетевое соединение. Простейшее сетевое соединение имеет два управляющих состояния: *соединено* и *разъединено*. Переход между этими состояниями происходит или при возникновении ошибки или посредством вызовов методов *установить соединение* (`connect`) и *разорвать соединение* (`disconnect`). В состоянии *соединено* может производиться получение (метод `receive`) и отправка (метод `send`) данных по соединению. В случае возникновения ошибки при передаче данных генерируется исключительная ситуация (`IOException`) и сетевое соединение разъединяется. В состоянии *разъединено* прием и отправка данных невозможна. При попытке осуществить передачу данных в этом состоянии объект также генерирует исключительную ситуацию.

Таким образом, интерфейс, который необходимо реализовать в классе `Connection`, должен выглядеть следующим образом (здесь и далее примеры приводятся на языке *Java*):

```
package connection;

import java.io.IOException;

public interface IConnection {
  public void connect() throws IOException;
  public void disconnect() throws IOException;
  public int receive() throws IOException;
  public void send(int value) throws IOException;
}
```

Основная идея паттерна ***State Machine*** заключается в разделении классов, реализующих логику переходов (*контекст*), конкретных состояний и модели данных. Для осуществления взаимодействия конкретных состояний с контекстом используются *события*, представляющие собой объекты, передаваемые состояниями контексту. Отличие от паттерна ***State*** состоит в методе определения следующего состояния при осуществлении перехода. Если в паттерне ***State*** следующее состояние указывается текущим состоянием, то в предлагаемом паттерне это выполняется путем уведомления класса контекста о наступлении события. После этого, в зависимости от события и текущего состояния, контекст устанавливает следующее состояние в соответствии с графом переходов.

Преимуществом такого подхода является то, что классам, реализующим состояния, не требуется «знать» друг о друге, так как выбор состояния, в которое производится переход, осуществляется контекстом в зависимости от текущего состояния и события.

Отметим, что графы переходов, применяемые для описания логики переходов при проектировании с использованием паттерна ***State Machine***, отличаются от графов переходов, рассмотренных в работах [11, 13, 24]. Применяемые графы переходов состоят только из состояний и переходов, помеченных событиями. Переход из текущего состояния S в следующее состояние S^* осуществляется по событию E, если на графе переходов существует дуга из S в S^*, помеченная

событием *E*. При этом из одного состояния не могут выходить две дуги, помеченные одним и тем же событием. Отметим, что на графе переходов не отображаются ни методы, входящие в интерфейс реализуемого объекта, ни условия порождения событий.

Граф переходов для описания поведения класса `Connection` приведен на рис. 1. В нем классы, реализующие состояния *соединено* и *разъединено*, называются соответственно `ConnectedState` и `DisconnectedState`, тогда как событие `CONNECT` обозначает установление связи, `DISCONNECT` — обрыв связи, а `ERROR` — ошибку передачи данных.

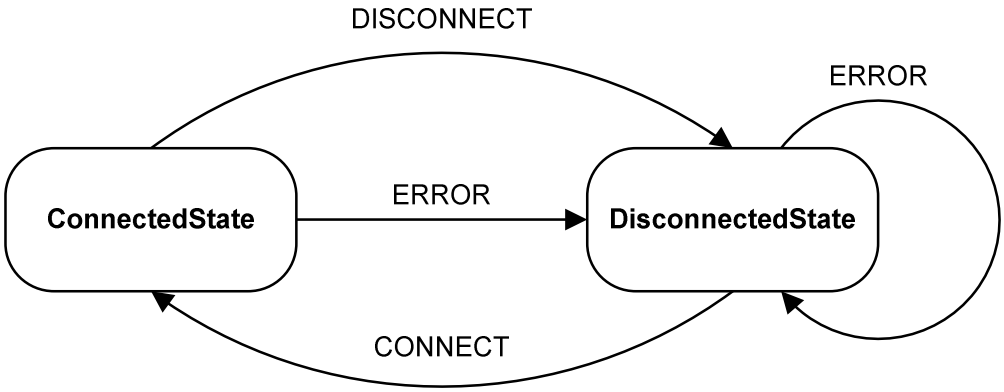

Рис. 1. Граф переходов для класса **`Connection`**

Рассмотрим в качестве примера обработку разрыва соединения при ошибке передачи данных. При реализации с использованием паттерна *State* состояние `ConnectedState` укажет контексту, что следует перейти в состояние `DisconnectedState`. В случае же паттерна *State Machine* контекст будет уведомлен о наступлении события `ERROR`, а тот осуществит переход в состояние `DisconnectedState`. Таким образом, классы `ConnectedState` и `DisconnectedState` не знают о существовании друг друга.

1.3. Применимость

Паттерн *State Machine* может быть использован в следующих случаях.

- *Поведение объекта существенно зависит от управляющего состояния. При этом реализация поведения объекта в каждом состоянии будет сконцентрирована в одном классе*. Этот вариант использования иллюстрируется в данной работе на примере класса, реализующего сетевое соединение.
- *Рефакторинг кода* [25]*, зависящего от состояния объекта*. Примером использования может служить рефакторинг кода, проверяющего права доступа к тем или иным функциям программного обеспечения в зависимости от текущего пользователя или приобретенной лицензии.
- *Повторное использование классов, входящих в паттерн, в том числе посредством создания иерархии классов состояний.* Пример такого использования будет рассмотрен в разд. 2.
- *Эмуляции абстрактных и структурных автоматов.*

Таким образом, область применимости паттерна *State Machine* шире, чем у паттерна *State*.

1.4. Структура

На рис. 2 изображена структура паттерна *State Machine*.

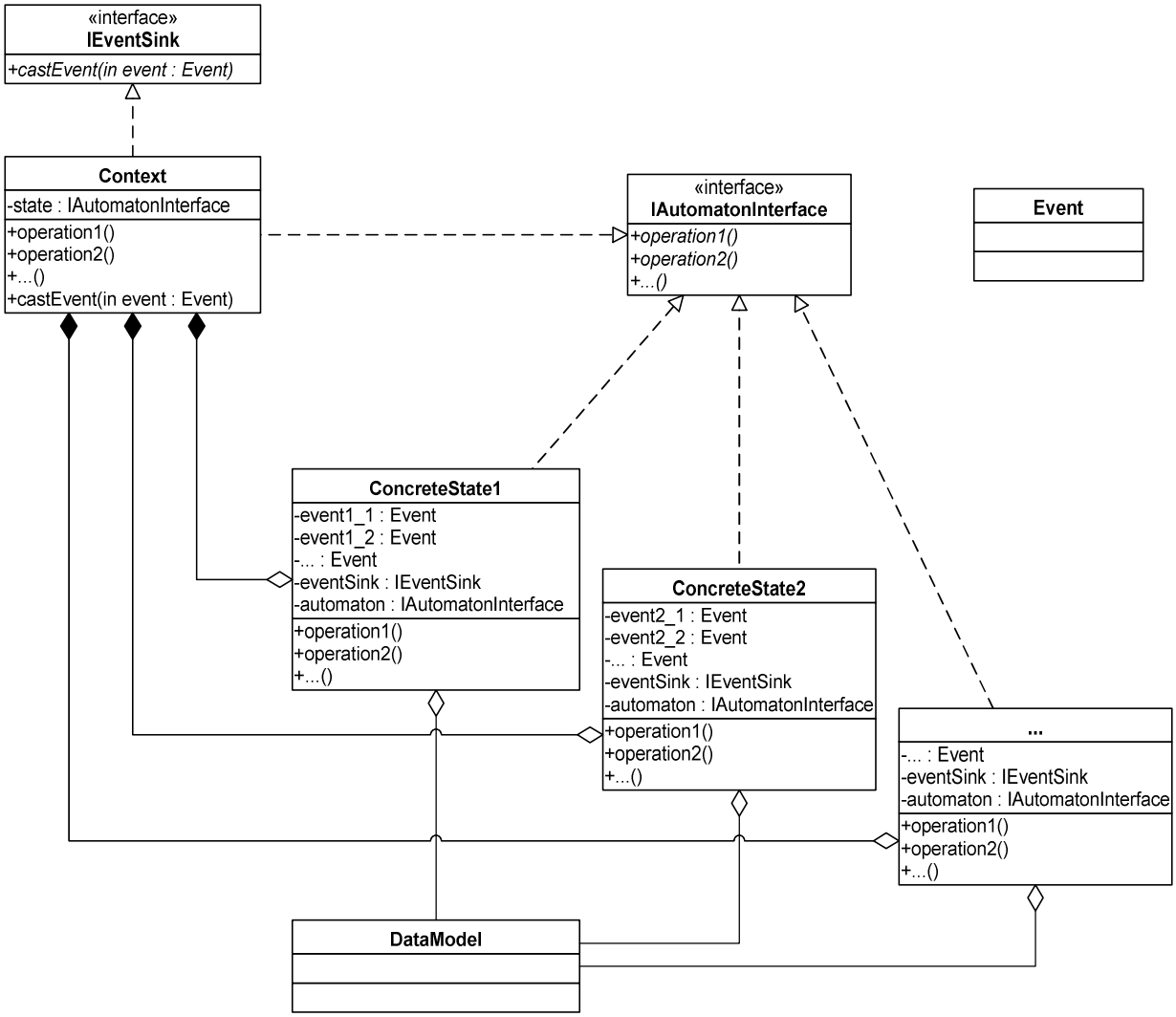

Рис. 2. Структура паттерна *State Machine*

Здесь `IAutomatonInterface` — интерфейс, реализуемого объекта, а `operation1`, `operation2`, … —методы этого интерфейса. Этот интерфейс реализуется основным классом `Context` и классами состояний `ConcreteState1`, `ConcreteState2`, …. Для смены состояния объекта используются события `event1_1`, `event2_1`, …, `event2_1`, `event2_2`, …, являющиеся объектами класса `Event`. Класс `Context` содержит ссылки на все состояния объекта (`ConcreteState1` и `ConcreteState2`), а также на текущее состояние (`state`). В свою очередь, классы состояний содержат ссылку на модель данных (`dataModel`) и интерфейс уведомления о событиях (`eventSink`). Для того чтобы не загромождать рисунок, на нем не отражены связи классов состояний и класса `Event`.

Классы `Context`, `ConcreteState1`, `ConcreteState2`, … реализуют интерфейс `IAutomatonInterface`. Класс `Context` содержит переменные типа `IAutomatonInterface`. Одна из них — текущее состояние автомата, а остальные хранят ссылки на классы состояний автомата. Отметим, что стрелки, соответствующие ссылкам на классы состояний, ведут к интерфейсу, а не к этим классам. Это следствие того, что все взаимодействие между контекстом и классами состояний производится через интерфейс автомата. Связи между контекстом и классами состояний отмечены стрелками с ромбом — используется агрегация.

1.5. Участники

Паттерн *State Machine* состоит из следующих частей.

- *Интерфейс автомата* (`IAutomatonInterface`) — реализуется контекстом и является единственным способом взаимодействия клиента с автоматом. Этот же интерфейс реализуется классами состояний.
- *Контекст* (`Context`) — класс, в котором инкапсулирована логика переходов. Он реализует интерфейс автомата, хранит экземпляры модели данных и текущего состояния.
- *Классы состояний* (`ConcreteState1`, `ConcreteState2`, …) — определяют поведение в конкретном состоянии. Реализуют интерфейс автомата.
- *События* (`event1_1`, `event1_2`, ...) — инициируются состояниями и передаются контексту, который осуществляет переход в соответствии с текущим состоянием и событием.
- *Интерфейс уведомления о событиях* (`IEventSink`) — реализуется контекстом и является единственный способом взаимодействия объектов состояний с контекстом.
- *Модель данных* (`DataModel`) — класс предназначен для хранения и обмена данными между состояниями.

Отметим, что в предлагаемом паттерне, интерфейс автомата реализуется как контекстом, так и классами состояний. Это позволяет добиться проверки целостности еще на этапе компиляции. В паттерне ***State*** такая проверка невозможна из-за различия интерфейсов контекста и классов состояний.

1.6. Отношения

При инициализации контекст создает экземпляр модели данных и использует его при конструировании экземпляров состояний. Кроме модели данных в конструктор класса состояния также передается интерфейс уведомления о событии (ссылка на контекст).

В процессе работы контекст делегирует вызовы методов интерфейса текущему экземпляру состояния. При исполнении делегированного метода объект, реализующий состояние, может сгенерировать событие — уведомить об этом контекст по интерфейсу уведомления о событиях.

Решение о смене состояния принимает контекст на основе события, пришедшего от конкретного объекта состояния.

1.7. Результаты

Сформулируем результаты, получаемые при использовании паттерна ***State Machine***.

- Также как и в паттерне ***State***, поведение, зависящее от состояния, локализовано в отдельных классах состояний.
- В отличие от паттерна ***State*** в предлагаемом паттерне логика переходов (сконцентрированная в классе контекста) отделена от реализации поведения в конкретных состояниях. В свою очередь, классы состояний обязаны только уведомить контекст о наступлении события (например, о разрыве соединения).
- Реализация интерфейса автомата в классе контекста может быть сгенерирована автоматически, а реализация логики переходов — по графу переходов специального вида, предложенного выше.
- Для повышения скорости смены состояний логика переходов может быть реализована специальной таблицей таким образом, что каждый переход осуществляется за одну операцию доступа по индексу.
- Паттерн ***State Machine*** предоставляет «чистый» (без лишних методов) интерфейс для пользователя. Для того чтобы клиенты не имели доступа к интерфейсу `IEventSink`, реализуемого классом контекста, следует использовать или *закрытое* (***private***) наследование (например, в языке *C++*) или определить закрытый конструктор и статический метод, создающий экземпляр контекста, возвращающий интерфейс автомата. Соответствующий фрагмент кода имеет вид:

```
class Automaton implements IAutomatonInterface {
  private Automaton() {}
  public static IAutomatonInterface CreateAutomaton() {
    return new Automaton();
  }
```

```
}
```

- Паттерн *State Machine*, в отличие от паттерна *State*, не содержит дублирующих интерфейсов для контекста и классов состояний — контекст и классы состояний реализуют один и тот же интерфейс.
- Возможно повторное использование классов состояний, в том числе посредством создания их иерархии. Заметим, что в работе [14] сказано, что «поскольку зависящий от состояния код целиком находится в одном из подклассов класса `State`, то добавлять новые состояния и переходы можно просто путем порождения новых подклассов». На самом же деле, добавление нового состояния зачастую влечет за собой модификацию остальных классов состояний, так как иначе переход в данное состояние не может быть осуществлен. Таким образом, расширение автомата, построенного на основе паттерна *State*, является проблематичным. Более того, при реализации наследования в паттерне *State* также затруднено и расширение интерфейса автомата. Скорее всего, именно по этим причинам в работе [14] не описано наследование автоматов.
- В работе [26] рассматривается задача реализации объектов, часть методов которых не зависит от состояния, для решения которой предложен паттерн *Three Level FSM*. Данная задача может быть также решена и при помощи паттерна *State Machine*. Для этого следует разделить интерфейс реализуемого объекта и интерфейс автомата. При этом последний реализуется контекстом в соответствии с описываемым паттерном. Для реализации полного интерфейса объекта создается наследник контекста, в котором определяются методы, не зависящие от состояния (рис. 3).

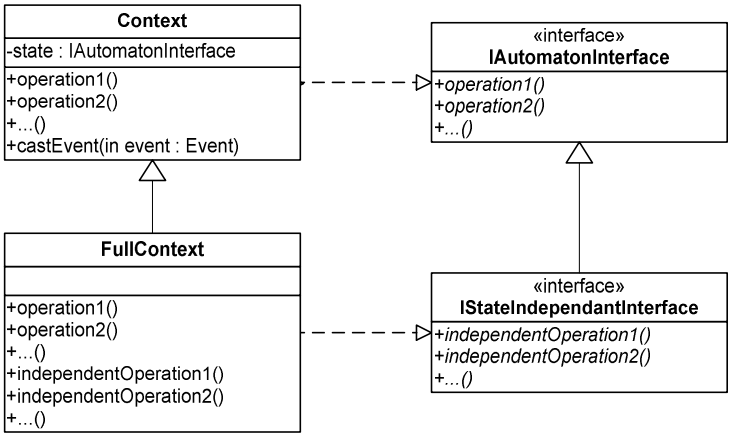

Рис. 3. Реализация методов, не зависящих от состояния

1.8. Реализация

Рассмотрим возможные модификации паттерна *State Machine*.

- *Хранение модели данных*. Контекст можно реализовать таким образом, чтобы он включал в себя модель данных, как предлагается в паттерне *State*. Тогда в конструктор объекта состояния передается только один параметр. Однако при таком подходе зависимость реализации состояний от контекста увеличивается, что усложняет повторное использование классов состояний.
- *Stateless и stateful классы состояний*. В паттерне *State Machine* контекст и классы состояний реализуют интерфейс автомата. Это достигается за счет того, что указанные классы содержат ссылки на модель данных и интерфейс уведомления о событиях. Таким образом, классы состояний являются *stateful* (зависят от предыстории). Такой подход не всегда приемлем, поскольку в некоторых ситуациях критичен расход памяти. Паттерн *State Machine* можно видоизменить так, чтобы состояния были *stateless* (не зависят от

предыстории). При этом для классов состояний придется определить новый интерфейс, отличающийся от интерфейса автомата тем, что в каждый метод добавлены параметры, через которые передаются ссылки на модель данных и интерфейс уведомления о событиях. Это приводит к фактическому дублированию интерфейсов, что затрудняет модификацию кода и его повторное использование, но экономит память.

- *Задание переходов*. Переходы между состояниями задаются в контексте. Это можно сделать, например, при помощи конструкции `switch`, как предлагается в *SWITCH-технологии* [27]. Переходы также можно задать таблицей, отображающей пару *<текущее состояние, событие>* в *<новое состояние>*. Инфраструктуру для реализации табличного подхода можно реализовать в базовом для всех контекстов классе.
- *Протоколирование*. Вынесение логики переходов в контекст позволяет осуществлять протоколирование переходов автоматов в терминах состояний и событий.
- *Создание модели данных*. Экземпляр модели данных может либо создаваться конструктором контекста (как описано выше), либо передаваться в параметрах конструктора контекста.

1.9. Пример кода

Коды всех примеров, описанных в статье, доступны по адресу [28].

В следующем примере приведен код на языке *Java*, реализующий класс *Соединение*, описанный в разд. 1.2. Это упрощенная модель произвольного удаленного соединения, через которое можно передавать данные.

Прежде всего, опишем интерфейсы и базовые классы, которые используются в данном примере. Эти классы вынесены в пакет `ru.ifmo.is.sm` (`sm` — сокращение от ***State Machine***). Диаграмма классов этого пакета приведена на рис. 4.

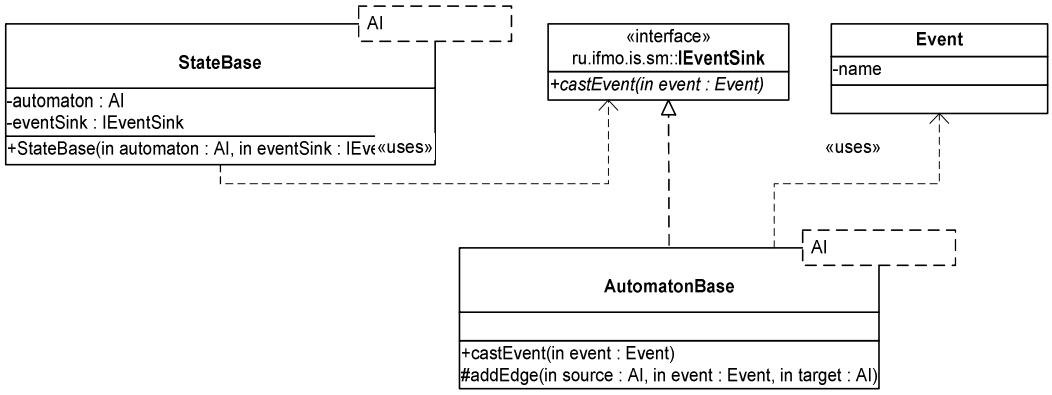

Рис. 4. Диаграмма классов пакета **`ru.ifmo.is.sm`**

Опишем классы и интерфейсы, входящие в него:

- `IEventSink` — интерфейс уведомления о событии:

```
package ru.ifmo.is.sm;

public interface IEventSink {
  public void castEvent(Event event);
}
```

- `Event` — класс события. Используется для уведомления контекста из классов состояний:

```
package ru.ifmo.is.sm;

public final class Event {
  private final String name;

  public Event(String name) {
    if (name == null) throw new NullPointerException();
    this.name = name;
```

```
  }

  public String getName() {
    return name;
  }
}
```

- `StateBase` — базовый класс для состояний. Для проверки типов во время компиляции применяются *параметры типа* (*generic*, *template*), появившееся в *Java 5.0*. В конструкторе запоминается интерфейс для приема событий:

```
package ru.ifmo.is.sm;

public abstract class StateBase<AI> {
  protected final AI automaton;
  protected final IEventSink eventSink;

  public StateBase(AI automaton, IEventSink eventSink) {
    if (automaton == null || eventSink == null) {
      throw new NullPointerException();
    }
    this.automaton = automaton;
    this.eventSink = eventSink;
  }

  protected void castEvent(Event event) {
    eventSink.castEvent(event);
  }
}
```

- `AutomatonBase` — базовый класс для всех автоматов. Он предоставляет возможность наследнику регистрировать переходы, используя метод `addEdge`. Дополнительно класс `AutomatonBase` реализует интерфейс уведомления о событии:

```
package ru.ifmo.is.sm;

import java.util.IdentityHashMap;
import java.util.Map;
import java.util.HashMap;

public abstract class AutomatonBase<AI> implements IEventSink {
  protected AI state;
  private final Map<AI, Map<Event, AI>> edges =
      new HashMap<AI, Map<Event, AI>>();

  protected void addEdge(AI source, Event event, AI target) {
    Map<Event, AI> row = edges.get(source);
    if (null == row) {
      row = new IdentityHashMap<Event, AI>();
      edges.put(source, row);
    }
    row.put(event, target);
  }

  public void castEvent(Event event) {
    try {
      state = edges.get(state).get(event);
    } catch (NullPointerException e) {
      throw new IllegalStateException("Edge is not defined");
    }
  }
}
```

Классы, созданные в соответствии с паттерном *State Machine*, образуют пакет `connection`. Диаграмма классов этого пакета приведена на рис. 5.

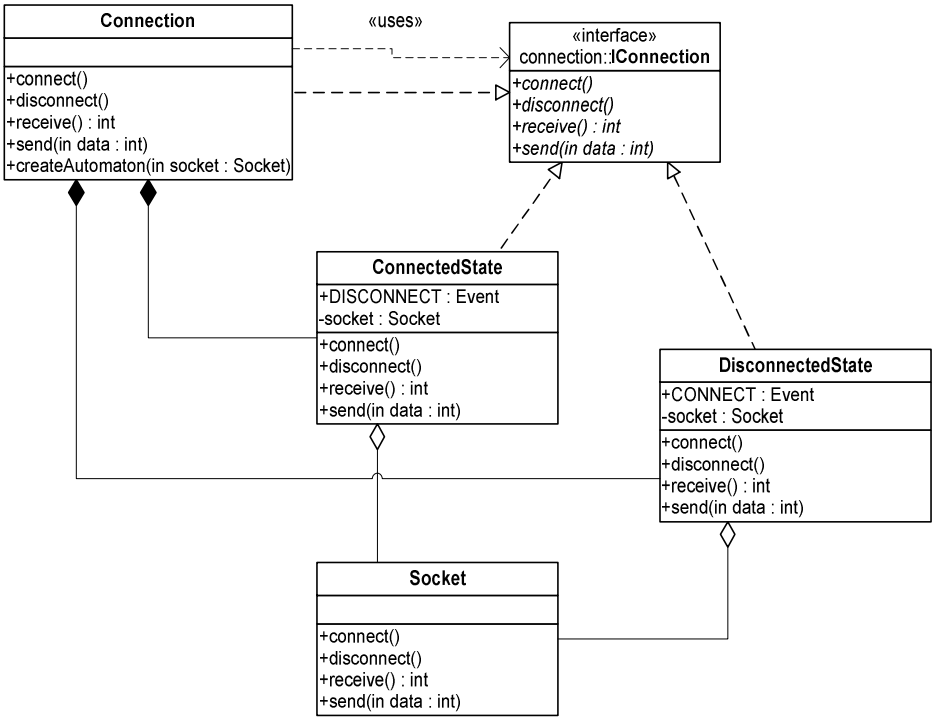

Рис. 5. Диаграмма классов пакета **`connection`**

В качестве модели данных автомата используется класс `Socket`, в рассматриваемом случае реализующий интерфейс `IConnection`.

После определения интерфейса для клиента и модели данных необходимо перейти к определению управляющих состояний автомата. Для данного примера реализуем классы `ConnectedState` и `DisconnectedState`. В состоянии `ConnectedState` могут произойти события `ERROR` и `DISCONECT`, а в состоянии `DisconnectedState` — события `CONNECT` и `ERROR` (рис. 1).

Обратим внимание, что на рис. 1 присутствуют дуги, помеченные одинаковыми событиями. В данном примере объекты событий создаются в классе состояния, из которого исходит переход. Например, событие `ERROR` на переходе из состояния `ConnectedState` в состояние `DisconnectedState` не совпадает с аналогичным событием на петле из состояния `DisconnectedState`. При другой реализации для события `ERROR` мог бы быть создан только один объект.

Приведем код классов состояний.

Класс `ConnectedState`:

```
package connection;

import ru.ifmo.is.sm.*;
import java.io.IOException;

public class ConnectedState <AI extends IConnection>
    extends StateBase<AI> implements IConnection {
  public static final Event DISCONNECT = new Event("DISCONNECT");
  public static final Event ERROR = new Event("ERROR");

  protected final Socket socket;

  public ConnectedState(AI automaton, IEventSink eventSink, Socket
        socket) {
    super(automaton, eventSink);
```

```
    this.socket = socket;
  }

  public void connect() throws IOException {
  }

  public void disconnect() throws IOException {
    try {
      socket.disconnect();
    } finally {
      eventSink.castEvent(DISCONNECT);
    }
  }

  public int receive() throws IOException {
    try {
      return socket.receive();
    } catch (IOException e) {
      eventSink.castEvent(ERROR);
      throw e;
    }
  }

  public void send(int value) throws IOException {
    try {
      socket.send(value);
    } catch (IOException e) {
      eventSink.castEvent(ERROR);
      throw e;
    }
  }
}
```

Обратим внимание, что класс состояния только частично специализирует параметр типа класса `StateBase`. При расширении интерфейса автомата наследники класса состояния еще более специализируют этот тип. Окончательная специализация указывается при создании класса состояния в конструкторе автомата. Это обеспечивает возможность последующего расширения интерфейса автомата и классов состояний.

Класс `DisconnectedState`:

```
package connection;

import java.io.IOException;
import ru.ifmo.is.sm.*;

public class DisconnectedState <AI extends IConnection>
    extends StateBase<AI> implements IConnection {
  public static final Event CONNECT = new Event("CONNECT");
  public static final Event ERROR = new Event("ERROR");

  protected final Socket socket;

  public DisconnectedState(AI automaton, IEventSink eventSink, Socket
      socket) {
    super(automaton, eventSink);
    this.socket = socket;
  }

  public void connect() throws IOException {
    try {
      socket.connect();
```

```
    } catch (IOException e) {
      eventSink.castEvent(ERROR);
      throw e;
    }
    eventSink.castEvent(CONNECT);
  }

  public void disconnect() throws IOException {
  }

  public int receive() throws IOException {
    throw new IOException("Connection is closed (receive)");
  }

  public void send(int value) throws IOException {
    throw new IOException("Connection is closed (send)");
  }
}
```

Остается описать класс `Connection` — контекст. В этом классе реализована логика переходов, в соответствии с графом переходов на рис. 1. Заметим, что последние четыре метода этого класса — делегирование интерфейса текущему состоянию:

```
package connection;

import java.io.IOException;
import ru.ifmo.is.sm.AutomatonBase;

public class Connection extends AutomatonBase<IConnection>
    implements IConnection {
  private Connection() {
    Socket socket = new Socket();

    // Создание объектов состояний
    IConnection connected = new ConnectedState<Connection>(this, this,
        socket);
    IConnection disconnected = new DisconnectedState<Connection>(this,
        this, socket);

    // Логика переходов
    addEdge(connected, ConnectedState.DISCONNECT, disconnected);
    addEdge(connected, ConnectedState.ERROR, disconnected);
    addEdge(disconnected, DisconnectedState.CONNECT, connected);
    addEdge(disconnected, DisconnectedState.ERROR, disconnected);

    // Начальное состояние
    state = disconnected;
  }

  // Создание экземпляра автомата
  public static IConnection createAutomaton() {
    return new Connection();
  }

  // Делегирование методов интерфейса
  public void connect() throws IOException { state.connect(); }
  public void disconnect() throws IOException { state.disconnect(); }
  public int receive() throws IOException { return state.receive(); }
  public void send(int value) throws IOException { state.send(value); }
}
```

Обратим внимание, что в классах состояний определена только логика генерации событий, а логика переходов сконцентрирована в классе контекста.

2. Повторное использование классов состояний

Рассмотрим два расширения класса `Connection`. Первое расширение будет демонстрировать возможность добавления методов в интерфейс класса, а второе — изменение поведения за счет введения новых состояний.

2.1. Расширение интерфейса автомата

Проиллюстрируем возможность расширения интерфейса автомата на примере добавления возможности возврата данных в объект соединения для их последующего считывания. Введем интерфейс `IPushBackConnection`, расширяющий интерфейс `IConnection` методом `pushBack`. Таким образом, расширенный интерфейс выглядит следующим образом:

```
package push_back_connection;

import connection.IConnection;
import java.io.IOException;

public interface IPushBackConnection extends IConnection {
  void pushBack(int value) throws IOException;
}
```

Отметим, что код для этого примера помещен в пакет `push_back_connection`, диаграмма классов которого представлена на рис. 6.

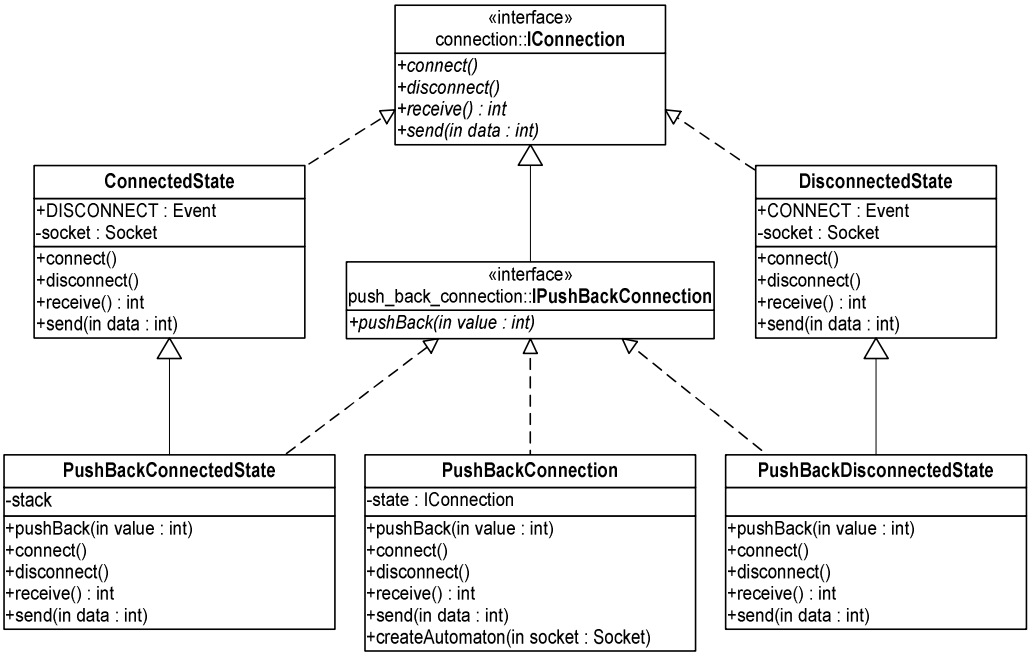

Рис. 6. Диаграмма классов пакета **`push_back_connection`**

При вызове метода `pushBack(int value)` значение, переданное в параметре этого метода, заносится в стек. При последующем вызове метода `receive` возвращается элемент из вершины стека, а если стек пуст, то значение извлекается из объекта `socket`.

Заметим, что в рассматриваемом случае количество управляющих состояний автомата не изменится, равно как и граф переходов автомата, но контекст и классы состояний должны реализовывать более широкий интерфейс `IPushBackConnection`. Назовем контекст нового автомата

BushBackConnection, а новые состояния — PushBackConnectedState и PushBackDisconnectedState.

Приведем реализацию класса PushBackConnectedState (класс PushBackDisconnectedState реализуется аналогично). При этом класс PushBackConnectedState расширяет класс ConnectedState, наследуя его логику:

```
package push_back_connection;

import connection.*;
import ru.ifmo.is.sm.IEventSink;
import java.util.Stack;
import java.io.IOException;

public class PushBackConnectedState <AI extends IPushBackConnection>
    extends ConnectedState<AI> implements IPushBackConnection {
  Stack<Integer> stack = new Stack<Integer>();

  public PushBackConnectedState(AI automaton, IEventSink eventSink,
        Socket socket) {
    super(automaton, eventSink, socket);
  }

  public int receive() throws IOException {
    if (stack.empty()) {
      return super.receive();
    }

    return stack.pop().intValue();
  }

  public void pushBack(int value) {
    stack.push(new Integer(value));
  }
}
```

Отметим, что в классе PushBackConnectedState параметр типа специализируется еще более чем в классе ConnectedState. Это требуется для того, чтобы поле automaton, определенное в классе StateBase имело правильный тип — IPushBackConnection. Таким образом, интерфейс автомата «протягивается» сквозь всю иерархию классов состояний через параметр типа.

Созданные классы состояний используются для реализации класса контекста PushBackConnection:

```
package push_back_connection;

import connection.Socket;
import ru.ifmo.is.sm.AutomatonBase;
import java.io.IOException;

public class PushBackConnection extends
        AutomatonBase<IPushBackConnection>
    implements IPushBackConnection {
  private PushBackConnection() {
    Socket socket = new Socket();

    // Создание объектов состояний
    IPushBackConnection connected = new
        PushBackConnectedState<PushBackConnection>(this, this, socket);
    IPushBackConnection disconnected = new
        PushBackDisconnectedState<PushBackConnection>(this, this,
        socket);
```

```
        // Логика переходов
        addEdge(connected, PushBackConnectedState.DISCONNECT, disconnected);
        addEdge(connected, PushBackConnectedState.ERROR, disconnected);
        addEdge(disconnected, PushBackDisconnectedState.CONNECT, connected);

        // Начальное состояние
        state = disconnected;
      }

      // Создание экземпляра автомата
      public static IPushBackConnection createAutomaton() {
        return new PushBackConnection();
      }

      // Делегирование методов интерфейса
      public void connect() throws IOException { state.connect(); }
      public void disconnect() throws IOException { state.disconnect(); }
      public int receive() throws IOException { return state.receive(); }
      public void send(int value) throws IOException { state.send(value); }
      public void pushBack(int value) throws IOException {
            state.pushBack(value); }
  }
```

Приведенный пример иллюстрирует, что классы состояний могут использоваться повторно в случае расширения интерфейса автомата.

2.2. Расширение логики введением новых состояний

Расширение логики поведения будем рассматривать на примере сетевого соединения, которое в случае возникновения ошибки при передаче данных закрывает канал и бросает исключение. При очередной попытке передачи данных производится попытка восстановить соединение и передать данные.

Для описания такого поведения требуется ввести новое состояние — *ошибка*, переход в которое означает, что канал передачи данных, закрыт из-за ошибки. Вызов любого метода передачи данных в этом состоянии будет приводить к установке нового соединения.

Таким образом, требуется реализовать класс `ResumableConnection`. Для этого необходимо дополнительно реализовать класс `ErrorState`, определяющий поведение в состоянии *ошибка*. Для состояний *соединено* и *разъединено* используются классы `ConnectedState` и `DisconnectedState`, уже разработанные в разд. 1.9. Граф переходов для класса `ResumableConnection` изображен на рис. 7.

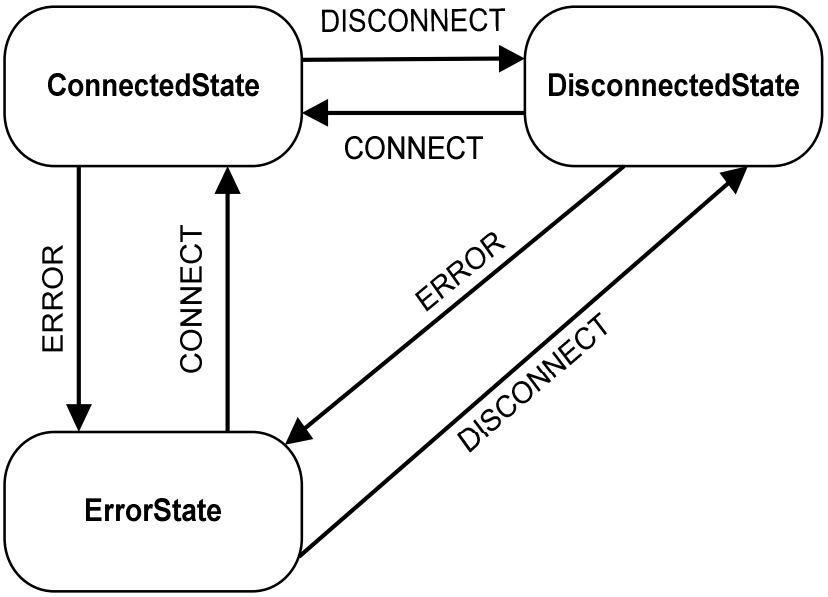

Рис. 7. Граф переходов класса **`ResumableConnection`**

Класс `ErrorState` реализуется следующим образом:

```
package resumable_connection;

import connection.*;
import ru.ifmo.is.sm.*;
import java.io.IOException;

public class ErrorState <AI extends IConnection>
    extends StateBase<AI> implements IConnection {
  public static final Event CONNECT = new Event("CONNECT");
  public static final Event DISCONNECT = new Event("DISCONNECT");

  protected final Socket socket;

  public ErrorState(AI automata, IEventSink eventSink, Socket socket) {
    super(automata, eventSink);
    this.socket = socket;
  }

  public void connect() throws IOException {
    socket.connect();
    castEvent(CONNECT);
  }

  public void disconnect() throws IOException {
    castEvent(DISCONNECT);
  }

  public int receive() throws IOException {
    connect();
    return automaton.receive();
  }

  public void send(int value) throws IOException {
    connect();
    automaton.send(value);
  }
```

```
}
```

Класс `ResumableConnection` реализуется следующим образом:

```
package resumable_connection;

import connection.*;
import ru.ifmo.is.sm.AutomatonBase;
import java.io.IOException;

public class ResumableConnection extends AutomatonBase<IConnection>
    implements IConnection {
  private ResumableConnection() {
    Socket socket = new Socket();

    // Создание объектов состояний
    IConnection connected = new
        ConnectedState<ResumableConnection>(this, this, socket);
    IConnection disconnected = new
        DisconnectedState<ResumableConnection>(this, this, socket);
    IConnection error = new ErrorState<ResumableConnection>(this, this,
        socket);

    // Логика переходов
    addEdge(connected, ConnectedState.DISCONNECT, disconnected);
    addEdge(connected, ConnectedState.ERROR, error);
    addEdge(disconnected, DisconnectedState.CONNECT, connected);
    addEdge(disconnected, DisconnectedState.ERROR , error);
    addEdge(error, ErrorState.CONNECT, connected);
    addEdge(error, ErrorState.DISCONNECT, disconnected);

    // Начальное состояние
    state = disconnected;
  }

  // Создание экземпляра автомата
  public static IConnection createAutomaton() {
    return new ResumableConnection();
  }

  // Делегирование методов интерфейса
  public void connect() throws IOException { state.connect(); }
  public void disconnect() throws IOException { state.disconnect(); }
  public int receive() throws IOException { return state.receive(); }
  public void send(int value) throws IOException { state.send(value); }
}
```

Из приведенного примера видно, что классы состояний могут быть использованы повторно при реализации других автоматов.

Выводы

Паттерн *State Machine* является усовершенствованием паттерна *State*. Он заимствует основную идею паттерна *State* — локализацию поведения, зависящего от состояния, в различных классах.

Новый паттерн устраняет ряд недостатков, присущих паттерну *State*.

- Паттерн *State Machine* позволяет разрабатывать отдельные классы независимыми друг от друга. Поэтому один и тот же класс состояния можно использовать в нескольких автоматах, каждый со своим набором состояний. Таким образом, устраняется главный недостаток паттерна *State* — сложность повторного использования.

- В паттерне *State* не описано каким образом обеспечивается чистота интерфейса, предназначенного для клиента. Предлагаемый паттерн устраняет эту проблему.
- В паттерне *State* логика переходов распределена по классам состояний, что порождает зависимости между классами и сложность восприятия логики переходов в целом. В паттерне *State Machine* логика переходов реализуется в контексте. Это позволяет разделить логику переходов и поведение в конкретном состоянии.
- Использование паттерна *State Machine* не приводит к дублированию интерфейсов.

Тем не менее, паттерн *State Machine* не устраняет такой недостаток паттерна *State*, как необходимость производить делегирование интерфейса автомата текущему объекту состояния вручную. Это ограничение можно обойти, используя автоматическую генерацию кода контекста или языки программирования, поддерживающие динамическое делегирование. Автоматическая генерация кода обычно используется в *CASE*-средствах. Второй подход можно реализовать, например, на языке *Self* [29], в котором описанное выше делегирование можно выполнить при помощи динамического наследования. Это обеспечивается тем, что язык позволяет непосредственно изменять класс объекта во время выполнения, а объекты в этом языке могут делегировать операции другим объектам.

Литература

1. **McCulloch W., Pitts W.** A Logical Calculus of Ideas Immanent in Nervous Activity //Bull. Math. Biophysics. 1943, 5. — P. 115-133.
2. **Гаврилов М.А.** Теория релейно-контактных схем. М.: Изд-во АН СССР, 1950. — 230 с.
3. **Huffman D.A.** The Synthesis of Sequential Switching Circuits //J. Franclin Inst. 1954. V.257, № 3, 4. — p. 161-190
4. **Mealy G.** A Method for Synthesizing Sequential Circuits //Bell System Technical Journal. 1955. V.34. № 5. — P.1045-1079
5. **Moore E.** Gedanken Experiments on Sequential Machines //В [6]. — P. 129-153.
6. **Automata Studies** //Ed. Shannon C.E., McCarthy J. Princeton Univ. Press, 1956. — P. 400 (Автоматы //Ред. Шеннона К.Э., МакКарти Дж. М.: Изд-во иностр. лит., 1956. — 451 с).
7. **Rubin M.**, **Scott D.** Finite automata and their decision problem //IBM J. Research and Development. 1959. V.3. № 2. — P. 115-125 (Кибернетический сборник. Вып.4. М.: Изд-во иностр. лит., 1962).
8. **Глушков В. М.** Синтез цифровых автоматов. М.: Изд-во физ.-мат. лит., 1962. — 476 с.
9. **Kleene S. C.** Representation of Events in Nerve Nets and Finite Automata //В [6]. — P. 3–41
10. **Thompson K.** Regular expression Search Algorithm //Communications of the ACM. 1966. V.11. № 6. — P. 419-422
11. **Aho A., Sethi R., Ullman J.** Compilers: Principles, Techniques and Tools. MA: Addison-Wesley, 1985, 500 p. (**Ахо А., Сети Р., Ульман Дж.** Компиляторы: принципы, технологии и инструменты. М.: Вильямс, 2001. —768 с).
12. **Hopcroft J., Motwani R., Ullman J.** Introduction to Automata Theory, Languages and Computation. MA: Addison-Wesley, 2001. — 521 p. (**Хопкрофт Д., Мотвани Р., Ульман Дж.** Введение в теорию автоматов, языков и вычислений. М.: Вильямс, 2001. — 528 с).
13. **Шалыто А. А.** SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998. — 628 с.
14. **Gamma E., Helm R., Johnson R., Vlissides J.** Design Patterns. MA: Addison-Wesley Professional. 2001. — 395 (**Гамма Э., Хелм Р., Джонсон Р., Влассидес Дж.** Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб.: Питер, 2001. — 368 с).
15. **Steling S., Maassen O.** Applied Java Patterns. Pearson Higher Education. 2001, P. 608 (**Стелтинг С., Массен О**. Применение шаблонов Java. Библиотека профессионала. М.: Вильямс, 2002. — 564 с).
16. **Grand M.** Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML. Wiley, 2002. — 544 p. (**Гранд М.** Шаблоны проектирования в Java. М.: Новое знание, 2004. — 560 с).
17. Java Data Objects (JDO). http://java.sun.com/products/jdo/index.jsp.
18. **Eliens A.** Principles of Object-Oriented Software Development. MA.: Addison-Wesley, 2000. — 502 p. (**Элиенс А.** Принципы объектно-ориентированной разработки программ. М.: Вильямс, 2002. — 496 с).

19. **Sandén B.** The state-machine pattern // Proceedings of the conference on TRI-Ada '96 http://java.sun.com/products/jdo/index.jsp.
20. **Adamczyk P.** The Anthology of the Finite State Machine Design Patterns. http://jerry.cs.uiuc.edu/~plop/plop2003/Papers/Adamczyk-State-Machine.pdf
21. **Odrowski J., Sogaard P.** Pattern Integration — Variations of State // Proceedings of PLoP96. http://www.cs.wustl.edu/~schmidt/PLoP-96/odrowski.ps.gz.
22. **Sane A., Campbell R.**. Object-Oriented State Machines: Subclassing, Composition, Delegation, and Genericity // OOPSLA ’95. http://choices.cs.uiuc.edu/sane/home.html.
23. **Шалыто А. А., Туккель Н. И.** От тьюрингова программирования к автоматному. // Мир ПК. 2002. № 2. — С. 144-149. (http://is.ifmo.ru, раздел «Статьи»).
24. **Harel D.** Statecharts: A visual formalism for complex systems //Sci. Comput. Program. 1987. Vol.8. — P. 231-274
25. **Fawler M.** Refactoring. Improving the Design of Existing Code. MA: Addison-Wesley. — 1999. — 431 p. (**Фаулер М.** Рефакторинг. Улучшение существующего кода. — М.: Символ-плюс, 2003. — 432 с).
26. **Martin R.** Three Level FSM // PLoPD, 1995. http://cpptips.hyperformix.com/cpptips/fsm5.
27. **Шалыто А. А., Туккель Н. И.** SWITCH-технология — автоматный подход к созданию программного обеспечения «реактивных» систем // Программирование. 2001. № 5. С. 45-62. (http://is.ifmo.ru, раздел «Статьи»).
28. Раздел «Статьи» сайта кафедры «Технологии программирования» Санкт-Петербургского государственного университета информационных технологий, механики и оптики (http://is.ifmo.ru/articles).
29. **The Self Language.** (http://research.sun.com/self/language.html).