

## **State Machine — расширение языка Java для эффективной реализации автоматов**

**Н. Н. Шамгунов,**

аспирант

**Г. А. Корнеев,**

аспирант

**А. А. Шалыто,**

д-р техн. наук, профессор

Санкт-Петербургский государственный университет информационных технологий, механики и оптики

*В данной статье предлагается новый язык объектно-ориентированного программирования State Machine, являющийся расширением языка программирования Java. В язык вводится ряд синтаксических конструкций, позволяющих программировать с использованием понятий автомат, состояние и событие. Для обработки новых синтаксических конструкций разработан препроцессор, преобразующий код на языке State Machine в код на языке Java. При этом новые синтаксические конструкции преобразуются в код на языке Java в соответствии с предложенным ранее авторами паттерном State Machine.*

*This paper presents a new Object-oriented programming language State Machine. This language is an extension of the Java. This language presents a number of syntactic constructions that provides an ability to create programs with notion of automaton, state and event. Author developed a preprocessor that transforms State Machine code to Java code. This preprocessor transforms new syntactic constructions to Java code according to State Machine pattern which was presented by these authors earlier.*

### **Введение**

В программировании часто возникает потребность в объектах, изменяющих свое поведение в зависимости от состояния. Обычно поведение таких объектов описывается при помощи конечных автоматов. Существуют различные паттерны проектирования для реализации указанных объектов, приведенные, например, в работах [1, 2]. В большинстве из этих паттернов или автоматы реализуются неэффективно, или сильно затруднено повторное использование компонентов автомата. Эти недостатки устранены в предложенном авторами паттерне *State Machine* [3].

В последнее время имеет место тенденция создания языков, ориентированных на предметную область [4]. В данном случае такой областью является автоматное программирование.

Одним из способов создания предметно-ориентированных языков является расширение существующих, например, за счет введения в них автоматных конструкций [5, 6].

В работе [7] был предложен язык *State* (расширяющий язык *C#* [8]), который предназначен для реализации объектов с изменяющимся поведением. Однако, конструкции, вводимые в этом языке, невозможно реализовать эффективно, поскольку в нем вызов метода объекта влечет за собой вычисление набора предикатов.

Зарекомендовавшим себя способом расширения языков программирования является встраивание в них поддержки паттернов проектирования. Например, в язык *C#* встроены паттерн *Observer* [1], широко используемый, например, для реализации графического интерфейса пользователя.

В данной работе предлагается язык программирования *State Machine*, расширяющий язык *Java* [9], который основывается на одноименном паттерне. В качестве основного языка был выбран язык программирования *Java*, так как для него существуют современные инструменты создания компиляторов, для которых есть не только документация, но и книга [11].

Статья состоит из пяти частей. В первой части описываются особенности предлагаемого языка *State Machine*. Во второй — приведен пример класса, реализующего сетевое соединение, и его реализация на предлагаемом языке. Третья часть содержит описание грамматики вводимых синтаксических конструкций. В четвертой части рассматривается повторное использование кода, написанного на предлагаемом языке. Пятая часть содержит описание реализации препроцессора.

## 1. Особенности языка *State Machine*

В предлагаемом языке, как и в паттерне *State Machine*, основной идеей является описание объектов, варьирующих свое поведение, в виде автоматов. В предложенном в работе [3] подходе разделяются классы, реализующие логику переходов (контексты), и классы состояний. Переходы инициируются состояниями путем уведомления контекста о наступлении событий. При этом в зависимости от события и текущего состояния, контекст устанавливает следующее состояние в соответствии с графом переходов.

Отметим, что если в паттерне *State* [1] следующее состояние указывается текущим, то в паттерне *State Machine* это выполняется путем уведомления класса контекста о наступлении события.

Логика переходов задается в терминах состояний и событий. При этом в языке *State Machine* полностью скрывается природа событий. Для пользователя они представляют собой сущности, принадлежащие классу состояний и участвующие в формировании таблицы переходов. Для описания логики переходов на этапе проектирования используются специальные графы переходов, предложенные в работе [3]. Эти графы состоят только из состояний и переходов, помеченных событиями.

По сравнению с языком *Java* в язык *State Machine* введены дополнительные конструкции, позволяющие описывать объекты с варьирующимся поведением в терминах автоматного программирования, определенных в работе [3]: *автоматов, состояний и событий*. Для описания автоматов и состояний в язык введены ключевые слова `automaton` и `state` соответственно, а для событий — ключевое слово `events`.

Отметим, что в предлагаемом языке, также как и паттерне *State Machine*, события являются основным способом воздействия объекта состояния на контекст. В указанном паттерне программист должен самостоятельно создавать объекты, представляющие события в виде статических переменных класса `Event`, в то время как в языке *State Machine* события введены как часть описания состояний. Это сделано для того, чтобы подчеркнуть их важность.

В паттерне *State Machine* реализация интерфейса автомата в контексте делегирует вызовы методов интерфейса текущему экземпляру состояния, причем делегирование реализовывалось вручную. В программе на языке *State Machine* это делать не требуется, так как препроцессор автоматически сгенерирует соответствующий код. Для этого используется технология *Reflection* [12]. Поэтому важно, чтобы препроцессор и генерируемый им код были реализованы на одном языке программирования. В частности, если бы язык расширял язык *C#* (как язык *State*), то и сам препроцессор необходимо было бы написать на языке *C#*.

Также как в паттерне *State Machine*, в предлагаемом языке состояние может делегировать дальнейшее выполнение действия автомату. При этом для ссылки на автомат используется ключевое слово `automaton` (также как и при описании автоматов).

Делегирование действия автомату требуется, например, при восстановлении после ошибки (соответствующий пример рассмотрен в разделе «Повторное использование»). В этом случае состояние, обрабатывающее ошибку, осуществляет действия по восстановлению и, в случае успеха, передает управление новому состоянию автомата.

Описание автоматов и состояний на языке *State Machine* помещаются в файлы с расширением `.sm`. Авторами разработан препроцессор, преобразующий код, написанный на предлагаемом языке, в код на языке *Java* (в файлы с расширением `.java`). При этом новые синтаксические конструкции преобразуются в соответствии с паттерном *State Machine*. Препроцессор генерирует код, содержащий параметры типа (`generics`) [10], что позволяет осуществлять проверку типов во время компиляции. Полученный код компилируется при помощи *Java*-компилятора, поддерживающего параметры типа.

## 2. Пример использования языка *State Machine*

В данном разделе особенности новых синтаксических конструкций языка *State Machine* рассматриваются на примере проектирования и реализации класса `Connection`, описанного в работе [3]. Приведем его краткое описание.

### 2.1. Описание примера

Требуется спроектировать класс `Connection`, представляющий сетевое соединение, имеющее два управляющих состояния: *соединено* и *разъединено*. Переход между ними 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. В случае возникновения ошибки при передаче данных генерируется исключительная ситуация (`IOException`) и сетевое соединение переходит в состояние *разъединено*, в котором прием и отправка данных невозможны. При попытке осуществить передачу данных в этом состоянии объект также генерирует исключительную ситуацию.

Интерфейс, который требуется реализовать в классе `Connection`, выглядит следующим образом:

```
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;
}
```

В работе [3] для реализации состояний *соединено* и *разъединено*, предложено использовать классы `ConnectedState` и `DisconnectedState` соответственно. Состояния уведомляют контекст о событиях: класс `ConnectedState` — о событиях `DISCONNECT` и `ERROR`, а класс `DisconnectedState` — о событиях `CONNECT` и `ERROR`.

Для рассматриваемого примера на рис. 1 представлен граф переходов вида, используемого в предлагаемом подходе [3].

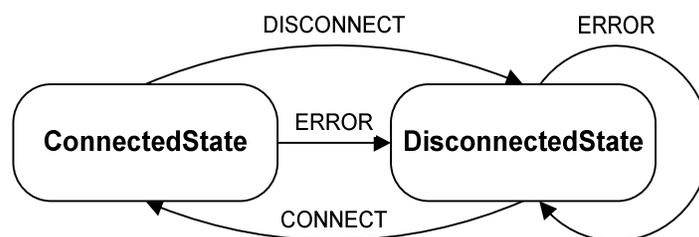


Рис. 1. Граф переходов для класса `Connection`

### 2.2. Описание состояний

Для описания состояния используется ключевое слово `state`. Приведем код состояния `ConnectedState` на языке *State Machine*:

```
package connection;

import java.io.IOException;

public state ConnectedState implements IConnection events DISCONNECT, ERROR {
    protected final Socket socket;

    public ConnectedState(Socket socket) {
```

```

        this.socket = socket;
    }

    public void connect() throws IOException {
    }

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

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

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

```

При описании состояния указан интерфейс автомата (`IConnection`) и список событий, которые это состояние может сгенерировать (`ERROR`, `DISCONNECT`). Также как в паттерне *State Machine*, контекст уведомляется о наступлении события вызовом метода `castEvent`. За исключением этого, состояния описываются аналогично классу на языке *Java*.

В предлагаемом языке состояние может реализовывать несколько интерфейсов. При этом первый из реализуемых состоянием интерфейсов будет считаться интерфейсом автомата.

Для реализации автомата `Connection`, необходимо также описать состояние `DisconnectedState`:

```

package connection;

import java.io.IOException;

public state DisconnectedState implements IConnection events CONNECT, ERROR {
    protected final Socket socket;

    public DisconnectedState(Socket socket) {
        this.socket = socket;
    }

    public void connect() throws IOException {
        try {
            socket.connect();
            castEvent(CONNECT);
        } catch (IOException e) {
            castEvent(ERROR);
            throw e;
        }
    }

    public void disconnect() throws IOException {
    }

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

```

```

    }

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

```

### 2.3. Описание автомата

В языке *State Machine* автомат предназначен для определения набора состояний и переходов.

Для описания автомата применяется ключевое слово `automaton`. Приведем код на предлагаемом языке для автомата `Connection`, реализующий граф переходов (рис. 1):

```

package connection;

public automaton Connection implements IConnection {
    state DisconnectedState disconnected(CONNECT -> connected, ERROR ->
        disconnected);
    state ConnectedState connected(ERROR -> disconnected, DISCONNECT ->
        disconnected);
    public Connection(Socket socket) {
        disconnected @= new DisconnectedState(socket);
        connected @= new ConnectedState(socket);
    }
}

```

Обратим внимание, что класс автомата должен реализовывать ровно один интерфейс, который и считается интерфейсом автомата. В данном примере — это интерфейс `IConnection`.

Состояния (`connected` и `disconnected` классов `ConnectedState` и `DisconnectedState` соответственно) описываются при помощи ключевого слова `state`. Первое из состояний, описанных в автомате, является стартовым. В данном примере это состояние `disconnected`.

Переходы по событиям описываются в круглых скобках, после имени состояния. Для одного состояния переходы разделяются запятыми. Например, для состояния `connected` переходами являются `DISCONNECT -> disconnected` и `ERROR -> disconnected`. Первый из них означает, что при поступлении события `DISCONNECT` в состоянии `connected` автомат переходит в состояние `disconnected`.

В конструкторе `public Connection(Socket socket)` производится создание объектов состояний. Отметим, что состояния, входящие в автомат, должны реализовать интерфейс автомата. Инициализация объектов состояний производится при помощи нового оператора `@=`, специально введенного для этой цели в язык *State Machine*. Таким образом, оператор `connected @= new ConnectedState(socket)` означает инициализацию состояния `connected` новым объектом класса `ConnectedState`.

За исключением этого, автомат описывается аналогично классу на языке *Java*.

Отметим, что состояния автомата перечисляются, но не определяются в нем. Таким образом, одни и те же состояния могут использоваться для реализации различных автоматов.

### 2.4. Компиляция примера

Для генерации *Java*-кода из файлов с расширением `.sm` необходимо выполнить команду `java ru.ifmo.is.sml.Main <имя файла1> [,имя файла2,...,имя файлаN]`.

В результате будет сформирован одноименный файл с расширением `.java`. Отметим, что для генерации класса `Connection` необходимо предварительно скомпилировать *Java*-компилятором интерфейс `IConnection`. Это требуется для генерации реализации этого интерфейса в соответствующем классе.

Для полной компиляции данного примера необходимо выполнить следующую последовательность команд:

```

rem Компиляция интерфейса автомата
javac IConnection.java

rem Преобразование состояний
java ru.ifmo.is.sml.Main ConnectedState.sm DisconnectedState.sm

```

```

rem Компиляция классов состояний
javac ConnectedState.java DisconnectedState.java

rem Преобразование автомата Connection
java ru.ifmo.is.sml.Main Connection.sm

rem Компиляция автомата
javac Connection.java

```

В результате будут сформированы соответствующие *Java*-файлы, которые будут скомпилированы *Java*-компилятором `javac`.

Отметим, что для компиляции и работы полученных классов требуется только класс `AutomatonBase`, определенный в пакете `ru.ifmo.is.sml.runtime`.

### 3. Грамматика описания автоматов и состояний

Как отмечено выше, язык программирования *State Machine* основан на языке *Java*, в который вводятся синтаксические конструкции для поддержки программирования в терминах *состояние* и *автомат*.

В данном разделе приводятся грамматики в расширенной форме Бэкуса-Наура [13] для описания этих конструкций.

#### 3.1. Грамматика описания состояния

```

state_decl      ::=  modifiers state
                  type
                  extends_decl?
                  implements_decl
                  events?
                  {
                  balanced
                  }
extends_decl    ::=  extends type
implements_decl ::=  implements type (, type)*
type           ::=  id (. id)*
events        ::=  events id (, id)*
balanced      ::=  <сбалансированная по скобкам
                  последовательность>
modifiers     ::=  (abstract | final | strictfp | public)*

```

Здесь и далее терминальные символы выделены полужирным шрифтом, а нетерминальные — наклонным. Для краткости не раскрывается определение нетерминала *balanced*. Он соответствует сбалансированной относительно использования круглых и фигурных скобок последовательности терминальных и нетерминальных символов [14].

Состояние должно реализовывать не менее одного интерфейса. При этом первый из них считается интерфейсом автомата.

В коде состояния возможно делегирование методов текущему состоянию автомата. Для этого используется ключевое слово `automaton`, которое имеет тип интерфейса автомата.

Отметим, что в данной версии языка состояния не могут содержать параметры типа.

## 3.2. Грамматика описания автомата

```
automaton_decl ::= modifiers automaton
                type
                implements_decl
                {
                state_var_decl+ balanced
                }
state_var_decl ::= state type id ( event_mapping ( , event_mapping)* ) ;
event_mapping ::= id ( , id)* -> id
```

Отметим, что интерфейс автомата должен совпадать у автомата и всех состояний, которые он использует. Это семантическое правило, поэтому оно не может быть выражено грамматикой.

Для инициализации состояний в конструкторе автомата применяется оператор @=. Слева от него указывается имя состояния, а справа — объект, реализующий это состояние. Тип указанного объекта должен в точности совпадать с типом, указанным при описании автомата.

В конструкторе все состояния автомата должны быть проинициализированы. При этом каждое — не более одного раза (как если бы они были обыкновенными переменными, описанными с модификатором `final`).

Использование оператора @= вне конструктора является ошибкой.

## 4. Повторное использование

Одним из преимуществ объектно-ориентированного программирования является возможность повторного использования кода. Эта возможность также поддерживается и в предлагаемом языке.

### 4.1. Допустимые способы повторного использования

В *Java* объектно-ориентированном программировании интерфейс объекта представляет собой некоторый контракт [18]. При этом возможно наследование класса, основываясь только на его контракте. Наследование от автомата как класса допустимо, но для расширения его поведения в наследнике необходимо изменять набор состояний и функцию переходов. Таким образом, пользователь не может воспринимать базовый автомат как черный ящик, поскольку доступ к реализации функции переходов базового автомата нарушает инкапсуляцию. Поэтому наследование автомата от класса или другого автомата в языке *State Machine* запрещено.

При этом предлагаемый язык допускает повторное использование классов состояний. Также как и в паттерне *State Machine*, это может быть сделано двумя способами:

- наследованием состояний;
- использованием классов состояний в нескольких автоматах.

В первом способе создаются наследники состояний, реализующие более широкий интерфейс по сравнению с базовым состоянием. Из полученных состояний конструируется новый автомат. Во втором способе одни и те же классы состояний используются для конструирования различных автоматов. В обоих случаях определяется новый автомат со своим набором состояний и переходов в нем. Также возможна комбинация этих подходов.

Ниже оба способа повторного использования классов состояний будут рассмотрены на примерах.

### 4.2. Описание примеров

Следуя работе [3], рассмотрим две модификации автомата `Connection`.

Автомат `PushBackConnection` предоставляет возможность возврата данных в объект соединения и их последующего считывания. Интерфейс этого автомата — `IPushBackConnection` расширяет интерфейс `IConnection`:

```
package push_back_connection;

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

public interface IPushBackConnection extends IConnection {
```

```

    void pushBack(int value) throws IOException;
}

```

Автомат `ResumableConnection` реализует следующую логику поведения класса, представляющего сетевое соединение. В случае возникновения ошибки при передаче данных автомат закрывает канал связи и генерирует исключительную ситуацию. При очередном вызове метода передачи данных производится попытка восстановить соединение.

Для описания такого поведения требуется ввести новое состояние — *ошибка*, переход в которое означает, что канал передачи данных закрыт из-за ошибки. Полученный граф переходов, используемого в предлагаемом подходе вида, изображен на рис. 2.

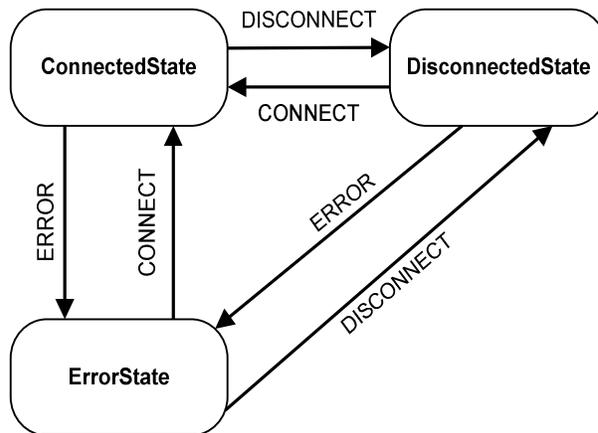


Рис. 2. Граф переходов автомата `ResumableConnection`

### 4.3. Наследование состояний

Для реализации автомата `PushBackConnection` необходимо описать два новых состояния `PushBackConnectedState` и `PushBackDisconnectedState`, унаследованных от состояний `ConnectedState` и `DisconnectedState` соответственно. Это позволяет обеспечить повторное использование кода состояний.

Приведем код состояния `PushBackConnectedState`:

```

package push_back_connection;

import connection.*;
import java.util.Stack;
import java.io.IOException;

public state PushBackConnectedState extends ConnectedState implements
    IPushBackConnection {
    private final Stack<Integer> stack = new Stack<Integer>();

    public PushBackConnectedState(Socket socket) {
        super(socket);
    }

    public int receive() throws IOException {
        return stack.empty() ? super.receive() : stack.pop();
    }

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

```

Состояние `PushBackDisconnectedState` реализуется аналогично:

```

package push_back_connection;

import connection.*;

```

```

import java.io.IOException;

public state PushBackDisconnectedState extends DisconnectedState implements
    IPushBackConnection {
    public PushBackDisconnectedState(Socket socket) {
        super(socket);
    }

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

```

Автомат PushBackConnection не наследует автомат Connection. Повторное использование кода достигается за счет применения наследников состояний ConnectedState и DisconnectedState.

```

package push_back_connection;

import connection.Socket;

public automaton PushBackConnection
    implements IPushBackConnection
    states
        PushBackConnectedState connected {
            ERROR -> disconnected,
            DISCONNECT -> disconnected
        },
        PushBackDisconnectedState disconnected {
            CONNECT -> connected,
            ERROR -> disconnected
        }
    {
        public PushBackConnection(Socket socket) {
            connected @= new PushBackConnectedState(socket);
            disconnected @= new PushBackDisconnectedState(socket);
        }
    }
}

```

#### 4.4. Использование одного состояния в различных автоматах

Для реализации автомата ResumableConnection необходимо дополнительно реализовать состояние ErrorState, определяющее поведение в состоянии *ошибка*. В автомате также будут использованы состояния ConnectedState и DisconnectedState, разработанные для автомата Connection.

Приведем код состояния ErrorState:

```

package resumable_connection;

import connection.*;
import java.io.IOException;

public state ErrorState implements IConnection events CONNECT, DISCONNECT {
    protected final Socket socket;

    public ErrorState(Socket socket) {
        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);
    }
}

```

Теперь можно определить автомат ResumableConnection:

```

package resumable_connection;

import connection.*;

public automaton ResumableConnection implements IConnection {
    state DisconnectedState disconnected(CONNECT -> connected, ERROR -> error);
    state ConnectedState connected(DISCONNECT -> disconnected, ERROR -> error);
    state ErrorState error(DISCONNECT -> disconnected, CONNECT -> connected);

    private ResumableConnection() {
        Socket socket = new Socket();

        connected @= new ConnectedState(socket);
        disconnected @= new DisconnectedState(socket);
        error @= new ErrorState(socket);
    }
}

```

Из приведенных примеров следует, что состояния могут быть использованы повторно.

## 5. Реализация препроцессора

Для реализации препроцессора были использованы инструменты создания компиляторов *JLex* и *Cup*, описанные в работе [11], которые распространяются по лицензии *open source license* [15]. Первый из них предназначен для построения лексических анализаторов, а второй — синтаксических [14].

В результате работы препроцессора производится преобразование переданных ему в качестве параметров файлов с расширением *.sm*, содержащих код автоматов и состояний на языке *State Machine*, в файлы с расширением *.java*. В процессе преобразования теряется исходное форматирование программы и комментарии. Однако это не является недостатком, поскольку получаемый код является промежуточным и не предназначен для редактирования вручную.

Препроцессор (в открытых кодах) можно скачать по адресу <http://is.ifmo.ru>, раздел *Статьи*. Для работы препроцессора необходимо также установить инструменты создания компиляторов *JLex* и *Cup*, доступные по адресу [16].

### 5.1. Генерация Java-классов по описанию состояний

Каждое состояние, описанное на языке *State Machine*, преобразуется в одноименный класс на языке *Java*. При этом все методы и их реализация переходят в генерируемый класс.

В случае если одно состояние расширяет другое, то генерируемый класс будет расширять соответствующий ему *Java*-класс. В противном случае, генерируемый класс будет расширять класс `AutomatonBase.StateBase`, входящий в пакет `ru.ifmo.is.sml.runtime`.

Указанный класс является базовым для всех классов состояний. В нем определено поле `automaton`, позволяющее вызывать методы автомата, в котором содержится данный экземпляр состояния. Отметим, что в языке *StateMachine* `automaton` является ключевым словом. Таким образом, конфликтов с полями, определенными пользователем, не возникает.

В классе `StateBase` также реализован метод `castEvent`, который состояния вызывают для уведомления автомата о наступлении события. Для реализации событий используется класс `AutomatonBase.StateBase.Event`.

Рассмотрим код, сгенерированный препроцессором для состояния `ConnectedState`. Отметим, что здесь и ниже форматирование сгенерированного кода и вставка комментариев выполнены вручную:

```
package connection;

import java.io.IOException;

public class ConnectedState<AI extends IConnection>
    // Базовый класс, для всех состояний
    extends ru.ifmo.is.language.AutomatonBase.StateBase<AI>
    implements IConnection {
    // События преобразуются в набор статических переменных
    public final static Event DISCONNECT = new Event(ConnectedState.class,
        "DISCONNECT", 0);
    public final static Event ERROR = new Event(ConnectedState.class, "ERROR",
        1);

    // В остальном -- без изменений

    protected final Socket socket;

    public ConnectedState(Socket socket) {
        this.socket = socket;
    }

    public void connect() throws IOException {}

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

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

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

Обратим внимание, что в этом коде, также как в паттерне *State Machine*, для представления событий используются статические переменные. При создании экземпляров событий (класса `Event`) первыми двумя параметрами конструктора являются ссылка на класс, определивший событие и имя события, которые могут быть использованы для автоматического протоколирования [17]. Третий параметр необходим для эффективной реализации графов переходов. Он представляет собой порядковый номер события в состоянии, отсчитываемый от нуля. При этом учитываются события, определенные в базовом состоянии, которым присваиваются меньшие номера.

## 5.2. Генерация *Java*-классов по описанию автоматов

Автомат в языке *State Machine* также преобразуется в одноименный класс, унаследованный от класса `AutomatonBase`, определенного в пакете `ru.ifmo.is.sml.runtime`.

Преппроцессор автоматически генерирует методы, реализующие интерфейс автомата. В каждом из них вызывается соответствующий метод текущего состояния. Такой метод гарантированно присутствует в состоянии, поскольку автомат и состояние реализуют интерфейс автомата.

Для автомата `Connection` преппроцессор генерирует следующий *Java*-класс:

```
package connection;

import java.io.IOException;
import ru.ifmo.is.sml.runtime.AutomatonBase;

public class Connection
    extends AutomatonBase<IConnection>
    implements IConnection
{
    private final static int[][] $TRANSITION_TABLE
        = new int[][] {
        {
            /* CONNECT      */ 0, /* connected */
            /* ERROR        */ 1, /* disconnected */
        },
        {
            /* DISCONNECT  */ 1, /* disconnected */
            /* ERROR        */ 1, /* disconnected */
        },
    };

    public Connection(Socket socket) {
        super(new IConnection[2/*количество состояний*/]);
        {
            DisconnectedState<IConnection> $state = new
                DisconnectedState<IConnection>(socket);
            state(0/*disconnected*/, $state, $state, this,
                $TRANSITION_TABLE[0/*disconnected*/]);
        }
        {
            ConnectedState<IConnection> $state = new
                ConnectedState<IConnection>(socket);
            state(1/*connected*/, $state, $state, this,
                $TRANSITION_TABLE[1/*connected*/]);
        }
    }

    // Делегирование методов интерфейса автомата
    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); }
}
```

Отметим, что использование имени `state` для поля, хранящего текущее состояние, не приводит к неоднозначности, так как в предлагаемом языке оно является ключевым словом. По этой же причине метод, связывающий состояние с автоматом, также называется `state`. Интересной особенностью является необходимость дважды передавать в этот метод ссылку на состояние (`state`), а так же ссылку на сам автомат (`this`). Это связано с невозможностью на языке *Java* определить класс, унаследованный от своего параметра типа.

На основе переходов, заданных в описании автомата, преппроцессор строит таблицу переходов, хранящуюся в статическом поле `$TRANSITION_TABLE`. Таблица представляет собой массив массивов целых чисел. В приведенном коде таблица переходов является матрицей  $2 \times 2$ . Однако, в общем случае, она матрицей не является, поскольку состояния могут порождать разное количество событий.

Строки таблицы переходов передаются объектам состояний при помощи вызова метода `init`, входящего в класс `AutomatonBase`. Это позволяет представить функцию уведомления о событии (`castEvent`) в компактном виде.

При такой реализации для осуществления каждого перехода требуется фиксированное время, существенно меньшее, чем в реализации, предложенной в работе [3]. Таким образом, базовый класс для всех автоматов выглядит следующим образом:

```

package ru.ifmo.is.sml.runtime;

public abstract class AutomatonBase<AI> {
    private final AI[] states;
    protected AI state;

    public AutomatonBase(AI[] states) {
        this.states = states;
    }

    protected void state(int index, StateBase<AI> state, AI stateI, AI automaton,
        int[] transitions) {
        states[index] = stateI;
        state.base = this;
        state.automaton = automaton;
        state.transitions = transitions;
        if (index == 0) this.state = states[index];
    }

    public static abstract class StateBase<AI> {
        protected AI automaton;
        private AutomatonBase<AI> base;
        private int[] transitions;

        protected void castEvent(Event event) {
            base.state = base.states[transitions[event.index]];
        }

        public final static class Event {
            private final Class state;
            private final String name;
            private final int index;

            public Event(Class state, String name, int index) {
                this.state = state;
                this.name = name;
                this.index = index;
            }
        }
    }
}

```

Обратим внимание, что класс AutomatonBase содержит класс StateBase как вложенный.

## Выводы

Предложенный язык программирования *State Machine* обладает следующими достоинствами.

1. Позволяет писать программы в терминах автоматного программирования.
2. Непосредственно поддерживает одноименный паттерн.
3. Повышает компактность и облегчает восприятие кода, по сравнению с реализацией паттерна *State Machine* непосредственно на языке *Java*.
4. Обеспечивает более быстрое выполнение переходов по сравнению с реализацией паттерна *State Machine*, приведенного в работе [3].
5. Предложенный синтаксис позволяет распознавать паттерн в коде, в том числе и автоматически, что может быть использовано при построении диаграмм и документации.

## Литература

1. **Gamma E., Helm R., Johnson R., Vlissides J.** Design Patterns. MA: Addison-Wesley Professional, 2001. — P.395 (**Гамма Э., Хелм Р., Джонсон Р., Влассидес Дж.** Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб.: Питер, 2001. — 368 с).
2. **Adamczyk P.** The Anthology of the Finite State Machine Design Patterns. <http://jerry.cs.uiuc.edu/~plop/plop2003/Papers/Adamczyk-State-Machine.pdf>

3. **Шамгунов Н. Н., Корнеев Г. А., Шалыто А. А.** Паттерн *State Machine* для объектно-ориентированного проектирования автоматов // Информационно-управляющие системы. 2004. № 5, с.
4. **Эндрю Х., Дэвид Т.** Программист-прагматик. М.: Лори, 2004. — 288 с.
5. Язык прикладного программирования STL 1.0. Руководство пользователя (v1.0.13b). [http://lmt-automation.ifmo.ru/pdfs/stlguide\\_1\\_0\\_13b.pdf](http://lmt-automation.ifmo.ru/pdfs/stlguide_1_0_13b.pdf)
6. **Ваганов С. А.** FloraWare — ускорить разработку приложений. <http://www.softcraft.ru/paradigm/oop/flora/index.shtml>
7. **Шамгунов Н. Н., Шалыто А. А.** Язык автоматного программирования с компиляцией в *Microsoft CLR*. // Microsoft Research Academic Days in St. Petersburg, 2004.
8. <http://msdn.microsoft.com/vcsharp/team/language/default.aspx> C# Language.
9. **Gosling J., Joy B., Steele G., Bracha G.** The Java Language Specification. [http://java.sun.com/docs/books/jls/second\\_edition/html/j.title.doc.html](http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html).
10. Generics. <http://java.sun.com/j2se/1.5.0/docs/guide/language/generics.html>
11. **Appel A. W.** Modern Compiler Implementation in Java. — NY, Cambridge, 1998. — 512 с.
12. **Green A.** Trail: The Reflection API. <http://java.sun.com/docs/books/tutorial/reflect/>
13. **Naur P. et al.** Revised Report on the Algorithmic Language ALGOL 60 //Communications of the ACM. 1960. Vol. 3. No.5, pp. 299-314.
14. **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).
15. *Open Source License*. <http://www.opensource.org/licenses/historical.php>
16. <http://www.cs.princeton.edu/~appel/modern/java/> Modern Compiler Implementation in Java.
17. **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»).
18. *Object-Oriented Programming Concepts* <http://java.sun.com/docs/books/tutorial/java/concepts/>