Санкт-Петербургский государственный университет информационных технологий, механики и оптики

Кафедра “Компьютерные технологии”

М.Г. Раер

Автоматное расширение языка C#

Санкт-Петербург

2006

ОГЛАВЛЕНИЕ

1. ВВЕДЕНИЕ

Часто поведение объекта, его реакция на определенные события зависит от состояния, в котором он находится. Такие объекты принято описывать конечными автоматами. Существует большое количество способов реализации автоматов в программировании [1, 2]. Рассматривается спектр вариантов, начиная от не объектно-ориентированного, когда автомат реализуется вложенными операторами *if* или *switch*, до полностью объектно-ориентированного, когда все элементы автомата (состояния, события, переходы, действия) реализованы как классы. Самой распространенной объектно-ориентированной реализацией является паттерн *State* [3]. В работе [1] рассмотрен список различных модификаций этого паттерна.

Паттерн проектирования *State* предлагает вынести код, отвечающий за поведение объекта в неком состоянии, в отдельный класс. При этом в методах класса в зависимости от условий производится переход в другое состояние. Таким образом, в данном случае код, отвечающий за смену состояний, рассредоточен по классам состояний. Это затрудняет модификацию кода, его поддержку и нарушает централизацию логики управления, как это предложено в SWITCH-технологии [4]. Кроме того, возникает зависимость между классами состояний, так как они должны “знать” друг о друге.

В работе [6] был предложен паттерн *State Machine*, устраняющий указанную зависимость и концентрирующий логику управления в одном месте. В этой работе приводилась реализация на языке *Java*, поэтому в классы состояний пришлось ввести члены-события (объекты специального класса `Event`) и наследовать состояния от базового класса `StateBase`. Это вносит некоторые ограничения на использование классов в качестве классов-состояний.

В работе [7] разработано расширение языка *Java* для более удобной реализации паттерна *State Machine*.

В настоящей работе предлагается реализация паттерна *State Machine* для языка C# [8], который снимает указанные выше ограничения. Также предлагается дополнение языка C# (*State#*) для более эффективной реализации паттерна. В качестве состояний становится возможным использовать произвольный класс, автомат может реализовывать произвольное количество автоматных интерфейсов. В язык *State#* вводится возможность автоматического протоколирования, событие смены состояния. Условия перехода можно задавать не только событиями, но и дополнительными условиями, предусмотрена возможность описывать действия на переходе.

2. ОБЗОР ПАТТЕРНОВ *STATE* И *STATE MACHINE*

2.1. Паттерн *State*

Паттерн `State` позволяет объекту изменять свое поведение в зависимости от своего состояния. Он входит в группу паттернов поведения.

Вводится абстрактный класс `State`, определяющий интерфейс объекта. Подклассы класса `State` задают поведение объекта. Каждому состоянию соответствует подкласс. Определяется основной класс `Context`. В нем хранится объект текущего состояния. Класс `Context` делегирует все запросы из интерфейса `State` этому объекту. Методы наследников класса `State` могут изменять значение этого объекта, изменяя тем самым состояние системы. Схематически структура паттерна `State` изображена на рис. 1.

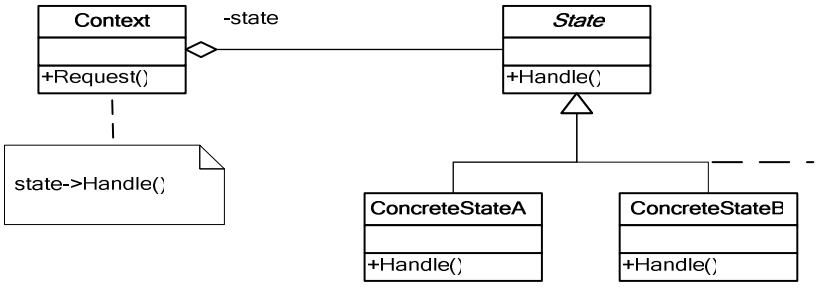

Рис. 1. Структура паттерна `State`

Введем некоторые пояснения.

`Context` – контекст. Предоставляет интерфейс для клиентов. Содержит экземпляр подкласса класса `State`, который определяет текущее состояние.

`State` – базовый класс, определяющий интерфейс.

Подклассы класса `State` – реализуют интерфейс класса `State` в каждом конкретном состоянии контекста `Context`.

Состояние контекста изменяется путем изменения переменной контекста из методов классов, являющихся наследниками класса `State`. Это приводит к тому, что классы состояний должны «знать» друг о друге. Кроме того, логика переходов из одного состояние в другое разбрасывается по коду по разным классам состояний.

2.2. Паттерн State Machine

В работе [6] авторами предлагается паттерн *`State Machine`*, устраняющий вышеуказанные недостатки. На **Ошибка! Источник ссылки не найден.** изображена его структура.

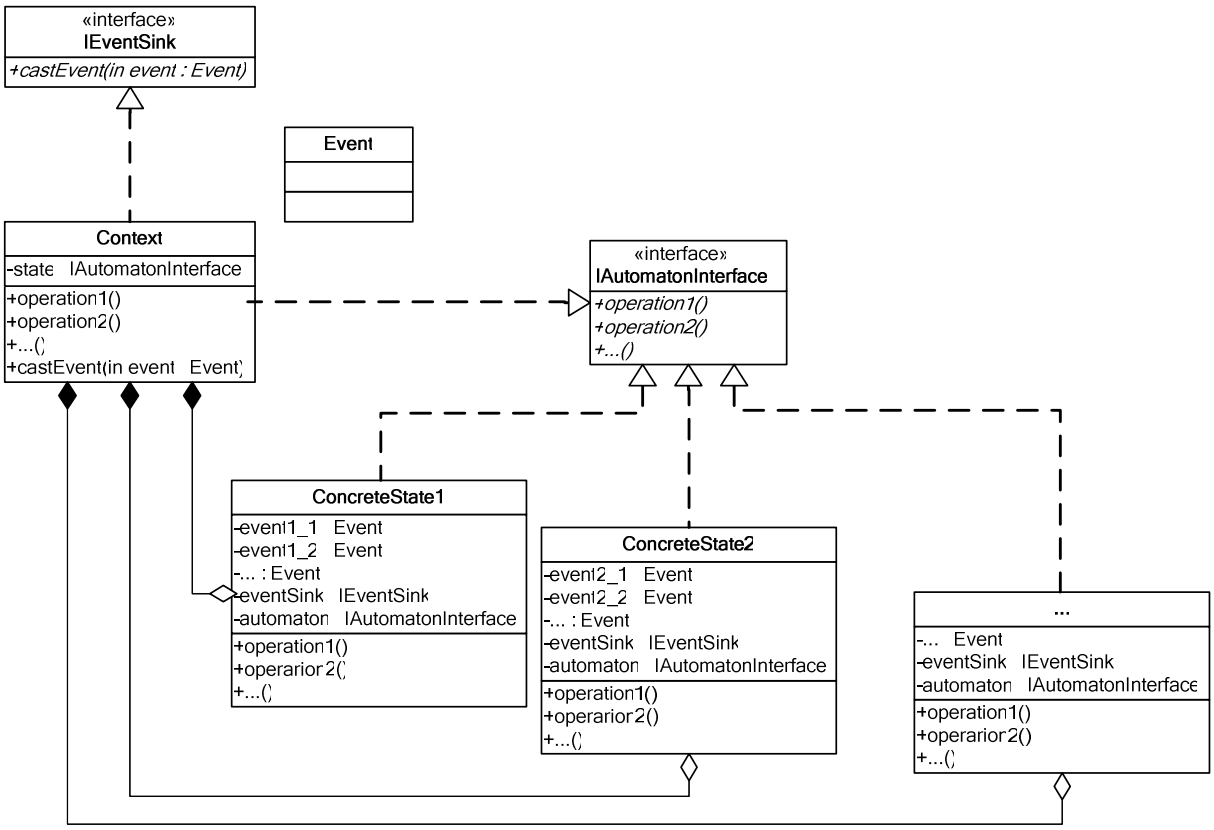

Рис. 2. Структура паттерна *State Machine*

Для того чтобы избавиться от зависимости классов-состояний между собой, в паттерне *State Machine* в классы-состояний были добавлены события.

События – это объекты, которые передаются состояниями контексту, и сообщают о необходимости смены состояния. Таким образом, вся логика переходов централизуется в классе-контекста. Этот класс на основе текущего состояния и пришедшего события определяет новое состояние.

В работе приведена реализация этого паттерна на языке *Java*.

Добавление классу-состояния события реализовывалось за счет добавления в этот класс члена-класса `Event`:

```
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");
…
}
```

Кроме того, любое состояние через базовый класс `StateBase` должно агрегировать интерфейс `IEventSink` для уведомления контекста о смене состояния:

```
public abstract class StateBase<AI> {
  protected final IEventSink eventSink;
…
}
```

Это приводит к необходимости специально разрабатывать классы-состояний с учетом изложенного выше и лишает разработчика возможности использовать в качестве классов-состояний другие классы, которые исходно не были предназначены для такого использования, например, классы стандартных библиотек.

Отсутствие в классе контекста членов-событий делает невозможным использование его в качестве класса-состояния, а, следовательно, и не возможным

использование автомата в качестве состояния другого автомата. Это значит, что невозможно, чтобы один автомат был вложен в другой.

3. РЕАЛИЗАЦИЯ ПАТТЕРНА *STATE* НА ЯЗЫКЕ *C#*

В этой работе предлагается реализация паттерна State на языке C#, которую будем называть `State#` или для краткости `S#`. Она снимает ограничения на классы состояния. На рис. 3 изображена структура реализации `State#`.

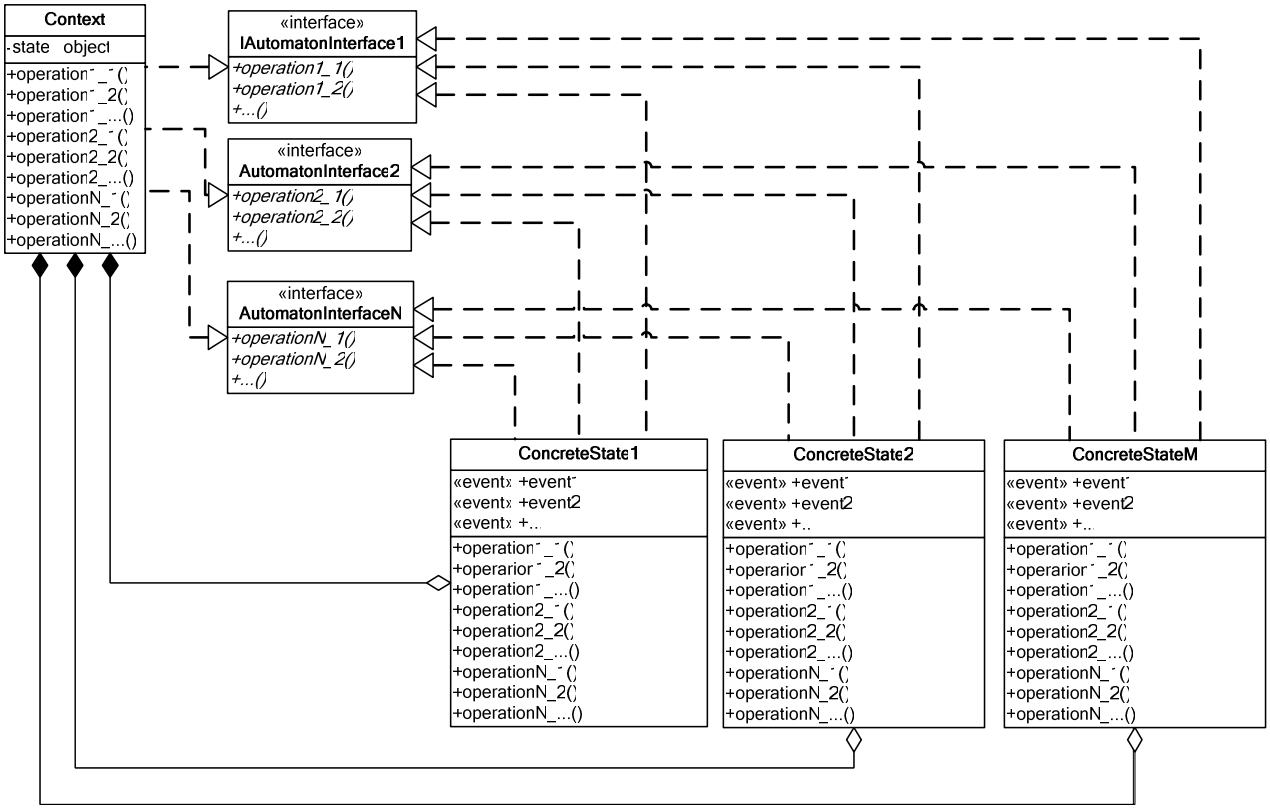

Рис. 3. Структура реализации `State#`

3.1. События

Для устранения недостатков, описанных в предыдущей главе, используем язык `C#` [8]. Этот язык содержит понятие `событие` на синтаксическом уровне.

События языка *C#* предлагается использовать в качестве событий, которые порождаются состояниями для извещения контекста.

```
public class State1 {
   public event EventHandler SomethingHappened;
…
}


public class Context {
   public Context() {
      state1.SomethingHappened += new EventHandler(state1_SomethingHappened);
      …
   }


   pulic void state1_SomethingHappened(object sender, EventArgs e) {
      CurrentState = state2;
   }
}
```

Таким образом, удается избавиться от использования объектов класса `Event`, и перейти к использованию синтаксически определенных в языке *C#* событий. События объявлены во многих библиотечных классах. Они являются стандартным механизмом оповещения о какой-либо ситуации. Это позволяет использовать библиотечные классы в качестве классов-состояний.

Также события могут присутствовать и в классе-контекста, а значит, возможно, использование класса-контекста в качестве класса-состояния для другого контекста. Это позволяет реализовать вложенные автоматы, когда какое-

либо состояние объемлющего автомата состоит из нескольких подсостояний вложенного:

```
public class Context1 {
  private State1 state1;
  private State2 state2;
  private State3 state3;
  …
}
public class Context2 {
  private State4 state1;
  private Context1 state2;
}
```

Здесь автомат `Context1` состоит из трех состояний: `State1`, `State2` и `State3`, а автомат `Context2` – из двух состояний: `State4` и `Context1`. Состояние `Context1`, так как, в свою очередь, является автоматом, состоит из трех подсостояний: `State1`, `State2`, `State3`.

3.2. Автоматный интерфейс

Автоматным интерфейсом назовем тот интерфейс, реализация методов которого зависит от состояния автомата.

Его методы реализуются в классе автомата делегированием вызова соответствующего метода у текущего состояния (рис. 4).

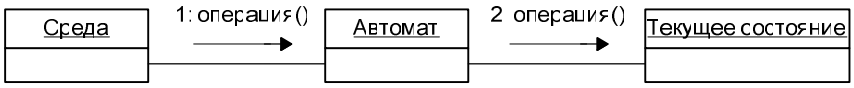

Рис. 4. Делегирование операций текущему состоянию автомата

Все классы состояний должны реализовывать все автоматные интерфейсы. Однако, это требование не распостраняется на неавтоматные интерфейсы. Класс состояния может и не реализовывать некий неавтоматный интерфейс, который реализуется автоматом.

В паттерне *State Machine* предполагалось, что автомат реализует один автоматный интерфейс. Это ограничение несколько искусственно. Предлагается использовать произвольное количество автоматных интерфейсов. Это может быть обосновано при использовании уже библиотечных интерфейсов. Единственное требование – каждый класс состояния автомата должен реализовывать все автоматные интерфейсы. При этом и класс контекста и классы состояний могут реализовывать произвольное количество неавтоматных интерфейсов.

3.3. Действия на переходах

Switch-технология [4] предлагает задавать действия, которые выполняются при переходе автомата из одного состояния в другое. Язык *State#* сохраняет эту возможность. Действия на переходе могут быть заданы в обработчике события перехода.

```
public class Context {

  public Context() {

    state1.SomethingHappened += new EventHandler(state1_SomethingHappened);

    …

  }


  public void state1_SomethingHappened(object sender, EventArgs e) {

    // Действия на переходе

    CurrentState = state2;

  }

}
```

Ниже на рис. 5 приводится диаграмма деятельности, иллюстрирующая переход автомата из одного состояния в другое.

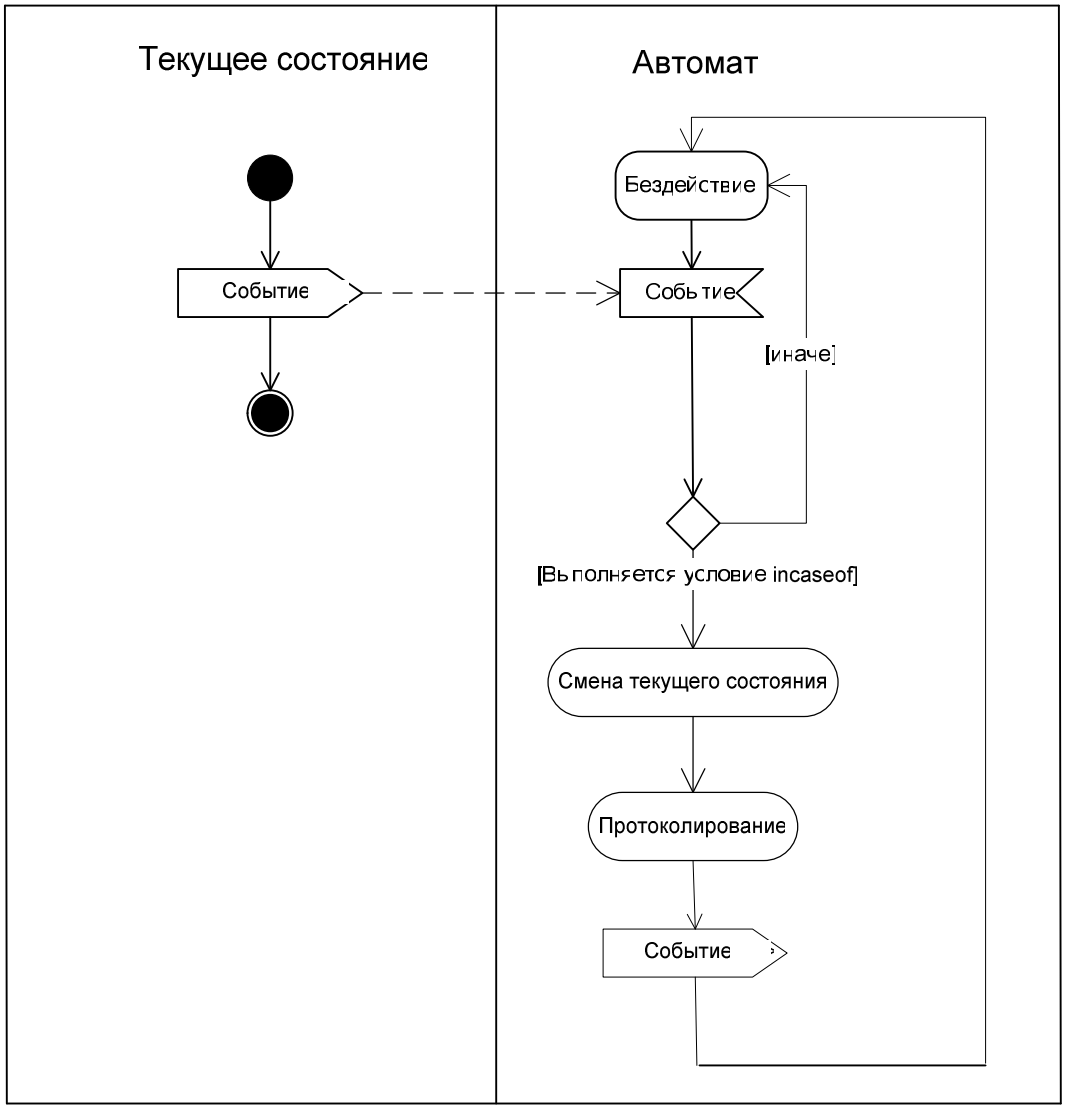

Рис. 5. Диаграмма деятельности при смене состояния

3.4. Сравнение реализации *State Machine* на языках *Java* и *C#*

Отсутствие встроенных в язык *Java* событий вынуждает к определению класса `Event` и добавлению в классы состояний экземпляров класса `Event`. Также класс состояния должен реализовать интерфейс `IEventSink` для сообщения контексту о смене состояния.

Встроенные в язык *C#* события избавляют от необходимости в событиях-объектах и реализации интерфейса.

В реализации паттерна *State Machine* на языке *Java* обработка событий происходила в одном методе `castEvent`, который был методом интерфейса `IEventSink`. В этом методе следующее состояние определяется либо с использованием хэш-таблиц, либо при помощи оператора *switch*. Первое решение не позволяет задавать действия на переходах и проверять дополнительные условия, а второе делает код метода громоздким.

Реализация на языке *C#* использует по одному методу для каждого перехода. Этот метод – обработчик события перехода для состояния. В метод можно добавить проверку дополнительных условий для перехода и действия на переходе (код, который должен выполняться при переходе).

4. ЯЗЫК *STATE#* (*S#*)

Для эффективной реализации подхода, изложенного в главе 3, предлагается расширение языка *C#*. Новый расширенный язык будет надмножеством языка *C#*. Из этого следует, что ему будут принадлежать и все программы, написанные на языке *C#*.

Введение в язык дополнительных синтаксических конструкций упрощает реализацию на нем шаблонных решений (паттернов).

Язык *C#* был разработан компанией *Microsoft* как язык для платформы *Microsoft .Net Framework*. *C#* – это объектно-ориентированный язык высокого уровня. Как и многие современные языки *C#* – это си-подобный язык. Однако для большей гибкости *C#* содержит такие дополнительные элементы как свойства, делегаты и события.

С помощью событий объект сообщает всем желающим о каком-нибудь изменении, произошедшим в нем. Событие вызывает код подписчиков, которые тем или иным образом реагируют на него.

Именно события позволяют устранить недостатки реализации паттерна, указанные в разд. 2.2. Любой класс, содержащий события и реализующий некий интерфейс, можно трактовать как состояние. Тогда другой класс (контекст), подписываясь на эти события, сможет сосредоточить логику переходов между состояниями.

4.1. Класс автомата

Для описания класса автомата в язык вводится новое ключевое слово *automaton* (автомат). Оно используется вместо ключевого слова *class*, которое применяется в языке *C#* и обозначает классы специального вида – классы автоматов.

```
public automaton MyAutomaton
```

Автомат может содержать все члены, которые может содержать класс языка *C#*: поля, методы, свойства, индексаторы, перегрузки операторов и события.

4.2. Наследование и реализация интерфейсов

Также как класс в языке *C#*, в языке *State#* автомат может наследовать один класс. В языке *State#* базовый класс может быть обычным классом или автоматом.

Однако класс автомата должен реализовывать как минимум один автоматный интерфейс и произвольное количество обычных интерфейсов. Перед автоматными интерфейсами ставится ключевое слово *auto*.

```
public automaton MyAutomaton : auto I1, I2, auto I3 {
…
}
```

В этом примере объявляется класс автомата `MyAutomaton`, который реализует автоматные интерфейсы `I1`, `I3` и обычный интерфейс `I2`.

Для всех состояний, включенных в автомат, классы этих состояний должны реализовывать все автоматные интерфейсы. При этом классы состояний могут реализовывать и другие интерфейсы дополнительно.

Для каждого метода из каждого автоматного интерфейса в классе автомата создается неявная реализация, которая вызывает соответствующий метод у текущего состояния. Поэтому явно реализовывать методы из автоматных интерфейсов в классе автомата нет необходимости.

4.3. Состояния

Состояниями в классе-контекста будут считаться члены-поля, объявленные с использованием ключевого слова *state* в начале объявления. Класс членов-состояний может быть любым, в том числе классом другого контекста.

```
public automaton MyAutomaton : auto I1, I2, auto I3 {
  state MyStateClass1 state1;
  …
}
```

В работе [7] для обозначения стартового состояния оно помещалось первым в класс контекста. Однако, такая семантика нетипична для таких объектно-ориентированных языков, как *Java* и *C#*. В них не важен порядок

объявления членов класса. Чтобы избежать нетипичной семантики, предлагается для стартового состояния использовать дополнительное ключевое слово *initial*:

```
public automaton MyAutomaton : auto I1, I2, auto I3 {
  state MyStateClass1 state1;
  initial state MyStateClass2 state2;
  state MyStateClass3 state3;
  …
}
```

После объявления состояния в фигурных скобках помещается информация о переходах из этого состояния.

Каждый переход начинается с ключевого слова *transition*. Затем идет ключевое слово *to*, за которым указывается имя члена-состояния, в которое осуществляется переход. Затем ключевое слово *when*, после которого имя публичного события, которое провоцирует переход и присутствует в классе состояния.

Затем идет необязательное ключевое слово *incaseof*, после которого в круглых скобках возможно указать булево выражение, являющееся условием перехода.

В конце можно указать действия, совершаемые на переходе. Для этого они указываются в фигурных скобках после ключевого слова *actions*:

```
public automaton MyAutomaton : auto I1, I2, auto I3 {
   state MyStateClass1 state1
{
   transition
   to state2
   when Happened1

   transition
   to state3
   when Happened2
};

   initial state MyStateClass2 state2
{
   transition
   to state1
   when SomethingHappened
   incaseof (someBooleanExpression)
   actions
   {
      Console.Writeln("!");
   }
};
   state MyStateClass3 state3;
   …
}
```

4.4. Автоматическое протоколирование

Одним из принципов SWITCH-технологии является протоколирование работы автомата. Протоколирование позволяет “отлавливать” ошибки. Поэтому в язык *State#* добавлена возможность автоматического протоколирования работы автомата.

Включение или выключение протоколирования не должно происходить в коде программы, а должно управляться извне.

Протоколирование включается в конфигурационном файле проекта. Он представляет собой *xml*-документ [12]. Протоколирование включается путем добавления в него секции `automatonLog` с атрибутом `Enabled` со значением `true`. Далее нужно включить протоколирование непосредственно для экземпляра автомата. Это делается добавлением в секцию `automatonLog` тэга `automatonLogInstance` со следующими параметрами:

- `Name` – имя протоколируемого автомата для ссылки на него;
- `Enabled` – `true` или `false` – включено или выключено протоколирование;
- `File` – путь к файлу, в который будет вестись протоколирование.

Формальное описание секции конфигурационного файла приведено ниже в формате `xsd` [13]:

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema id="AutomatonLogConfiguration"
targetNamespace=http://tempuri.org/AutomatonLogConfiguration.xsd
elementFormDefault="qualified"
xmlns=http://tempuri.org/AutomatonLogConfiguration.xsd
xmlns:mstns=http://tempuri.org/AutomatonLogConfiguration.xsd
xmlns:xs=http://www.w3.org/2001/XMLSchema
xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
  <xs:element name="automatonLog">
    <xs:complexType>
      <xs:attribute name="Enabled" type="xs:boolean" default="True"
      use="optional" />
        <xs:sequence>
          <xs:element name="automatonLogInstance">
            <xs:complexType>
              <xs:attribute name="Enabled" type="xs:boolean" default="True"
              use="optional" />
              <xs:attribute name="Name" type="xs:string" use="required" />
              <xs:attribute name="File" type="xs:string" use="required" />
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
</xs:schema>
```

Для того, чтобы связать экземпляр автомата с записью в конфигурационном файле ему необходимо задать имя. Для этого используется конструктор с именем в качестве первого параметра. Такие конструкторы будут создаваться

автоматически для каждого конструктора, написанного пользователем. Если автомат будет создан без передачи имени в конструкторе, то оно будет назначено автоматически в формате:

<имя_класса_автомата>_<номер экземпляра>

Приведем пример создания объекта-автомата:

```
MyAutomaton myAutomaton = new MyAutomaton(”MyAutomaton1”);
```

В результате в файле, имя которого задано в конфигурационном файле, для автомата с именем MyAutomaton1 будет сохранена вся информация о переходах автомата из одного состояния в другое.

4.5. Извещениие о смене состояния

Кроме автоматического протоколирования, пользователю предоставляется возможность гибко настраивать реакцию системы на смену состояния автомата. Для этого каждый класс контекста предоставляет событие StateChanged типа StateChangedEventHandler, которое происходит каждый раз при смене состояния:

```
delegate void StateChangedEventHandler (object sender, StateChangedEventArgs e)
```

В параметрах события sender представляет собой объект контекста. Параметр e содержит два открытых свойства OldState и NewState типа string, возвращающих имена членов состояний в классе контекста.

4.6. Сравнение языков *State Machine* и *State#*

Язык *State Machine* [7] является развитием языка *Java* и транслируется в этот язык.

Язык *State#* является развитием языка *C#* и транслируется в язык *C#*.

Язык *State#* не определяет синтаксиса для определения классов состояний, так как для реализации паттерна *State Machine* на языке *State#* в качестве классов состояний используются обычные классы.

Рассмотрим недостатки языка *State Machine*:

1. Класс автомата должен реализовывать ровно один интерфейс, который и считается автоматным;
2. Инициализация объектов состояний в конструкторе автомата описывается с помощью нового оператора `@=` вместо оператора `=;`
3. Стартовым считается состояние, описанное в автомате первым. Описание состояний в автомате похоже на описание членов класса. Наделять семантикой порядок определения членов несколько нетрадиционно.

Язык *State#* устраняет эти недостатки, а также предоставляет дополнительные возможности.

1. Задавать булевские условия перехода.
2. Задавать действия на переходах.
3. Генерировать события о смене состояния.

4. Настраиваемое протоколирование.

5. ТРАНСЛЯТОР ЯЗЫКА *S#*

Для написания транслятора с языка *S#* на язык *C#* использовался инструмент *ANTLR* (ANother Tool for Language Recognition) [11], предназначенный для создания распознавателей, компиляторов и трансляторов на основе описанной грамматики и действий на языке *C++*, *Java* или *C#*. В настоящей работе используется язык *C#*.

Распознавание языков стало распространенной задачей. В то время как компиляторы для традиционных языков, таких как *C* или *Java*, все еще создаются, их количество гораздо меньше количества компиляторов мини-языков, для которых разрабатываются распознаватели и трансляторы. Распознаватели создаются для форматов баз данных, графических файлов (*PostScript*, *AutoCAD*), текстовых описаний (*HTML*, *SGML*, *XML*). Инструмент *ANTLR* предназначен для построения такого рода распознавателей и трансляторов.

Этот инструмент работает с тремя видами грамматик:

- парсеров (синтаксических анализаторов);
- лексеров (лексических анализаторов);
- синтаксических деревьев.

Рассматриваемый инструмент использует *LL(k)-анализ* для всех видов грамматик. Поэтому форматы описания грамматик имеют много общего.

Лексические анализаторы работают с самим исходным текстом. Входом для них является поток символов, а выходом – поток токенов.

Входом для синтаксического анализатора является поток токенов, в процессе обхода которых совершаются семантические действия, и может создаваться синтаксическое дерево (*Abstract Syntax Tree*).

Входом для парсера синтаксических деревьев является синтаксическое дерево. В процессе его обхода могут совершаться семантические действия и создаваться другое дерево.

Транслятор *S#* использует все три типа парсеров. Лексический анализатор распознает токены на входе, синтаксический применяет к потоку токенов грамматику языка и создает синтаксическое дерево, а парсер дерева обходит дерево и выводит оттранслированный текст в выходной поток.

5.1. Грамматика языка *S#*

Грамматика языка *S#* основана на грамматике языка *C#* (*ECMA-334*). Ниже приводятся только изменения, внесенные в грамматику языка *C#*. Грамматика приведена в формате *ANTLR* [11].

```
typeDeclaration :
  classDeclaration     |
  automatonDeclaration |
  structDeclaration    |
  interfaceDeclaration |
  enumDeclaration      |
  delegateDeclaration;
```

```
automatonDeclaration :
   AUTOMATON identifier automatonBase automatonBody
   ( options { greedy = true; } : SEMI )?;



automatonBase :
   (
     COLON (AUTO type | type)
        (
          COMMA (AUTO type | type)
        )*
   )?;



automatonBody :
   OPEN_CURLY automatonMemberDeclarations CLOSE_CURLY;



automatonMemberDeclarations :
   (automatonMemberDeclaration)*;



automatonMemberDeclaration :
   attributes modifiers
   (destructorDeclaration | typeAutomatonMemberDeclaration);



typeAutomatonMemberDeclaration :
   commonMemberDeclaration |
   autoConstructorDeclaration  |
   autoStateDeclaration;



autoConstructorDeclaration :
   identifier OPEN_PAREN (formalParameterList)? CLOSE_PAREN
   (constructorInitializer)? automatonConstructorBody;
```

```
automatonConstructorBody :
  body;



autoStateDeclaration :
  (INITIAL)? STATE type stateDeclarator SEMI;



stateDeclarator :
  identifier
  (
    OPEN_CURLY (transition)+ CLOSE_CURLY
  )?
  (ASSIGN variableInitializer)?;



transition :
  TRANSITION
  TO identifier
  WHEN qualifiedIdentifier
  (INCASEOF OPEN_PAREN booleanExpression CLOSE_PAREN)?
  (ACTIONS block)?;
```

5.2. Семантика языка *S#*

Ниже приводится семантика языка *S#*.

Рассмотрим определения, которые могут встречаться в единице трансляции:

```
typeDeclaration :
  classDeclaration     |
  automatonDeclaration |
  structDeclaration    |
  interfaceDeclaration |
  enumDeclaration      |
  delegateDeclaration;
```

Определение автомата *automatonDeclaration* может встречаться там же, где может встречаться описание класса, структуры, интерфейса, перечисления или делегата.

Рассмотрим определение автомата:

```
automatonDeclaration :
  AUTOMATON identifier automatonBase automatonBody
  ( options { greedy = true; } : SEMI )?;
```

Это определение похоже на определение класса, но вместо ключевого слова *class* используется ключевое слово *automaton*. Определение автомата транслируется в определение класса с тем же именем. Транслятор автоматически добавляет следующие члены к определению класса:

- событие, подписка на которое информирует о смене состояния:

```
public event StateSharp.Runtime.StateChangedEventHandler StateChanged;
```

- член класса, содержащий ссылку на текущее состояние:

```
protected object currentState;
```

- ссылка на менеджер потока, в который осуществляется протоколирование:

```
protected StreamWriter streamWriter;
```

- имя автомата:

```
protected string name;
```

- счетчик автоматов для автоматического именования:

```
protected static int counter;
```

- внутренняя переменная, отражающая состояние инициализации:

```
private bool isInit;


protected void init() {
```

Метод `init` выставляет значение переменной `currentState` в ссылку на начальное состояние, инициализирует имя автомата, если оно не было задано в

конструкторе, инициализирует значение переменной `streamWriter` и производит подписку на события состояний.

Перейдем к определению базовых классов.

В качестве базового класса автомат может иметь один класс. Кроме того, автомат должен реализовывать как минимум один автоматный интерфейс и произвольное количество просто интерфейсов. Для того чтобы обозначить автоматный интерфейс будем использовать ключевое слово *auto* перед именем интерфейса. Для каждого автоматного интерфейса, который реализуется автоматом, транслятор автоматически генерирует имплементацию его методов. Она состоит в вызове одноименного метода у текущего состояния.

Грамматика описания базовых классов имеет вид:

```
automatonBase :
  (
    COLON (AUTO type | type)
      (
        COMMA (AUTO type | type)
      )*
  )?;
```

Перейдем к рассмотрению членов класса автомата. Ими могут быть те же объекты что и члены класса и состояния:

```
automatonMemberDeclaration :
  attributes modifiers
  (destructorDeclaration | typeAutomatonMemberDeclaration);



typeAutomatonMemberDeclaration :
  commonMemberDeclaration      |
  autoConstructorDeclaration   |
  autoStateDeclaration;
```

Рассмотрим определение конструктора автомата. Для каждого конструктора транслятор создает еще один конструктор, отличающийся от исходного наличием дополнительного параметра name типа *string*. С помощью вызова такого конструктора можно задать имя автомата. В противном случае имя будет сгенерировано автоматически в формате *<имя класса автомата>_<значение счетчика counter>*.

```
autoConstructorDeclaration :
  identifier OPEN_PAREN (formalParameterList)? CLOSE_PAREN
  (constructorInitializer)? automatonConstructorBody;
```

Все состояния транслируются в закрытые члены класса. Поэтому использование модификатора доступа при объявлении состояния является ошибкой. Автомат должен иметь ровно одно начальное состояние. Оно помечается ключевым словом *initial*.

```
autoStateDeclaration :
   (INITIAL)? STATE type stateDeclarator SEMI;
```

Переходы транслируются в подписку на события в методе `init` и в обработчик события. Обработчик события проверяет условие перехода (если оно было задано), изменяет значение переменной `currentState`, выполняет действия на переходе (если они были заданы), осуществляет протоколирование и посылает сообщение о смене состояния.

```
stateDeclarator :
   identifier
   (
     OPEN_CURLY (transition)+ CLOSE_CURLY
   )?
   (ASSIGN variableInitializer)?;



transition :
   TRANSITION
   TO identifier
   WHEN qualifiedIdentifier
   (INCASEOF OPEN_PAREN booleanExpression CLOSE_PAREN)?
   (ACTIONS block)?;
```

5.3. Структура решения

Решение состоит из пяти проектов (рис. 6).

1. `StateSharp.`
2. `StateSharp.Lexer.`

3. StateSharp.Parser.

4. StateSharp.Printer.

5. StateSharp.Runtime.

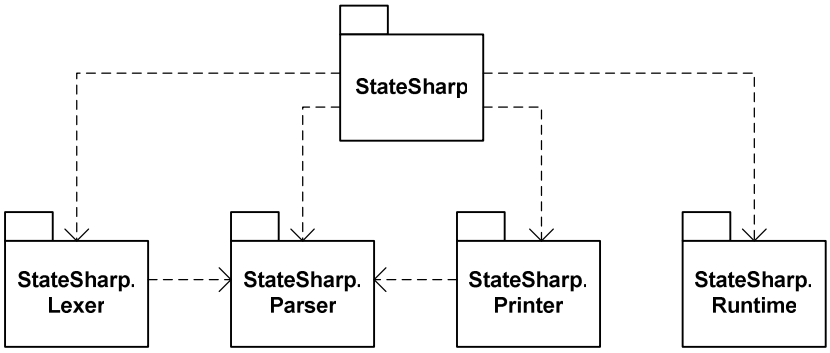

Рис. 6. Диаграмма пакетов решения

StateSharp.Lexer – проект, ответственный за лексический анализатор. Содержит главный класс лексического анализатора `CSharpLexer`.

StateSharp.Parser – проект, ответственный за синтаксический анализ и формирование синтаксического дерева. Содержит главный класс синтаксического анализатора `CSharpParser`, вспомогательные классы для представления узлов дерева, классы, выполняющие модификацию дерева и хранящие информацию об автомате.

StateSharp.Printer – проект, ответственный за вывод синтаксического дерева в виде *C#*-кода.

StateSharp.Runtime – проект, необходимый для работы скомпилированного автомата. Содержит классы для работы с конфигурационным

файлом проекта и классы, определяющие событие и его аргументы при смене состояния автомата.

StateSharp – интерфейс командной строки для запуска транслятора.

5.4. Запуск транслятора

Для использования транслятора необходимо наличие *Microsoft.Net Framework 1.1*, установленного на компьютере.

Транслирование вызывается путем вызова транслятора *StateSharp.exe* со следующими параметрами:

StateSharp.exe -print [-references <reference1>;<reference2>;...] <file name>

Результатом этого запуска будет вывод оттранслированного файла с именем *file name* в основной поток вывода.

Если необходимо вывести результат трансляции в другой файл, то можно использовать синтаксис перенаправления потоков:

StateSharp.exe -print [-references <reference1>;<reference2>;...] <file name> > <output file name>.

Результатом этого запуска будет вывод оттранслированного файла с именем *file name* в файл с именем *output file name*.

Ключ *-references* используется для задания сборок, в которых содержаться типы, используемые в качестве автоматных интерфейсов и классов состояний. После указания ключа *-references* имена сборок, перечисляются через точку с запятой. Сборки ищутся согласно стандартному механизму загрузки сборок в *Microsoft .Net Framework* [14].

5.5. Пример использования

Предположим, что требуется реализовать класс соединение, которое может находиться в соединенном и разъединенном состояниях. Его поведение описывается диаграммой состояний (рис. 8).

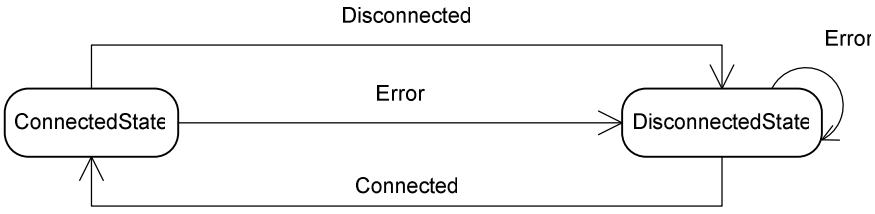

Рис. 7. Диаграмма состояний примера

Диаграмма классов, реализующая этот автомат на основе реализации *State#* (разд. 3), приведена на рис. 8.

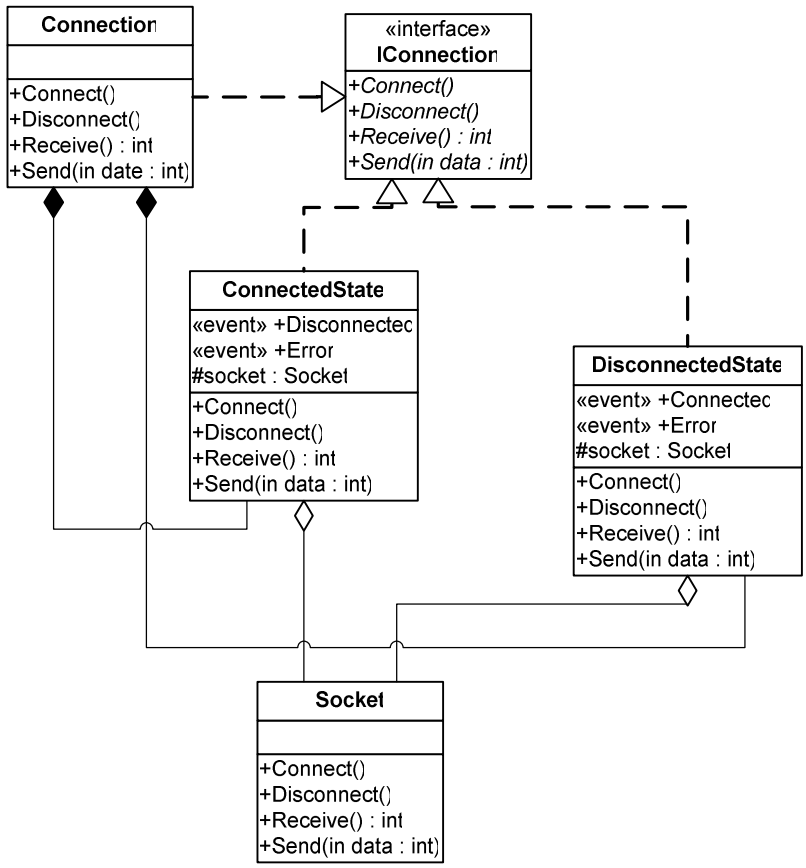

Рис. 8. Диаграмма классов примера

Реализуем эту диаграмму классов на языке *State#*:

- интерфейс *IConnection*, описывающий соединение:

```
public interface IConnection {

   void Connect();

   void Disconnect();

   int Receive();

   void Send(int value);

}
```

- класс-состояние *ConnectedState*, описывающий соединение в соединенном состоянии:

```
public class ConnectedState : IConnection {
  protected Socket socket;
  public event EventHandler Disconnected;
  public event EventHandler Error;

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

  #region IConnection Members
  public void Connect() {
  }

  public void Disconnect() {
    try {
      socket.Disconnect();
    } finally {
      if (Disconnected != null) {
        Disconnected();
      }
    }
  }
  public int Receive() {
    try {
      return socket.Receive();
    } catch (IOException ex) {
      if (Error != null) {
        Error();
      }
    }
```

```
  }
  public void Send(int value) {
    try {
      socket.Send(value);
    } catch (IOException ex) {
      if (Error != null) {
        Error();
      }
    }
  }
  #endregion
}
```

- класс-состояние *DisconnectedState*, описывающий соединение в разъединенном состоянии:

```
public class DisconnectedState : IConnection {
  protected Socket socket;
  public event EventHandler Connected;
  public event EventHandler Error;

  public DisconnectedState(Socket socket) {
    this.socket = socket;
  }
  #region IConnection Members
  public void Connect() {
    try {
      socket.Connect();
    } catch (IOException ex) {
      if (Error != null) {
```

```
            Error();
          }
        }
        if (ConnectedState != null) {
          ConnectedState();
        }
      }
      public void Disconnect() {
      }
      public int Receive() {
        throw new Exception("Connection is closed");
      }
      public void Send(int value) {
        throw new Exception("Connection is closed");
      }
      #endregion
}
```

- класс *Connection*, реализующий автомат, описывающий поведение соединения:

```
public automaton Connection : auto IConnection {
  initial state DisconnectedState disconnectedState {
    transition
    to connectedState
    when Connected

    transition
    to disconnectedState
    when Error
```

```
  };

  state ConnectedState connectedState {
    transition
    to disconnectedState
    when Disconnected

    transition
    to disconnectedState
    when Error
  };

  public Connection(Socket socket) {
    disconnectedState = new DisconnectedState(socket);
    connectedState    = new ConnectedState(socket);
  }
}
```

Класс *Socket*, отвечающий за организацию передачи данных, изображен на диаграмме классов. Однако код этого класса здесь не приводится, так как детали его реализации не важны.

Используя разработанный автором транслятор, который размещен на сайте http://is.ifmo.ru в разделе “Работы”, получим по приведенному выше коду на языке *State#* следующий код на языке *C#*:

```
public class Connection  : IConnection {
  DisconnectedState  disconnectedState;
  ConnectedState  connectesState;

  public Connection(Socket  socket) {
    disconnectedState = new DisconnectedState (socket);
    connectedState    = new ConnectedState (socket);
    init();
  }

  public Connection(string name, Socket  socket) {
    this.name = name;
    disconnectedState = new DisconnectedState (socket);
    connectedState    = new ConnectedState (socket);
    init();
  }

  public event StateSharp.Runtime.StateChangedEventHandler  StateChanged;
  protected object currentState;
  protected StreamWriter streamWriter;
  protected string name;
  protected static int counter;
  private bool isInit;

  protected void init() {
    if(isInit)  return;
    isInit = true;
    currentState = disconnectedState;
    if (name == null) {
      name = this.GetType().Name + "_" + counter;
```

```
        counter++;
      }
      StateSharp.Runtime.AutomatonLogConfig automatonLogConfig =
(StateSharp.Runtime.AutomatonLogConfig )
System.Configuration.ConfigurationSettings.GetConfig("automatonLog");

      if (automatonLogConfig.Contains(name)) {
        streamWriter = new StreamWriter (automatonLogConfig[name]);
      }
      disconnectedState.Error += new System.EventHandler (disconnectedState_Error);
      disconnectedState.Connected += new System.EventHandler
(disconnectedState_Connected);
      connectesState.Error += new System.EventHandler (connectesState_Error);
      connectesState.Disconnected += new System.EventHandler
(connectesState_Disconnected);
    }

    public void Send(System.Int32 value) {
      ((IConnection ) currentState).Send(value);
    }

    public System.Int32 Receive() {
      return ((IConnection ) currentState).Receive();
    }

    public void Disconnect() {
      ((IConnection ) currentState).Disconnect();
    }

    public void Connect() {
```

```
     ((IConnection ) currentState).Connect();
   }

   private void disconnectedState_Error (System.Object  sender, System.EventArgs e)
{
     if (Object.ReferenceEquals(currentState, disconnectedState)) {
       currentState = disconnectedState;
       if (streamWriter != null) {
         streamWriter.WriteLine(DateTime.Now + "Error Happened => Transition from
state disconnectedState to disconnectedState ");
       }
       if (StateChanged != null) {
         StateChanged(this, new StateSharp.Runtime.StateChangedEventArgs
(disconnectedState, disconnectedState));
       }
     }
   }

   private void disconnectedState_Connected (System.Object  sender,
System.EventArgs e) {
     if (Object.ReferenceEquals(currentState, disconnectedState)) {
       currentState = connectedState;
       if (streamWriter != null) {
         streamWriter.WriteLine(DateTime.Now + "Connected Happened => Transition
from state disconnectedState to connectedState ");
       }
       if (StateChanged != null) {
         StateChanged(this, new StateSharp.Runtime.StateChangedEventArgs
(disconnectedState, connectedState));
       }
```

```
    }
  }
  private void connectesState_Error (System.Object  sender, System.EventArgs e) {
    if (Object.ReferenceEquals(currentState, connectesState)) {
      currentState = disconnectedState;
      if (streamWriter != null) {
        streamWriter.WriteLine(DateTime.Now + "Error Happened => Transition from
state connectesState to disconnectedState ");
      }
      if (StateChanged != null) {
        StateChanged(this, new StateSharp.Runtime.StateChangedEventArgs
(connectesState, disconnectedState));
      }
    }
  }

  private void  connectesState_Disconnected (System.Object sender,
System.EventArgs e) {
    if (Object.ReferenceEquals(currentState, connectesState)) {
      currentState = disconnectedState;
      if (streamWriter != null) {
        streamWriter.WriteLine(DateTime.Now + "Disconnected Happened =>
Transition from state connectesState to disconnectedState ");
      }
      if (StateChanged != null) {
        StateChanged(this, new StateSharp.Runtime.StateChangedEventArgs
(connectesState, disconnectedState));
      }
    }
```

```
    }

}
```

ЗАКЛЮЧЕНИЕ

В настоящей работе рассмотрен паттерн проектирования *State Machine*, являющийся развитием паттерна *State*.

Предлагается реализация паттерна *State Machine* на языке *C#*, основанная на использовании событий этого языка как средства взаимодействия классов состояний и класса контекста. Это позволяет использовать стандартные, библиотечные классы в качестве классов состояний. Применение событий может существенно ускорить разработку, так как позволит использовать уже готовые решения.

В рамках настоящей работы предложено расширение языка *C#* (язык *State#*) для более удобной реализации этого паттерна. Предложен синтаксис описания автоматов и разработан транслятор с языка *State#* на язык *C#*.

Транслятор поддерживает использование протоколирования. Протоколирование может быть полезно как средство отладки и тестирования. Управление протоколированием производится в конфигурационном файле без перекомпиляции проекта. Кроме того, в автомат транслятором добавляется событие о смене состояния, которое может быть использовано из вне.

В качестве расширения данной работы может быть рассмотрено написание модуля для *Microsoft Visual Studio* для автоматической трансляции

программ на языке *State#*, написание транслятора с языка *C# 2.0* или *C# 3.0* и трансляция непосредственно в *Microsoft Intermediate Language*.

ЛИТЕРАТУРА

1. *P. Adamczyk.* The Antology of the Finite State Machine Design Patterns. // http://jerry.cs.uiuc.edu/~plop/plop2003/Paperes/Adamczyk-State-Machine.pdf
2. *P. Adamczyk.* Selected Patterns for Implementing Finite State Machines. // http://pinky.cs.uiic.edu/~padamczy/fsm
3. *Э.Гамма, Р.Хелм, Р.Джонсон, Дж. Влиссидес.* Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб.:Питер, 2001.
4. *А.А. Шалыто.* Switch-технология. Алгоритмизация и программирование задач логического управления. СПб.:Наука, 1998.
5. *А. Николаенко.* Static Finite State Machime. // RSDN Magazine. 2005. №3.
6. *Н.Н. Шамгунов, Г.А.Корнеев, А.А. Шалыто.* State Machine – новый паттерн объектно-ориентированного проектирования. // Информационно-управляющие системы. 2004. №5.
7. *Н.Н. Шамгунов, Г.А.Корнеев, А.А. Шалыто.* State Machine – расширение языка Java для эффективной реализации автоматов. // Информационно-управляющие системы. 2005. №1.
8. *Эндрю Троелсен.* C# и платформа .NET. СПб.: Питер, 2002.
9. *Джесс Либерти.* Программирование на C#. СПб.-Москва: Символ, 2003.

10. Microsoft Developer Network // http://msdn.microsoft.com

11. ANTLR // http://www.antlr.org

12. XML Specification // http://www.w3.org/XML/

13. XML Schema // http://www.w3.org/XML/Schema/

14. *Джеффри Рихтер*. Программирование на платформе Microsoft .Net Framework. СПб.: Питер, 2005.