Санкт-Петербургский государственный университет информационных технологий,
точной механики и оптики

Кафедра компьютерных технологий

П.Ю. Маврин

Реализация диаграмм состояний

Бакалаврская работа

Научные руководители: Корнеев Г.А., Шалыто А.А.

Санкт-Петербург
2006

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Часто при проектировании сталкиваются с ситуацией, когда поведение объекта определяется его некоторым текущим состоянием. Например, электронные наручные часы могут находиться в состояниях: «время», «дата», «секундомер» и т. д. [14].

Как правило, подобные конструкции естественно появляются при проектировании реактивных систем — событийно-управляемых систем, реагирующих на воздействия внешней среды. Существующие средства построения подобных систем: SWITCH-технология [8], паттерны проектирования [3, 10] или различные средства генерации кода [15, 17, 25] часто оказываются недостаточно мощными или недостаточно гибкими для построения систем с большим числом состояний и сложной их структурой. Другой проблемой является отсутствие общепринятой семантики таких систем.

В этой работе исследуются общие принципы построения систем с выделенными состояниями, поведение которых описано при помощи диаграмм состояний (Statechart), предложенных Дэвидом Харелом. Диаграммы состояний выбраны как наиболее гибкая, функциональная, универсальная и наглядная, на наш взгляд, форма представления поведения системы. Впервые эти диаграммы были предложены Харелом в 1987 году в статье [14]. Впоследствии эти диаграммы почти без изменений были включены в язык UML [2].

В главе 1 выполнен обзор существующих методов построения реактивных систем на основе диаграмм состояний и схожих конструкций.

В главе 2 предлагается новый метод построения таких систем. Метод включает описание структуры модели, семантики и паттерн

проектирования, позволяющий ее реализовать.

В главе 3 описана библиотека на языке *Java*, реализующая предложенный паттерн, и приведены примеры использования этой библиотеки.

ГЛАВА 1. ПОСТРОЕНИЕ РЕАКТИВНЫХ СИСТЕМ НА ОСНОВЕ КОНЕЧНЫХ АВТОМАТОВ

В этой главе рассмотрим основные способы построения реактивных систем на базе конечных автоматов.

В разд. 1.1 описаны некоторые семантики, используемые при реализации реактивных систем.

В разд. 1.2 и 1.3 описаны способы построения программ для реактивных систем. Эти способы разделены на две группы:

- программы для генерации кода;
- паттерны проектирования.

1.1. Семантики конечных автоматов

Семантика — основная часть любого языка. Это тем более верно для языке, описывающего поведение. Однако, по непонятным причинам, язык *UML* не предоставляет достаточно полного и четкого описания семантики диаграмм состояний [1, 2]. Отсутствие общепринятой семантики приводит к тому, что разные реализации одной и той же модели могут работать по-разному.

Например, в работе [11] приведен пример диаграммы состояний, которая интерпретируется по-разному при использовании различных семантик.

Рассмотрим подробнее особенности некоторых семантик, используемых при построении реактивных систем.

1.1.1. Семантика SWITCH-технологии

SWITCH-технология — это комплексная технология алгоритмизации и программирования задач логического управления [8], основной упор в которой делается на конечные автоматы, как форму

описания поведения программы. При таком подходе этой технологии просто необходимо было ввести формальную семантику, что и было сделано в работе [8]. Именно благодаря формальной семантике достигается изоморфизм графа переходов и программы. Эта технология успешно зарекомендовала себя при создании систем логического управления.

В дальнейшем SWITCH-технология стала применяться для построения разнообразных программ. На ее базе студентами и аспирантами СПбГУ ИТМО было создано около 100 проектов [6].

Однако, графы переходов, используемые в SWITCH-технологии, уступают по изобразительной мощности и гибкости диаграммам состояний. Подробнее SWITCH-технология рассмотрена в разд. 1.2.2.

1.1.2. Семантики *Statemate* и *Rhapsody*

Рассмотрим семантику, используемую в средстве *I-Logix Statemate* [19]. Это средство было создано специально для построения реактивных систем, поведение которых заданно диаграммой состояний. Поэтому и семантика, используемая в нем [16], также основана на диаграммах состояний. Основным недостатком этой семантики является недостаточно полное и точное описание. Из-за этого программист, использующий это средство для построения кода не может точно знать, как будет себя вести созданная им программа. Кроме того, эта семантика недостаточно практична. Например, в некоторых местах в ее описании фигурируют действия, которые должны происходить за нулевое время.

Семантика, используемая в программе *I-Logix Rhapsody* [15, 18] построена на базе семантики *Statemate*. В ней были исправлены некоторые неточности *Statemate*, например, требования о нулевом времени выполнения действий. Однако, проблема неполного и

неточного описания, все-таки, остается.

Кроме перечисленного, отметим, что программы *Statemate* и *Rhapsody* являются коммерческими продуктами, поэтому не только исходные коды, но даже и исполняемые файлы не предоставлены в свободном доступе. Это существенно затрудняет понимание идей их авторов.

1.2. Средства генерации кода

В настоящее время появилось достаточно много средств для работы с диаграммами *UML*. Во многих из них существует возможность автоматического построения кода программы по *UML*-диаграммам классов.

Однако программ, позволяющих генерировать код по диаграммам состояний, довольно мало. Отчасти это связано с отсутствием общепринятой семантики (см. разд. 1.1), отчасти — с техническими трудностями, возникающими при построении таких программ.

Рассмотрим некоторые из таких средств.

1.2.1. *Statemate* и *Rhapsody*

К сожалению, как уже говорилось, программы *I-Logic Statemate* [19] и *I-Logic Rhapsody* [18] являются коммерческими продуктами, что не позволяет их изучить. При этом отметим, что единственное, с чем можно ознакомиться по этим средствам — используемые семантики [16, 15, 18].

1.2.2. SWITCH-технология

Общий принцип очень прост и естественен. Занумеруем состояния числами от 1 до n и будем хранить в некоторой переменной `currentState` номер текущего состояния. Теперь, если требуется

совершить некоторое действие в зависимости от текущего состояния, напишем конструкцию **switch** (или несколько **if**), которая в зависимости от значения переменной `currentState` будет выполнять разные участки кода.

Такой подход описан в нескольких книгах, посвященных программированию автоматов и исполняемому *UML* [7, 23].

Дальнейшее развитие этот метод получил в SWITCH-технологии [8], о которой уже было сказано в разд. 1.1.1. Существуют средства для генерации кода в стиле SWITCH-технологии, например *Visio2Switch* [26].

Отметим, что подход, используемый в SWITCH-технологии, работает только при отсутствии вложенных состояний. Графы переходов, применяемые в этой технологии недостаточно мощны для компактного описания сложного поведения [24], даже несмотря на возможность использования вложенных в состояния автоматов.

1.2.3. *UniMod*

UniMod [25] — это плагин к среде разработки *Eclipse* [12], позволяющий задавать логику класса в виде диаграммы состояний *UML*. При этом он поддерживает все основные идеи SWITCH-технологии. Таким образом, *UniMod* адаптирует SWITCH-технологию к объектно-ориентированному программированию и *UML*. *UniMod* позволяет не только генерировать код, но и интерпретировать построенные модели.

К сожалению, *UniMod* не может использовать все возможности диаграмм состояний *UML*.

1.3. Паттерны проектирования

Существует множество паттернов проектирования, предназначенных для реализации поведения, изменяющегося в зависимо-

сти от состояния [3, 10]. В этом разделе рассмотрены некоторые из них.

1.3.1. Паттерн *State*

Паттерн *State*, описанный в работе [3] выглядит очень просто и элегантно, однако не приспособлен для реализации сложных конструкций диаграмм состояний, например, вложенных состояний. Существуют несколько его расширений [10], однако ни одно из них не подходит для корректной реализации сложного поведения, описанного диаграммой состояний.

1.3.2. Паттерн *State Machine*

Паттерн *State Machine*, описанный в работе [9] расширяет возможности паттерна *State*, однако в оригинальной форме он так же не приспособлен для реализации сложной логики, заданной диаграммой состояний.

1.4. Выводы по главе 1

1. Выполнен обзор существующих методов реализации автоматов.
2. Установлено, что известные семантики либо недостаточно формализованы, либо неприменимы на практике из-за сложности их формального описания.

ГЛАВА 2. ПАТТЕРН ПРОЕКТИРОВАНИЯ *O-STATE*

Как было показано в предыдущей главе, в настоящее время существуют определенные сложности при построении реактивных систем на базе конечных автоматов.

В этой главе делается попытка решить эти проблемы. При этом первоначально рассматривается структура используемой модели. Далее описывается ее семантика. В конце главы предлагается паттерн проектирования, позволяющий реализовать эту модель, используя описанную семантику.

2.1. Структура модели

Диаграммы состояний *UML* могут содержать довольно много различных элементов. Это, с одной стороны, помогает описывать логику поведения более компактным образом, но, с другой стороны, затрудняет описание семантики. В этом разделе будут выделены основные элементы диаграмм состояний, которые используются в настоящей работе. В разд. 2.2.4 приведено несколько примеров реализации других элементов диаграмм состояний *UML*, с помощью модели, описанной в этом разделе.

Перейдем к описанию основных элементов диаграмм состояний, которые применяются в создаваемой модели.

2.1.1. События

Из всех типов событий, представленных в языке *UML*, в предлагаемой модели сохранится только один тип — сигналы. Все остальные типы могут быть промоделированы с использованием сигналов. Будем разделять события (сигналы) на внешние и внутренние. Внутренние события посылаются системой самой себе в процессе обработки другого события. Все остальные события — внешние.

2.1.2. Состояния

В рассматриваемой модели сохранены только основные типы состояний, используемые в диаграммах состояний *UML*. Далее в разд. 2.2.4 будет показано, как в рамках предлагаемой модели реализовать различные псевдосостояния, представленные в *UML*.

2.1.2.1. Простое состояние

Простое состояние изображается в виде прямоугольника со скругленными краями, внутри которого написано название состояния.

Например, состояние «`Ожидание`» изображено на рис. 2.1.

Рис. 2.1. Простое состояние

2.1.2.2. OR-состояние

OR-состояние — это состояние, представляющее группу из нескольких подсостояний. Оно так же, как и простое состояние, изображается в виде прямоугольника со скругленными краями, внутри которого располагаются его подсостояния.

Например, составное состояние часов «`Включены`», содержащее подсостояния «`Время`» и «`Дата`», изображено на рис. 2.2.

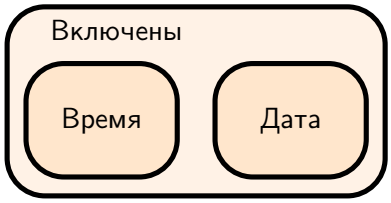

Рис. 2.2. OR-состояние

2.1.2.3. AND-состояние

AND-состояние — это состояние, объединяющее несколько OR-состояний, отвечающих за деятельности, которые не зависят друг от друга. Оно изображается в виде прямоугольника со скругленными краями, разделенного на части пунктирными линиями. Различные части соответствуют OR-состояниям, объединенным этим AND-состоянием. Например, совместное состояние часов «`Включены`», объединяющее составные состояния «`Изображение`» и «`СостояниеБудильника`», изображено на рис. 2.3.

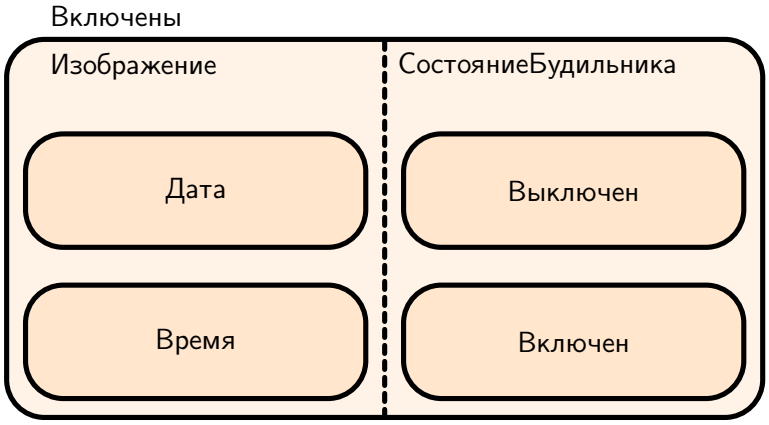

Рис. 2.3. AND-состояние

В предлагаемой модели будем считать, что есть состояние, содержащее все остальные состояния. Будем называть его базовым состоянием. Если в исходной диаграмме состояний такого состояния нет, то его можно получить, добавив OR-состояние, содержащее все состояния исходной диаграммы.

2.1.3. Дерево состояний

Структура состояний естественно представляется в виде дерева, в корне которого находится базовое состояние, а в листьях — простые состояния. Это дерево будем называть деревом состояний. На рис. 2.4 изображено дерево состояний для структуры, представленной на рис. 2.3.

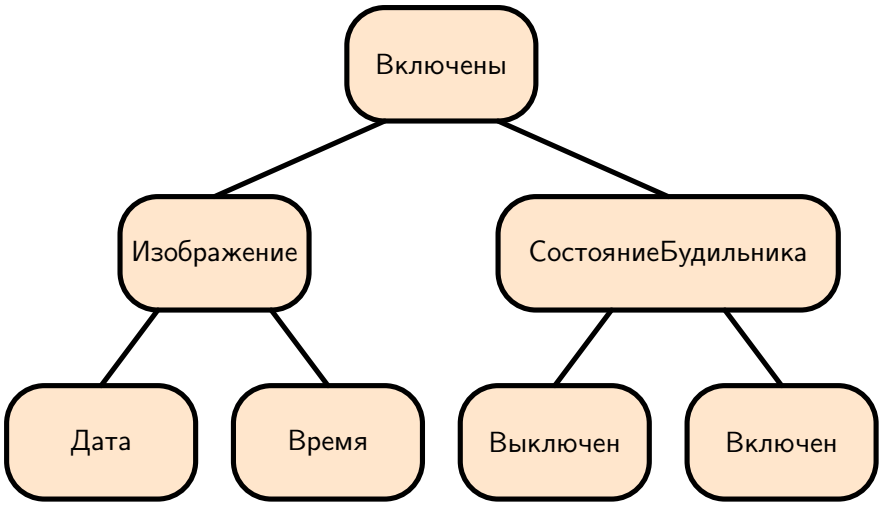

Рис. 2.4. Дерево состояний

2.1.4. Активные состояния. Дерево активных состояний

В каждый момент времени некоторые состояния являются активными. При этом соблюдаются следующие условия:

- базовое состояние активно;
- если OR-состояние активно, то активно ровно его подсостояний, иначе все его подсостояния неактивны;
- если AND-состояние активно, то активны все OR-состояния, объединенные им, иначе все они неактивны.

Таким образом, активные состояния также могут быть естественно представлены в виде дерева, в корне которого находится базовое состояние, а в листьях — простые состояния. Это дерево будем называть деревом активных состояний. Такое дерево как граф является подграфом дерева состояний. Кроме того отметим, что все множество активных состояний однозначно задается множеством активных простых состояний. Поэтому состояние системы иногда будет задаваться перечислением активных простых состояний.

2.1.5. Переходы

Переходы определяют правила изменения дерева активных состояний при появлении событий. Формально эти правила будут изложены в разд. 2.2. В предлагаемой модели будет только один тип перехода — простой переход.

Простой переход задается исходным состоянием, целевым состоянием, активирующим событием и защитным условием. Он изображается стрелкой от исходного к целевому состоянию. Он помечается следующим образом: `<активирующие событие>[[<защитное условие>]]`. Например, упрощенное описание поведения будильника может быть изображена как на рис. 2.5.

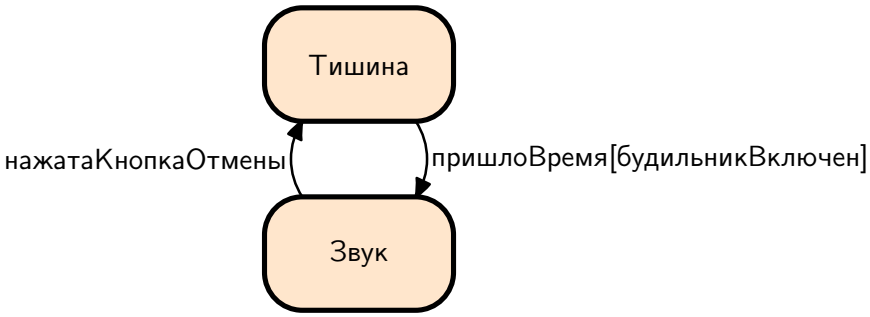

Рис. 2.5. Простые переходы

По этой схеме переход активности от состояния «Тишина» к состоянию «Звук» происходит при появлении события «пришлоВремя», если в этот момент истинно значение переменной «будильникВключен». Обратный переход выполняется при появлении события «нажатаКнопкаОтмены» независимо от значений переменной.

2.1.6. Действия

Действие — это набор простых операций, выполнение которых занимает время, много меньшее характерного времени системы. (В качестве характерного времени системы можно взять, на-

пример, среднюю величину интервала времени между событиями). Ограничение на время выполнения действия связано с тем, что при создании модели предполагается, что обработкой события не могут заниматься несколько потоков одновременно. Если события будут приходить быстрее, чем будут успевать обрабатываться, возможен бесконечный рост очереди сообщений.

2.1.6.1. Действия при входе и выходе

Действие при входе в состояние — это действие, совершаемое системой в момент, когда это состояние становится активным. Аналогично, действие при выходе из состояния, — это действие, совершаемое системой в момент, когда это состояние перестает быть активным.

На схеме действия при входе и выходе обозначаются надписями: **entry/**<действие при входе> и **exit/**<действие при выходе> соответственно.

2.1.6.2. Действие при переходе

Действие при переходе — это действие, которое выполняется при осуществлении перехода. Оно выполняется после действий при выходе и до действий при входе в состояние. Подробнее процесс выполнения перехода описан в разд. 2.2.

На диаграмме действие при переходе обозначается добавлением к пометке перехода строки **/**<действие при переходе>.

2.2. Семантика

Формальное описание семантики модели — одна из основных задач этой работы.

2.2.1. Выполнение перехода

Переход в предлагаемой модели выполняется по следующему алгоритму:

Алгоритм 1. Выполнение перехода
Дано: Transition t — переход, который требуется выполнить.
Надо: Выполнить переход.

```
1:    void makeTransition(Transition t) {
2:        State b = a.target;
3:        List<State> l = new ArrayList<State>();
4:        while (!b.isActive()) {
5:            l.add(b);
6:            b = b.parent;
7:        }
8:        removeBranch(b.children[0]);
9:        t.action.run();
10:       addChainToTree(l, b);
11:       fillBranch(B.children[0]);
12:       startNewStates(B.children[0]);
13:   }
```

Рассмотрим этот алгоритм подробнее. Разделим выполнение перехода на четыре этапа (рис. 2.6). Пусть осуществляется переход в состояние A (рис. 2.6, а).

Этап 1 (строки 2—7). Пойдем от состояния A вверх по дереву состояний до первого активного состояния. Пусть это состояние B (рис. 2.6, б).

Этап 2 (строка 8). Заметим, что состояние B обязано являться составным. Поэтому у него есть ровно один ребенок в дереве активных состояний. Отрежем всю ветку, соответствующую этому ребенку, выполнив предварительно действие при выходе для каждого состояния этой ветки. Используем для этого следующий алгоритм:

Алгоритм 2. Удаление ветки дерева состояний
Дано: State a — состояние, удаляемое из дерева текущих состояний.
Надо: Удалить состояние и все его подсостояния, предварительно вызвав действие при выходе, от детей к корню.

```
1:    void removeBranch(State a) {
2:        for (State b : a.children) {
3:            removeBranch(b);
4:        }
5:        a.exit();
6:        a.parent.removeChild(a);
7:    }
```

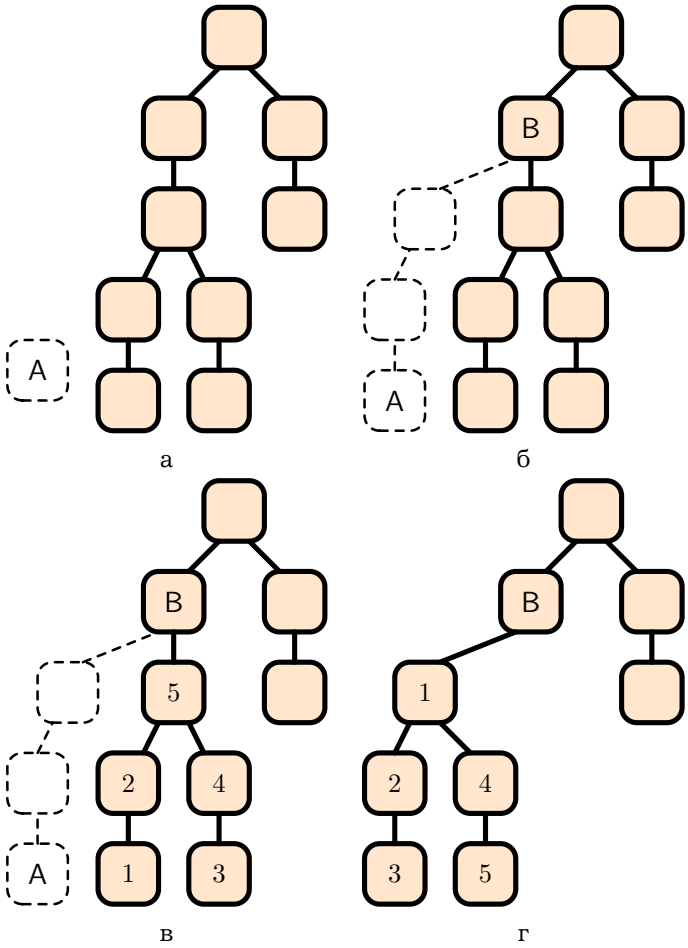

Рис. 2.6. Выполнение перехода

Отметим, что действие при выходе выполняется от детей к корню — для подсостояний оно всегда выполняется раньше, чем для надсостояний. Однако точный порядок специфицирован не будет. Один из возможных порядков выполнения действий при выходе отмечен на рис. 2.6, в цифрами от 1 до 5.

Этап 3 (строка 9). Выполним действие при переходе.

Этап 4 (строки 10—12). Подвесим новую ветку к состоянию B, дополним ее состояниями так, чтобы выполнялись условия для активных состояний. Используем для этого следующий алгоритм:

Алгоритм 3. Дополнение ветки дерева недостающими состояниями
Дано: State a — корень заполняемой ветки.
Надо: Дополнить ветку недостающими состояниями, чтобы получить корректное дерево активных состояний.

```
1:    void fillBranch(State a) {
2:        switch (a.stateType) {
3:        case OR_STATE :
4:            if (a.children.length == 0) {
5:                a.addChild(a.initial);
6:            }
7:            break;
8:        case AND_STATE :
9:            for (State b : a.parts) {
10:               if (!b.isActive()) {
11:                   a.addChild(b);
12:               }
13:           }
14:       }
15:       for (State b : a.children) {
16:           fillBranch(b);
17:       }
18:   }
```

Выполним для всех новых состояний действие при входе. Отметим, что такое действие, в отличие от действия при выходе, выполняется от корня к детям. Используем следующий алгоритм:

Алгоритм 4. Выполнение действия при входе для новых состояний
Дано: State a — корень новой ветки дерева активных состояний.
Надо: Выполнить действие при входе для новых состояний, от корня к детям.

```
1:    void startNewStates(State a) {
2:        a.start();
3:        for (State b : a.children) {
4:            startNewStates(b);
5:        }
6:    }
```

Точный порядок выполнения действий при входе также не будет специфицирован. Один из возможных порядков отмечен на рис. 2.6, г цифрами от 1 до 5.

2.2.2. Обработка события

Будем считать, что система обрабатывает внешние события по одному, причем за время обработки новые внешние события не поступают. Будем действовать следующим образом: делегируем событие всем активным состояниям, по возможности выполним ак-

тивированные при этом переходы. Далее проделаем то же самое со всеми полученными в процессе обработки внутренними событиями. Будем действовать так, пока система не придет в устойчивое состояние. Эта идея реализована следующим алгоритмом:

Алгоритм 5. Обработка события
Дано: Event *e* — пришедшее событие.
Надо: Обработать событие.

```
1:      Queue<Event> eq = new LinkedList<Event>();
2:      Queue<Transition> tq = new LinkedList<Transition>();
3:      void processEvent(Event e) {
4:          offerEvent(e);
5:          while (!eq.isEmpty()) {
6:              delegateEventToStates(eq.remove(), activeStatesTreeRoot);
7:              while (!tq.isEmpty()) {
8:                  Transition t = tq.remove();
9:                  if (t.source.isActive()) {
10:                     performTransition(t);
11:                 }
12:             }
13:         }
14:     }
15:     void offerEvent(Event e) {
16:         eq.offer(e);
17:     }
18:     void offerTransition(Transition t) {
19:         tq.offer(t);
20:     }
```

Алгоритм работает следующим образом. Формируются очереди событий и переходов (строки 1, 2). Далее, пока очередь не опустеет, приходящие события делегируются активным состояниям (строки 5—13). Если в процессе обработки события активируется переход, то он добавляется в очередь вызовом метода `offerTransition`. Если было активировано внутреннее событие, то оно добавляется в очередь вызовом метода `offerEvent`.

Алгоритм делегирования события активным состояниям может быть различным в разных системах. Например, в некоторых системах может быть удобно обрабатывать события от корня к детям, в других — от детей к корню, в третьих — событие передается состоянию только если ни одно из его подсостояний не обработало его [24].

Для примера рассмотрим алгоритм делегирования состояний, обрабатывающий состояния от детей к корню:

Алгоритм 6. Обработка события
Дано: Event e — обрабатываемое событие. State a — корень ветки дерева активных состояний, которой требуется передать на обработку пришедшее событие.
Надо: Передать пришедшее событие всем состояниям ветки от детей к корню.

```
1:  void delegateEventToStates(Event e, State a) {
2:      for (State b : a.children) {
3:          delegateEventToStates(e, b);
4:      }
5:      a.processEvent(e);
6:  }
```

2.2.3. Классификация систем по степени детерминированности их поведения

Заметим, что алгоритм, предложенный для обработки события, также не указывает точный порядок обработки событий. Поэтому порядок выполнения переходов не однозначен. Кроме того, алгоритм выполнения перехода не указывает, в каком порядке выполнять действия при входе и выходе. Это может стать причиной недетерминированности поведения системы.

Разобьем системы на три класса в зависимости от степени детерминированности их поведения.

2.2.3.1. Сильно детерминированные системы

Сильно детерминированными будем называть системы, при работе которых невозможны ситуации, когда у предложенного алгоритма обработки события будет выбор.

Например, сильно детерминированной является любая система, диаграмма состояний которой представляет собой простой автомат (все состояния — простые), в котором из каждого состояния по каждому событию существует не более одного перехода (рис. 2.5).

К сожалению, класс сильно детерминированных систем очень узок. Поэтому введем следующий класс.

2.2.3.2. Слабо детерминированные системы

Слабо детерминированными или детерминированными будем называть системы, при работе которых, независимо от того, какой недетерминированный выбор сделает алгоритм, общий эффект от обработки события окажется одним и тем же. Здесь под общим эффектом понимается изменение дерева активных состояний и изменение переменных окружения.

Например, система на рис. 2.7 является детерминированной, но не сильно детерминированной. Действительно, если система находится в состоянии (D, F) и пришло событие e, то активируются переходы (D→E) и (F→G). Они могут быть выполнены в любом порядке. Однако независимо от порядка их выполнения, система переходит в состояние (E, G). Поэтому она является детерминированной.

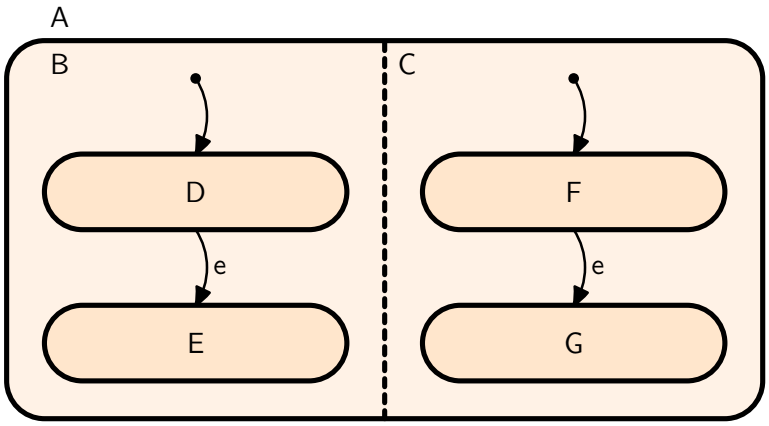

Рис. 2.7. Детерминированная система

Если на переходах в схеме, изображенной на рис. 2.7, выполняются действия z_1 и z_2 (рис. 2.8), то ее детерминированность существенно зависит от типа действий.

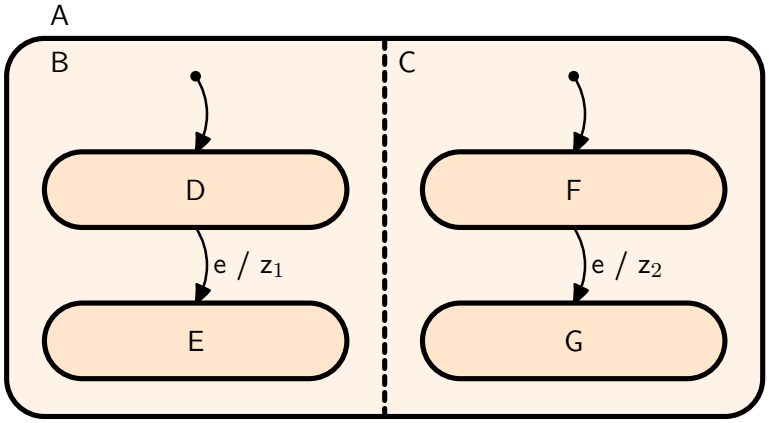

Рис. 2.8. Детерминированность системы зависит от действий

Например, если действие $z_1 : a = 0$, а действие $z_2 : a = 1$, то такая система не является детерминированной, поскольку общий эффект действий зависит от порядка их выполнения.

Если же действие $z1 : a = 0$, а действие $z_2 : b = 0$, то такая система детерминирована, так как при любом порядке выполнения действий общий эффект остается одним и тем же: $a = 0, b = 0$.

Видимо, большинство асинхронных реактивных систем могут быть описаны в предложенной модели таким образом, чтобы быть детерминированными в данном смысле.

2.2.3.3. Недетерминированные системы

Недетерминированные системы часто возникают в синхронных системах — в системах, которые опрашивают внешние переменные с некоторым интервалом времени, и при определенных значениях этих переменных делают переходы и выполняют действия. В предложенной модели такие системы описываются с помощью введения одного события (назовем его `t`), которое будет посылаться системе на обработку через определенные интервалы времени.

Проблемы недетерминированности, возникающие в синхронных системах, связаны с тем, что за интервал времени сразу несколько внешних переменных могут изменить значение.

Например, если на бомбе с часовым заводом (диаграмма состояний которой изображена на рис. 2.9), находящейся в состоянии «Детонация», кнопка «Отмена» будет нажата в тот момент, когда закончится время, то в зависимости от того, какой переход выполнится первым, эффект может быть различным.

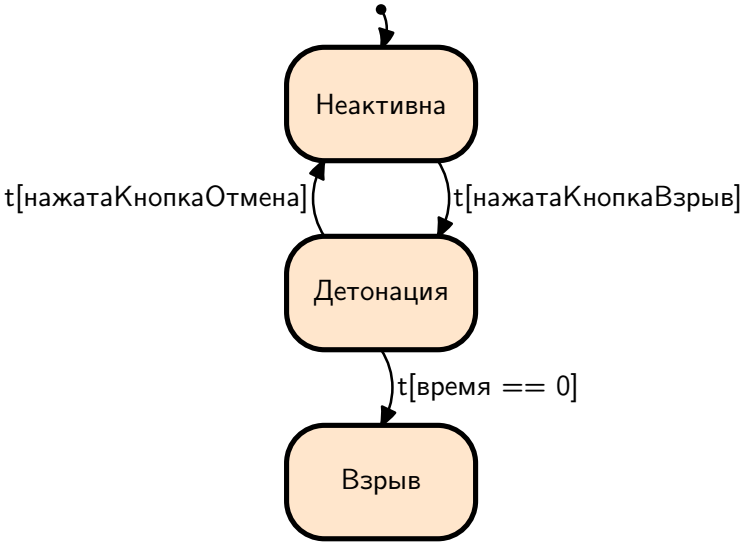

Рис. 2.9. Пример недетерминированной системы

На практике, как правило, такие системы либо приводят к детерминированному виду (рис. 2.10), либо вообще оставляют это без внимания, считая, что при нескольких возможных вариантах, все они являются удовлетворительными. При этом принятое решение зависит от реализации.

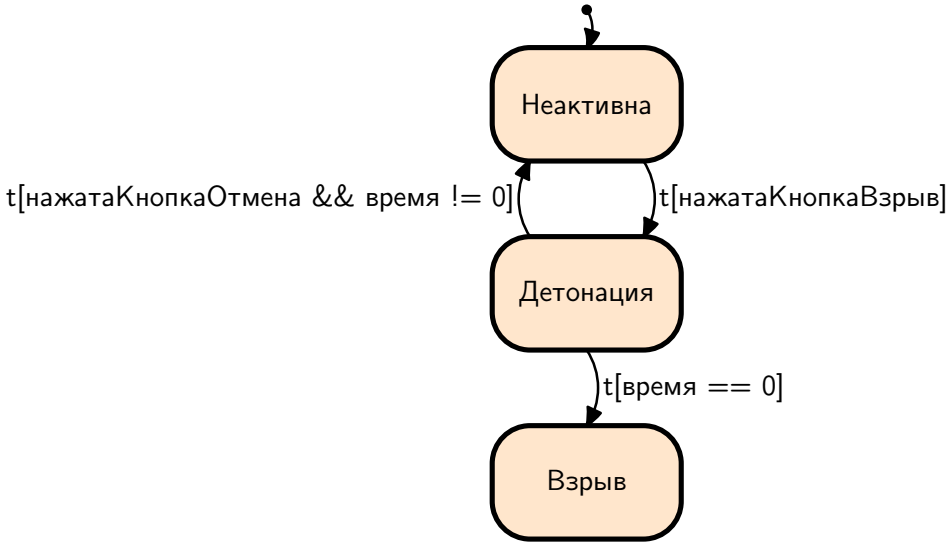

Рис. 2.10. Детерминированная версия системы

2.2.4. Примеры реализации других элементов диаграмм состояний *UML* на базе описанной модели

Как отмечалось выше, предлагаемая модель содержит лишь основные элементы диаграмм состояний *UML*. В этом разделе рассмотрим примеры реализации некоторых других элементов диаграмм состояний *UML* при помощи описанной модели.

2.2.4.1. Условное псевдосостояние

Условное псевдосостояние — это псевдосостояние, осуществляющее один из переходов в зависимости от выполнения условий (рис. 2.11).

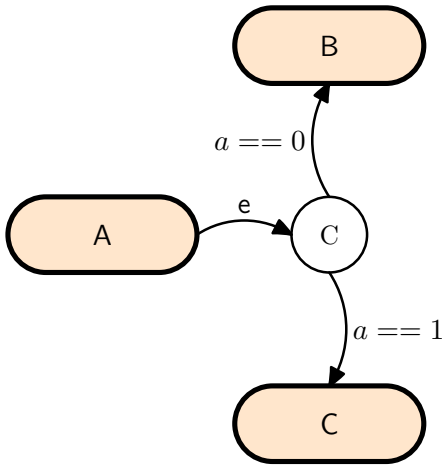

Рис. 2.11. Пример диаграммы состояний, содержащей условное псевдосостояние

В предлагаемой модели то же поведение можно реализовать так, как показано на рис. 2.12.

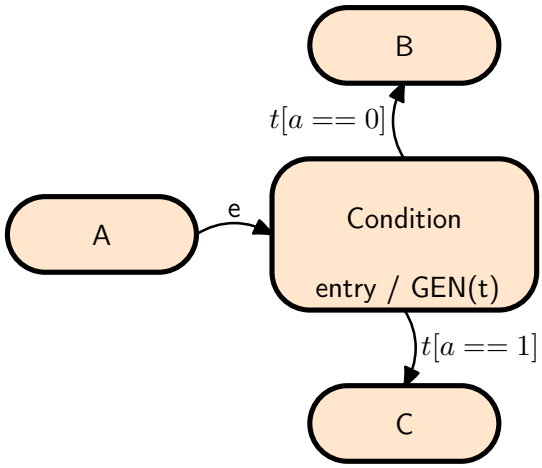

Рис. 2.12. Пример реализации условного псевдосостояния при помощи описанной модели

2.2.4.2. Вилка

Вилка — это сложный переход с несколькими целевыми состояниями (рис. 2.13).

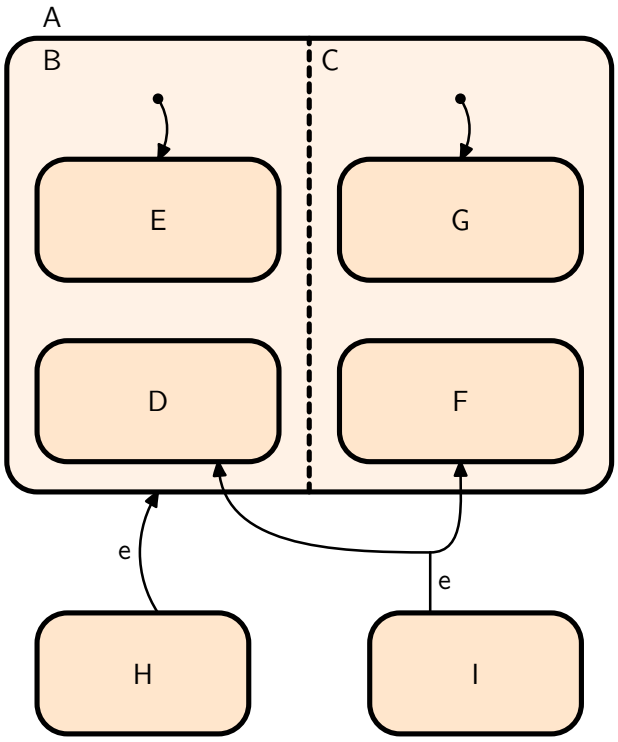

Рис. 2.13. Пример диаграммы состояний, содержащей вилку

В предлагаемой модели то же поведение можно реализовать так, как показано на рис. 2.14.

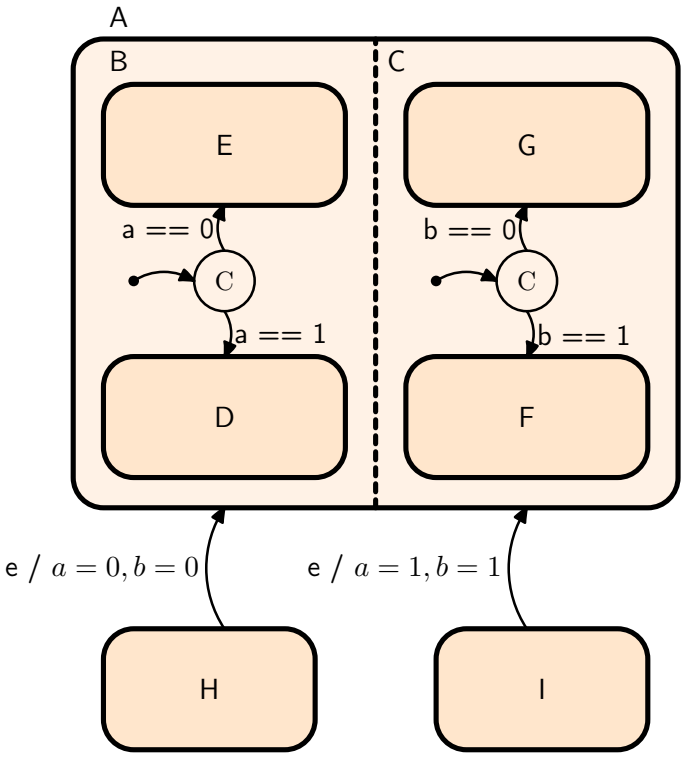

Рис. 2.14. Пример реализации вилки при помощи описанной модели

На основе изложенного материала опубликована следующая работа [5]

2.3. Паттерн проектирования *O-State*

Наиболее известной реализацией объекта, изменяющего поведение в зависимости от состояния, является паттерн *State*. Паттерн *State* не может использоваться напрямую для реализации предложенной модели, так как он не предусматривает сложных структур, используемых в ней. Паттерн *State* может быть применен для реализации предложенной модели в том случае, если все состояния — простые. В этом случае у системы всегда ровно одно активное состояние. Это существенно упрощает реализацию, но в то же время существенно сокращает возможности модели.

В настоящей работе предлагается новый паттерн, являющийся, в некотором смысле, расширением паттерна *State*, способным реализовывать конструкции, содержащие OR- и AND-состояния.

2.3.1. Назначение

Паттерн *O-State* предназначен для создания объектов, поведение которых варьируется в зависимости от состояния и задано при помощи модели, описанной в разд. 2.1.

2.3.2. Применимость

Паттерн *O-State* может быть использован в следующих случаях:

- когда поведение объекта зависит от его состояния. При этом объект может находиться в нескольких состояниях одновременно, а правила переходов могут быть описаны с помощью модели, предложенной в разд. 2.1;

- когда использование паттерна *State* затруднено или невозможно из-за сложной структуры состояний.

2.3.3. Структура

Диаграмма классов, описывающая структуру паттерна, изображена на рис. 2.15.

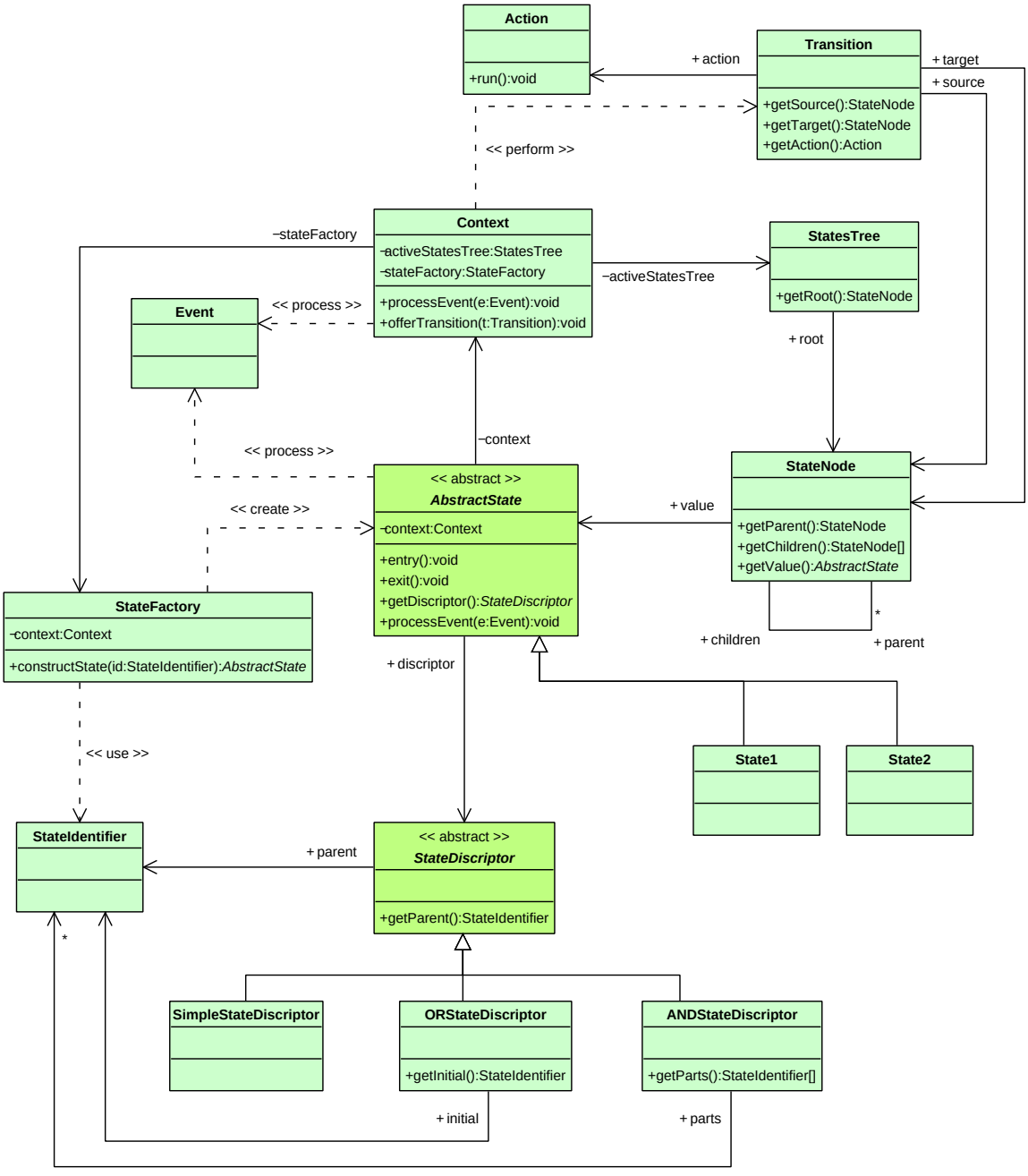

Рис. 2.15. Общая структура паттерна *O-State*

2.3.4. Участники

- **Context** — контекст. С ним работают клиенты, посылая на обработку события с помощью вызовов метода `processEvent`, и состояния, активирующие переход с помощью вызовов метода `offerTransition`;
- **Event** — событие. Класс является предком для всех классов событий, которые могут произойти в системе;
- **AbstractState** — абстрактное состояние. Абстрактный класс, являющийся предком для всех классов, задающих состояния объекта;
- **StateIdentifier** — идентификатор состояния. Экземпляры этого класса идентифицируют состояния объекта;
- **StateDiscriptor** — описание состояния. Этот класс и его потомки задают описание положения состояния относительно других состояний;
- **SimpleStateDiscriptor** — описание простого состояния;
- **ORStateDiscriptor** — описание OR-состояния;
- **ANDStateDiscriptor** — описание AND-состояния;
- **StateFactory** — фабрика состояний. Этот класс отвечает за построение новых состояний по их идентификаторам (используется паттерн *Factory* [3]);
- **Transition** — Переход. Инициируется состоянием во время обработки события, обрабатывается контекстом;
- **StatesTree**, **StateNode** — служебные классы, необходимые для построения дерева активных состояний.

2.3.5. Отношения

- класс **Context** делегирует активным состояниям события, полученные от клиентов, вызывающих метод

`processEvent`. При этом класс **Context** действует в соответствии с алгоритмом обработки события, описанным в разд. 2.2.2;

- если состояние в процессе обработки события должно измениться, оно вызывает метод `offerTransition` у объекта класса **Context**;
- при смене состояния, объект класса **Context** использует экземпляр класса **StateFactory** для построения новых состояний;
- класс **StateFactory** строит объекты **AbstractState** по их идентификаторам **StateIdentifier**.

2.3.6. Результаты

- предложенный паттерн обладает всеми достоинствами паттерна *State* (разделенное по классам поведение, легкость масштабирования, явное выделение состояний и переходов);
- паттерн обеспечивает возможность реализации систем со сложным поведением, которые содержат OR- и AND-состояния. Проектирование таких систем без этого паттерна сложно. Это связано с тем, что, например, при выполнении одного перехода могут быть активированы и дезактивированы сразу несколько состояний, для которых необходимо выполнить действия при входе или при выходе. Таким образом, выполнение перехода в этих случаях представляет собой достаточно сложный процесс. Описание этого процесса обеспечивается разработанной семантикой, а использование предложенного паттерна при проектировании позволяет в дальнейшем реализовать этот процесс корректно.

2.3.7. Реализация

Для корректной реализации следует учесть некоторые особенности данного паттерна:

- необходимо, чтобы построенная система четко следовала алгоритму обработки события, предложенному в разд. 2.2.2. Это осложнено тем, что реализация этого алгоритма в объекте класса **Context** не монолитна, так как в процессе он должен периодически отдавать управление активным состояниям;
- правильное построение дерева состояний, позволяющее эффективно реализовать алгоритм выполнения перехода, также может представлять некоторые сложности;
- объект класса **StateFactory** должен уметь создавать объекты всех существующих классов, задающих состояния. Таким образом, он должен знать обо всех существующих состояниях. Это несколько затрудняет масштабирование системы. При реализации паттерна на языке *Java*, содержащем механизм *Reflection*, можно вместо класса **StateIdentifier** использовать класс **java.lang.Class**. В этом случае процесс построения объекта состояния можно несколько унифицировать, устранив необходимость для класса **StateFactory** иметь информацию обо всех состояниях, в которых может находиться система.

Эти особенности учтены при реализации библиотеки, описанной в следующем разделе.

2.4. Выводы по главе 2

1. Разработана формальная семантика диаграмм состояний, удобная для практического применения.

2. Разработан паттерн программирования *O-State*, помогающий реализовать диаграммы состояний, используя предложенную семантику.
3. Указаны проблемы, возникающие при реализации предложенного паттерна. Поэтому будет разработана библиотека, содержащая реализацию на языке *Java* основных классов паттерна.

ГЛАВА 3. РЕАЛИЗАЦИЯ ПРЕДЛОЖЕННОГО МЕТОДА

В этой главе описаны примеры использования метода, предложенного в главе 2.

3.1. Библиотека *O-State*

Как было показано в разд. 2.3.7, реализация паттерна *O-State* представляет некоторые технические сложности. Для упрощения реализации была разработана библиотека *O-State* на языке *Java* [13]. Для большего удобства при построении программ, библиотека существенно использует особенности языка *Java*, и особенности ее версии 1.5 [13]. Например, в предлагаемой библиотеке используются механизмы *Reflection* [22], *Annotations* [20] и *Generics* [21].

Паттерн *O-State*, описанный в разд. 2.3, был несколько модифицирован с учетом возможностей языка *Java* 1.5. Опишем выполненные изменения:

- помимо приема и обработки событий, возможно также общение с внешней средой через некоторый интерфейс (как в паттерне *State*). Вызовы методов регистрируются так же, как события, и делегируются состояниям, так же, как это делается при обработке события;
- абстрактный класс **AbstractState** заменен на соответствующий интерфейс **IState**;
- описание класса получается не вызовом метода `getDiscriptor`, а при помощи механизма Annotations (рис. 3.2);
- роль **StateIdentifier** выполняет класс **java.lang.Class**.

Структура основных классов библиотеки *O-State* отражена на диаграмме, изображенной на рис. 3.1.

Поясним функциональность основных классов библиотеки.

- **StateMachineFactory** — класс-фабрика для построения экземпляров автомата;
- **IStateMachine** — базовый интерфейс автомата;
- **StateMachineManager** — класс, реализующий базовые операции автомата;
- **IState** — базовый интерфейс состояния автомата;
- **IStateFactory** — базовый интерфейс класса-фабрики для построения экземпляров состояния;
- **Transition** — переход;
- **Event** — событие;
- **StateMachineImpl** — реализация базового интерфейса автомата;
- **StatesTree** — реализация дерева состояний;
- **StatesTreeController** — реализация сложных действий на дереве состояний.

В заключение отметим, что в предыдущем разделе были перечислены некоторые особенности, которые необходимо учесть при реализации предложенного паттерна. Они учтены при создании библиотеки следующим образом:

- алгоритм обработки события разбит на несколько частей и реализован в классах **StateMachineManager** и **StatesTreeController**;
- дерево состояний реализовано в классе **StatesTree**. Реализации сложных операций вынесены в класс **StatesTreeController**;

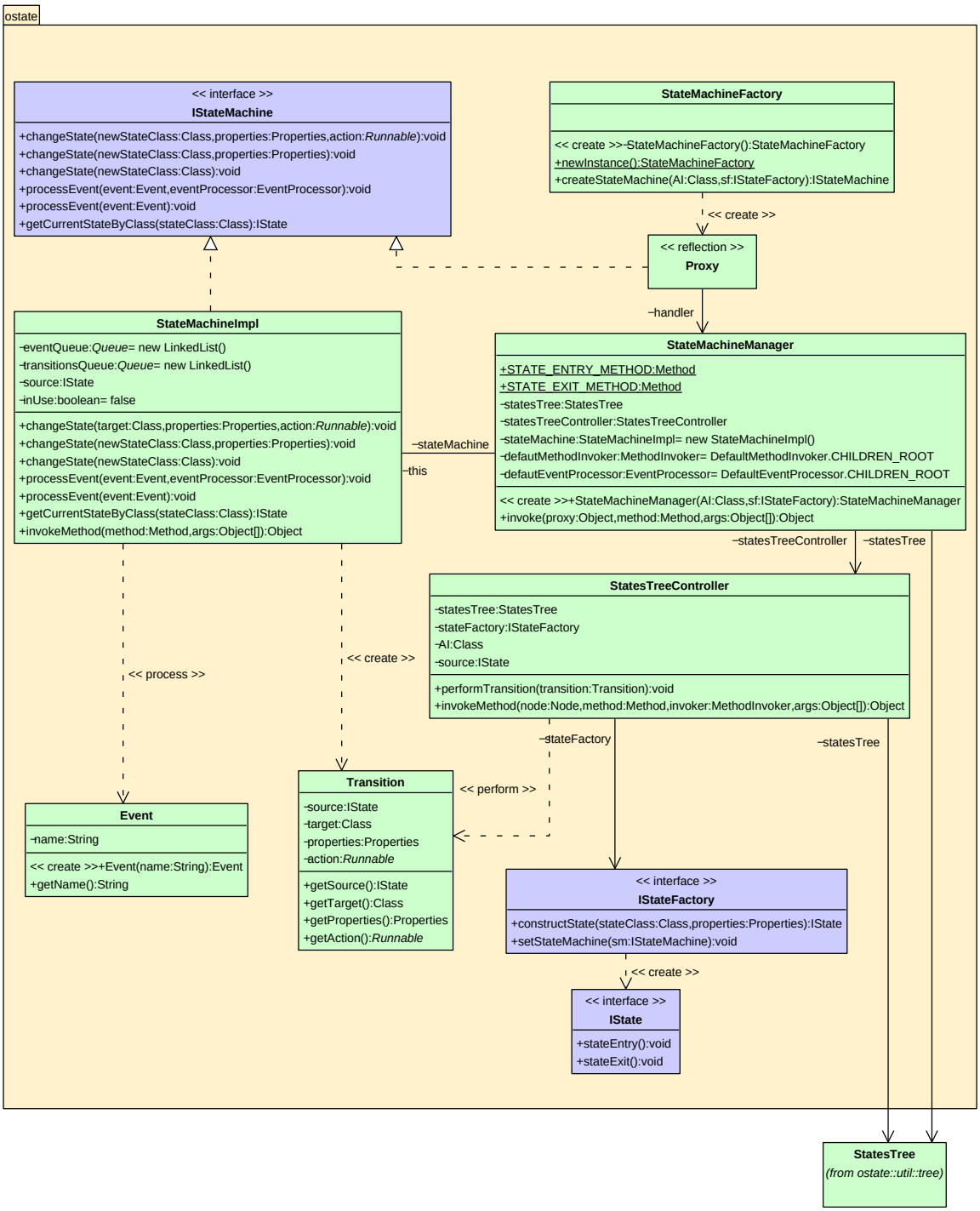

Рис. 3.1. Основные классы библиотеки *O-State*

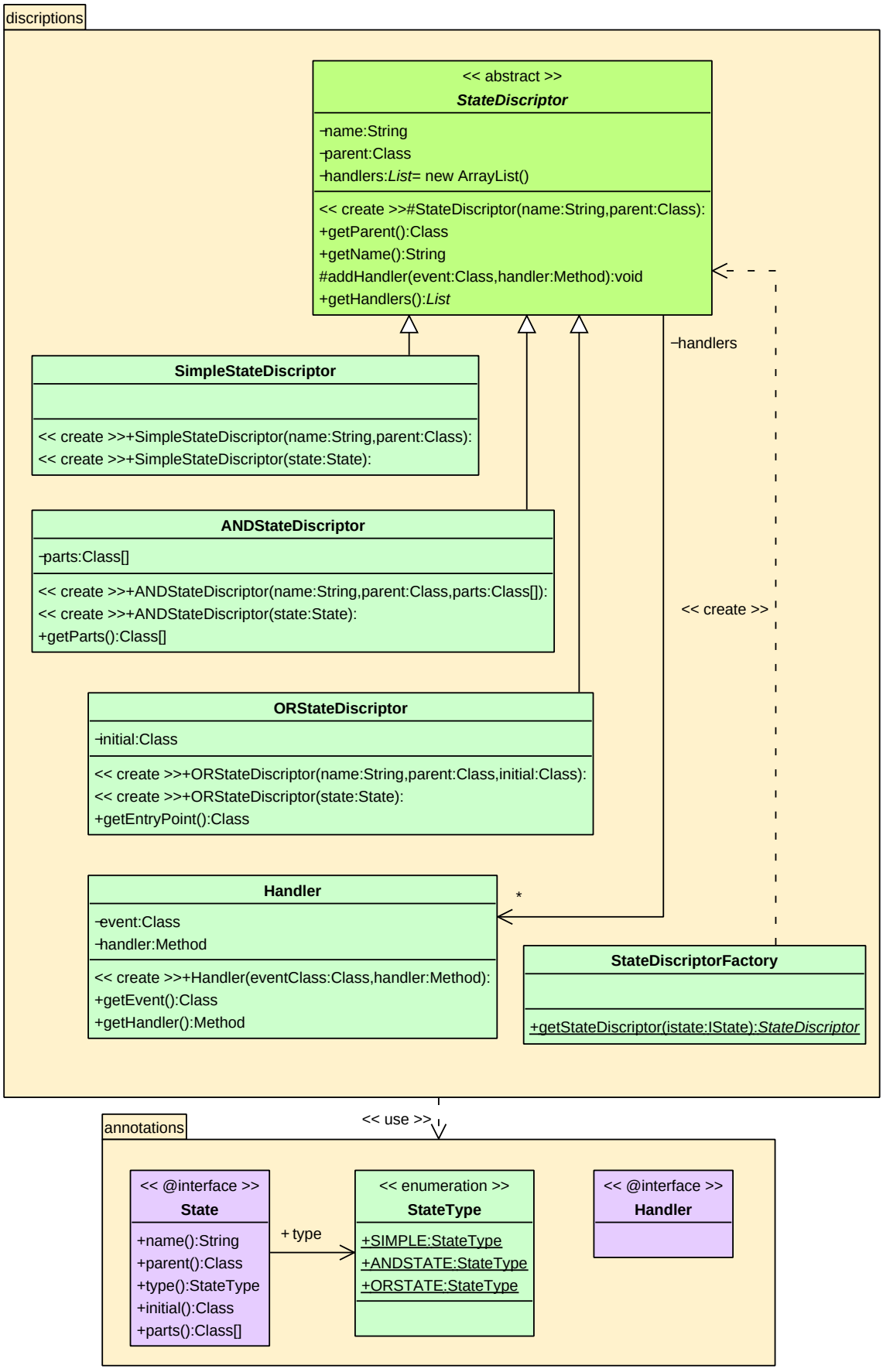

Рис. 3.2. Использование механизма *Annotations* в библиотеке *O-State*

- в качестве класса **StateIdentifier** используется класс **java.lang.Class**. Благодаря этому, процесс построения экземпляра состояния в реализации интерфейса **IStateFactory** может быть унифицирован.

3.2. Пример использования

Рассмотрим простой пример. Пусть требуется построить сетевое соединение, поведение которого заданно диаграммой состояний, изображенной на рис. 3.3.

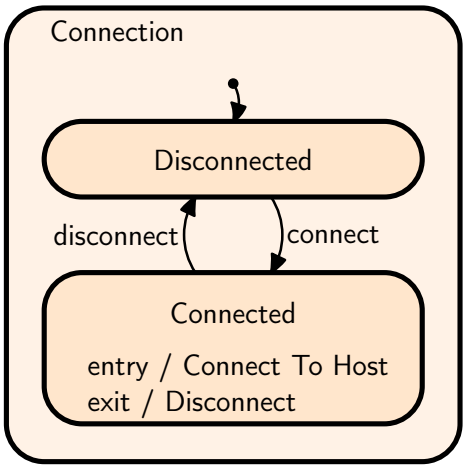

Рис. 3.3. Диаграмма состояний сетевого соединения

Первоначально опишем внешний интерфейс соединения — интерфейс **Connection** (рис 3.4).

```
public interface Connection {

    void connect();

    void disconnect();

}
```

Далее реализуем абстрактный класс **AbstractConnectionState**, являющийся предком всех классов, описывающих состояния соединения. Все эти классы должны реализовывать интерфейсы **IState<Connection>** и **Connection**, поэтому реализуем их в классе **AbstractConnectionState**.

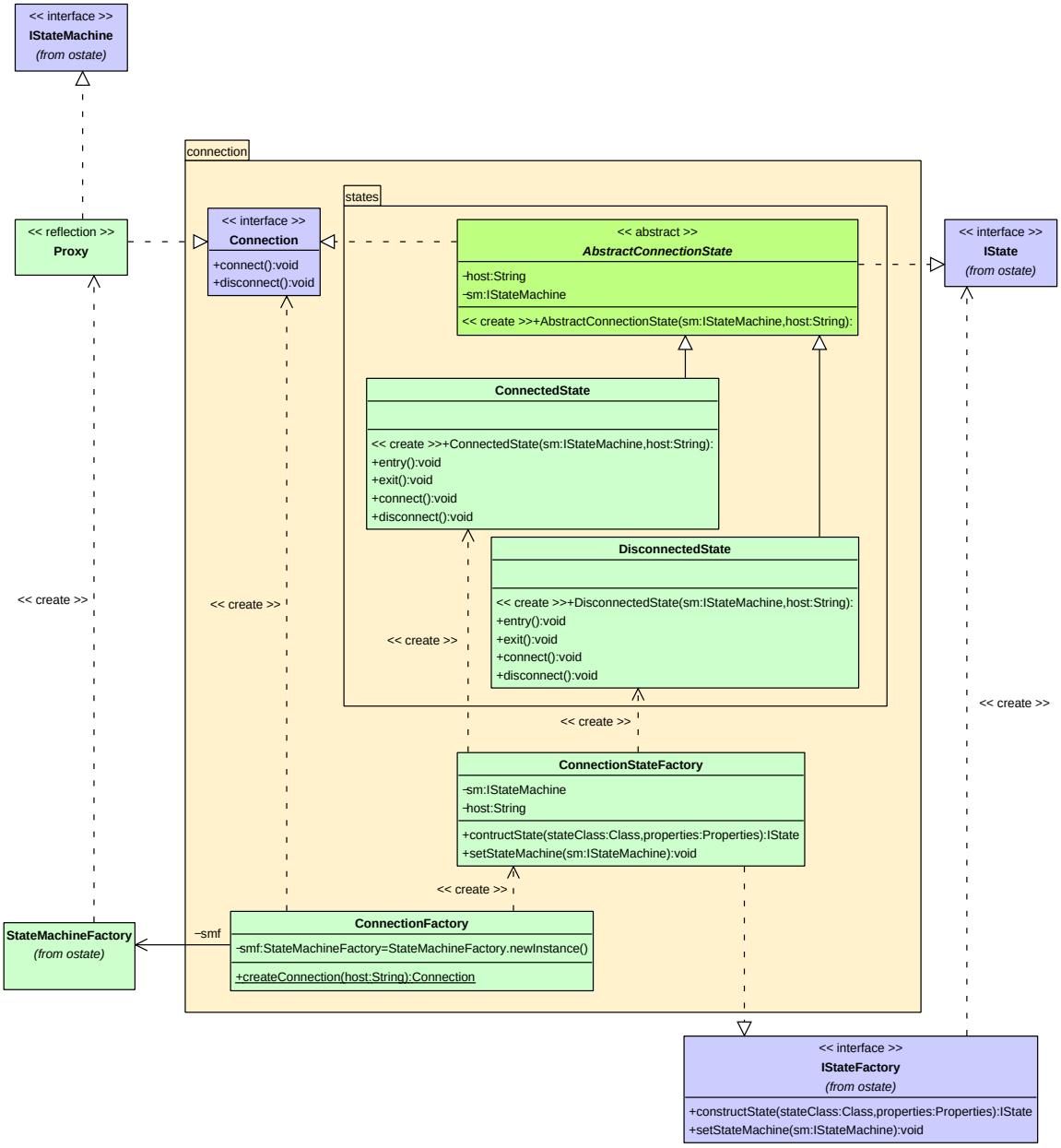

Рис. 3.4. Диаграмма классов сетевого соединения

```
abstract public class AbstractConnectionState
                  implements IState<Connection>, Connection {

    protected final IStateMachine<Connection> sm;

    protected final String host;


    public AbstractConnectionState(
                IStateMachine<Connection> sm,
                String host) {
        this.sm = sm;
        this.host = host;
    }

    public void entry() {
    }

    public void exit() {
    }

    public void connect() {
    }

    public void connect() {
    }

}
```

Теперь реализуем классы состояний: **ConnectedState** и **DisconnectedState**.

```
@State (
        name = "disconnected"
)
public class DisconnectedState extends AbstractConnectionState {

    public DisconnectedState(
                IStateMachine<Connection> sm,
                String host) {
        super(sm, host);
    }

    public void connect() {
        sm.changeState(ConnectedState.class);
    }

}
```

```
@State (
        name = "connected"
)
public class ConnectedState extends AbstractConnectionState {

    public ConnectedState(
                IStateMachine<Connection> sm,
                String host) {
        super(sm, host);
    }

    public void entry() {
        // Connect to Host
    }

    public void exit() {
        // Disconnect
    }

    public void disconnect() {
        sm.changeState(DisconnectedState.class);
    }

}
```

Реализуем класс **ConnectionStateFactory**, используемый для построения экземпляров классов состояния соединения. При этом применяется механизм Reflection. Заметим, в классах состояний должен быть конструктор с параметрами типа: **IStateMachine** и **String**.

```
public class ConnectionStateFactory implements IStateFactory<Connection> {

    private IStateMachine<Connection> sm;

    private final String host;

    public ConnectionStateFactory(
               String host) {
        this.host = host;
    }

    public void setStateMachine(IStateMachine<Connection> sm) {
        this.sm = sm;
    }

    public <T extends IState<Connection>>T constructState(
               Class<T> stateClass,
               Properties properties) {
        try {
            return stateClass.getConstructor(
                       IStateMachine.class,
                       String.class
                   ).newInstance(sm, host);
        } catch (Exception e) {
            return null;
        }
    }

}
```

Теперь все готово для создания объекта, реализующего диаграмму состояний, изображенную на рис. 3.3. Поскольку это действие не тривиально, реализуем его в специальном классе **ConnectionFactory**.

```
public class ConnectionFactory {

    private static final
        StateMachineFactory<Connection> SMF =
        StateMachineFactory.newInstance();

    public static Connection createConnection(
               String host) {
        IStateMachine<Connection> sm =
                SMF.createStateMachine(
                    Connection.class,
                    new ConnectionStateFactory(host)
                );
        sm.changeState(Disconnected.class);
        return (Connection)sm;
    }

}
```

3.3. Использование библиотеки *O-State* в ядре *Taiga* системы *PCMS4*

Система *PCMS* предназначена для автоматического тестирования задач по информатике и программированию. Система используется при проведении олимпиад и чемпионатов различного уровня, например, при проведении Всероссийской олимпиады по информатике и полуфинала NEERC международного чемпионата по программированию ACM ICPC. В настоящее время идет разработка четвертой версии системы.

Одна из основных особенностей системы — компонентная реализация, при которой компоненты могут работать независимо друг от друга. Таким образом, если одна из компонент выйдет из строя, она может быть исправлена и перезапущена. При этом компоненты, не использующие ее, в перезапуске не нуждаются.

Ядро системы *PCMS4*, которое носит название *Taiga*, предназначено для обеспечения взаимодействия компонент (поддержка динамического запуска, остановки и перезапуска компонент, прослеживание их зависимостей друг от друга, динамическое подключение друг к другу и т.д.).

Жизненный цикл компоненты довольно сложен: компонента ожидает, пока будут зарегистрированы все необходимые ей сервисы, затем стартует, работает и, при необходимости, останавливает работу. Общий жизненный цикл компоненты представлен на рис. 3.5.

Выбор диаграмм состояний как метода формального описания жизненного цикла компонент обоснован в работе [4].

Таким образом, часть ядра *Taiga* должна следить за текущим состоянием каждой компоненты и реализовать логику ее жизненного цикла.

В предыдущих версиях системы эта часть ядра писалась тра-

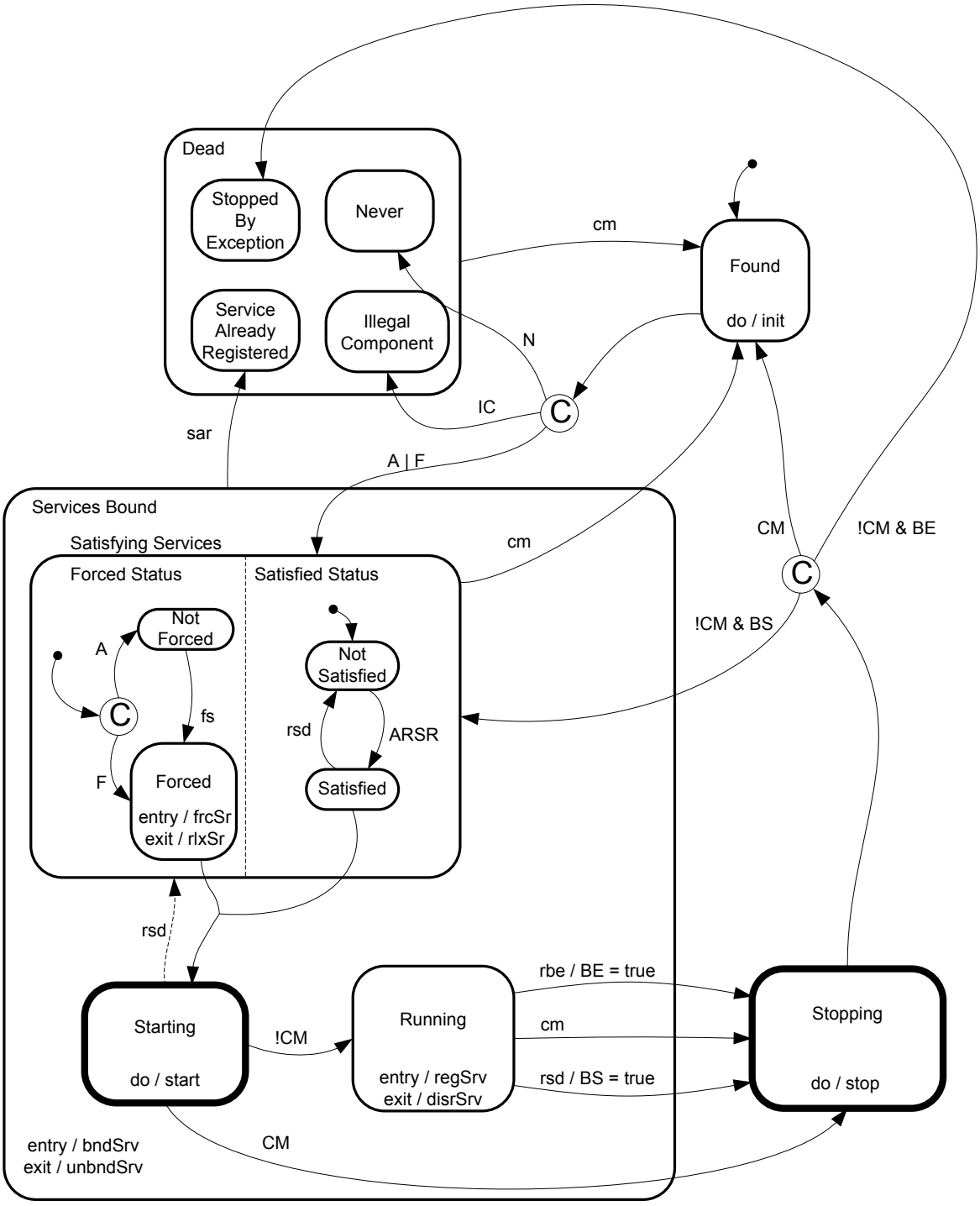

Рис. 3.5. Жизненный цикл компоненты PCMS4

диционным образом, с использованием флагов, а так как она, как отмечено выше, реализует сложную логику, то уверенность в правильности ее работы во всех режимах отсутствовала.

Для устранения этого недостатка, во-первых, функциональность этой части ядра была формально специфицирована с помощью диаграммы состояний, описывающей ее жизненный цикл, а во-вторых, эта диаграмма была корректно реализована с помощью разработанной в настоящей работе библиотеки *O-State*.

Исходные коды разработанной библиотеки приведены на сайте http://is.ifmo.ru/ в разделе «Работы».

3.4. Выводы по главе 3

1. Разработана библиотека *O-State* для реализации диаграмм состояний на основе одноименного паттерна.
2. Библиотека апробирована при реализации части ядра системы тестирования олимпиадных задач, отвечающей за реализацию жизненного цикла компонент системы.

ЗАКЛЮЧЕНИЕ

В работе получены следующие научные результаты:

1. Из всех элементов, существующих в диаграммах состояний UML, выделены основные, базовые элементы. Это было сделано для упрощения модели и облегчения построения формальной семантики.
2. Построена формальная семантика для выделенного подмножества элементов диаграмм состояний UML, Введены три класса систем, по степени их детерминированности.
3. Были исследованы существующие паттерны проектирования, используемые для построения реактивных систем, разработан новый паттерн проектирования O-State, помогающий реализовать построенную семантику.
4. Разработана библиотека O-State на языке Java, содержащая классы и интерфейсы, необходимые для реализации предложенного паттерна проектирования.
5. Результаты работы внедрены в ядре Taiga системы PCMS4.

Возможны следующие пути развития работы:

1. Расширение семантики для поддержки многопоточной обработки событий.
2. Модификации и расширения библиотеки O-State для более удобного использования.
3. Рассмотреть варианты использования предложенной семантики для не объектно-ориентированных языков.

ИСТОЧНИКИ

[1] *Буч Г., Джекобсон А., Рамбо Д.* UML. Руководство пользователя. М.: ДМК Пресс, 2004.

[2] *Буч Г., Якобсон А., Рамбо Д.* UML. СПб.: Питер, 2006.

[3] *Гамма Э., Хелм Р., Джонсон Р., Влассидес Д.* Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб.: Питер, 2001.

[4] *Маврин П., Корнеев Г., Станкевич А., Шалыто А.* Моделирование жизненного цикла компоненты программного комплекса с использованием диаграмм состояний. http://is.ifmo.ru/works/_taiga.pdf.

[5] *Маврин П., Корнеев Г., Шалыто А.* Формальная семантика диаграмм состояний, удобная для практического применения. http://is.ifmo.ru/works/_semantics.pdf.

[6] Сайт, посвященный автоматному программированию. http://is.ifmo.ru/.

[7] *Фаулер М., Скотт К.* UML. Основы. Краткое руководство по унифицированному языку моделирования. М.: Символ-Плюс, 2002.

[8] *Шалыто А. А.* SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998. http://is.ifmo.ru/books/switch/1.

[9] *Шамгунов Н., Корнеев Г., Шалыто А.* State machine — новый паттерн объектно-ориентированного проектирования // *Информационно-управляющие системы*. 2004. № 5. С. 13–25. http://is.ifmo.ru/works/pattern.pdf.

[10] *Adamczyk P.* The anthology of the finite state machine design patterns. http://jerry.cs.uiuc.edu/~plop/plop2003/Papers/Adamczyk-State-Machine.pdf.

[11] *Crane M. L., Dingel J.* Uml vs. classical vs. rhapsody statecharts: Not all models are created equal. // MoDELS. 2005. Pp. 97–112.

[12] Eclipse. The open source community whose projects are focused on providing an extensible development platform and application frameworks for building software. http://www.eclipse.org/.

[13] *Gosling J., Joy B., Steele G., Bracha G.* The Java Language Specification, Third Edition. Boston, Mass.: Addison-Wesley, 2005. http://java.sun.com/docs/books/jls/download/langspec-3.0.pdf.

[14] *Harel D.* Statecharts: A visual formalism for complex systems // *Science of Computer Programming*. June 1987. Vol. 8, no. 3. Pp. 231–274. citeseer.ist.psu.edu/harel87statecharts.html.

[15] *Harel D., Kugler H.* The RHAPSODY Semantics of Statecharts (or, On the Executable Core of the UML) // Integration of Software Specification Techniques for Application in Engineering. Vol. 3147 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 2004. Pp. 325–354.

[16] *Harel D., Lachover H., Naamad A., Pnueli A., Politi M., Sherman R., Shtull-Trauring A., Trakhtenbrot M. B.* STATEMATE: A working environment for the development of complex reactive systems // *Software Engineering*. 1990. Vol. 16, no. 4. Pp. 403–414. citeseer.ist.psu.edu/harel90statemate.html.

[17] *Harel D., Naamad A.* The STATEMATE semantics of statecharts // *ACM Transactions on Software Engineering and Methodology*. 1996. Vol. 5, no. 4. Pp. 293–333. citeseer.ist.psu.edu/harel96statemate.html.

[18] I-Logix Rhapsody: Model-Driven Development Software with UML 2.0. http://www.ilogix.com/rhapsody/rhapsody.cfm.

[19] I-Logix Statemate, Embedded Systems Design Software. http://www.ilogix.com/statemate/statemate.cfm.

[20] Java Tutorial: Annotations. http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html.

[21] Java Tutorial: Generics. http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf.

[22] Java Tutorial: Reflection. http://java.sun.com/docs/books/tutorial/reflect/.

[23] *Mellor S. J., Balcer M. J.* Executable UML. Addison-Wesley, 2004.

[24] *Samek M.* Practical Statecharts in C/C++. CMP Books, 2002.

[25] UniMod. Инструмент, поддерживающий SWITCH-технологию. http://unimod.sourceforge.net/.

[26] Visio2Switch, конвертор для автоматической генерации C/C++ кода по автоматным графам, нарисованных в Visio в соответствии с требованиями SWITCH-технологии. http://is.ifmo.ru/progeny/visio2switch/.