Санкт-Петербургский государственный университет информационных технологий, механики и оптики
Факультет информационных технологий и программирования
Кафедра компьютерных технологий

Д. Ю. Кочелаев

Проектирование, спецификация и реализация автоматизированных классов

Магистерская работа

Научный руководитель – докт. техн. наук, профессор А. А. Шалыто

Санкт-Петербург
2009

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ 5

ГЛАВА 1. ПОСТАНОВКА ЗАДАЧИ 7

1.1. Основные понятия 7

1.2. Постановка задачи 7

Выводы по главе 1 8

ГЛАВА 2. ОБЗОР СУЩЕСТВУЮЩИХ ПОДХОДОВ ПОСТРОЕНИЯ ПРОГРАММ И ИХ РЕАЛИЗАЦИИ 9

2.1. Классификация существующих подходов 9

2.2. Генераторы декларативной части 10

2.2.1. *IBM Rational Rose* 10

2.2.2. *Power Designer* 11

2.2.3. *Enterprise Architect* 13

2.3. Генераторы логики 14

2.3.1. *Windows Workflow Foundation* 14

2.3.2. *UniMod* 16

2.3.3. *State Machine Designer* 18

2.4. Автоматизированные абстрактные типы данных 19

Выводы по главе 2 19

ГЛАВА 3. БИБЛИОТЕКА *EIFFEL-STATE* 21

3.1. Язык программирования Eiffel 21

3.1.1. Основные черты *Eiffel* 21

3.2. Реализация ААТД. Автоматизированные классы 23

3.2.1. Основные принципы 23

3.2.2. Управляющие и вычислительные состояния 24

3.2.3. Автоматизированные классы 25

3.2.4. Проектирование автоматизированных классов 27

3.2.5. Архитектура библиотеки 28

3.2.6. Вопросы реализации 32
3.2.7. Наследование автоматизированных классов 35
3.3. Визуальное проектирование системы 37
3.3.1. Преобразование диаграммы автоматов в исходный код 38
3.3.2. Преобразование исходного кода в диаграмму автоматов 39
Выводы по главе 3 40
ГЛАВА 4. ПРИМЕР ИСПОЛЬЗОВАНИЯ БИБЛИОТЕКИ 41
4.1. Описание задачи 41
4.2. Решение задачи в терминах объектно-ориентированного программирования 42
4.3. Решение задачи с использованием автоматизированных классов 44
Выводы по главе 4 48
ЗАКЛЮЧЕНИЕ 49
ИСТОЧНИКИ 50
ПРИЛОЖЕНИЯ 53
Приложение 1. Исходные коды основных классов библиотеки EiffelState 53
Класс *AUTOMATED* 53
Класс *STATE* 54
Класс *STATE_DEPENDENT_FUNCTION* 55
Класс *STATE_DEPENDENT_PROCEDURE* 60
Приложение 2. Исходные коды традиционной реализации игры «крестики-нолики» 63
Класс *APPLICATION* 63
Класс *FIELD_CELL* 65
Класс *GAME* 69
Класс *GAME_MANAGER* 81
Класс *INTERFACE_NAMES* 83
Класс *MAIN_WINDOW* 84
Класс *WINNER_FIRST_MANAGER* 91
Приложение 3. Исходные коды реализации игры «крестики-нолики», базирующейся на автоматизированных классах 93
Класс *FIELD_CELL* 93

Класс *GAME* 97
Класс *GAME_MANAGER* 109
Класс *WINNER_FIRST_MANAGER* 111

ВВЕДЕНИЕ

В настоящее время при разработке программного обеспечения широко используется язык *UML* [1 – 3]. При этом поведение программной системы описывается с помощью диаграмм состояний детерминированных конечных автоматов [4].

Конечный автомат – это математическая абстракция, позволяющая описывать изменения состояния объекта в зависимости от его текущего состояния и входных данных, при условии, что число состояний конечно.

Важным является то, что рассматриваемые конечные автоматы, описывающие поведение, являются детерминированными. Детерминированным конечным автоматом называется такой автомат, в котором при любой последовательности входных данных существует лишь одно состояние, в которое автомат может перейти из текущего состояния. Переход в новое состояние зависит от того, какие условия на переходах выполняются. Также при переходе возможен вызов различных действий, которые изменяют окружение автомата.

Отметим, что визуальное проектирование поведения системы с помощью диаграмм состояний значительно облегчает задачу описания логики системы. В настоящее время существует ряд средств для визуального проектирования программных систем [5 – 7]. Однако, в этих средствах проектирование системы необходимо производить с нуля, а также они накладывают ограничения на архитектуру создаваемой системы. Данные ограничения делают невозможным использование этих инструментальных средств при работе над уже существующими проектами, а также не предоставляют простого способа для перехода от существующей архитектуры к архитектуре, совместимой с этими инструментальными средствами.

Рассмотрим основные задачи, решаемые в данной работе:

– разработка метода использования автоматного программирования, позволяющего встраивать автоматы в существующие объектно-ориентированные системы;

– реализация предложенного метода и обоснование простой интеграции в существующие системы;

– разработка алгоритма перехода от диаграмм состояний к исходным кодам и обратно.

В главе 1 приведена формальная постановка задачи и введены основные понятия, используемые в работе.

В главе 2 приведен обзор существующих средств для поддержки автоматного программирования и определены основные недостатки и возможные проблемы при их использовании.

В главе 3 описывается решение вопросов, возникающих при реализации метода, который предложен Н. Поликарповой [8, 9], и приводится описание реализации этого метода. Также в этой главе приведен алгоритм для перехода от диаграмм состояний к исходному коду и обратно.

В главе 4 проведен анализ предлагаемого подхода, и выполнено его сравнение с существующими подходами. Также в этой главе приведен пример интеграции в существующую систему с использованием подхода, описанного в главе 3.

ГЛАВА 1. ПОСТАНОВКА ЗАДАЧИ

1.1. Основные понятия

Приведем основные понятия, используемые в данной работе.

Детерминированный конечный автомат – математическая абстракция, которая состоит из следующих компонент [10]:

- конечное множество состояний;
- конечное множество входных воздействий;
- функция переходов, аргументами которой является текущее состояние и входное воздействие, а значением – новое состояние;
- начальное состояние;
- множество заключительных, или допускающих, состояний.

Автоматное программирование – метод проектирования программных систем с явным выделением состояний [11].

Диаграмма автоматов (диаграмма состояний) — диаграмма, на которой представлен конечный автомат с простыми состояниями, переходами и композитными состояниями [2].

Автоматизированный абстрактный тип данных (ААТД) — абстрактный тип данных, который включает в себя автомат, описывающий его поведение [8].

1.2. Постановка задачи

В 2006 году Н. Поликарпова в бакалаврской работе, выполненной на кафедре «Компьютерные технологии» СПбГУ ИТМО, предложила объектно-ориентированный подход к моделированию и спецификации сущностей со сложным поведением [8]. В основе этого подхода лежит автоматное программирование, при использовании которого поведение объектов задается с

помощью конечных автоматов. Такие объекты было предложено называть автоматизированными абстрактными типами данных (ААТД).

Целью работы Н. Поликарповой было построение математической модели сущности, поведение которой зависит от состояния. Таким образом, работа являлась теоретической и не предлагала какой-либо реализации для ААТД. Также остался открытым ряд вопросов, которые связаны с реализацией предложенной методики.

Целью настоящей работы является расширение методики, предложенной Н. Поликарповой, а также практическая реализация данной методики.

Основные направления работы:

1. Выявление и решение открытых вопросов, связанных с практической реализацией методики.
2. Практическая реализация методики в виде библиотеки на языке *Eiffel* [12].
3. Разработка алгоритма перехода от реализации поведения ААТД на языке *Eiffel* к визуальному представлению в виде диаграммы автоматов и обратно.

Выводы по главе 1

1. Введены основные понятия, используемые в данной работе.
2. Выполнена постановка решаемой задачи и определены основные направления работы.

ГЛАВА 2. ОБЗОР СУЩЕСТВУЮЩИХ ПОДХОДОВ ПОСТРОЕНИЯ ПРОГРАММ И ИХ РЕАЛИЗАЦИИ

2.1. Классификация существующих подходов

Язык *UML* позволяет описать поведение (с помощью диаграмм автоматов) и структуру (с помощью диаграммы классов) программы. При этом на основе этих диаграмм может быть построен исходный код на целевом языке программирования. Отметим, что различные подходы и инструментальные средства, которые их реализуют, позволяют описать с помощью диаграмм либо поведение, либо декларативную часть программы. На основании этой характеристики подходы и их реализации могут быть разделены на две группы. При этом один и тот же подход может попадать в обе группы.

Основной акцент в первой группе подходов делается на построение декларативной части – иерархии классов, их методов и полей. При этом поведение спроектированной системы должно быть прописано разработчиком вручную и никак не отражается при проектировании.

Вторая группа подходов, наоборот, предоставляет возможности для описания поведения разрабатываемой системы. При этом визуальное описанное поведение системы преобразуется в код на целевом языке программирования. При этом может строиться и декларативная часть системы.

Другим важным атрибутом подхода является возможность вносить в систему изменения после построения кода на целевом языке программирования – возможность совместить визуальное проектирование системы и непосредственное кодирование частей функциональности.

2.2. Генераторы декларативной части

Рассмотрим инструментальные средства из первой группы подходов – средства, которые предоставляют возможность для визуального описания только декларативной части. К этим средствам можно отнести такие программные продукты как *IBM Rational Rose* [13], *Power Designer* [14], *Enterprise Architect* [15] и т. д. Остановимся подробнее трех указанных инструментальных средствах.

2.2.1. *IBM Rational Rose*

Программный продукт *IBM Rational Rose* (интерфейс продукта представлен на рис. 1), выпущенный компанией *IBM Rational Software*, поддерживает визуальное объектно-ориентированное моделирование. Для визуального моделирования используются *UML*-диаграммы. Данное средство поддерживает построение кода по диаграммам, а также обратное проектирование – построение модели по программному коду для большого числа языков программирования. *IBM Rational Rose* поддерживает большинство *UML*-диаграмм для подробного моделирования проекта, однако, многие из этих диаграмм не влияют на генерируемый код.

Отметим, что данное инструментальное средство поддерживает только построение декларативной части кода – иерархию классов спроектированной системы, а также артефакты (поля, методы и т.д.) конкретных классов. Вся функциональность, которая отвечает за поведение системы, должна быть написана разработчиком вручную.

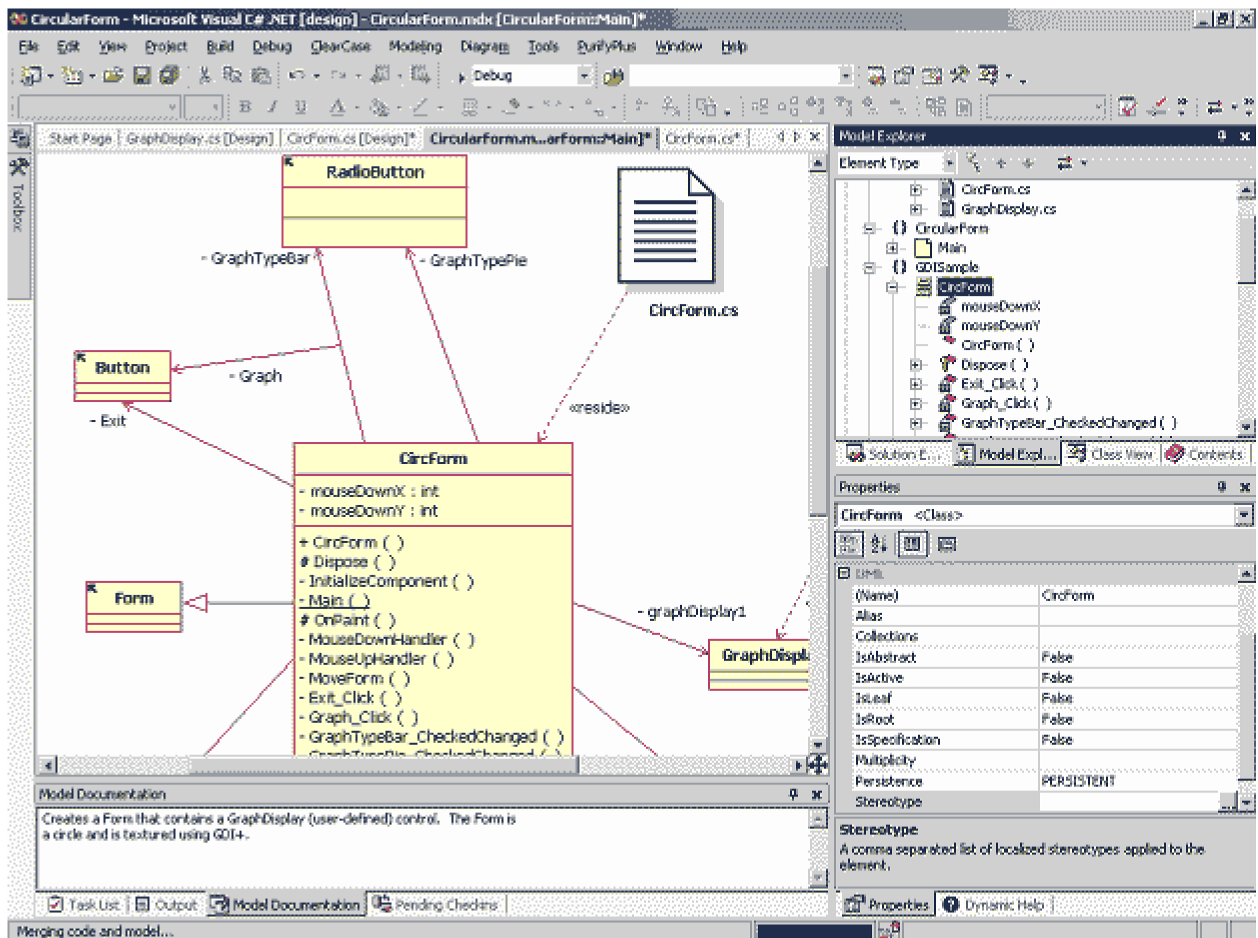

Рис. 1. Интерфейс *IBM Rational Rose*

2.2.2. *Power Designer*

Программный продукт *Power Designer* (интерфейс продукта представлен на рис. 2) от компании *Sybase* предоставляет возможности для создания полнофункциональных бизнес-приложений. Отметим некоторые средства, которые предоставляет данный продукт:

- средства для моделирования бизнес-процессов;
- средства для проектирования баз данных (с последующей синхронизацией структуры базы и ее модели)
- средства для построения кода по модели для языков *Java*, *C#*, *C++*, *PowerBuilder*, *VB.Net* и т.д.

При этом для генерации кода с помощью данного программного продукта возможно использование современных технологий программирования, например, для языка *Java* – *Enterprise Java Beans* (*EJB*).

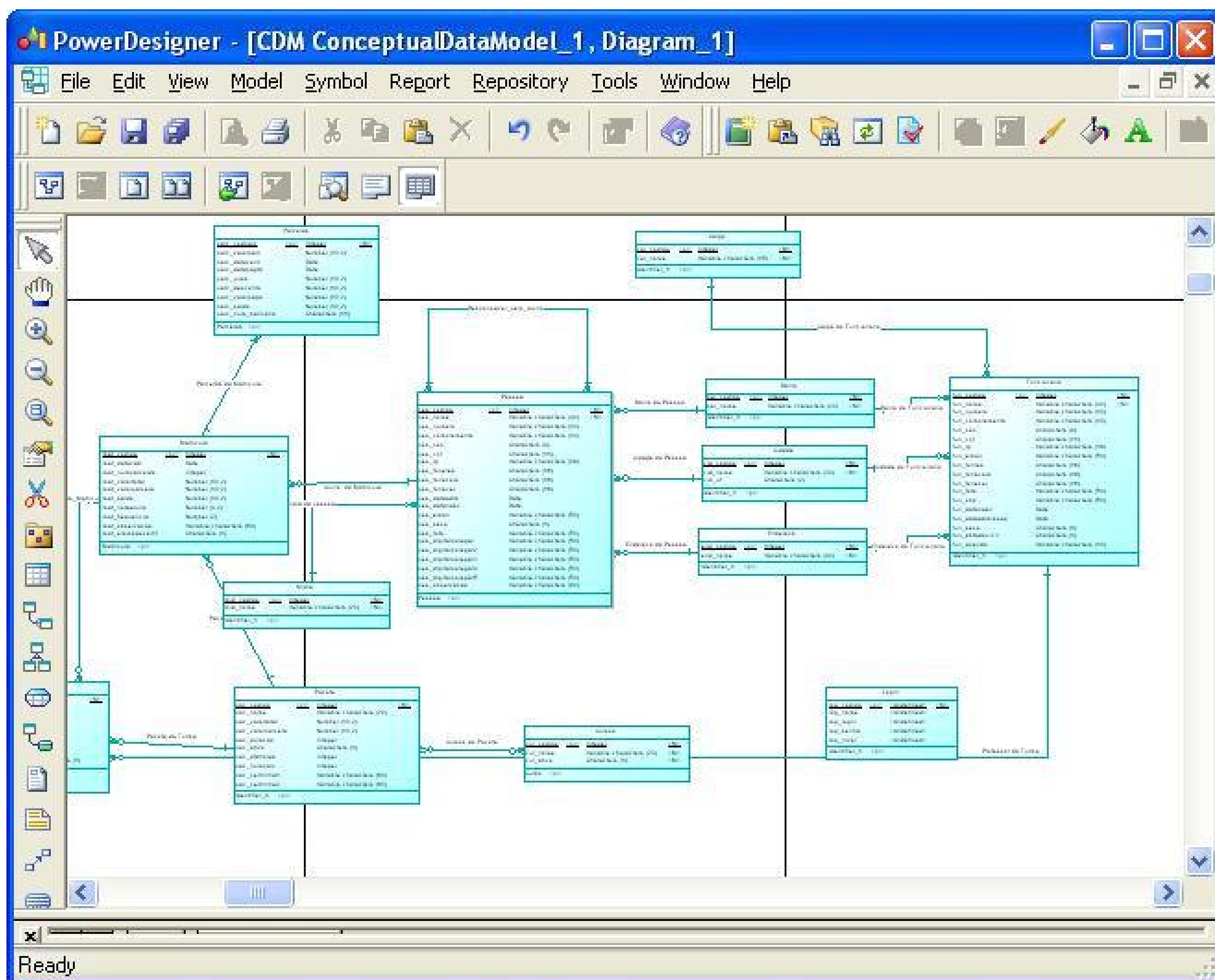

Рис. 2. Интерфейс *Sybase Power Designer*

Для описания модели используются диаграммы *UML 2*, с возможностью обратного проектирования для диаграмм, которые описывают декларативную часть проекта. Однако, процессы построения кода по диаграммам и восстановление диаграмм из кода не имеют двусторонней синхронизации.

Приведем пример потери информации при построении кода. Пусть была создана некоторая модель, описывающая структуру (поля и методы) класса *SampleClass*. После этого были запрограммированы действия, выполняемые в методах, а также добавлены дополнительные методы (путем их ручного кодирования). Потом для добавления методов в модель был запущен процесс обратного проектирования. Заметим, однако, что логика методов никак не отражается на диаграмме.

2.2.3. *Enterprise Architect*

Программный продукт *Enterprise Architect* (интерфейс продукта представлен на рис. 3) разработан компанией *Sparx Systems*. Данный продукт поддерживает диаграммы *UML 2*. В том числе поддерживаются и диаграммы *StateChart*, которые позволяют описать поведение проектируемой системы в терминах конечных автоматов.

Построение кода по диаграммам в этом продукте также возможно только для декларативной части – иерархии классов и артефактов спроектированных классов. Возможность построения кода по автоматной модели не предусмотрена. Данный программный продукт имеет еще один существенный недостаток – отсутствует связь с какой-либо средой разработки, что значительно затрудняет дальнейшую работу с построенным автоматически кодом.

Приложение *Enterprise Architect* стоит в некоторой степени между приложениями, которые позволяют автоматизировать построение кода для декларативной части системы, и приложениями, которые предоставляют возможность построения кода, отвечающего за поведение проектируемой системы.

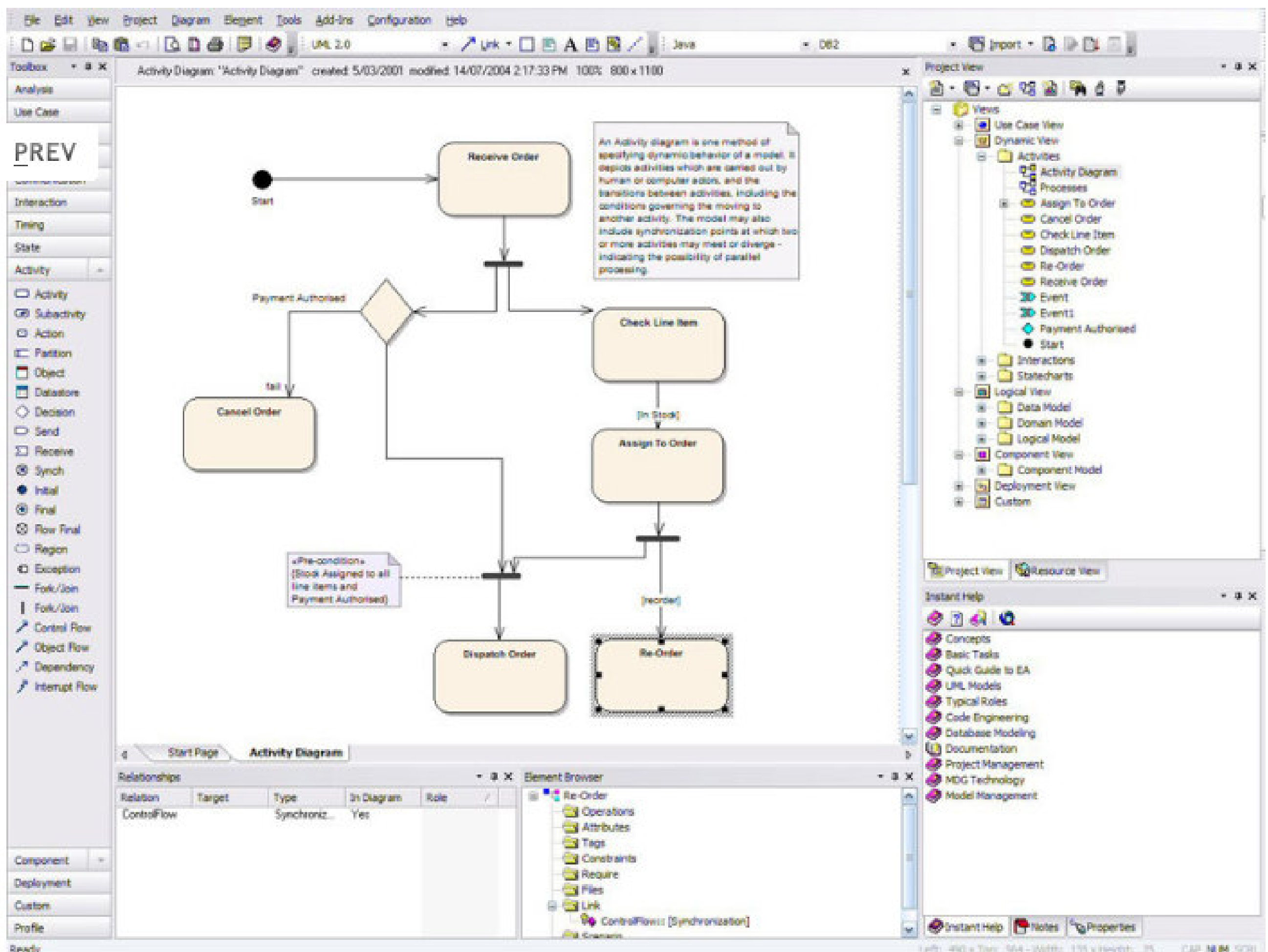

Рис. 3. Интерфейс *Enterprise Architect*

2.3. Генераторы логики

Рассмотрим программные продукты, которые реализуют второй подход. Эти инструментальные средства позволяют строить исходный код, который отвечает за поведение проектируемой системы на основе диаграмм автоматов.

2.3.1. *Windows Workflow Foundation*

Технология *Windows Workflow Foundation* [16] разработана корпорацией *Microsoft*. Реализация этой технологии распространяется вместе со средой разработки *Visual Studio* (интерфейс продукта представлен на рис. 4). Данный продукт предоставляет широкие возможности по описанию модели

проектируемой системы и автоматическому построению кода по модели. Он поддерживает построение кода для части системы, которая отвечает за поведение. Таким образом, *Windows Workflow Foundation* предоставляет возможности для визуального проектирования поведения системы с дальнейшим автоматическим построением кода по полученной модели.

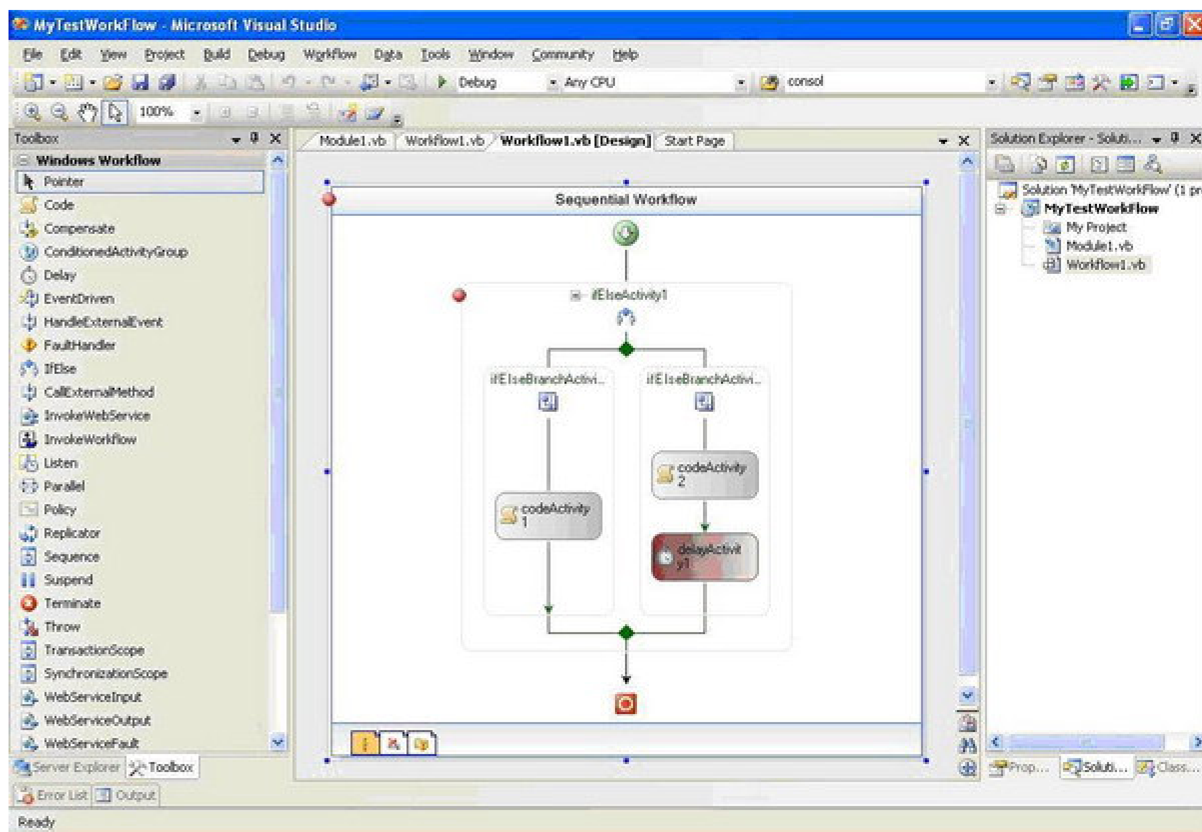

Рис. 4. Интерфейс *Windows Workflow*

Перечислим некоторые недостатки данного приложения и генерируемого им кода. Во-первых, среда разработки *Visual Studio*, в которую встроены средства для построения модели проектируемой системы, является платной. Во-вторых, подход, который был положен корпорацией *Microsoft* в основу своего программного продукта, затрудняет встраивание в уже существующие системы, что является существенным недостатком. Еще один недостаток – сложность практического применения классов, которые используются при автоматическом построении кода, без использования визуального редактора. Иными словами,

использовать библиотеку *Windows Workflow Foundation* без визуального редактора неудобно.

2.3.2. *UniMod*

Инструментальное средство *UniMod* [6, 17, 18] разработано на кафедре «Компьютерных технологий» СПбГУ ИТМО (интерфейс инструментального средства представлен на рис. 5). Фактически инструментальное средство *UniMod* является одной из реализаций «исполняемого *UML*» [19]. При использовании автоматного подхода программа в целом проектируется с помощью двух типов *UML*-диаграмм: диаграмм классов, которые представляются в виде схемам связей автоматизированных объектов, и диаграмм состояний, которые реализуют автоматы, интерфейс которых указан на диаграмме классов. Эти диаграммы исполняются автоматически, а фрагменты программ, соответствующие входным и выходным воздействиям, пишутся вручную. Таким образом, это инструментальное средство позволяет при создании программ совмещать разные уровни абстракции (диаграммы и текст на языке *Java*) и разные стили программирования (графический и текстовый).

Инструментальное средство *UniMod* предоставляет набор возможностей, которые значительно облегчают проектирование:

- Возможность визуального построения диаграмм классов и состояний. Средство обеспечивает тесную интеграцию между визуальным представлением и реализацией представленных объектов в коде.
- Возможность не только интерпретировать и компилировать построенную программу, но и осуществлять визуальную отладку, добавляя точки останова на состояния и переходы.

- Возможность валидации построенных автоматных моделей.

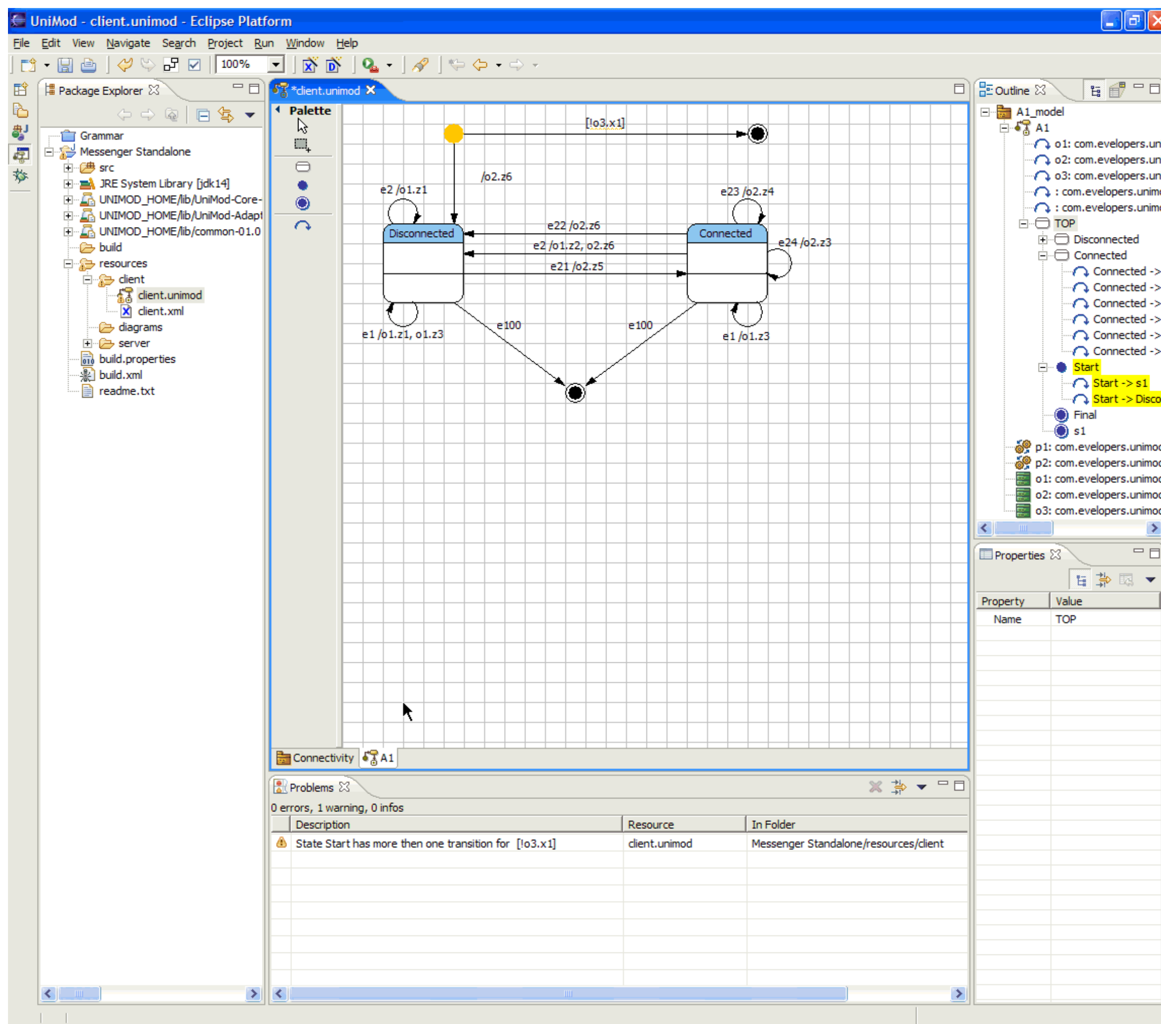

Рис. 5. Интерфейс инструментального средства *UniMod*

При этом отметим, что подход, который используется в инструментальном средстве *UniMod*, имеет ряд недостатков. Во-первых, все поведение программы должно быть описано с помощью автоматов, которые спроектированы с помощью этого инструментального средства, что делает невозможным встраивание полученного кода в существующие системы. Таким образом, данное инструментальное средство пригодно только для вновь создаваемых систем. Кроме того, эти системы должны быть небольшими (разрабатываться одним–двумя человеками), так как многие не обладают автоматным стилем мышления. Во-вторых, невозможно использовать библиотеки системы при

написании программ без использования графического редактора инструментального средства *UniMod*.

2.3.3. *State Machine Designer*

Инструментальное средство *State Machine Designer* также разработано на кафедре «Компьютерных технологий» СПбГУ ИТМО [7]. Интерфейс приложения представлен на рис. 6.

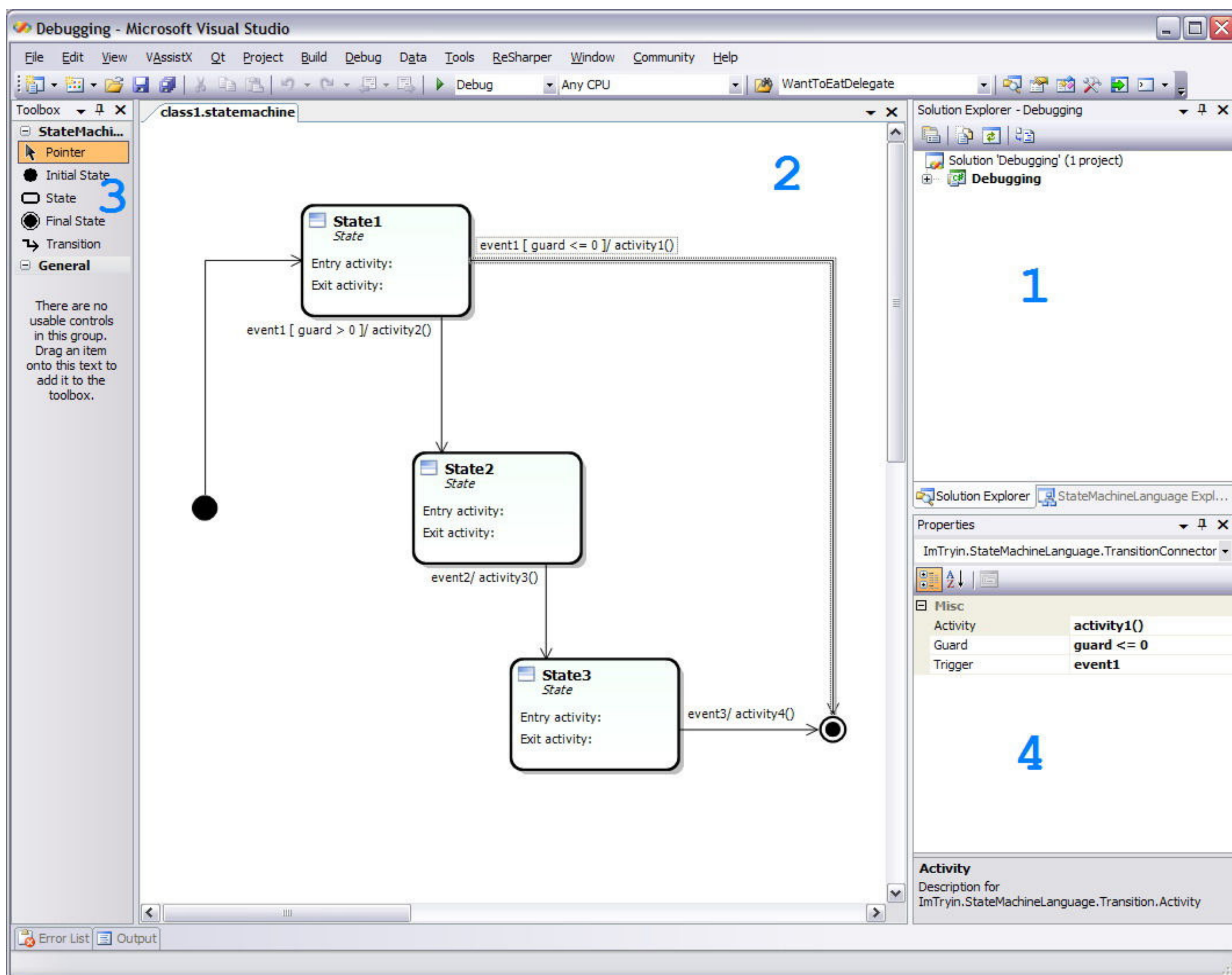

Рис. 6. Интерфейс приложения *State Machine Designer*

Данное инструментальное средство является дополнением к среде разработки *Visual Studio*. В основе этого программного продукта лежат принципы, который используются в инструментальном средства *UniMod*. Как

следствие *State Machine Designer* имеет те же недостатки, что и инструментальное средство *UniMod*.

2.4. Автоматизированные абстрактные типы данных

Объектно-ориентированный и автоматный подходы к разработке программного обеспечения обладают рядом преимуществ, поэтому решение задачи совместного использования этих двух подходов важно как с теоретической, так и с практической точки зрения. В работах [20, 21] описывается подход, названный *объектно-ориентированным программированием с явным выделением состояний*.

Однако в этих работах решается вопрос о реализации конечных автоматов с помощью объектов. Таким образом, моделирование системы выполняется исключительно в терминах автоматов, а объектно-ориентированный подход применяется позднее — на стадии реализации. Подход, предложенный автором работы [8], принципиально иной – автоматы используются для описания поведения объекта. Соответственно на стадии проектирования используются одновременно и автоматный, и объектно-ориентированный подходы.

Как было отмечено выше, одной из проблем в существующих подходах является невозможность интеграции автоматного подхода в уже существующие системы. Эта проблема решается при использовании автоматного подхода для описания поведения отдельных объектов, а не системы в целом. Так как автоматная реализация поведения инкапсулируется, то пользователь может обращаться с ААТД также как с обычным типом данных.

Выводы по главе 2

1. Проведена классификация существующих подходов.

2. Выполнен обзор генераторов декларативной части проектируемых систем. Выявлен основной недостаток этих генераторов – отсутствие возможности визуального проектирования поведения системы.
3. Проведен обзор генераторов логики проектируемых систем. Выявлены основные недостатки этих генераторов:
   - сложность, а часто и отсутствие возможности, встраивания в существующие программные системы;
   - невозможность использования библиотек, которые реализуют автоматное представление поведения, без применения визуального редактора.
4. Рассмотрен подход с использованием автоматизированных абстрактных типов данных. При его применении не возникает проблема при встраивании автоматов в уже существующие объектно-ориентированные программы. Отметим, что для данного подхода отсутствует реализация, и существуют открытые вопросы, связанные с ней.

ГЛАВА 3. БИБЛИОТЕКА *EIFFEL-STATE*

3.1. Язык программирования Eiffel

Для реализации методики ААТД предлагается использовать язык программирования *Eiffel*, разработанный Бертраном Мейером. *Eiffel* – объектно-ориентированный язык программирования, который был спроектирован для предоставления программистам возможности рационально разрабатывать расширяемые, переиспользуемые, надежные программные системы. *Eiffel* используется как язык для обучения принципам программирования и проектирования [22], так и в качестве платформы для разработки коммерческих приложений в различных прикладных областях.

3.1.1. Основные черты *Eiffel*

При проектировании в язык *Eiffel* был внесен ряд отличительных черт. Рассмотрим некоторые из них более подробно.

Все типы в *Eiffel* представляются классами. Для повышения эффективности классы в *Eiffel* разделены на ссылочные и расширенные. При этом ссылочные классы содержат только ссылку на объект, а расширенные (например, *INTEGER_8* и *CHARACTER_32*) – непосредственно значение.

Проектирование по контракту (*design by contract* [23]) — основная отличительная черта языка, которая состоит в том, что для любого метода могут быть установлены пред- и постусловия, которые будут проверяться при входе и выходе из метода соответственно. Также для каждого класса могут быть установлены инварианты, которые будут проверяться при создании новых экземпляров класса и обращении к методам класса.

Eiffel поддерживает автоматическое управление памятью – очистка памяти

при удалении всех ссылок на объект осуществляется автоматически сборщиком мусора. Причем, в качестве положительной черты *Eiffel* декларируется то, что в многопоточной программе сборщики мусора для каждой из нитей работают независимо друг от друга. Это возможно, поскольку в *Eiffel* нет глобальных и статических переменных.

Механизм агентов в *Eiffel* позволяет передавать метод объекта в качестве аргумента функции. Поэтому отпадает необходимость в создании классов оберток в случаях, когда необходимо передать в функцию код для выполнения.

Еще одной уникальной чертой *Eiffel* являются однократные (*once*) методы. Эти методы выполняются только при первом обращении к ним. В чем-то данная функциональность схожа с макросами в языках *C*/*C*++*,* но в отличие от макросов в однократных методах имеется возможность работать с объектами, которые существуют во время исполнения программы.

Eiffel позволяет работать с набором значений разных типов. Для этого существует специальный класс *TUPLE*. При объявлении переменных этого класс они параметризуются типами значений, которые будут содержаться в наборе. Обращение к значениям из набора может осуществляться как по индексу значения, так и по имени, которое также задается при объявлении переменной класса *TUPLE*.

```
Local
  book_description: TUPLE [name: STRING; author: STRING;
number_of_pages: INTEGER]
do
  -- Создание объекта типа TUPLE
  book_description := ["Hamlet", "Shakespeare", 50]
```

```
  -- Обращение к полям объекта по имени
  book_description.author := "William Shakespeare"

  -- Обращение к полям объекта по индексу
  book_description.item (1)
end
```

Отметим также, что программы на языке *Eiffel* являются переносимыми между различными платформами на уровне компиляции. При этом сначала код на языке *Eiffel* компилируется в исходный код на языке программирования *C* и только потом в исполняемый бинарный код.

3.2. Реализация ААТД. Автоматизированные классы

3.2.1. Основные принципы

Сущности с простым поведением всегда имеют одну и ту же реакцию на некоторое входное воздействие. При этом с точки зрения объектно-ориентированного подхода класс является моделью сущности с простым поведением. В свою очередь, реакция сущности со сложным поведением на некоторое входное воздействие зависит от последовательности предыдущих воздействий на сущность. Примерами сущности со сложным поведением являются элементы графического интерфейса, сетевые протоколы и задачи, связанные с имитацией искусственного интеллекта.

Таким образом, сущность со сложным поведением имеет набор возможных состояний и находится в одном из них. Реакция сущности зависит от входного воздействия и текущего состояния. При этом в результате воздействия

текущее состояние может измениться. Поведение такой сущности удобно представлять с помощью детерминированного конечного автомата. Основной целью данной работы является разработать объектно-ориентированную реализацию сущностей со сложным поведением, которая бы являлась обычным классом с точки зрения клиентов этой сущности.

3.2.2. Управляющие и вычислительные состояния

Атрибуты сущности со сложным поведением определяют набор ее вычислительных состояний как комбинацию всех возможных значений атрибутов. При этом число вычислительных состояний хоть и конечно, но очень велико. При этом вычислительные состояния отличаются друг от друга в основном лишь количественно. В случае автоматного подхода для каждой сущности со сложным поведением определяется набор управляющих состояний. Число управляющих состояний обычно мало, что делает их более удобным объектом для работы по сравнению с вычислительными состояниями. Каждое управляющее состояние в отличие от вычислительных состояний имеет семантическое значение. При этом управляющие состояния отличаются друг от друга качественно.

Хорошим примером, описывающим разницу между вычислительными и управляющими состояниями, является машина Тьюринга [10]. Управляющими состояниями в данном случае являются состояния управляющего конечного автомата, а вычислительными – все возможные состояния ленты.

Управляющие состояния могут быть как функциями от атрибутов сущности со сложным поведением, так и быть ортогональны им за счет добавления новой информации. Например, поведение стека [24] зависит от того, в каком управляющем состоянии находится стек. В качестве его управляющих состояний можно выделить: «стек пуст», «стек полон» и «стек не пуст и

не полон». В свою очередь, эти управляющие состояния являются функциями от таких атрибутов стека как текущий размер и вместимость.

Примером управляющих состояний, которые добавляют новую информацию, может служить автоответчик. Управляющими состояниями в этом случае: «автоответчик отключен», «автоответчик включен» и «настраивается фраза приветствия для автоответчика». Вычислительные состояния сущности определяются наличием входящего звонка и тем, происходит ли запись на автоответчик. Однако по этим вычислительным состояниям невозможно определить в каком из управляющих состояний находится автоответчик.

3.2.3. Автоматизированные классы

Описание сущностей со сложным поведением может быть разделено на описание логики и семантики. Описание логики включает в себя управляющие состояния, переходы между этими состояниями и выполняется с помощью конечного автомата. Также в логическую часть описания сущности со сложным поведением входят реакции на входные воздействия, которые зависят не только входного воздействия, но и от текущего состояния.

Семантическое описание включает в себя набор вычислительных состояний (атрибутов) и действия, которые может выполнять сущность. С точки зрения объектно-ориентированного подхода, семантическая часть описывается классом. Вычислительные состояния при этом задаются атрибутами этого класса, а действия – командами класса.

Логическая часть сущности со сложным поведением называется контроллером, а семантическая – объектом управления. Объект управления, интегрированный вместе с контроллером в единую сущность, называется автоматизированным объектом управления [8]. Реализацию автоматизированного объекта управления в терминах объектно-

ориентированного программирования будем называть *автоматизированным классом*.

Важным вопросом при проектировании автоматизированных классов является выбор набора управляющих состояний. Отметим, что число управляющих состояний напрямую связано с уровнем абстракции автоматизированного класса. Слишком высокий уровень абстракции (небольшое число управляющих состояний) ведет к усложнению самого объекта управления и уменьшает преимущества, получаемые от использования автоматного программирования. В случае низкого уровня абстракции (большое число управляющих состояний) значительно усложняется контроллер, что, в свою очередь, усложняет его проектирование и поддержку.

Обратим внимание, что автоматизированные классы остаются обычными объектами для своих клиентов. Это достигается за счет инкапсуляции автоматной реализации логики класса. Данное свойство играет большую роль при интеграции в существующие объектно-ориентированные системы, так как позволяет использовать автоматизированные классы без перепроектирования всей системы в автоматном стиле. При этом методы автоматизированного класса могут вызываться также как методы обычного класса. В этом случае выполняемые действия и возвращаемые значения зависят от текущего состояния контроллера автоматизированного класса.

Отметим также, что автоматизированные классы, в отличие от большинства других случаев использования автоматного программирования [6, 7], являются пассивными – все действия выполняются после входных воздействий. Автомат, реализующий контроллер автоматизированного класса, базируется на автомате Мили, в котором действия зависят не только состояния, в котором находится автомат, но и от входных воздействий.

3.2.4. Проектирование автоматизированных классов

Как уже было отмечено, одной из наиболее важных задач при использовании автоматного программирования является интеграция автоматных фрагментов в существующие объектно-ориентированные системы без полного перепроектирования этих систем. Для решения этой задачи автоматная реализация логики классов должна возникать только на поздней стадии проектирования и не должна выходить за рамки отдельных модулей. Рассмотрим далее основные этапы проектирования автоматизированных классов.

- На первом этапе выполняется декомпозиция решаемой задачи. Части исходной задачи реализуются с помощью набора взаимодействующих объектов.
- На втором этапе из полученных объектов выделяются сущности со сложным поведением, которые будут реализованы с помощью автоматного подхода (автоматизированных классов). Остальные сущности проектируются и программируются с помощью стандартного объектно-ориентированного подхода.
- На третьем этапе для каждой сущности со сложным поведением определяется набор управляющих состояний и переходов между ними, а также условия и действия на переходах. Один из наиболее удобных видов описания сущностей со сложным поведением – диаграмма состояний. Напомним, что выбор управляющих состояний определяется уровнем абстракции системы.
- На последнем этапе реализуется объект управления, который предоставляет запросы и команды. Запросы используются в условиях на переходах, а команды – в качестве действий, выполняемых при

осуществлении переходов.

3.2.5. Архитектура библиотеки

Рассмотрим основные классы библиотеки *EiffelState*, их устройство и способы взаимодействия. Как видно из диаграммы классов, представленной на рис. 7, в основе библиотеки лежат четыре класса: *AUTOMATED*, *STATE*, *STATE_DEPENDENT_FUNCTION* и *STATE_DEPENDENT_PROCEDURE*.

Эти классы доступны клиентам библиотеки и позволяют построить на своей основе автоматизированный класс, который бы реализовывал поведение в конкретной задаче. Остановимся подробнее на устройстве и назначении каждого из этих классов.

Класс *AUTOMATED* является абстрактным базовым классом для всех автоматизированных классов, которые разрабатываются на основе библиотеки *EiffelState*. Этот класс предоставляет средства для хранения текущего состояния объекта управления, а также вспомогательные методы для проверки доступности вызываемых методов в текущем состоянии. Таким образом, класс *AUTOMATED* скорее является *маркировочным* классом – наследование от этого класса сигнализирует о том, что класс потомок является автоматизированным классом.

Класс *STATE* является вспомогательным классом, который описывает состояние системы. В библиотечной версии этого класса описание состояния сводится к его имени, однако, отметим, что при разработке систем, которые используют библиотеку *EiffelState*, данный класс может быть расширен дополнительной информацией при помощи механизма наследования.

Два класса, которые используются при описании поведения, *STATE_DEPENDENT_FUNCTION* и *STATE_DEPENDENT_PROCEDURE* имеют схожее назначение. Рассмотрим подробнее функциональность, которая

предоставляется этими классами.

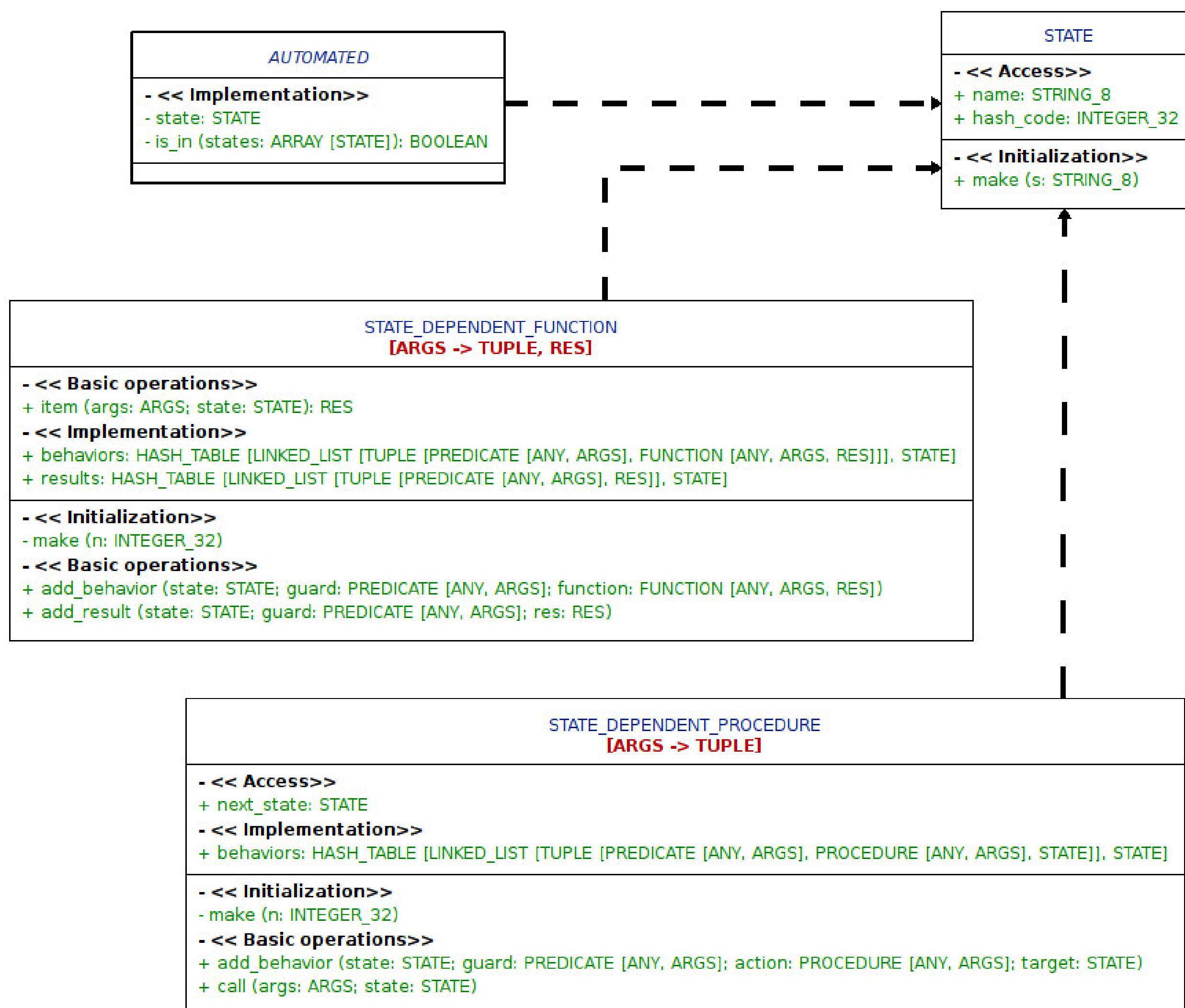

Рис. 7. Диаграмма основных классов библиотеки *EiffelState*

Эти классы используются для описания действий и условий на переходах (*STATE_DEPENDENT_PROCEDURE*), а также для описания функций, результат которых зависит от текущего состояния автоматизированного класса (*STATE_DEPENDENT_FUNCTION*). Напомним, что методы, которые

вызываются на переходах, не возвращают результат, в то время как функции, которые возвращают результат, не изменяют текущее состояние автоматизированного класса.

Оба класса имеют метод *add_behavior*, который позволяет определить поведение функции или процедуры, в зависимости от текущего состояния. Входными параметрами для этого метода являются:

- состояние автоматизированного класса, которое определяет, какое поведение будет иметь процедура (функция);
- предикат, выполнение которого также определяет поведение процедуры (функции). Это позволяет задавать различное поведение для одного состояния;
- процедура (в случае *STATE_DEPENDENT_PROCEDURE*) или функция (в случае *STATE_DEPENDENT_FUNCTION*), которая будет выполняться в указанном состоянии и при условии выполнения указанного предиката;
- также в случае *STATE_DEPENDENT_PROCEDURE* передается еще один параметр – новое состояние системы.

Вызов процедур и функций, поведение которых зависит от состояния, выполняется аналогично подходу, традиционному для языка программирования *Eiffel*.

Таким образом, для получения результата функции вызывается метод *item* класса *STATE_DEPENDENT_FUNCTION*. Аргументами этого метода являются текущее состояние и набор параметров, которые необходимо передать выполняемой функции. Также эти аргументы будут использованы для вычисления предикатов. После вызова метода *item* выполняется поиск всех функций, добавленных с помощью метода *add_behavior*, которые определены для данного состояния. После этого из них выбирается та функция, у которой

выполняется ассоциированный с ней предикат. У этой функции, в свою очередь, вызывается метод *item*, и результат вызова этого метода возвращается в клиенту *STATE_DEPENDENT_FUNCTION*.

Вызов процедур осуществляется аналогично, с той разницей, что для вызова самой процедуры используется метод *call*, а новое состояние автоматизированного класса можно получить с помощью *next_state*. Отметим, что выбор процедуры, которая будет реально выполнена, для класса *STATE_DEPENDENT_PROCEDURE* производится также как и выбор функции в случае класса *STATE_DEPENDENT_FUNCTION*.

Для удобства и улучшения производительности в классе *STATE_DEPENDENT_FUNCTION* есть возможность указать непосредственный результат выполнения для некоторого состояния и предиката. Таким образом можно избежать дополнительных вычислений, если результат функции в этом состоянии и при указанных аргументах заранее известен.

Перечислим теперь основные этапы разработки автоматизированного класса на основе библиотеки *EiffelState*:

- создается класс, который является наследником класса *AUTOMATED*;
- далее создаются экземпляры класса *STATE*, которые соответствуют управляющим состояниям, выбранным на этапе проектирования класса;
- для каждого действия, выделенного на этапе проектирования, создаются экземпляры *STATE_DEPENDENT_PROCEDURE*. Переходы и условия на них задаются с помощью метода *add_behavior*;
- для каждой функции, результат которой зависит от состояния, создаются экземпляры *STATE_DEPENDENT_FUNCTION*;
- создаются методы класса, которые будут составлять его интерфейс и инкапсулировать его автоматную реализацию.

3.2.6. Вопросы реализации

При создании библиотеки *EiffelState* был решен ряд вопросов, связанных с реализацией компонент библиотеки. Остановимся подробнее на некоторых из этих вопросах и их решении.

В первую очередь встает вопрос о возможности совмещения действий, которые зависят от текущего состояния автоматизированного класса, и действий, логика которых не зависит от состояния. Возможными вариантами решения являются либо запрет независящих от состояния действий, либо разрешение таких действий. С одной стороны, независящие от состояния действия могут быть реализованы с помощью тех же механизмов, что и зависящие – действие может быть добавлено для каждого состояния с условием, которое всегда выполняется. Однако, такой подход может оказаться неудобным по двум причинам – во-первых, диаграмма состояний будет загромождаться излишними петлями (независящие от состояния действия не меняют состояние автоматизированного класса), а во-вторых, для описания такого действия вручную потребуется значительное количество однотипного кода. Поэтому было принято решение разделить все действия автоматизированного класса на зависящие и независящие от состояния. При этом действия, независящие от состояния, реализуются как методы обычного класса.

Второй существенный вопрос – описание условий на переходах. Традиционное решение – представление условия в виде булевого выражения, переменными в котором являются значения полей объектов управления. Однако, при использовании библиотеки вручную такой подход может показаться искусственным. Кроме того, в отличие от традиционного подхода описывающий поведение автомат ограничен одним автоматизированным классом. Поэтому в качестве условий на переходах было решено использовать произвольные

методы, возвращающие значение типа *BOOLEAN*. Это реализовано в библиотеке с помощью возможности языка *Eiffel* передавать функцию в качестве аргумента метода при помощи механизма агентов. Входными параметрами для этих условий являются аргументы действия на переходе. Для передачи произвольного числа именованных параметров был использован механизм *TUPLE* классов. Отметим, что если для вычисления условия требуются значения большего числа переменных, нежели для выполнения действия, то экземпляр *TUPLE*-класса может содержать дополнительные параметры, которые будут игнорироваться действием, но будут использованы при вычислении условия.

Для представления действий, которые выполняются при переходах, также существует несколько вариантов. Действие может быть представлено в виде воздействия на объект управления, последовательности таких воздействий или иметь более общую структуру и представлять собой последовательность произвольных действий, в том числе и над объектом управления. С одной стороны, хотелось бы сопоставить одно действие на переходе с одним воздействием на объект управления. С другой стороны, такой подход либо имеет недостаточную выразительность, либо ведет к усложнению возможных воздействий на объект управления. В текущей реализации библиотеки *EiffelState* действие на переходе является агентом. Таким образом, имеет место подход, в котором действие на переходе является последовательностью произвольных действий. Отметим, что воздействие на объект управления может быть использовано как агент. Это позволяет не создавать дополнительных методов оберток, если действие на переходе совпадает с единичным воздействием на объект управления.

В связи с тем, что действия, которые выполняются на переходах, не возвращают никаких значений, за исключением нового состояния автоматизированного класса, встает вопрос о возможности вызова

возвращающих значение функций. Отметим, что если поведение функции не зависит от состояния автоматизированного класса, то она реализуется как обычный возвращающий значение метод класса в рамках традиционного объектно-ориентированного подхода. Функции, значение которых зависит от текущего состояния автоматизированного класса, предлагается реализовывать с помощью класса библиотеки *STATE_DEPENDENT_FUNCTION*. Напомним, что этот класс позволяет задавать реально вызывающуюся функцию для любой пары состояние-условие. Также отметим, что данные функции являются запросами и не изменяют текущее состояние автоматизированного класса. При этом несколько подряд идущих вызовов функции возвращают одно и тоже значение. Аргументы, используемые в условиях при выборе поведения, также как и в случае условий на переходах, совпадают с аргументами, которые передаются в саму функцию.

Действия и запросы, поведение которых зависит от состояния, могут быть не определены в некоторых состояниях системы. Поэтому встает вопрос, каким должно быть поведение автоматизированного класса при вызове действия (или запроса) в состоянии, в котором поведение действия (запроса) не определено. Рассмотрим три возможных варианта поведения.

Первым и простейший вариант – игнорирование таких действий (система не изменяет состояние и не выполняет никаких реальных действий). Однако, при таком решении вопроса возникает две проблемы: невозможность проигнорировать запрос, так как он должен вернуть результат; возможное не прогнозируемое клиентом поведение автоматизированного класса. Таким образом, данное решение не является подходящим.

Второй вариант поведения автоматизированного класса при вызове метода в состоянии, на котором оно не определено, – переход в специальное состояние *ERROR*. Однако, однажды произведенный ошибочный вызов метода класса

ведет к тому, что этот экземпляр автоматизированного класса больше невозможно использовать, без дополнительных действий. Такое поведение неудобно, так как возникает необходимость отслеживать, не перешел ли автоматизированный класс в состояние *ERROR,* и решать каким образом выводить его из этого состояния.

В текущей версии библиотеки реализован третий вариант поведения – в случае ошибочного вызова метода при проверке предусловий методов *item* и *call* генерируется исключение, которое может быть обработано клиентом. При этом автоматизированный класс останется в том же состоянии, в котором он был перед ошибочным вызовом.

3.2.7. Наследование автоматизированных классов

Определим, какие изменения могут быть внесены при наследовании автоматизированного класса.

1. Могут быть переопределены или добавлены новые процедуры или функции, поведение которых не зависит от состояния**.** В данном случае никаких отличий от аналогичных действий при наследовании обычных классов нет.
2. Могут быть добавлены новые процедуры или функции, поведение которых зависит от состояния**.** Тот факт, что класс, в который добавляются новые процедуры или функции, является наследуемым, в данной ситуации никак не влияет на процесс.
3. Могут быть изменены существующие процедуры или функции, поведение которых зависит от состояния. Без ограничения общности можно рассмотреть только процесс изменения существующей процедуры, так как для функции действия аналогичны.

Рассмотрим подробнее возможные изменения, которые могут быть

внесены в процедуру, поведение которой зависит от состояния, при наследовании автоматизированного класса. Изменения могут быть трех видов:

- добавление нового перехода;
- изменение условия на существующем переходе;
- изменение выполняемого при переходе действия.

Остановимся подробнее на добавлении нового перехода. Переход из нового состояния добавляется вызовом метода *add_behavior* и не порождает новых задач. В случае *переходов из состояний, для которых уже существуют переходы,* может возникнуть неоднозначность при выборе перехода. Поясним это на примере. Пусть из состояния *S1* есть два перехода — в состояние *S2* при ($x > 0$) и в состояние *S3* при ($x <= 0$) (рис. 8).

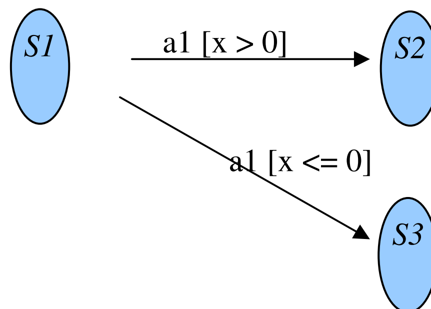

Рис. 8. ***Диаграмма состояний родительского класса***

Пусть, в наследуемом классе была добавлена еще одна переменная — y, а также переход в состояние *S4* при ($y >0$ && $x > 10$) (рис. 9).

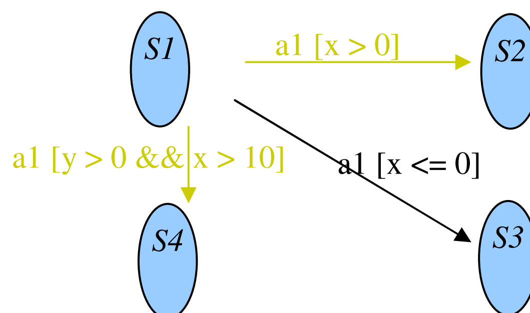

Рис. 9. ***Диаграмма состояний дочернего класса***

Тогда при ($y = 1$) и ($x = 11$) выполняются условия для переходов и в *S2*, и в *S4*. Таким образом, необходимо разрешить конфликт неоднозначности выбора

перехода.

Эта неоднозначность может быть устранена расстановкой приоритетов переходов. Рассмотрим механизм выбора перехода при наличии приоритетов. Пусть имеется список переходов из текущего состояния *S1*, которые отсортированы по приоритету в порядке убывания. Тогда для выполнения выбирается первый переход, условие на котором выполняется.

Остановимся подробнее на возможных реализациях расстановки приоритетов у переходов. Первый вариант реализации предусматривает ручное задание приоритетов переходов на этапе их добавления. При этом приоритет задается некоторым числом. Такой вариант задания приоритетов позволяет расставить переходы, добавленные в дочерних классах, в любом порядке – как до родительских, так и после них. Однако, задание приоритетов в ручную вынуждает пользователя библиотеки писать дополнительный код, а также может не решить задачу неоднозначности выбора переходов, если двум переходам будет назначен одинаковый приоритет.

Второй вариант – назначение более высокого приоритета вновь добавленным переходам. Таким образом, переходы, которые были добавлены в дочерних классах, будут обрабатываться до переходов, которые были объявлены в родительском классе. В этом случае более высокий приоритет предлагается давать переходам из дочерних классов, так как в их условиях могут использоваться переменные, отсутствовавшие в родительском классе. В библиотеке *EiffelState* реализован второй вариант назначения приоритетов.

3.3. Визуальное проектирование системы

Как уже было отмечено, наиболее удобным способом описания поведения программ являются диаграммы автоматов. В рамках библиотеки *EiffelState* были разработаны алгоритм перехода от диаграммы автоматов к коду, который

реализует автоматизированный класс с соответствующим поведением, и алгоритм построения диаграммы автоматов по исходному коду автоматизированного класса, написанного с помощью библиотеки *EiffelState*.

Отметим, что таким образом достигается двухстороннее преобразование диаграмм автоматов в исходный код и обратно без потерь дополнительной информации, доступной только в одном из представлений. Для исходного кода такой информацией является реализация поведения процедур и функций, для диаграмм – расположение состояний и переходов.

3.3.1. Преобразование диаграммы автоматов в исходный код

Рассмотрим поэтапно алгоритм построения исходного кода по диаграмме автоматов.

1. Выделяется список состояний, представленных на диаграмме. Для каждого состояния заводится переменная типа *STATE*.
2. Выделяется список процедур, поведение которых зависит от состояния, представленных на диаграмме. Для каждой процедуры выполняются следующие действия:
   - 2.1. Выделяется список переходов с условиями, в которых задействованы процедуры.
   - 2.2. Для каждой процедуры в коде описывается переменная типа *STATE_DEPENDENT_PROCEDURE*.
   - 2.3. Переменная инициализируется в конструкторе класса и для каждого перехода задается поведение – в код добавляется вызов метода *add_behavior*. Если поведение было представлено на диаграмме, как вызов методов других класса, то оно кодируется как встроенный (*inline*) агент, в противном случае – создается дополнительный метод класса. Реализация этого метода остается

за разработчиком.

Рассмотрим, как изменится алгоритм в случае, если исходный код для данного автоматизированного класса уже существовал. На шаге 2.2. перед созданием новой переменной проверяется, что процедуры с таким именем еще не существует. Изменится также и шаг 2.3 – вызовы метода *add_behavior* для переходов или с условиями, которые отсутствовали на диаграмме состояний, удаляются. Для оставшихся методов обновляется поведение в соответствии с указаниями на диаграмме. Для новых переходов действия аналогичны действиям для нового класса. Аналогичным образом обрабатываются и состояния.

3.3.2. Преобразование исходного кода в диаграмму автоматов

Этапы преобразования исходного кода автоматизированного класса в диаграмму автоматов во многом схожи с преобразованием, описанным в разд. 3.3.1. Рассмотрим их подробнее.

1. Производится синтаксический разбор исходного кода реализации автоматизированного класса.
2. Выделяются все переменные типа *STATE*. Для каждой переменной создается соответствующее состояние на диаграмме.
3. Выделяются все переменные типа *STATE_DEPENDENT_PROCEDURE*. Для каждой переменной выполняется поиск всех вызовов метода *add_behvior*. Каждому вызову на диаграмме будет соответствовать переход. Начальное и конечное состояния, условие на переходе и выполняемое действие получаются из аргументов передаваемых в метод.

Рассмотрим, как изменится алгоритм в случае, если диаграмма автоматов для данного автоматизированного класса уже существовала.

На втором этапе алгоритма создаются только те состояния, которых еще нет на диаграмме. При этом состояния, которые присутствуют только на диаграмме, удаляются.

Поведение при создании переходов аналогично. Пусть требуется добавить переход из состояния *S1* в состояние *S2* при условии *cond.* Если такой переход уже существует, то обновляется действие, выполняемое при этом переходе, в противном случае будет создан новый переход. Все переходы на диаграмме автоматов, которые не обновлены на третьем этапе, удаляются.

Выводы по главе 3

1. Проведен обзор языка программирование *Eiffel*.
2. Приведены основные принципы *ААТД*.
3. Введено понятие автоматизированного класса, как реализации *ААТД*.
4. Рассмотрены основные этапы проектирования автоматизированных классов.
5. Описана архитектура библиотеки *EiffelState*, которая реализует работу с автоматизированными классами на языке *Eiffel*.
6. Рассмотрены некоторые детали реализации библиотеки *EiffelState*.
7. Решена задача выбора перехода при наследовании автоматизированных классов.
8. Приведены алгоритмы перехода от исходных кодов к диаграммам автоматов и обратно.

ГЛАВА 4. ПРИМЕР ИСПОЛЬЗОВАНИЯ БИБЛИОТЕКИ

4.1. Описание задачи

В качестве примера программной системы, реализованный при помощи библиотеки *EiffelState*, рассмотрим игру «крестики-нолики» для двух человек. При этом в начале реализуем игру в терминах традиционного объектно-ориентированного программирования. Далее покажем, что автоматизированные классы могут быть интегрированы в существующий программный продукт без смены архитектуры. Конечным результатом пошаговой замены классов со сложным поведением на автоматизированные классы станет реализация игры «крестики-нолики» на основании библиотеки *EiffelState*.

«Крестики-нолики» – это игра для двух человек, в которой каждый из игроков по очереди ставит крестик (нолик) на поле размером три на три. Существуют разновидности игры с другими размерами поля, в том числе и неограниченным по размерам полем. Однако в данной главе рассматривается самая простая версия. Это позволит сосредоточить внимание на архитектуре программы, а не на реализации механизма игры.

Цель игры – выставить три крестика (нолика) в ряд, столбик или по диагонали. Если на поле не осталось свободных клеток, а ни один из игроков не выполнил условие, необходимое для победы, игра считается окончившейся вничью.

Программная реализация игры «крестики-нолики» представляет собой приложение с девятью кнопками. Каждая кнопка соответствует клетке поля, а нажатие на кнопку – установке крестика или нолика, в зависимости от очередности хода. По окончанию игры предлагается начать новую партию. При этом первым будет ходить тот игрок, кто выиграл последнюю партию.

4.2. Решение задачи в терминах объектно-ориентированного программирования

Как уже было отмечено, для демонстрации простоты интеграции автоматизированных классов в существующие системы реализуется игра «крестики-нолики» с помощью традиционного объектно-ориентированного программирования, а затем в реализации некоторые из классов заменяются на автоматизированные.

В этой секции дано описание «традиционной» версии, которая была реализована во время написания данной работы. На рис. 10 изображена диаграмма классов этой реализации. Приложение реализуется с использованием паттерна проектирования *Model-View-Controller* (*MVC*) [24]. Напомним, что архитектура приложения, которое основывается на паттерне *MVC*, подразумевает строгое разделение данных, их представления и управления.

Рассмотрим подробнее основные классы данной программной системы.

- Запуск приложения начинается с корневого класса *APPLICATION*. В нем выполняется подготовка и запуск графического пользовательского интерфейса.
- Интерфейс пользователя реализуется в классах *INTERFACE_NAMES* и *MAIN_WINDOW*. Они и образуют часть *MVC*-системы, отвечающей за представление данных. При этом класс *INTERFACE_NAMES* содержит основные ресурсы приложения – названия окон, текст диалоговых окон и т. п.

  Класс *MAIN_WINDOW* содержит игровое поле, которое представляет собой девять кнопок, расположенных квадратом со стороной три.
- Класс *FIELD_CELL* отвечает за хранение состояние клетки поля.

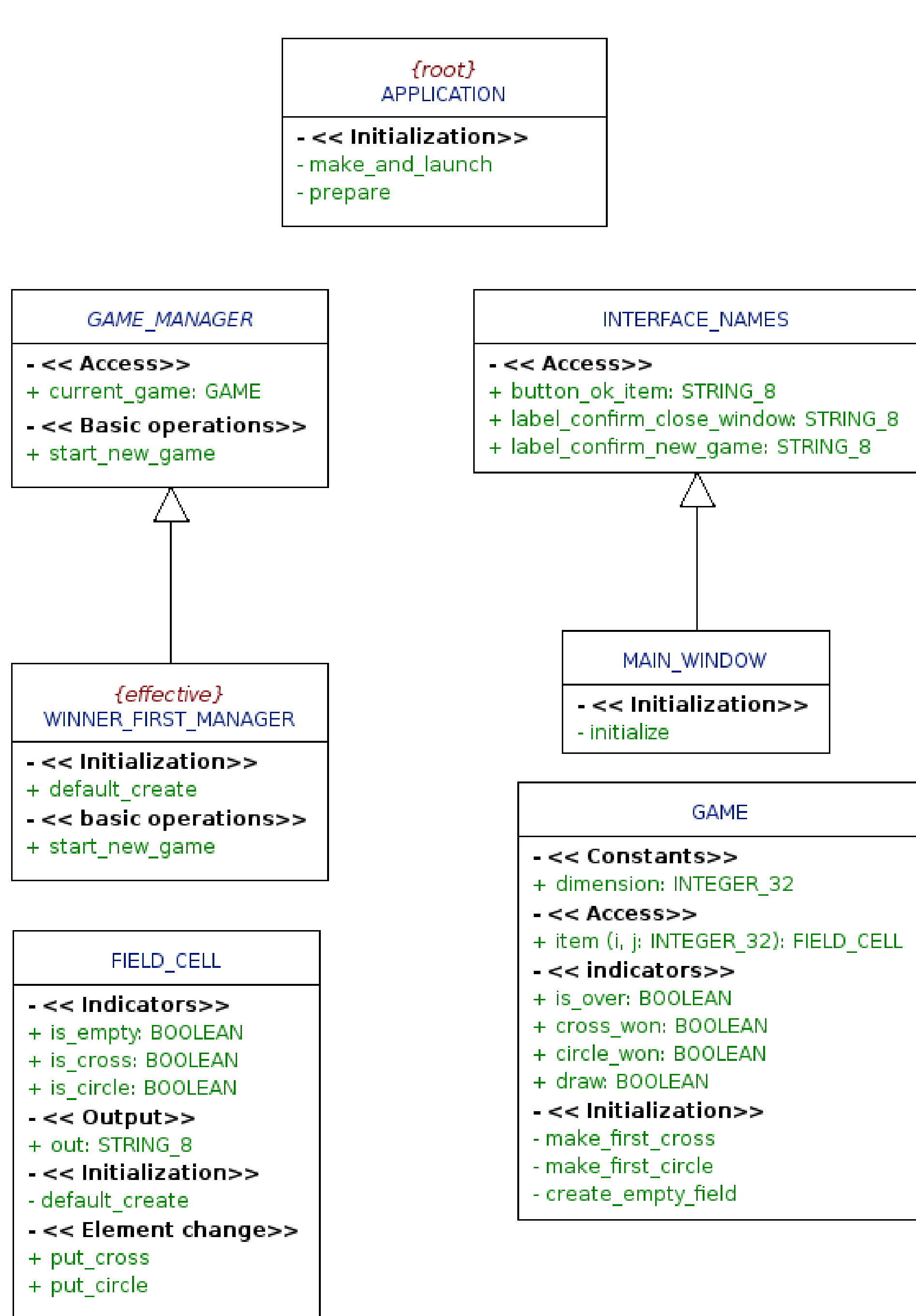

Рис. 10. ***Диаграмма классов традиционной реализации игры «крестики-нолики»***

- Абстрактный класс *GAME_MANAGER* объявляет операции, предназначенные для управления программной системой между партиями игры. Примером такого управления является выбор права первого хода. Реализацией этого класса является *WINNER_FIRST_MANAGER*. В этой реализации первым ходит игрок,

который выиграл последнюю партию.

- Класс *GAME*, как и класс *GAME_MANAGER*, относится к управляющей части программной системы и реализует управление в процессе игровой партии. В данном классе осуществляются все проверки, связанные с возможностью хода и окончанием игры.

4.3. Решение задачи с использованием автоматизированных классов

Рассмотрим теперь решение этой задачи в терминах автоматизированных классов и с использованием созданной библиотеки *EiffelState*. Построение решения будем выполнять в соответствии с шагами, описанными в разд. 3.2.4.

Первый этап (декомпозиция задачи) уже выполнен при решении задачи с помощью традиционного объектно-ориентированного подхода.

На втором этапе выделяются сущности со сложным поведением. В случае нашей реализации игры «крестики-нолики» — *FIELD_CELL*, *GAME_MANAGER* и *GAME*. Реализация остальных сущностей может быть взята из разд. 4.2.

Третий этап включает в себя проектирование поведения сущностей со сложным поведением – выделение управляющих состояний и определение переходов между ними. Выполним эти действия для каждой сущности.

Автоматизированный класс *FIELD_CELL*, реализующий соответствующую сущность, имеет три управляющих состояния: *Empty* – клетка пуста, *Cross* – клетка содержит крестик, *Circle* – клетка содержит нолик. Переходы между состояниями выполняются при вызове двух процедур: *put_cross* и *put_circle*. Диаграмма состояний автоматизированного класса *FIELD_CELL* представлена на рис. 11.

Р

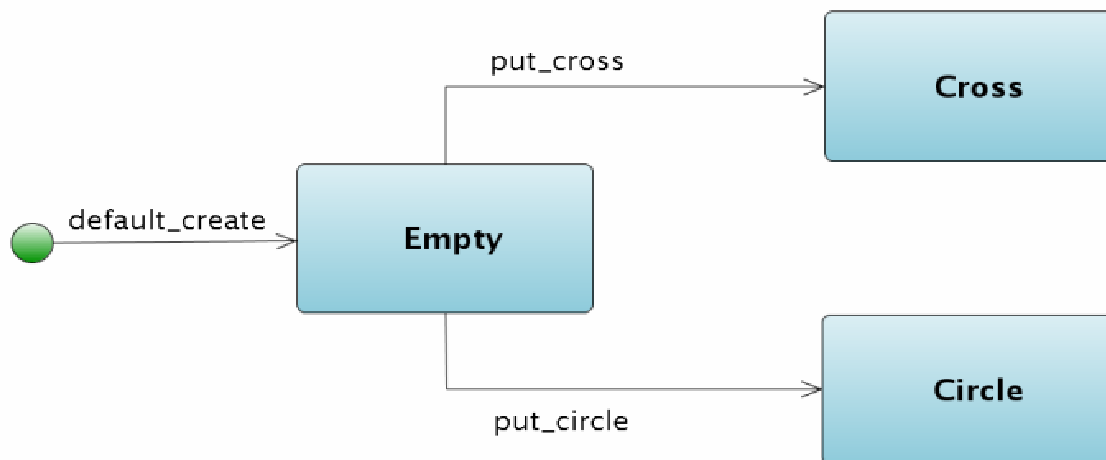

Рис. 11. ***Диаграмма состояний автоматизированного класса FIELD_CELL***

Автоматизированный класс *GAME_MANAGER* имеет всего два управляющих состояния: *First_turn_cross* и *First_turn_circle*. Нахождение в этих состояниях определяет право первого хода для крестиков и ноликов соответственно. Переход между состояниями происходит при начале новой игры и вызове процедуры *start_new_game*. Так как класс *GAME_MANAGER* абстрактный, то точная реализация переходов задается его наследниками, и в данном случае классом *WINNER_FIRST_MANAGER* (рис. 12).

Автоматизированный класс *GAME* управляет логикой партии игры и содержит состояния, которые описывают как очередность хода, так и состояние самой партии. Этими состояниями являются: *Cross_turn* – ход крестиков, *Circle_turn* – ход ноликов, *Cross_win* – победа крестиков, *Circle_win* – победа ноликов и *Draw* – ничья. Переходы между состояниями осуществляются при вызове процедуры *make_turn* (рис. 13).

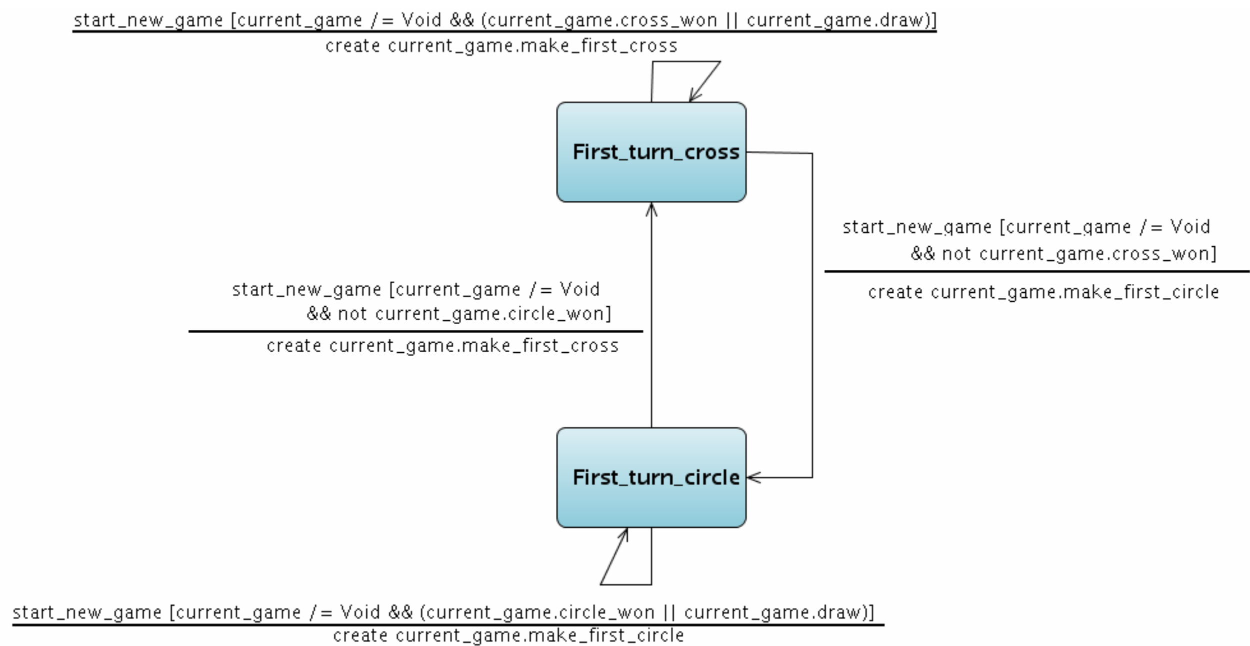

Рис. 12. ***Диаграмма состояний автоматизированного класса WINNER_FIRST_MANAGER***

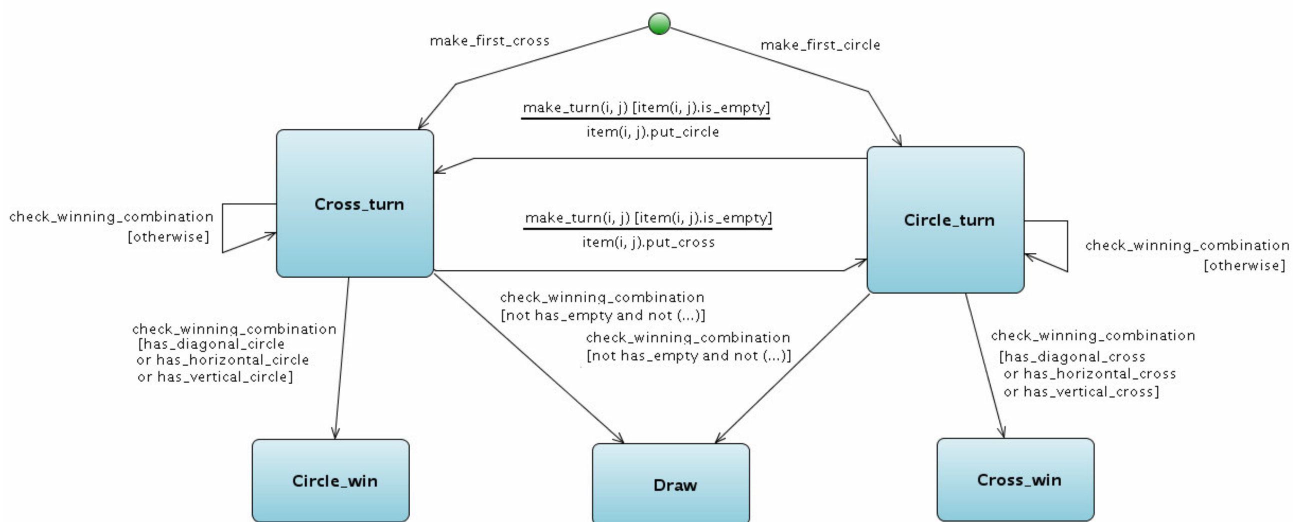

Рис. 13. ***Диаграмма состояний автоматизированного класса GAME***

Отметим, что интерфейс автоматизированных классов полностью совпадает с интерфейсом одноименных классов из «традиционной» реализации (рис. 14). Таким образом, автоматная реализация логики этих

классов полностью инкапсулирована, и они могут заменить соответствующие классы из «традиционной» реализации без изменения остальных классов программной системы.

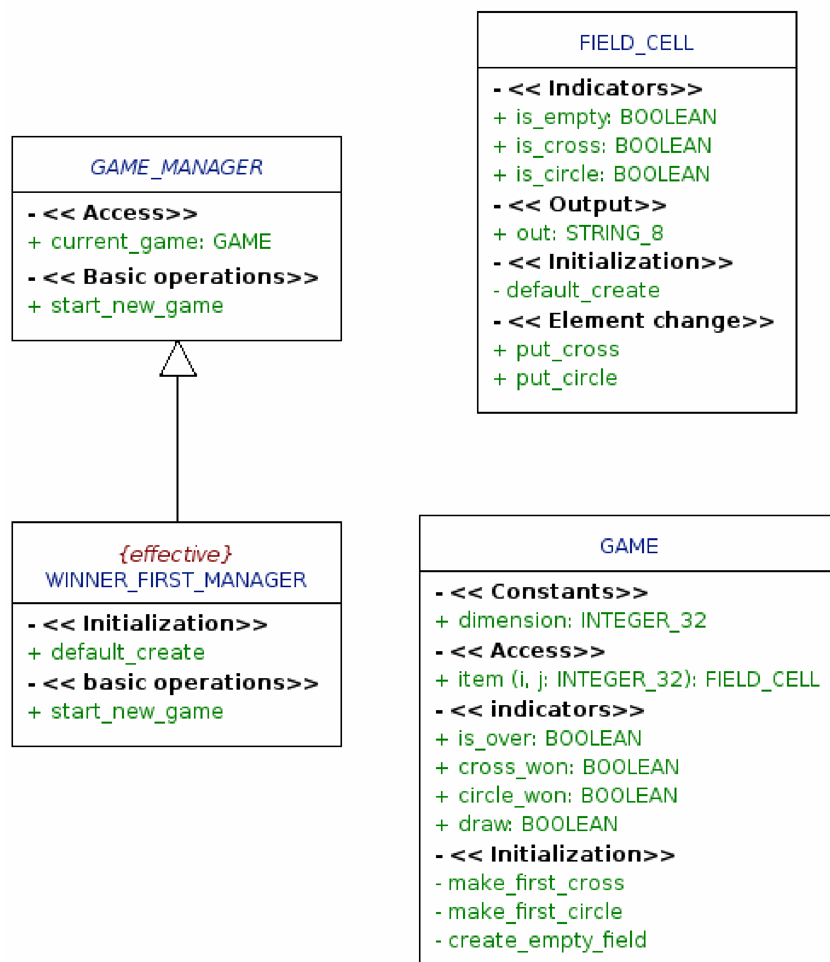

Рис. 14. ***Диаграмма автоматизированных классов***

Выводы по главе 4

1. Приведено описание пробной задачи, выбранной для демонстрации простоты интеграции автоматизированных классов в существующие программные системы.
2. Выполнена реализация пробной задачи в терминах объектно-ориентированного программирования.
3. Выполнена реализация пробной задачи в терминах автоматизированных классов. При этом интерфейсы автоматизированных классов совпадают с интерфейсами соответствующих классов «традиционной» реализации. Этот факт доказывает возможность легкой интеграции в существующие программные системы.

ЗАКЛЮЧЕНИЕ

В данной работе получены следующие результаты:

1. Выявлены основные проблемы в существующих подходах построения автоматных программ и их реализации.
2. Введено понятие автоматизированного класса, как реализации ААТД.
3. Разработан алгоритм проектирования автоматизированных классов.
4. Спроектирована и реализована на языке программирования *Eiffel* библиотека для работы с автоматизированными классами.
5. Решена задача наследования автоматизированных классов в рамках библиотеки *EiffelState*.
6. Разработаны алгоритмы для перехода от диаграммы автоматов к исходным кодам автоматизированного класса и обратно.
7. На примере продемонстрирована возможность легкой интеграции автоматизированных классов в существующие программные системы.

Возможны следующие направления развития работы:

1. Изучение вопроса верификации автоматизированных классов.
2. Разработка графического редактора диаграмм состояний, интегрированного в среду разработки *EiffelStudio*.

ИСТОЧНИКИ

1. *Гома Х.* UML. Проектирование систем реального времени, распределенных и параллельных приложений. М.: ДМК, 2002.
2. *Буч Г.*, *Рамбо Д.*, *Джекобсон А.* Язык UML. Руководство пользователя. М.: ДМК, 2000.
3. *Larman C.* Applying UML and Patterns. Prentice Hall PTR, 1997.
4. *Mellor S.*, *Scott K.*, *Uhl A.*, *Weise A.*, Introduction to Model Driven Architecture. Addison-Wesley, 2003.
5. *Гуров В. С.*, *Нарвский А. С.*, *Шалыто А. А.* Автоматизация проектирования событийных объектно-ориентированных программ с явным выделением состояний /Труды X Всероссийской научно-методической конференции «Телематика-2003». СПб.: СПбГИТМО (ТУ). 2003. http://tm.ifmo.ru
6. Веб-сайт проекта *UniMod*. http://unimod.sourceforge.net/
7. *Решетников Е. О.* Инструментальное средство для визуального проектирования автоматных программ на основе *Microsoft Domain-Specific Language Tools*. Бакалаврская работа. СПбГУ ИТМО. 2007. http://is.ifmo.ru/papers/reshetnikov_bachelor/
8. Поликарпова Н. И. Объектно-ориентированный подход к моделированию и спецификации сущностей со сложным поведением. Бакалаврская работа. СПбГУ ИТМО. 2006. http://is.ifmo.ru/papers/oosusch/
9. Поликарпова Н. И., Шалыто А. А. Автоматное программирование. СПб.: Питер, 2009.
10. *Хопкрофт Дж.*, *Мотвани Р.*, *Ульман Дж.* Введение в теорию автоматов, языков и вычислений. М.: Вильямс, 2008.
11. *Шалыто А. А.* SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998.

http://is.ifmo.ru/books/switch/1

12. Веб-сайт *Eiffel Software*. http://eiffel.com
13. Веб-сайт продукта *Rational Rose*. http://www-304.ibm.com/jct03001c/software/awdtools/developer/rose/
14. Веб-сайт продукта *Sybase PowerDesigner*. http://www.sybase.com/products/modelingmetadata/powerdesigner
15. Веб-сайт продукта *Enterprise Architect Professional*. http://www.sparxsystems.com/
16. Веб-сайт *Microsoft Windows Workflow Foundation*. http://msdn2.microsoft.com/en-us/library/ms735967.aspx
17. Гуров В. С., Мазин М. А., Нарвский А. С., Шалыто А. А. Инструментальное средство для поддержки автоматного программирования // Программирование. 2007. № 6, с. 65 – 80. http://is.ifmo.ru/works/_2008_01_27_gurov.pdf
18. *Kochelaev D. Y.*, *Khasanzyanov B. S.*, *Yaminov B. R.*, *Shalyto A. A.* Instrumetnal Tool for Automata Based Software Development UniMod 2 // Proceedings of the Second Spring Young Researchers' Colloquium on Software Engineering (SYRCoSE 2008). SPbSU. 2008. V. 1, pp. 55 – 58.
19. *Mellor S.*, *Balcer M.* Executable UML: A Foundation for Model-Driven Architecture. Addison-Wesley, 2002.
20. *Туккель Н. И.*, *Шалыто А. А.* Система управления танком для игры Robocode. Объектно-ориентированное программирование с явным выделением состояний. http://is.ifmo.ru/projects/tanks/
21. *Шалыто А. А.*, *Наумов Л. А.* Методы объектно-ориентированной реализации реактивных агентов на основе конечных автоматов. http://is.ifmo.ru/works/_aut_oop.pdf
22. *Мейер Б.* Объектно-ориентированное конструирование программных

систем. М.: Русская Редакция, 2005.

23. *Meyer B.*: Applying «Design by Contract» //Computer (IEEE), 25, 10, October 1992, pp. 40 – 51.

24. *Гамма Э.*, *Хелм Р.*, *Джонсон Р.*, *Влиссидес Дж.* Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб.: Питер, 2007.

ПРИЛОЖЕНИЯ

Приложение 1. Исходные коды основных классов библиотеки EiffelState

Класс *AUTOMATED*

```
indexing
     description: "Automated objects."
     author: "D. Kochelaev

deferred class
     AUTOMATED

feature {NONE} -- Implementation
state: STATE
          -- Current control state

is_in (states: ARRAY [STATE]): BOOLEAN is
          -- Indicator predicate
     do
          Result := states.has (state)
     ensure
          Result = (states.has (state))
     end
end
```

Класс *STATE*

```
indexing
    description: "Named control states."
    author: ""D. Kochelaev

class
    STATE

inherit
    HASHABLE

create
    make

feature -- Initialization
make (s: STRING) is
        -- Create a state with name `s'
    require
        s_exists: s /= Void
        s_non_empty: not s.is_empty
    do
        name := s
    ensure
        name_set: name = s
    end

feature -- Access
```

```
    name: STRING
            -- Name

    hash_code: INTEGER is
            -- Hash code
        do
            Result := name.hash_code
        end
    end
```

Класс *STATE_DEPENDENT_FUNCTION*

```
    indexing
        description: "Functions whose behavior depends on
the control state."
        author: "D. Kochelaev"

    class
        STATE_DEPENDENT_FUNCTION  [ARGS -> TUPLE, RES]

    create
        make

    feature {NONE} -- Initialization
    make (n: INTEGER) is
            -- Create a procedure valid for at least `n'
states
        require
```

```
			n_positive: n > 0
		do
			create behaviors.make (n)
			behaviors.compare_objects
			create results.make (n)
			results.compare_objects
		end

	feature -- Basic operations
	add_behavior (state: STATE; guard: PREDICATE [ANY,
ARGS]; function: FUNCTION [ANY, ARGS, RES]) is
			-- Make function return the result of
`function' when called in `state' and `guard' holds
		local
			list: LINKED_LIST [TUPLE [guard: PREDICATE
[ANY, ARGS]; function: FUNCTION [ANY, ARGS, RES]]]
		do
			behaviors.search (state)
			if behaviors.found then
				behaviors.found_item.put_front ([guard,
function])
			else
				create list.make
				list.put_front ([guard, function])
				behaviors.extend (list, state)
			end
		end
```

```
	add_result (state: STATE; guard: PREDICATE [ANY,
ARGS]; res: RES) is
			-- Make function return `r' when called in
`state' and `guard' holds
		local
			list: LINKED_LIST [TUPLE [guard: PREDICATE
[ANY, ARGS]; res: RES]]
		do
			results.search (state)
			if results.found then
				results.found_item.extend ([guard, res])
			else
				create list.make
				list.put_front ([guard, res])
				results.extend (list, state)
			end
		end

	item (args: ARGS; state: STATE): RES is
			-- Function result in `state' with `args'
(default value if no specific behavior defined)
		require
			results.has (state) or behaviors.has (state)
--check if function is defined in this state
		local
			found: BOOLEAN
```

```
        do
            results.search (state)
            if results.found then
                from
                    results.found_item.start
                until
                    results.found_item.after or found
                loop
                    if
results.found_item.item.guard.item (args) then
                        found := True
                        Result :=
results.found_item.item.res
                    end
                    results.found_item.forth
                end
            end
            behaviors.search (state)
            if behaviors.found and not found then
                from
                    behaviors.found_item.start
                until
                    behaviors.found_item.after or found
                loop
                     if
behaviors.found_item.item.guard.item (args) then
                         found := True
```

```
                          Result :=
behaviors.found_item.item.function.item (args)
                      end
                     behaviors.found_item.forth
                end
            end
        end

    feature -- Implementation
    behaviors: HASH_TABLE [LINKED_LIST [TUPLE [guard:
PREDICATE [ANY, ARGS]; function: FUNCTION [ANY, ARGS,
RES]]], STATE]
            -- Function behaviors in different states,
when different guards hold

    results: HASH_TABLE [LINKED_LIST [TUPLE [guard:
PREDICATE [ANY, ARGS]; res: RES]], STATE]
            -- Function results in different states, when
different guards hold

    invariant
        behaviors_exists: behaviors /= Void
        results_exists: results /= Void
    end
```

Класс *STATE_DEPENDENT_PROCEDURE*

```
    indexing
        description: "Procedures whose behavior depends
on the control state."
        author: ""D. Kochelaev

    class
        STATE_DEPENDENT_PROCEDURE [ARGS -> TUPLE]

    create
        make

    feature {NONE} -- Initialization
    make (n: INTEGER) is
            -- Create a procedure valid for at least `n'
states
        require
            n_positive: n > 0
        do
            create behaviors.make (n)
            behaviors.compare_objects
        end

    feature -- Access
    next_state: STATE
            -- State to transit to after calling
procedure
```

```
    feature -- Basic operations
    add_behavior (state: STATE; guard: PREDICATE [ANY,
ARGS]; action: PROCEDURE [ANY, ARGS]; target: STATE) is
            -- Make procedure execute `action' and
transit to `target' when called in `state' and `guard'
holds
        local
            list: LINKED_LIST [TUPLE [guard: PREDICATE
[ANY, ARGS]; action: PROCEDURE [ANY, ARGS]; target:
STATE]]
        do
            behaviors.search (state)
            if behaviors.found then
                behaviors.found_item.put_front ([guard,
action, target])
            else
                create list.make
                list.put_front ([guard, action, target])
                behaviors.extend (list, state)
            end
        end

    call (args: ARGS; state: STATE) is
            -- Call procedure in `state' with `args'
        require
            behaviors.has (state)
```

```
        local
            found: BOOLEAN
        do
            found := False
            next_state := state
            behaviors.search (state)
            if behaviors.found then
                from
                    behaviors.found_item.start
                until
                    behaviors.found_item.after or found
                loop
                     if
behaviors.found_item.item.guard.item (args) then
                         found := True

behaviors.found_item.item.action.call (args)
                         next_state :=
behaviors.found_item.item.target
                     end
                    behaviors.found_item.forth
                end
            end
        end

    feature -- Implementation
    behaviors: HASH_TABLE [LINKED_LIST [TUPLE [guard:
```

```
PREDICATE [ANY, ARGS]; action: PROCEDURE [ANY, ARGS];
target: STATE]], STATE]
        -- Procedure behaviors in different states, when
different guards hold

    invariant
        behaviors_exists: behaviors /= Void
    end
```

Приложение 2. Исходные коды традиционной реализации игры «крестики-нолики»

Класс *APPLICATION*

```
    indexing
    description  : "Root class for this application."
    author       : "D. Kochelaev"

    class
        APPLICATION

    inherit
        EV_APPLICATION

    create
        make_and_launch

    feature {NONE} -- Initialization
```

```
    make_and_launch is
            -- Initialize and launch application
        do
            default_create
            prepare
            launch
        end

    prepare is
            -- Prepare the first window to be displayed.
            -- Perform one call to first window in order
to
            -- avoid to violate the invariant of class
EV_APPLICATION.
        do
                -- create and initialize the first
window.
            create first_window

                -- Show the first window.
                --| TODO: Remove this line if you don't
want the first
                --|       window to be shown at the
start of the program.
            first_window.show
        end
```

```
feature {NONE} -- Implementation

first_window: MAIN_WINDOW
        -- Main window.

end -- class APPLICATION
```

Класс *FIELD_CELL*

```
indexing
description: "Cells of the game field."
author: ""

class
    FIELD_CELL

inherit
    AUTOMATED
    redefine
        default_create,
        out
    end

feature {NONE} -- Initialization
default_create is
        -- Create an empty cell
    do
        state := Empty
```

```
        ensure then
                is_empty: is_empty
        end

feature -- State dependent: Element change
put_cross is
                -- Put cross into the cell
        do
                sd_put_cross.call([], state)
                state := sd_put_cross.next_state
        end

put_circle is
                -- Put circle into the cell
        do
                sd_put_circle.call([], state)
                state := sd_put_circle.next_state
        end

feature -- State dependent: Indicators
is_empty: BOOLEAN is
                -- Is the cell empty?
        do
                Result := is_in (<<Empty>>)
        end

is_cross: BOOLEAN is
```

```
            -- Is the cell cross?
        do
            Result := is_in (<<Cross>>)
        end

    is_circle: BOOLEAN is
            -- Is the cell circle?
        do
            Result := is_in (<<Circle>>)
        end

    feature -- State dependent: Output
    out: STRING is
            -- String representation of the cell
        do
            Result := sd_out.item ([], state)
        end

    feature {NONE} -- Automaton
    Empty: STATE is once create Result.make ("Empty") end

    Cross: STATE is once create Result.make ("Cross") end

    Circle: STATE is once create Result.make ("Circle")
end

    sd_put_cross: STATE_DEPENDENT_PROCEDURE [TUPLE] is
```

```
            -- State dependent procedure for `put_cross'
        once
            create Result.make (1)
            Result.add_behavior (Empty,
                agent : BOOLEAN do Result := True end,
                agent do_nothing,
                Cross)
        end

    sd_put_circle: STATE_DEPENDENT_PROCEDURE [TUPLE] is
            -- State dependent procedure for `put_circle'
        once
            create Result.make (1)
            Result.add_behavior (Empty,
                agent : BOOLEAN do Result := True end,
                agent do_nothing,
                Circle)
        end

    sd_out: STATE_DEPENDENT_FUNCTION [TUPLE, STRING] is
            -- State dependent function for `out'
        once
            create Result.make (3)
            Result.add_result (Empty, agent : BOOLEAN do
Result := True end, "")
            Result.add_result (Cross, agent : BOOLEAN do
Result := True end, "X")
```

```
            Result.add_result (Circle, agent : BOOLEAN do
Result := True end, "O")
        end



    end
```

Класс *GAME*

```
    indexing
        description: "Tic-Tac-Toe games."
        author: "D. Kochelaev"

    class
        GAME

    inherit
        AUTOMATED

    create
        make_first_cross,
        make_first_circle

    feature {NONE} -- Initialization
    make_first_cross is
            -- Create a new game with first turn of
crosses
        do
```

```
            create_empty_field
            build_sd_make_turn
            build_sd_check_winning_combination
            state := Cross_turn
        ensure
            field_empty: field.for_all (agent
{FIELD_CELL}.is_empty)
        end

    make_first_circle is
            -- Create a new game with first turn of
circles
        do
            create_empty_field
            build_sd_make_turn
            build_sd_check_winning_combination
            state := Circle_turn
        ensure
            field_empty: field.for_all (agent
{FIELD_CELL}.is_empty)
        end

    create_empty_field is
            -- Create field with empty cells
        local
            i, j: INTEGER
        do
```

```
            create field.make (Dimension, Dimension)
            from
                i := 1
            until
                i > Dimension
            loop
                from
                    j := 1
                until
                    j > Dimension
                loop
                    field.put (create {FIELD_CELL}, i,
j)
                    j := j + 1
                end
                i := i + 1
            end
        end

    feature -- Constants
    Dimension: INTEGER is 3
            -- Number of rows and columns on the field

    feature -- Access
    item (i, j: INTEGER): FIELD_CELL  is
            -- Value of (`i', `j') cell
        require
```

```
            i_not_too_small: i >= 1
            i_not_too_large: i <= Dimension
            j_not_too_small: j >= 1
            j_not_too_large: j <= Dimension
        do
            Result := field.item (i, j)
        end

    feature -- State-dependent: indicators
    is_over: BOOLEAN is
            -- Is game over?
        do
            Result := is_in (<<Cross_win, Circle_win,
Draw>>)
        end

    cross_won: BOOLEAN is
            -- Did the crosses win?
        do
            Result := is_in (<<Cross_win>>)
        end

    circle_won: BOOLEAN is
            -- Did the crosses win?
        do
            Result := is_in (<<Circle_win>>)
        end
```

```
feature -- State dependent: basic operations
make_turn (i, j: INTEGER) is
		-- Make turn, filling cell (`i', `j')
	require
		i_not_too_small: i >= 1
		i_not_too_large: i <= Dimension
		j_not_too_small: j >= 1
		j_not_too_large: j <= Dimension
	do
		sd_make_turn.call ([i, j], state)
		state := sd_make_turn.next_state
		sd_check_winning_combination.call ([], state)
		state :=
sd_check_winning_combination.next_state
	end

feature {NONE} -- Predicates
has_empty: BOOLEAN is
		-- Is there empty cell?
	local
		k, l:INTEGER
	do
		from
			k := 1
		until
			k > Dimension or Result
```

```
            loop
                from
                    l := 1
                until
                    l > Dimension or Result
                loop
                    Result := item (k, l).is_empty
                    l := l + 1
                end
                k := k + 1
            end
        end

    has_horizontal_cross : BOOLEAN is
            -- Is there horizontal cross-winning
combination?
        local
            k: INTEGER
        do
            from
                Result := False
                k := 1
            until
                k > Dimension or Result
            loop
                Result := item (1, k).is_cross and
item(2, k).is_cross and item(3, k).is_cross
```

```
                k := k + 1
            end
        end

    has_horizontal_circle : BOOLEAN is
            -- Is there horizontal circle-winning
combination?
        local
            k: INTEGER
        do
            from
                Result := False
                k := 1
            until
                k > Dimension or Result
            loop
                Result := item (1, k).is_circle and
item(2, k).is_circle and item(3, k).is_circle
                k := k + 1
            end
        end

    has_vertical_cross : BOOLEAN is
            -- Is there vertical cross-winning
combination?
        local
            k: INTEGER
```

```
        do
            from
                Result := False
                k := 1
            until
                k > Dimension or Result
            loop
                Result := item (k, 1).is_cross and
item(k, 2).is_cross and item(k, 3).is_cross
                k := k + 1
            end
        end

    has_vertical_circle : BOOLEAN is
            -- Is there vertical circle-winning
combination?
        local
            k: INTEGER
        do
            from
                Result := False
                k := 1
            until
                k > Dimension or Result
            loop
                Result := item (k, 1).is_circle and
item(k, 2).is_circle and item(k, 3).is_circle
```

```
                k := k + 1
            end
        end

    has_diagonal_cross : BOOLEAN is
            -- Is there diagonal cross-winning
combination?
        do
            Result := item (1, 1).is_cross and item(2,
2).is_cross and item(3, 3).is_cross
            Result := Result or (item (1, 3).is_cross and
item(2, 2).is_cross and item(3, 1).is_cross)
        end

    has_diagonal_circle : BOOLEAN is
            -- Is there diagonal circle-winning
combination?
        do
            Result := item (1, 1).is_circle and item(2,
2).is_circle and item(3, 3).is_circle
            Result := Result or (item (1, 3).is_circle
and item(2, 2).is_circle and item(3, 1).is_circle)
        end

    feature {NONE} -- Automaton
    Cross_turn: STATE is once create Result.make ("Cross
turn") end
```

```
    Circle_turn: STATE is once create Result.make
("Circle turn") end
    Cross_win: STATE is once create Result.make ("Cross
win") end
    Circle_win: STATE is once create Result.make ("Circle
win") end
    Draw: STATE is once create Result.make ("Draw") end

    sd_make_turn: STATE_DEPENDENT_PROCEDURE [TUPLE
[INTEGER, INTEGER]]
            -- State-dependent procedure for `make_turn'

    sd_check_winning_combination:
STATE_DEPENDENT_PROCEDURE [TUPLE]
            -- State-dependent procedure for
`check_winning_combination'. It checks if there is a
winning combination after opponents move

    build_sd_make_turn is
            -- Build `sd_make_turn'
        do
            create sd_make_turn.make (2)
            sd_make_turn.add_behavior (Cross_turn,
                agent (i, j: INTEGER): BOOLEAN do Result
:= True and has_empty end,
                agent (i, j: INTEGER)
                    do
```

```
                        item (i, j).put_cross
                    end,
                Circle_turn)

            sd_make_turn.add_behavior (Circle_turn,
                agent (i, j: INTEGER): BOOLEAN do Result
:= True end,
                agent (i, j: INTEGER)
                    do
                        item (i, j).put_circle
                    end,
                Cross_turn)
        end

    build_sd_check_winning_combination is
            -- Build `_sd_check_winning_combination'
        do
            create sd_check_winning_combination.make (6)

            sd_check_winning_combination.add_behavior
(Cross_turn,
                agent : BOOLEAN do Result := True end,
                agent do end, Cross_turn)
            sd_check_winning_combination.add_behavior
(Circle_turn,
                agent : BOOLEAN do Result := True end,
                agent do end, Circle_turn)
```

```
            sd_check_winning_combination.add_behavior
(Cross_turn,
                agent : BOOLEAN do Result := not
has_empty end,
                agent do end, Draw)
            sd_check_winning_combination.add_behavior
(Circle_turn,
                agent : BOOLEAN do Result := not
has_empty end,
                agent do end, Draw)

            sd_check_winning_combination.add_behavior
(Cross_turn,
                agent : BOOLEAN do Result :=
has_diagonal_circle or has_horizontal_circle or
has_vertical_circle end,
                agent do end, Circle_win)
            sd_check_winning_combination.add_behavior
(Circle_turn,
                agent : BOOLEAN do Result :=
has_diagonal_cross or has_horizontal_cross or
has_vertical_cross end,
                agent do end, Cross_win)
        end
```

```
feature {NONE} -- Implementation
field: ARRAY2 [FIELD_CELL]
        -- Game field

invariant
field_exists: field /= Void
field_width_correct: field.width = Dimension
field_height_correct: field.height = Dimension
end
```

Класс *GAME_MANAGER*

```
indexing
    description: "Managers that may start games,
collect statistics, etc."
    author: "D. Kochelaev"

deferred class
    GAME_MANAGER

inherit
    AUTOMATED

feature -- Access
current_game: GAME

feature -- State dependent: basic operations
```

```
start_new_game is
        -- Start a new game
    do
        sd_start_new_game.call([], state)
        state := sd_start_new_game.next_state
    end

feature {NONE} -- Automaton
First_turn_cross: STATE is once create Result.make
("First turn cross") end
First_turn_circle: STATE is once create Result.make
("First turn circle") end

sd_start_new_game: STATE_DEPENDENT_PROCEDURE [TUPLE]
        -- State-dependent procedure for
`start_new_game'

build_sd_start_new_game is
        -- Build `sd_start_new_game'
    deferred
    end
end
```

Класс *INTERFACE_NAMES*

```
    indexing
         description : "Strings for the Graphical User
Interface"
         author : "D. Kochelaev"

    class
         INTERFACE_NAMES

    feature -- Access

    Button_ok_item: STRING is "OK"
         -- String for "OK" buttons.

    Label_confirm_close_window: STRING is "You are about
to close this window.%NClick OK to proceed."
         -- String for the confirmation dialog box that
appears
         -- when the user try to close the first window.

    Label_confirm_new_game: STRING is "Game is over. Play
again?"
         -- String for the confirmation dialog box for
         -- new game
    end -- class INTERFACE_NAMES
```

Класс *MAIN_WINDOW*

```
indexing
    description : "Main window for this application"
    author      : "Dmitry Kochelaev"

class
    MAIN_WINDOW

inherit
    EV_TITLED_WINDOW
        redefine
            initialize
        end

    INTERFACE_NAMES
        export
            {NONE} all
        undefine
            default_create, copy
        end
create
    default_create

feature {NONE} -- Initialization
    initialize is
            -- Build the interface for this window.
        do
```

```
                Precursor {EV_TITLED_WINDOW}

    --            create {INTERCHANGE_MANAGER}
game_manager
                create {WINNER_FIRST_MANAGER}
game_manager

                build_main_container
                extend (main_container)
                close_request_actions.extend (agent
close_window)
                set_title (Window_title)
                set_size (Window_width, Window_height)

                game_manager.start_new_game
            end

    feature {NONE} -- Model
        game_manager: GAME_MANAGER
                -- Game manager

        update_buttons is
                -- Update `buttons' according to `game'
            local
                i, j: INTEGER
            do
                from
```

```
				i := 1
			until
				i > {GAME}.Dimension
			loop
				from
					j := 1
				until
					j > {GAME}.Dimension
				loop
					buttons.item (i, j).set_text
(game_manager.current_game.item (i, j).out)
					j := j + 1
				end
				i := i + 1
			end
		end

	on_button_click (i: INTEGER; j: INTEGER) is
			-- Process i j button click
		require
			i_not_too_small: i >= 1
			i_not_too_large: i <= {GAME}.Dimension
			j_not_too_small: j >= 1
			j_not_too_large: j <= {GAME}.Dimension
		do
			game_manager.current_game.make_turn (i,
j)
```

```
                update_buttons
                if game_manager.current_game.is_over then
                    request_new_game
                end
            end

    feature {NONE} -- View
        build_main_container is
                -- Create and populate `main_container'.
            require
                main_container_not_yet_created:
main_container = Void
            local
                i: INTEGER
                j: INTEGER
                button: EV_BUTTON
                vertical_boxes: ARRAY [EV_VERTICAL_BOX]
            do
                create main_container
                create vertical_boxes.make (1,
{GAME}.Dimension)
                create buttons.make ({GAME}.Dimension,
{GAME}.Dimension)

                from
                    j := 1
                until
```

```
                    j > {GAME}.Dimension
                loop
                    vertical_boxes.put (create
{EV_VERTICAL_BOX}, j)
                    main_container.extend
(vertical_boxes.item (j))
                    from
                        i := 1
                    until
                        i > {GAME}.Dimension
                    loop
                        create button
                        buttons.put (button, i, j)
                        button.select_actions.extend
(agent on_button_click (i, j))
                        vertical_boxes.item (j).extend
(button)
                        i := i + 1
                    end
                    j := j + 1
                end
            ensure
                main_container_created: main_container /=
Void
            end
```

```
        request_new_game is
                -- Ask, whether user wants to start new
game
            local
                question_dialog: EV_CONFIRMATION_DIALOG
            do
                create question_dialog.make_with_text
(Label_confirm_new_game)
                question_dialog.show_modal_to_window
(Current)

                if
question_dialog.selected_button.is_equal ((create
{EV_DIALOG_CONSTANTS}).ev_ok) then
                    game_manager.start_new_game
                    update_buttons
                else
                    close_window
                end
            end

        close_window is
                -- Close the window
            do
                destroy;
                (create
{EV_ENVIRONMENT}).application.destroy
```

```
            end

        main_container: EV_HORIZONTAL_BOX
                -- Main container (contains all widgets
displayed in this window)

        buttons: ARRAY2 [EV_BUTTON]
                -- Buttons for cells

    feature {NONE} -- Constants

        Window_title: STRING is "Tic Tac Toe sample.
Implemented using automata-based programming (manual,
static)."
                -- Title of the window.

        Window_width: INTEGER is 400
                -- Initial width for this window.

        Window_height: INTEGER is 400
                -- Initial height for this window.

    invariant
        game_manager_exists: game_manager /= Void
        main_container_exists: main_container /= Void
        buttons_exists: buttons /= Void
        each_button_exists: not buttons.has (Void)
```

```
end -- class MAIN_WINDOW
```

Класс *WINNER_FIRST_MANAGER*

```
indexing
    description: "Game managers that let the winner
of the previous game do the first turn and change the
first player in case of draw."
    author: "D. Kochelaev"

class
    WINNER_FIRST_MANAGER

inherit
    GAME_MANAGER
    redefine
        default_create
    end

feature -- Initialization
default_create is
        -- Create a manager with first turn cross
    do
        build_sd_start_new_game
        state := First_turn_cross
    end

feature -- State dependent: basic operations
```

```
    build_sd_start_new_game is
            -- Build `sd_start_new_game'
        do
            create sd_start_new_game.make (4)
            sd_start_new_game.add_behavior
(First_turn_cross,
                agent: BOOLEAN do Result := True end,
                agent do create
current_game.make_first_cross end,
                First_turn_circle)
            sd_start_new_game.add_behavior
(First_turn_circle,
                agent: BOOLEAN do Result := True end,
                agent do create
current_game.make_first_circle end,
                First_turn_cross)
            sd_start_new_game.add_behavior
(First_turn_cross,
                agent: BOOLEAN do Result := current_game
/= Void and then current_game.circle_won end,
                agent do create
current_game.make_first_circle end,
                First_turn_cross)
            sd_start_new_game.add_behavior
(First_turn_circle,
                agent: BOOLEAN do Result := current_game
/= Void and then current_game.cross_won end,
```

```
                agent do create
current_game.make_first_cross end,
                First_turn_circle)

        end
    end
```

Приложение 3. Исходные коды реализации игры «крестики-нолики», базирующейся на автоматизированных классах

Класс *FIELD_CELL*

```
    indexing
    description: "Cells of the game field."
    author: "D. Kochelaev"

    class
        FIELD_CELL

    inherit
        AUTOMATED
        redefine
            default_create,
            out
        end

    feature {NONE} -- Initialization
    default_create is
            -- Create an empty cell
```

```
        do
            state := Empty
        ensure then
            is_empty: is_empty
        end

feature -- State dependent: Element change
put_cross is
            -- Put cross into the cell
        do
            sd_put_cross.call([], state)
            state := sd_put_cross.next_state
        end

put_circle is
            -- Put circle into the cell
        do
            sd_put_circle.call([], state)
            state := sd_put_circle.next_state
        end

feature -- State dependent: Indicators
is_empty: BOOLEAN is
            -- Is the cell empty?
        do
            Result := is_in (<<Empty>>)
        end
```

```
    is_cross: BOOLEAN is
            -- Is the cell cross?
        do
            Result := is_in (<<Cross>>)
        end

    is_circle: BOOLEAN is
            -- Is the cell circle?
        do
            Result := is_in (<<Circle>>)
        end

    feature -- State dependent: Output
    out: STRING is
            -- String representation of the cell
        do
            Result := sd_out.item ([], state)
        end

    feature {NONE} -- Automaton
    Empty: STATE is once create Result.make ("Empty") end

    Cross: STATE is once create Result.make ("Cross") end

    Circle: STATE is once create Result.make ("Circle")
end
```

```
    sd_put_cross: STATE_DEPENDENT_PROCEDURE [TUPLE] is
            -- State dependent procedure for `put_cross'
        once
            create Result.make (1)
            Result.add_behavior (Empty,
                agent : BOOLEAN do Result := True end,
                agent do_nothing,
                Cross)
        end

    sd_put_circle: STATE_DEPENDENT_PROCEDURE [TUPLE] is
            -- State dependent procedure for `put_circle'
        once
            create Result.make (1)
            Result.add_behavior (Empty,
                agent : BOOLEAN do Result := True end,
                agent do_nothing,
                Circle)
        end

    sd_out: STATE_DEPENDENT_FUNCTION [TUPLE, STRING] is
            -- State dependent function for `out'
        once
            create Result.make (3)
            Result.add_result (Empty, agent : BOOLEAN do
Result := True end, "")
```

```
            Result.add_result (Cross, agent : BOOLEAN do
Result := True end, "X")
            Result.add_result (Circle, agent : BOOLEAN do
Result := True end, "O")
        end

    end
```

Класс *GAME*

```
    indexing
        description: "Tic-Tac-Toe games."
        author: ""D. Kochelaev

    class
        GAME

    inherit
        AUTOMATED

    create
        make_first_cross,
        make_first_circle

    feature {NONE} -- Initialization
    make_first_cross is
            -- Create a new game with first turn of
crosses
```

```
		do
			create_empty_field
			build_sd_make_turn
			build_sd_check_winning_combination
			state := Cross_turn
		ensure
			field_empty: field.for_all (agent
{FIELD_CELL}.is_empty)
		end

	make_first_circle is
			-- Create a new game with first turn of
circles
		do
			create_empty_field
			build_sd_make_turn
			build_sd_check_winning_combination
			state := Circle_turn
		ensure
			field_empty: field.for_all (agent
{FIELD_CELL}.is_empty)
		end

	create_empty_field is
			-- Create field with empty cells
		local
			i, j: INTEGER
```

```
        do
            create field.make (Dimension, Dimension)
            from
                i := 1
            until
                i > Dimension
            loop
                from
                    j := 1
                until
                    j > Dimension
                loop
                    field.put (create {FIELD_CELL}, i,
j)
                    j := j + 1
                end
                i := i + 1
            end
        end

    feature -- Constants
    Dimension: INTEGER is 3
            -- Number of rows and columns on the field

    feature -- Access
    item (i, j: INTEGER): FIELD_CELL  is
            -- Value of (`i', `j') cell
```

```
        require
            i_not_too_small: i >= 1
            i_not_too_large: i <= Dimension
            j_not_too_small: j >= 1
            j_not_too_large: j <= Dimension
        do
            Result := field.item (i, j)
        end

    feature -- State-dependent: indicators
    is_over: BOOLEAN is
            -- Is game over?
        do
            Result := is_in (<<Cross_win, Circle_win,
Draw>>)
        end

    cross_won: BOOLEAN is
            -- Did the crosses win?
        do
            Result := is_in (<<Cross_win>>)
        end

    circle_won: BOOLEAN is
            -- Did the crosses win?
        do
            Result := is_in (<<Circle_win>>)
```

```
		end

	feature -- State dependent: basic operations
	make_turn (i, j: INTEGER) is
			-- Make turn, filling cell (`i', `j')
		require
			i_not_too_small: i >= 1
			i_not_too_large: i <= Dimension
			j_not_too_small: j >= 1
			j_not_too_large: j <= Dimension
		do
			sd_make_turn.call ([i, j], state)
			state := sd_make_turn.next_state
			sd_check_winning_combination.call ([], state)
			state :=
sd_check_winning_combination.next_state
		end

	feature {NONE} -- Predicates
	has_empty: BOOLEAN is
			-- Is there empty cell?
		local
			k, l:INTEGER
		do
			from
				k := 1
			until
```

```
                    k > Dimension or Result
                loop
                    from
                        l := 1
                    until
                        l > Dimension or Result
                    loop
                        Result := item (k, l).is_empty
                        l := l + 1
                    end
                    k := k + 1
                end
            end

        has_horizontal_cross : BOOLEAN is
                -- Is there horizontal cross-winning
combination?
            local
                k: INTEGER
            do
                from
                    Result := False
                    k := 1
                until
                    k > Dimension or Result
                loop
                    Result := item (1, k).is_cross and
```

```
item(2, k).is_cross and item(3, k).is_cross
                k := k + 1
            end
        end

    has_horizontal_circle : BOOLEAN is
            -- Is there horizontal circle-winning
combination?
        local
            k: INTEGER
        do
            from
                Result := False
                k := 1
            until
                k > Dimension or Result
            loop
                Result := item (1, k).is_circle and
item(2, k).is_circle and item(3, k).is_circle
                k := k + 1
            end
        end

    has_vertical_cross : BOOLEAN is
            -- Is there vertical cross-winning
combination?
        local
```

```
            k: INTEGER
        do
            from
                Result := False
                k := 1
            until
                k > Dimension or Result
            loop
                Result := item (k, 1).is_cross and
item(k, 2).is_cross and item(k, 3).is_cross
                k := k + 1
            end
        end

    has_vertical_circle : BOOLEAN is
            -- Is there vertical circle-winning
combination?
        local
            k: INTEGER
        do
            from
                Result := False
                k := 1
            until
                k > Dimension or Result
            loop
                Result := item (k, 1).is_circle and
```

```
item(k, 2).is_circle and item(k, 3).is_circle
                k := k + 1
            end
        end

    has_diagonal_cross : BOOLEAN is
            -- Is there diagonal cross-winning
combination?
        do
            Result := item (1, 1).is_cross and item(2,
2).is_cross and item(3, 3).is_cross
            Result := Result or (item (1, 3).is_cross and
item(2, 2).is_cross and item(3, 1).is_cross)
        end

    has_diagonal_circle : BOOLEAN is
            -- Is there diagonal circle-winning
combination?
        do
            Result := item (1, 1).is_circle and item(2,
2).is_circle and item(3, 3).is_circle
            Result := Result or (item (1, 3).is_circle
and item(2, 2).is_circle and item(3, 1).is_circle)
        end

    feature {NONE} -- Automaton
    Cross_turn: STATE is once create Result.make ("Cross
```

```
turn") end
    Circle_turn: STATE is once create Result.make
("Circle turn") end
    Cross_win: STATE is once create Result.make ("Cross
win") end
    Circle_win: STATE is once create Result.make ("Circle
win") end
    Draw: STATE is once create Result.make ("Draw") end

    sd_make_turn: STATE_DEPENDENT_PROCEDURE [TUPLE
[INTEGER, INTEGER]]
            -- State-dependent procedure for `make_turn'

    sd_check_winning_combination:
STATE_DEPENDENT_PROCEDURE [TUPLE]
            -- State-dependent procedure for
`check_winning_combination'. It checks if there is a
winning combination after opponents move

    build_sd_make_turn is
            -- Build `sd_make_turn'
        do
            create sd_make_turn.make (2)
            sd_make_turn.add_behavior (Cross_turn,
                agent (i, j: INTEGER): BOOLEAN do Result
:= True and has_empty end,
                agent (i, j: INTEGER)
```

```
                    do
                        item (i, j).put_cross
                    end,
                Circle_turn)

            sd_make_turn.add_behavior (Circle_turn,
                agent (i, j: INTEGER): BOOLEAN do Result
:= True end,
                agent (i, j: INTEGER)
                    do
                        item (i, j).put_circle
                    end,
                Cross_turn)
        end

    build_sd_check_winning_combination is
            -- Build `_sd_check_winning_combination'
        do
            create sd_check_winning_combination.make (6)

            sd_check_winning_combination.add_behavior
(Cross_turn,
                agent : BOOLEAN do Result := True end,
                agent do end, Cross_turn)
            sd_check_winning_combination.add_behavior
(Circle_turn,
                agent : BOOLEAN do Result := True end,
```

```
                agent do end, Circle_turn)

            sd_check_winning_combination.add_behavior
(Cross_turn,
                agent : BOOLEAN do Result := not
has_empty end,
                agent do end, Draw)
            sd_check_winning_combination.add_behavior
(Circle_turn,
                agent : BOOLEAN do Result := not
has_empty end,
                agent do end, Draw)

            sd_check_winning_combination.add_behavior
(Cross_turn,
                agent : BOOLEAN do Result :=
has_diagonal_circle or has_horizontal_circle or
has_vertical_circle end,
                agent do end, Circle_win)
            sd_check_winning_combination.add_behavior
(Circle_turn,
                agent : BOOLEAN do Result :=
has_diagonal_cross or has_horizontal_cross or
has_vertical_cross end,
                agent do end, Cross_win)
        end
```

```
feature {NONE} -- Implementation
field: ARRAY2 [FIELD_CELL]
        -- Game field

invariant
field_exists: field /= Void
field_width_correct: field.width = Dimension
field_height_correct: field.height = Dimension
end
```

Класс *GAME_MANAGER*

```
indexing
    description: "Managers that may start games,
collect statistics, etc."
    author: "D. Kochelaev"

deferred class
    GAME_MANAGER

inherit
    AUTOMATED

feature -- Access
current_game: GAME
```

```
feature -- State dependent: basic operations
start_new_game is
        -- Start a new game
    do
        sd_start_new_game.call([], state)
        state := sd_start_new_game.next_state
    end

feature {NONE} -- Automaton
First_turn_cross: STATE is once create Result.make
("First turn cross") end
First_turn_circle: STATE is once create Result.make
("First turn circle") end

sd_start_new_game: STATE_DEPENDENT_PROCEDURE [TUPLE]
        -- State-dependent procedure for
`start_new_game'

build_sd_start_new_game is
        -- Build `sd_start_new_game'
    deferred
    end
end
```

Класс *WINNER_FIRST_MANAGER*

```
    indexing
        description: "Game managers that let the winner
of the previous game do the first turn and change the
first player in case of draw."
        author: ""D. Kochelaev

    class
        WINNER_FIRST_MANAGER

    inherit
        GAME_MANAGER
        redefine
            default_create
        end

    feature -- Initialization
    default_create is
            -- Create a manager with first turn cross
        do
            build_sd_start_new_game
            state := First_turn_cross
        end

    feature -- State dependent: basic operations
    build_sd_start_new_game is
            -- Build `sd_start_new_game'
```

```
        do
                create sd_start_new_game.make (4)
                sd_start_new_game.add_behavior
(First_turn_cross,
                    agent: BOOLEAN do Result := True end,
                    agent do create
current_game.make_first_cross end,
                    First_turn_circle)
                sd_start_new_game.add_behavior
(First_turn_circle,
                    agent: BOOLEAN do Result := True end,
                    agent do create
current_game.make_first_circle end,
                    First_turn_cross)
                sd_start_new_game.add_behavior
(First_turn_cross,
                    agent: BOOLEAN do Result := current_game
/= Void and then current_game.circle_won end,
                    agent do create
current_game.make_first_circle end,
                    First_turn_cross)
                sd_start_new_game.add_behavior
(First_turn_circle,
                    agent: BOOLEAN do Result := current_game
/= Void and then current_game.cross_won end,
                    agent do create
current_game.make_first_cross end,
```

```
                   First_turn_circle)

        end
    end
```