Санкт-Петербургский государственный университет информационных технологий, механики и оптики

На правах рукописи

Шопырин Данил Геннадьевич

Методы объектно-ориентированного проектирования и реализации программного обеспечения реактивных систем

Специальность 05.13.13 – Телекоммуникационные системы и компьютерные сети

Диссертация на соискание ученой степени
кандидата технических наук

Научный руководитель –
доктор технических наук,
профессор Шалыто А.А.

Санкт-Петербург
2005

Оглавление

Введение

Актуальность проблемы. При разработке объектно-ориентированного программного обеспечения телекоммуникационных систем актуальна задача проектирования, описания и реализации их поведения. Программные системы можно разделить на следующие классы [1]:

- *преобразующие* системы – это системы, завершающие свое выполнение после преобразования входных данных (например, архиватор, компилятор). В таких системах обычно входные данные известны и доступны на момент запуска системы, а выходные данные доступны после ее завершения;
- *интерактивные* системы – это системы, взаимодействующие с окружающей средой в режиме диалога (например, текстовый редактор). Характерной особенностью таких систем является то, что они могут контролировать скорость взаимодействия с окружающей средой – заставлять окружающую среду «ждать»;
- *реактивные* системы – это системы, взаимодействующие с окружающей средой посредством обмена сообщениями в темпе, задаваемом средой.

Многие телекоммуникационные системы являются представителями класса *реактивных* систем. Реактивные системы имеют следующие особенности [1]:

- время отклика *реактивной* системы задается ее окружением;
- поведение *реактивных* систем детерминировано;
- для *реактивных* систем характерен параллелизм;
- сбои в работе *реактивных* систем крайне нежелательны.

Все вышеуказанные свойства характерны и для многих телекоммуникационных систем, компоненты которых взаимодействуют между собой посредством обмена сообщениями. В качестве каналов связи используются радиосвязь, спутниковая связь, Internet-соединения и т.д. Скорость обработки сообщений обычно строго регламентируется используемыми протоколами и/или конфигурацией системы. Телекоммуникационным системам присущи детерминированное поведение и параллелизм. Сбои в работе *ответственных* телекоммуникационных систем могут привести к катастрофическим последствиям.

Наиболее часто для решения задач проектирования, описания и реализации телекоммуникационных систем используются такие средства как:

- язык описания и спецификации ***SDL*** (Specification and Description Language) [2];
- унифицированный язык моделирования ***UML*** (Unified Modeling Language) [3];
- язык синхронного программирования ***SyncCharts*** [4, 5].

Вышеперечисленные языки являются графическими и в качестве средства описания поведения объектов и процессов в них используются диаграммы, основанные на теории автоматов. Однако эти диаграммы обладают рядом недостатков:

- громоздкость;
- отсутствие удобных и непротиворечивых способов перехода от спецификации к реализации.

Для устранения этих недостатков с 1991 года в России разрабатывается ***SWITCH***-технология [6], также известная как «автоматное программирование», которая предназначена для спецификации и реализации систем со сложным поведением, в том числе и телекоммуникационных. Графы переходов, используемые в *автоматном программировании*, весьма компактны, так

как применяются совместно со схемами связей автоматов, подробно описывающими их интерфейс. Однако в автоматном программировании остаются недостаточно проработанными следующие вопросы:

- совместное использование объектно-ориентированного и автоматного программирования;
- объектно-ориентированное проектирование программного обеспечения *реактивных* систем;
- объектно-ориентированная реализация программного обеспечения *реактивных* систем.

Поэтому исследования, направленные на решение проблем проектирования и реализации реактивных систем на основе совместного использования объектно-ориентированной и автоматной технологий, весьма актуальны.

Целью диссертационной работы является разработка методов объектно-ориентированного проектирования и реализации программного обеспечения *реактивных* систем, главным образом телекоммуникационных, на основе конечных автоматов.

Основные задачи исследования состоят в следующем:

- совершенствование методов реализации автоматных систем на основе ***SWITCH***-технологии;
- разработка графической нотации для проектирования автоматных объектов, которая позволяет описывать декомпозицию и структурирование их логики с помощью наследования;
- разработка методов реализации автоматных объектов на универсальных языках программирования, обеспечивающих декомпозицию и структурирование логики с помощью наследования.

Методы исследования. В работе использованы методы теории автоматов, объектно-ориентированного проектирования и программирования.

Научная новизна. В работе получены следующие научные результаты, которые выносятся на защиту.

1. Предложен метод реализации автоматных систем на основе библиотеки ***STOOL*** (*SWITCH*-Technology Object Oriented Library), устраняющий ряд недостатков ***SWITCH***-технологии.
2. Разработана графическая нотация для проектирования автоматных объектов, которая позволяет описывать декомпозицию и структурирование их логики с помощью наследования.
3. Разработан метод реализации автоматных объектов на основе виртуальных методов, обеспечивающий декомпозицию и структурирование логики с помощью наследования.
4. Разработан метод реализации автоматных объектов на основе виртуальных вложенных классов, обеспечивающий декомпозицию и структурирование логики с помощью наследования.

Результаты диссертации были получены в ходе выполнения научно-исследовательских работ «Разработка технологии программного обеспечения систем управления на основе автоматного подхода», выполненной по заказу Министерства образования РФ в 2001 – 2005 гг., и «Разработка технологии автоматного программирования», выполненной по гранту Российского фонда фундаментальных исследований № 02-07-90114 в 2002 – 2003 гг. (http://is.ifmo.ru, раздел «Наука»).

Достоверность научных положений, выводов и практических рекомендаций, полученных в диссертации, подтверждается корректным обоснованием постановок задач, точной формулировкой критериев, компьютерным моделированием, а также результатами внедрения методов, предложенных в диссертации, на практике.

Практическое значение работы заключается в том, что все полученные результаты могут быть использованы, а некоторые уже используются, в

практической деятельности. Предложенные методы проектирования и реализации автоматных объектов упрощают поддержку и сопровождение программ за счет сокращения дублирования логики автоматов. Предложенные методы реализации автоматных объектов основаны только на *стандартных* возможностях используемых языков программирования, что снижает *риски* при их практическом применении. Практическая ценность подтверждается внедрением результатов работы.

Внедрение результатов работы. Результаты, полученные в диссертации, используются на практике:

- в компании *Транзас* (Санкт-Петербург) при разработке системы мониторинга мобильных объектов ***Navi-Manager***, и в частности, ее коммуникационного сервера;
- в компании *Транзас* (Санкт-Петербург) при разработке каркаса для построения редакторов пространственных данных ***Iris***;
- в учебном процессе на кафедре «Компьютерные технологии» СПбГУ ИТМО при чтении лекций по курсу «Теория автоматов в программировании».

Апробация диссертации. Основные положения диссертационной работы докладывались на научно-методических конференциях «Телематика-2003», «Телематика-2004», «Телематика-2005» (Санкт-Петербург) и XXXIII конференции профессорско-преподавательского состава СПбГУ ИТМО (Санкт-Петербург, 2004).

Публикации. По теме диссертации опубликовано 7 печатных работ, в том числе в журналах «Информационно-управляющие системы» и «Информационные технологии моделирования и управления».

Структура диссертации. Диссертация изложена на 174 страницах и состоит из введения, пяти глав и заключения. Список литературы содержит

78 наименований. Работа иллюстрирована 53 рисунками и содержит три таблицы.

В первой главе приведен обзор современного состояния проблемы проектирования и реализации телекоммуникационных систем на основе конечных автоматов. Во второй главе описан предлагаемый метод реализации автоматных систем на основе библиотеки ***STOOL***. Третья глава содержит описание предлагаемой графической нотации для проектирования автоматных объектов. В четвертой главе описаны методы реализации автоматных объектов на основе виртуальных методов и на основе виртуальных вложенных классов. Пятая глава содержит описание результатов внедрения предложенных методов проектирования и реализации автоматных объектов на практике, в том числе и в телекоммуникационных системах.

Глава 1. Обзор методов проектирования и реализации реактивных систем на основе конечных автоматов

1.1. Язык описания и спецификации *SDL*

*Я*зык ***SDL*** (Specification and Description Language) является языком проектирования и реализации, предназначенным для разработки сложных, распределенных приложений [7]. *Я*зык ***SDL*** используется для описания структуры, поведения и потоков данных системы. Типичной областью применения языка ***SDL*** является отрасль телекоммуникаций.

Разработка языка ***SDL*** началась в 1972 году в ***CCITT*** (Comité Consultatif International de Télégraphique et Téléphonique). Первая версия языка ***SDL*** была выпущена в 1976 году. Последующие версии языка выпускались каждые четыре года. На сегодняшний день ***SDL*** является во всех смыслах *полным* и *зрелым* языком.

Отличительные черты современных версий языка ***SDL***:

- язык ***SDL*** стандартизован международным стандартом ***ITU-T*** (International Telecommunications Union Technical standards group) Z.100 [8];
- язык ***SDL*** является формальным языком, гарантирующим точность, целостность и четкость дизайна;
- язык ***SDL*** имеет изоморфные друг другу графическое и текстовое представления;
- язык ***SDL*** является самодокументирующимся языком проектирования и реализации;
- язык ***SDL*** является объектно-ориентированным языком, поддерживающим инкапсуляцию, полиморфизм и динамическое связывание;

- язык ***SDL*** имеет высокий уровень тестируемости, являющийся следствием высокой формальности языка;
- язык ***SDL*** не зависит от используемой платформы, операционной системы и языка программирования;
- язык ***SDL*** предоставляет возможность повторного использования дизайна;
- язык ***SDL*** предоставляет возможность применения развитых методов оптимизации.

Система на языке ***SDL*** состоит из следующих компонентов:

- структура – иерархия систем, блоков, процессов и процедур;
- коммуникации – система каналов и передаваемых по ним сигналов;
- поведение – процессы системы;
- данные – используемые абстрактные типы данных;
- наследование – отношения наследования и специализации между компонентами системы.

Конечные автоматы применяются в языке ***SDL*** для описания поведения системы. В диаграммах ***SDL*** для описания поведения системы используется более богатый набор элементов, чем в графах переходов. Основные обозначения, используемые при описании поведения системы, приведены на рис. 1.

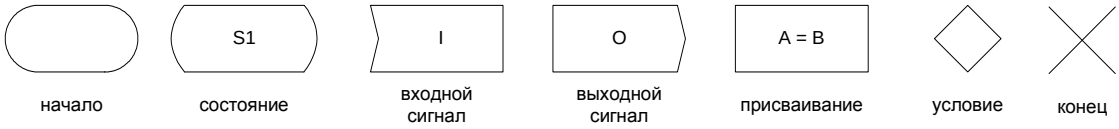


Рис. 1. Основные обозначения, используемые в ***SDL***-диаграммах

На рис. 2 в качестве примера приведена ***SDL***-диаграмма процесса, управляющего доступом к некоторому ресурсу [9].

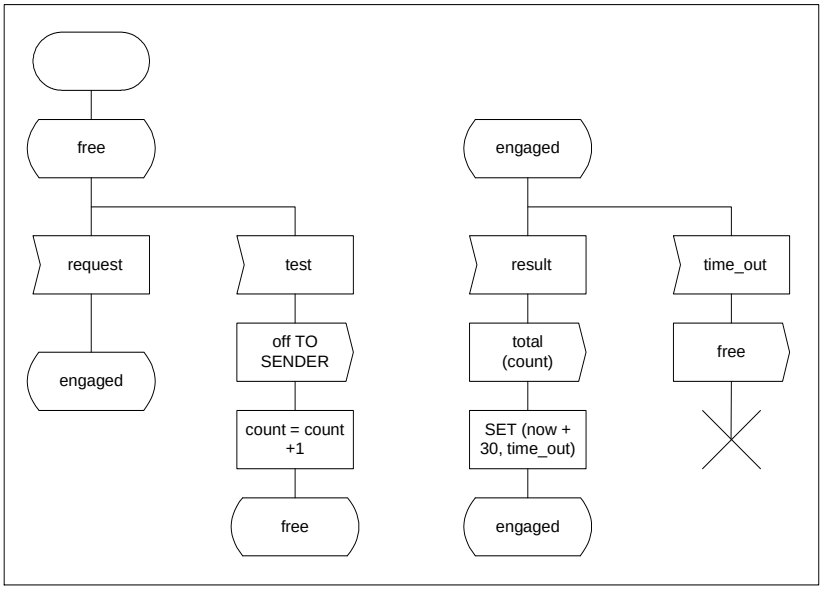


Рис. 2. Пример ***SDL***-диаграммы

Процесс ожидает поступления входного сигнала в своем текущем состоянии. После получения одного из ожидаемых сигналов процесс изменяет свое состояние. Переход в следующее состояние сопровождается последовательностью элементарных действий, таких как изменение значений переменных, отправка сигналов другим процессам, установка тайм-аута и т.д.

1.2. Графический язык *Statecharts*

Графический язык проектирования программных систем ***Statecharts*** был предложен в работе [10]. Большинство известных подходов к проектированию систем с использованием конечных автоматов основаны, в той или иной степени, на языке ***Statecharts***. Язык ***Statecharts*** является, в своем роде, родоначальником целого семейства языков и технологий, использующих конечные автоматы.

Язык ***Statecharts*** является расширением традиционной модели конечных автоматов [11], облегчающим разработку и спецификацию сложных реактивных систем. В традиционную модель добавляются элементы, позволяющие описывать такие понятия, как *иерархия* и *параллелизм* [12, 13].

Иерархия вводится путем *вложенных состояний*. Вложение состояний семантически соответствует операции XOR (исключительное или). Если состояния A и B вложены в состояние C, как показано на рис. 3, то система может находиться только в одном из них, но не в обоих одновременно. Состояние C является абстракцией состояний A и B.

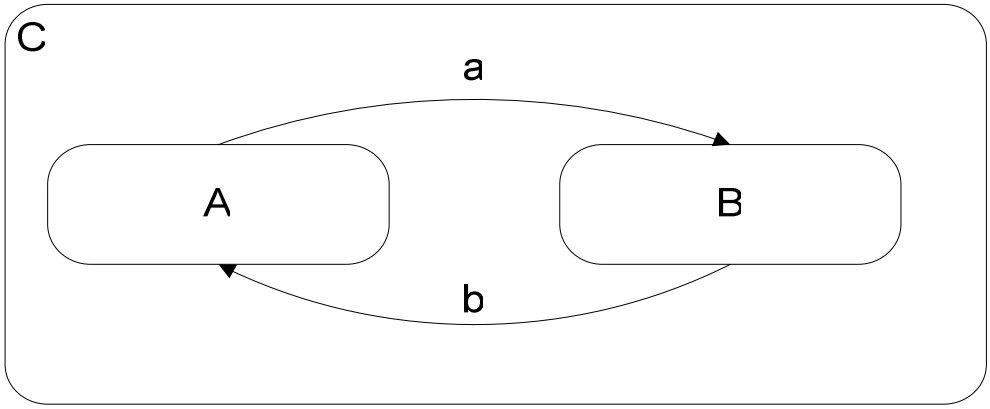


Рис. 3. Вложение состояний

Например, микроволновая печь может находиться в состояниях ON и OFF (рис. 4,a). Причем, в состояние ON вложены состояния WAVE и GRILL. Достоинством вложения состояний является то, что на некотором уровне детализации, пользователь может *абстрагироваться* от наличия состояний WAVE и GRILL (рис. 4,b), что чрезвычайно важно при проектировании сложных многомодульных систем.

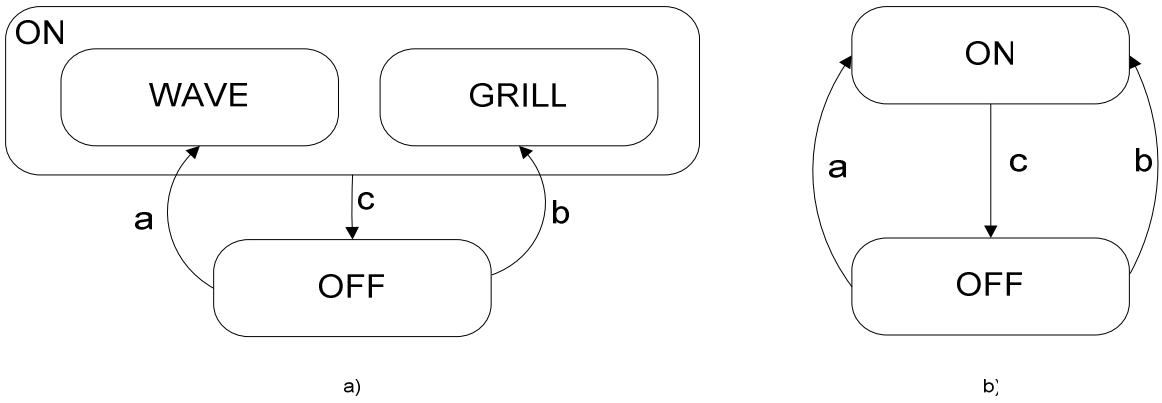


Рис. 4. Пример абстрагирования посредством вложения состояний

Параллелизм вводится путем *ортогональных состояний*. Ортогонализация состояний семантически соответствует операции AND (логическое и).

Если состояния A и B ортогональны друг другу и вложены в состояние C, как показано на рис. 5, то система обязана находиться в обоих этих состояниях *одновременно* (ортогональные состояния в языке ***Statecharts*** разделяются пунктирной линией). Таким образом, состояния A и B параллельны друг другу, а состояние C является их *ортогональным произведением*.

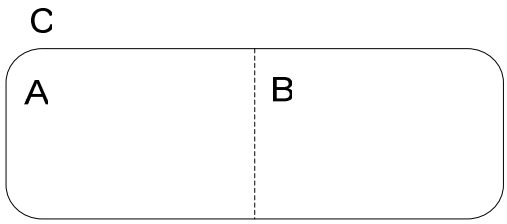


Рис. 5. Ортогональные состояния

Пусть, например, самолет состоит из двух *независимых* подсистем – NAVIGATION и RADAR. Подсистема NAVIGATION содержит состояния ON_GROUD, CLIMB и CRUISE, а подсистема RADAR содержит состояния OFF и ON. Благодаря ортогонализации, эта система в целом может быть изображена так, как показано на рис. 6.

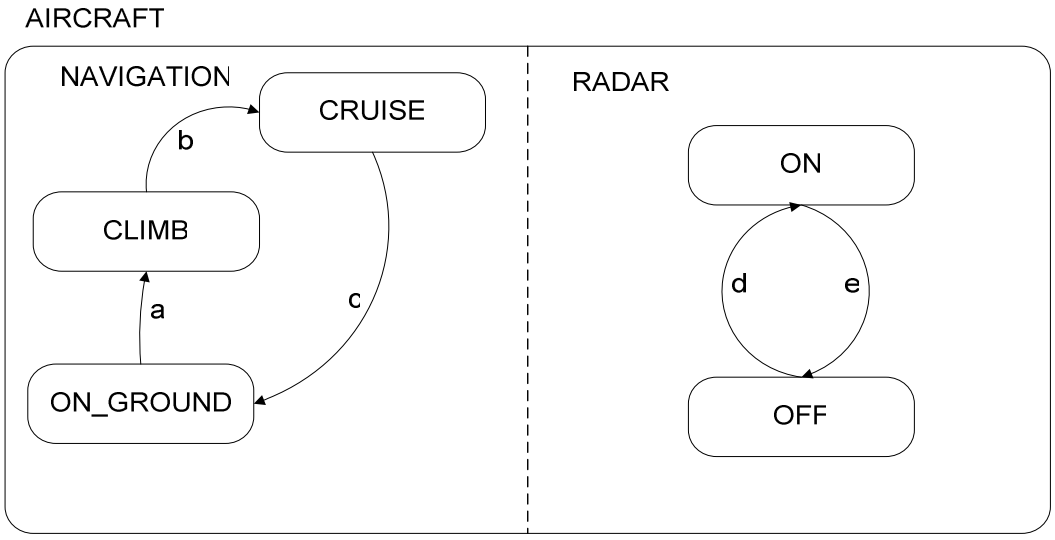


Рис. 6. Пример ортогональных состояний

Соответствующий граф переходов традиционного конечного автомата (рис. 7) является *произведением* [14] конечных автоматов рассмотренных подсистем и содержит шесть состояний (*3 x 2*). Если бы каждая из подсистем

содержала тысячу состояний, то результирующий граф переходов содержал бы миллион состояний. Данный эффект называется *комбинаторным взрывом* и является основным недостатком традиционных конечных автоматов. Отметим, что язык `Statecharts` успешно преодолевает эту проблему благодаря механизму ортогональных состояний.

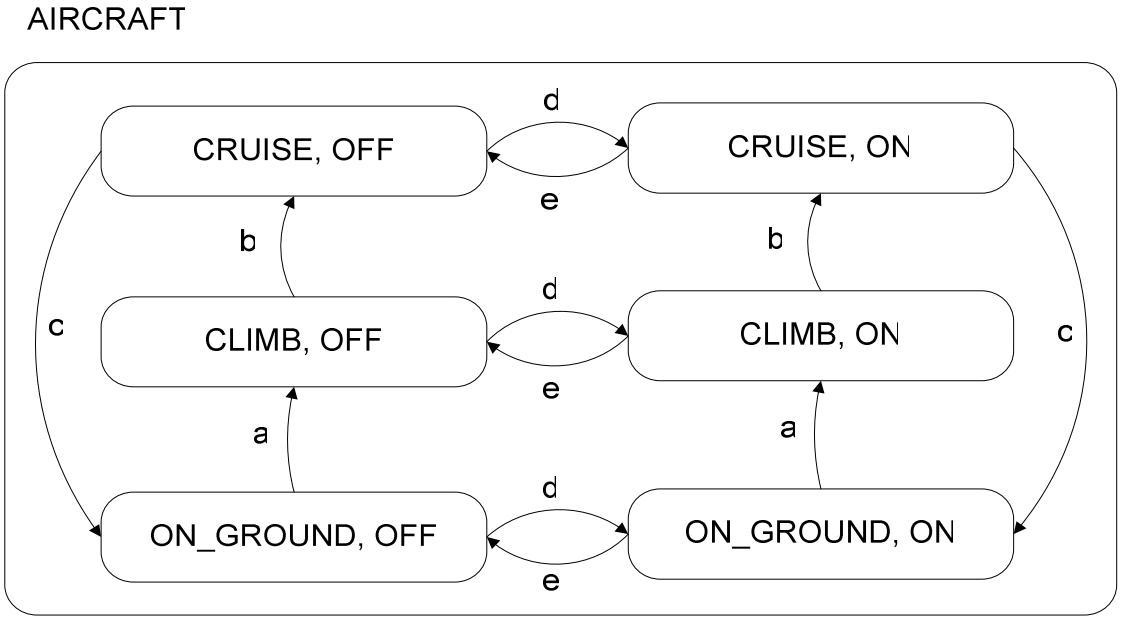


Рис. 7. Комбинаторный взрыв при применении классических автоматов

Дополнительным механизмом декомпозиции системы являются *широковещательные события*. События, сгенерированное внешним окружением или одной из подсистем, одновременно обрабатывается *всеми* параллельными состояниями системы, что значительно уменьшает связность подсистем.

Например, подсистемы самолета `NAVIGATION` и `RADAR` могут содержать состояния `MALFUNCTION`, в которое они переходят по событию `FAIL` (рис. 8). Допустим, что подсистема `NAVIGATION` находится в состоянии `CRUISE`, а подсистема `RADAR` – в состоянии `ON`. В случае если окружением системы будет сгенерированно событие `FAIL`, обе подсистемы перейдут в свои состояния `MALFUNCTION` *одновременно*.

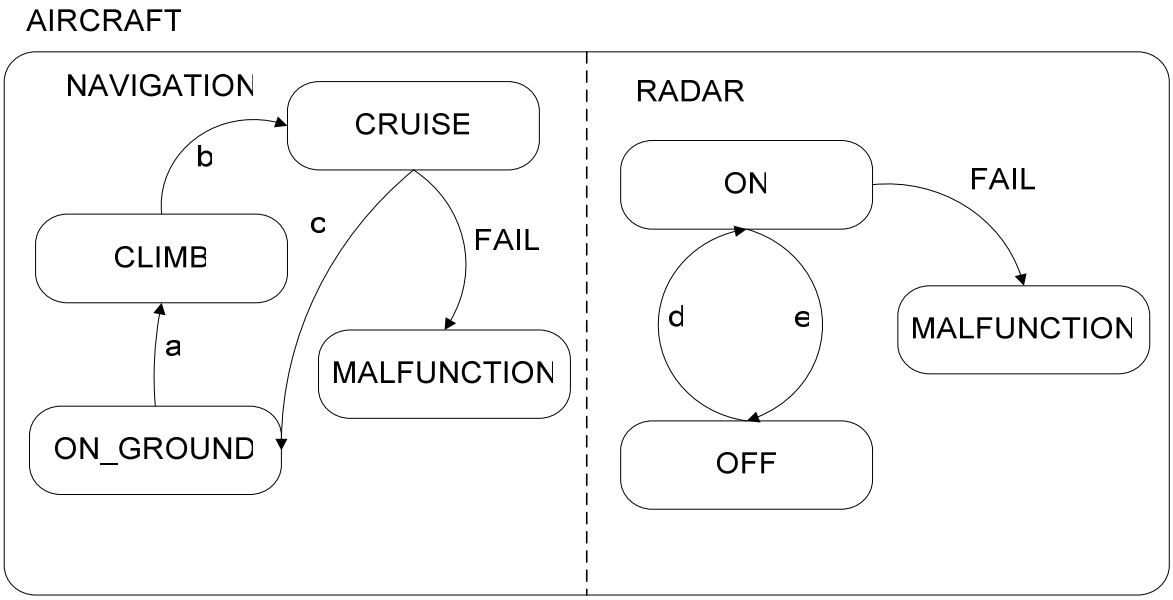


Рис. 8. Пример использования широковещательных событий

Модуль (package) ***UML State Machines*** является объектно-ориентированным вариантом языка ***Statecharts*** [3, 15]. Автоматы в языке ***UML*** используются для задания поведения отдельных объектов. Конечные автоматы являются удобным способом описания поведения объектов или порядка вызова их операций.

Согласно спецификации [15], язык ***UML State Machines*** отличается от классического языка ***Statecharts*** следующим образом***:***

- события могут содержать параметры, в то время как в языке ***Statecharts*** они являются примитивными сигналами;
- не поддерживаются предопределенные действия языка ***Statecharts***, такие как `entered(s)`, `exited(s)` и т.д.
- не поддерживаются широковещательные события (*broadcast*), однако поддерживаются события предписанные множеству объектов;
- поддерживается понятие синхронного взаимодействия между автоматами;
- действия на переходе выполняются в порядке перечисления;

- поддерживаются точки динамического выбора (dynamic choice point);
- модель выполнения опирается на потоки (threads), в отличие от языка ***Statecharts***, опирающегося на «синхронное выполнение».

1.3. Синхронное программирование

С начала 90-х годов в Западной Европе развиваются технологии программирования реактивных систем под общим названием «синхронное программирование» [4, 16].

Объясним основную идею синхронного программирования. Известно, что схемы делятся на синхронные, асинхронные и апериодические (с самосинхронизацией) [17]. При построении вычислительных машин в основном используются синхронные схемы, важнейшей составляющей которых являются синхронные автоматы с памятью. Каждый из таких автоматов содержит две составляющие (рис. 9):

- автомат без памяти, осуществляющий функциональное преобразование;
- тактируемые элементы памяти, хранящие состояние автомата.

Посредством элементов памяти удается обеспечить *синхронную* передачу сигналов в обратной связи с выходов автомата – на его входы, устранив состязания сигналов в обратной связи. Это позволяет разделить состояния автомата в последовательные моменты времени. Тактируемые элементы памяти обеспечивают также *синхронизацию* поступающих на вход автомата входных воздействий и сигналов обратной связи.

Рис. 9. Синхронный автомат с памятью

В синхронном программировании вводится термин *реакция*. Реакция – это неделимый такт работы системы, в процессе которого все ее компоненты синхронно обрабатывают входные сигналы, изменяют свое состояние и формируют выходные сигналы. Считается, что длительность любой реакции равна нулю. В результате, *интервалы реакции* превращаются в *моменты реакции*. Следовательно, реакции не перекрываются друг с другом. Таким образом, система «работает» только в моменты реакций. Между моментами реакций система «бездействует». Это предположение называется *синхронной гипотезой* или *гипотезой атомарных реакций* [18].

Таким образом, в синхронном программировании вводится абстрактное дискретное время, каждый момент которого соответствует одной реакции системы.

Существуют две основные модели реактивных систем:

- системы, управляемые событиями (рис. 10,а);
- системы, управляемые таймером (рис. 10,б).

```
<Инициализация>
<Для каждого> <входного события> <повторять>
    <Вычислить выходные данные>
    <Вывести выходные данные>
<Конец цикла>
```

а) Система, управляемая собыиями

```
<Инициализация>
<Для каждого> <срабатывания таймера> <повторять>
    <Прочитать входные данные>
    <Вычислить выходные данные>
    <Вывести выходные данные>
<Конец цикла>
```

б) Система, управляемая таймером

Рис. 10. Модели реактивных систем

Математически, синхронная система может быть определена как совокупность компонентов, задаваемых:

- вектором входных переменных $\tilde{x}_i$;
- состоянием y_i;
- вектором выходных воздействий $\tilde{z}_i$.

Тогда реакция системы в целом есть комбинация реакций всех компонентов системы. При этом для каждого компонента справедлива система соотношений:

$$y_i^{n+1} = f(y_i^n, \tilde{x}_i^n);$$
$$\tilde{z}_i^n = g(y_i^n, \tilde{x}_i^n).$$

Комбинация реакций отдельных компонентов обеспечивается тем, что входные или выходные значения *i*-го компонента могут являться также входными значениями для *j*-го компонента:

$$x_i^n(k) = z_j^n(l),$$

или

$$x_i^n(k) = x_j^n(l),$$

где $x_i^n(k)$ означает *k*-ю координату вектора $\tilde{x}_i^n$.

Отметим, что система, заданная таким образом не всегда имеет однозначное детерминированное поведение. Дело в том, что в ней могут возникнуть мгновенные обратные связи. Эта ситуация проиллюстрирована в терминах синхронных конечных автоматов на рис. 11 (мгновенная обратная связь выделена пунктиром).

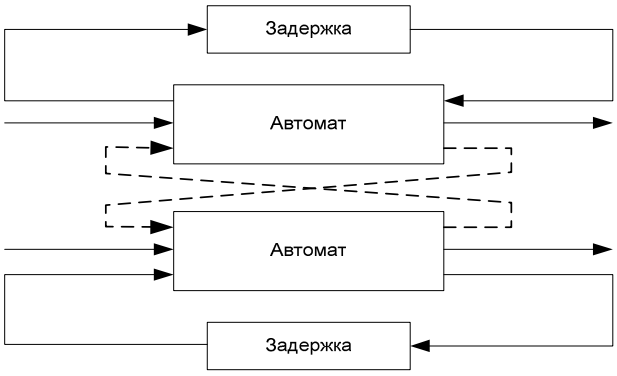


Рис. 11. Мгновенная обратная связь при соединении двух автоматов

Эта проблема может быть решена как минимум четырьмя разными способами [4]:

- каждая реакция системы разбивается на последовательность элементарных *микрошагов* (что приводит к многочисленным противоречивым интерпретациям [12]). Данный подход используется в ***Very High Speed Integrated Circuit Hardware Description Language (VHDL)*** [19], языке ***Statecharts*** [10] и других системах.
- вводится требование, что система не может содержать мгновенных обратных связей. Данный подход используется в языке программирования ***Lustre*** [20].
- вводится требование, что каждой реакции системы соответствует ровно один вариант перехода в следующее состояние. Данный подход используется в языке ***Esterel*** [21].
- вводится требование, что каждой реакции может соответствовать ни одного (программа заблокирована), один или более вариантов перехода в следующее состояние. В последнем случае имеет место недетерминированное поведение. Данный подход используется в языке ***Signal*** [22].

1.3.1. Язык *Argos*

Язык ***Argos*** – это графический язык программирования, оперирующий конечными автоматами [23]. Синтаксис языка ***Argos*** похож на синтаксис языка ***Statecharts***. Основным отличием является тот факт, что язык ***Argos*** опирается на *синхронную гипотезу*.

Также как и ***Statecharts***, язык ***Argos*** оперирует конечными автоматами, расширяя синтаксис последних введением вложенных и ортогональных состояний.

В качестве примера на рис. 12 приведена программа, которая по каждому четвертому событию `e1` генерирует событие `e2`.

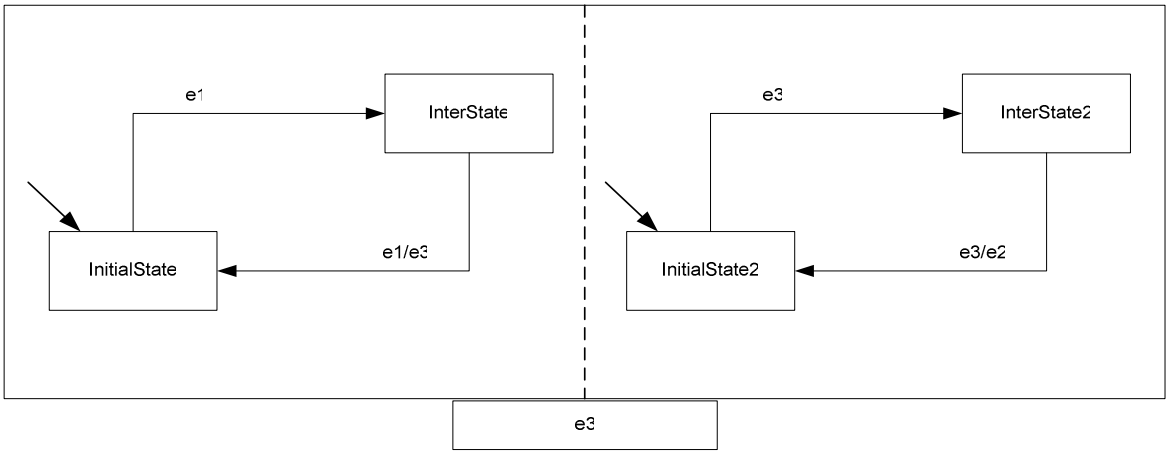


Рис. 12. Пример программы на языке ***Argos***

Множество состояний на графе переходов обозначается множеством прямоугольников. Граф переходов имеет одно начальное состояние, помеченное дугой без начальной вершины. Множество переходов между состояниями представлено множеством помеченных дуг. Метки переходов состоят из входного и выходного воздействий, записанных в формате `I[/O]`, где `I` – входное воздействие, а `O` – необязательное выходное воздействие.

Автоматы, разделенные пунктирной линией, выполняются параллельно. Автоматы могут взаимодействовать между собой посредством событий. Выходные события одного автомата могут быть входными событиями для другого автомата. Язык ***Argos*** позволяет задать область видимости локального события. На рис. 12 область видимости события `e3` ограничена прямоугольником, охватывающим оба автомата.

1.3.2. Язык *SyncCharts*

Язык ***SyncCharts*** – это графический язык программирования [5, 24], семантика которого полностью соответствует семантике языка ***Esterel*** [21]. Многие возможности унаследованы [25] из языков ***Statecharts*** и ***Argos*** .

Язык ***SyncCharts*** предоставляет мощный и гибкий синтаксис. Особенностью данного языка являются вытесняющие инструкции. Вытеснение (*pre-*

emption) – это отказ некоторому процессу в праве выполняться навсегда (сильная форма) или временно (слабая форма) [26].

Простой пример программы на языке ***SyncCharts***, взятый из работы [24], показан на рис. 13. Данная программа подсчитывает количество сигналов `T` и в момент, когда счетчик достигает значения пять, генерирует сигнал тревоги `Alarm`. Счетчик запускается посредством сигнала `set` и может быть прерван в любой момент сигналом `reset`.

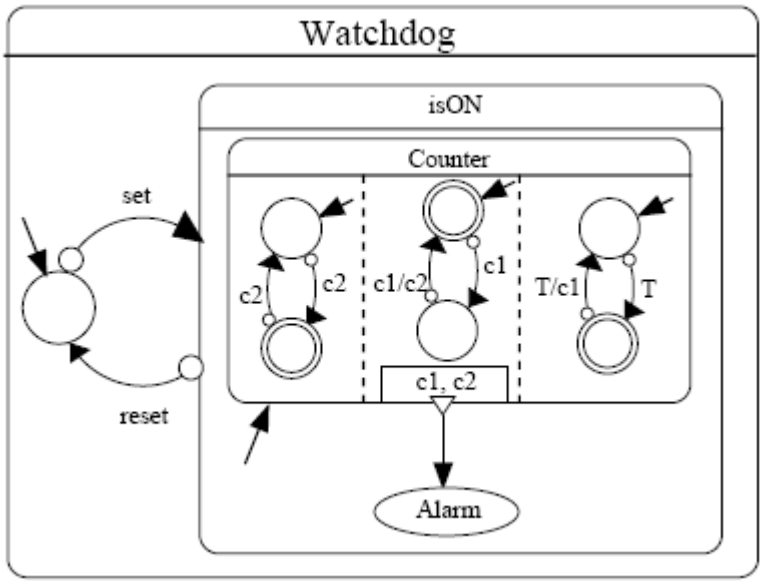


Рис. 13. ***Watchdog*** – пример использования языка ***SyncCharts***

Конечное состояние обозначается двойной окружностью. Начальное состояние помечается дугой перехода, не имеющей начала. Нормальное завершение обозначается стрелкой с треугольником. Строгое вытеснение обозначается стрелкой с окружностью.

Примером строгого вытеснения может служить обработка сигнала `reset`. В момент возникновения этого сигнала все процессы, вложенные в состояние `isON`, немедленно прерываются и система покидает это состояние.

1.4. *SWITCH*-технология

С 1991 года для программирования реактивных систем развивается ***SWITCH***-технология [6, 27–36], также известная как «автоматное программи-

рование» или «программирование с явным выделением состояний», которая в качестве языка спецификации использует графы переходов.

Семантика используемых в ***SWITCH***-технологии графов переходов близка к семантике языка ***Statecharts***, но не эквивалентна ей. В ***SWITCH***-технологии вводятся понятия *автомат* и *вложение автоматов*, и отсутствуют понятия *вложенных* и *ортогональных* состояний. Вложение и ортогонализация состояний в ***SWITCH***-технологии реализуется посредством вложения автоматов и введения понятия «система автоматов».

Рассмотрим реализацию примеров, приведенных в разд. 1.2 с использованием ***SWITCH***-технологии (используя синтаксис предлагаемый в работе [27]). На рис. 14 представлен аналог системы, изображенной на рис. 4.

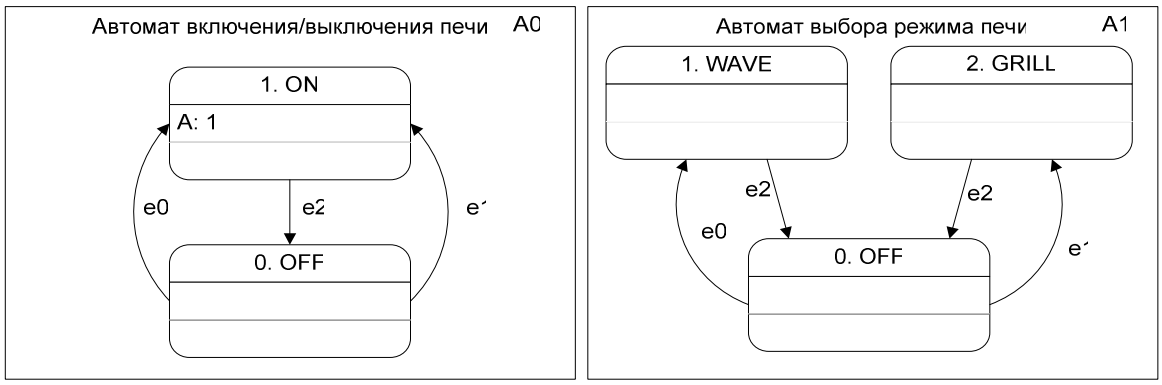


Рис. 14. Пример абстрагирования посредством вложения автоматов

Недостатком реализации на основе ***SWITCH***-технологии, по сравнению с языком ***Statecharts,*** является наличие дублирующего состояния `OFF` в автомате `A1`.

Достоинство ***SWITCH***-технологии состоит в лучшей инкапсуляции, так как:

- факт наличия разных режимов работы неизвестен на уровне автомата `A0`;
- отсутствуют переходы между разными уровнями детализации как, например, это имеет место при переходе из состояния `OFF` в состояние `WAVE` на рис. 4.

На рис. 15 представлен аналог системы, изображенной на рис. 6.

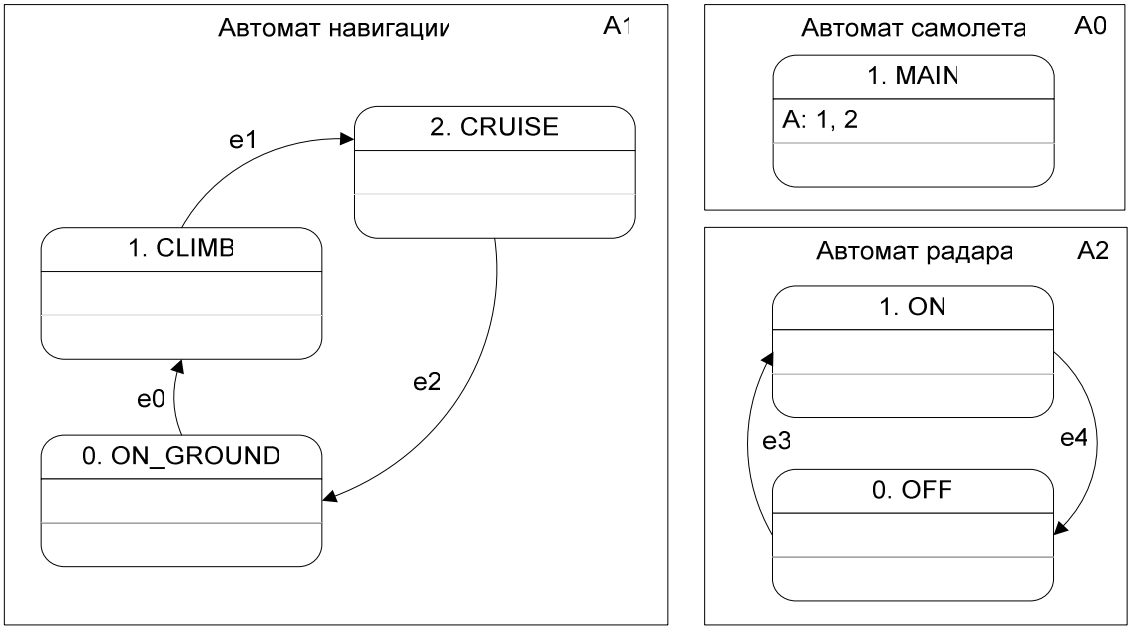


Рис. 15. Пример ортогонализации посредством вложения автоматов

К достоинствам, по сравнению со ***Statecharts***-версией, можно отнести тот факт, что при использовании вложения автоматов не возникает проблемы с нотацией, как это имеет место при использовании вложенных ортогональных состояний [10]. Так, например, в языке ***Statecharts*** возникают трудности с расположением названия *макросостояния*, содержащего вложенные ортогональные состояния. Поэтому на рис. 6 название макросостояния `AIRCRAFT` изображено вовне прямоугольника этого состояния.

Вложение автоматов избавляет также и от необходимости вводить понятие «историческое состояние».

Графы переходов ***SWITCH***-технологии проще воспринимаются в случае больших и сложных систем, за счет лучшей инкапсуляции. Также немаловажен тот факт, что данный подход более удобен для получения *твердых копий* документации, когда документ может быть напечатан в режиме «один автомат – одна страница».

1.5. Реализация систем на основе *SWITCH*-технологии

Согласно работе [29], автоматное программирование не противопоставляется процедурному или объектно-ориентированному программированию, и может быть использовано совместно с ними. Однако до сих пор остаются нерешенными некоторые проблемы, связанные с совместным использованием автоматного и объектно-ориентированного программирований.

В работе [29] предлагается хранить текущее состояние автомата в виде одной многозначной переменной интегрального типа (в частности, перечислимого типа). В этом случае поведение автомата реализуется с помощью оператора `switch` по этой переменной с вложенными условными операторами `if` по входным событиям.

Основными недостатками данного подхода являются:

- использование двух последовательно выполняющихся операторов `switch`, что приводит к размазыванию и частичному дублированию логики;
- отсутствие механизмов автоматического сохранения и контроля изменений предыдущего состояния автомата;
- отсутствие механизмов автоматического протоколирования.

Описанный подход хорошо подходит для автоматической генерации кода, но использование его при ручном программировании затруднительно. Указанные недостатки являются источником ошибок и затрудняют поддержку и сопровождение программных систем.

Использование двух последовательно выполняющихся операторов `switch` и «ручной» учет предыдущего состояния автомата связаны с необходимостью различения действий и деятельностей в состояниях [37].

Эти недостатки, также как и «ручное» протоколирование, связаны с использованием в качестве переменной состояния переменных примитивных типов (интегральных или перечислимых) и недостаточной формализации

входных и выходных воздействий. Перечисленные проблемы решаются в библиотеке ***STOOL***, описанной в разд. 2.

1.6. Конечные автоматы в объектно-ориентированных системах

Конечные автоматы широко используются при проектировании реактивных систем. Однако после перехода на объектно-ориентированную методологию, использование конечных автоматов сократилось по причине отсутствия полностью объектно-ориентированных методов их реализации. Многие номинально объектно-ориентированные методы реализации конечных автоматов, на самом деле, больше относятся к *структурному* программированию [38]. Наличие же больших блоков *структурного* кода в объектно-ориентированной программе ведет к потере однородности и, вследствие чего, усложнению системы.

Наиболее распространенные *не* объектно-ориентированные методы реализации конечных автоматов основаны на:

- вложенных условных операторах `if` [39];
- вложенных операторах `switch` [40];
- операторах `goto` [41];
- таблицах переходов [42, 43].

Первые три подхода приводят к написанию больших *монолитных* блоков кода. Подобный код труден для чтения и понимания. Монолитность получаемого исходного кода делает невозможными декомпозицию и структурирования логики конечных автоматов с помощью наследования.

Применение четвертого метода снижает читабельность исходного кода. Табличный подход противоречит *декларативности* современных языков программирования высокого уровня, так как в этом случае исходный код аналогичен коду, получаемому при использовании ассемблеров. Вместо ис-

пользования высокоуровневых концепций, заданных в используемом языке программирования, таких как *класс* или *метод*, используются *самодельные* таблицы вызовов, что значительно затрудняет отладку и поддержку исходного кода. Это происходит потому, что данные конструкции *чужды* используемому отладчику и компилятору.

Библиотека ***STOOL***, преложенная в настоящей работе, основана на *частично* объектно-ориентированном способе реализации автоматных объектов, так как логика поведения автоматных объектов реализуется с помощью оператора `switch`, что делает невозможным ее декомпозицию и структурирование с помощью наследования.

Основным достоинством перечисленных не объектно-ориентированных методов реализации конечных автоматов является высокая скорость выполнения. Основной недостаток – трудность поддержки и сопровождения получаемого исходного кода. С ростом производительности современной вычислительной техники стоимость поддержки и сопровождения программных систем выходит, по сравнению с другими составляющими, на передний план. Это, в частности, и послужило стимулом к созданию полностью объектно-ориентированных методов реализации конечных автоматов.

Отметим, что объектно-ориентированные методы реализации конечных автоматов не отменяют использование средств автоматической генерации кода. В тех случаях, когда скорость выполнения сгенерированного кода не критична, то объектно-ориентированные методы реализации конечных автоматов могут быть использованы наравне с традиционными методами, такими как реализация конечных автоматов на основе оператора `switch`.

Обзоры существующих объектно-ориентированных методов реализации автоматных объектов приведены в работах [38, 44–46]. В настоящей работе предлагаются два новых метода реализации автоматных объектов:

- метод реализации автоматных объектов на основе виртуальных методов (разд. 4.1);

- метод реализации автоматных объектов на основе виртуальных вложенных классов (разд. 4.3).

Отличительной особенностью обоих предложенных методов является возможность декомпозиции и структурирования логики автоматных объектов с помощью наследования. В разд. 3.4 описывается графическая нотация для проектирования автоматных объектов, также позволяющая описывать наследование их логики. Ниже в данной главе приведен краткий обзор методов, родственных двум предложенным методам реализации автоматных объектов.

В настоящей работе принята следующая терминология. Объект, поведение которого зависит от его текущего состояния и реализовано на основе конечных автоматов, будем называть *автоматным объектом*.

К сожалению, в современной иностранной и отечественной литературе сложилась путаница в отношении термина *состояние* (*state*). Основными альтернативными вариантами являются:

- *macrostate* (*макросостояние*) – используется достаточно редко (упоминается в [47]), конфликтует с термином «макросостояние» языка ***Statecharts***;
- *управляющее или автоматное состояние* – используется в отечественных работах по ***SWITCH***-технологии [48].
- *mode* (*режим работы*) – новый, еще не устоявшийся, термин [47];

В результате этой несогласованности иногда возникают весьма курьезные ситуации. Например, работа [47] называется «Methods for *states*», но в ней используется термин *mode*. В настоящей работе в качестве основного термина используется *управляющее состояние*, или просто *состояние*. Однако в первой главе, при обзоре существующих подходов к реализации автоматных объектов, используется оригинальная терминология.

1.6.1. Паттерн проектирования *State* и его варианты

Наиболее известным методом реализации автоматных объектов является паттерн проектирования ***State,*** описанный в работе [49]. Многие современные методы реализации автоматных объектов, в том числе и предложенный в данной работе метод на основе виртуальных вложенных классов, являются усовершенствованными вариантами паттерна ***State***.

Согласно работе [49], *паттерн проектирования* – это описание взаимодействия объектов и классов, адаптированных для решения общей задачи проектирования в конкретном контексте. Паттерн проектирования именует, абстрагирует и идентифицирует ключевые аспекты структуры общего решения, которые и позволяют применить его для создания повторно используемого дизайна.

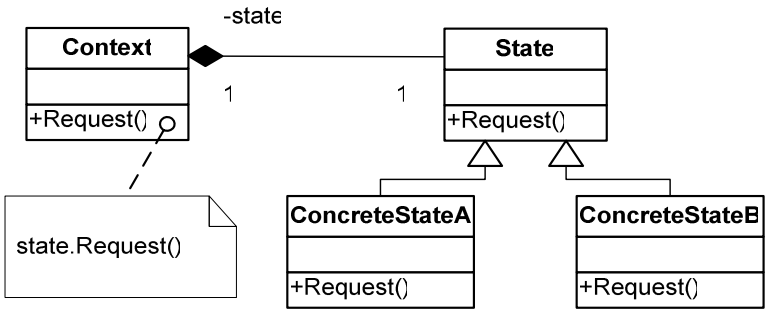


Рис. 16. Структура паттерна проектирования State

Структура паттерна ***State*** приведена на рис. 16. Использование этого паттерна позволяет варьировать поведение объекта в зависимости от его внутреннего состояния таким образом, что извне создается впечатление изменения класса объекта.

Кроме *управляющего состояния*, или просто *состояния*, кардинальным образом влияющего на поведение объекта, объект может также иметь *вычислительное состояние*. Под вычислительным состоянием понимается совокупность всех атрибутов объекта, не управляющая его поведением непосредственно. Например, объект, управляющий поведением микроволновой печи,

может находиться в *управляющих состояниях* `OFF`, `WAVE` и `GRILL`. Кроме того, этот объект должен хранить температуру в печи, время приготовления и другие параметры. Множество подобных атрибутов автоматного объекта является его вычислительным состоянием.

В паттерне ***State*** участвуют следующие объекты:

- контекст – определяет интерфейс автоматного объекта, хранит экземпляр текущего конкретного состояния и переадресует ему все вызовы;
- состояние – определяет интерфейс для инкапсуляции поведения в конкретном состоянии;
- конкретное состояние – реализует поведение автоматного объекта в одном из его состояний.

Предлагается несколько вариантов отношений между объектами:

- контекст может передавать указатель на самого себя конкретным состояниям. В этом случае экземпляры состояний могут совместно пользоваться методами и полями контекста. Однако в результате этого увеличивается связность между контекстом и конкретными состояниями;
- вычислительное состояние автоматного объекта может храниться в контексте или в конкретных состояниях. Второй вариант предпочтительнее в тех случаях, когда в зависимости от текущего состояния значительно изменяется состав хранимой информации. Кроме того, второй вариант упрощает поддержание целостности хранимой информации;
- экземпляры конкретных состояний могут создаваться *по требованию* или создаваться в виде статических полей в процессе инициализации системы. Использование первого или второго вариантов зависит от частоты смены состояния объектами. В случае

частой смены состояния второй вариант предпочтительнее за счет экономии времени, которое расходуется на создание и уничтожение экземпляров состояний;

- переход в следующее состояние может осуществляться либо контекстом, либо объектом состояния. Первый вариант приводит к централизации логики автоматного объекта [50]. Второй — способствует увеличению гибкости системы [51], и, в частности, делает возможным расширение логики автоматного объекта с помощью наследования;

Достоинства паттерна проектирования ***State***:

- не зависящее от управляющего состояния поведение автоматного объекта локализуется в классах управляющих состояний;
- переходы между состояниями выполняются явно.

Недостатки паттерна проектирования ***State***:

- данный паттерн неудобен в случае большого числа состояний, так как приводит к написанию большого числа конкретных состояний;
- контекст отвечает за корректную диспетчеризацию всех внешних вызовов, что повышает связность системы.

На данный момент известно множество альтернативных вариантов паттерна ***State***, некоторые из которых описаны ниже.

1.6.2. Реализация автоматных объектов на основе методов

Рассмотрим альтернативный подход к реализации автоматных объектов, предложенный в работе [47]. Метод, поведение которого зависит от текущего состояния объекта, реализуется совокупностью методов, имеющих одинаковую сигнатуру и описывающих какой-то один вариант поведения объекта. В этом случае, состояние автоматного объекта хранится в виде на-

бора указателей на методы, реализующие поведение объекта в данном состоянии. Таким образом, состояние автоматного объекта хранится не в виде *скалярной* переменной, а в виде вектора указателей на методы. Переключение состояний автоматного объекта осуществляется путем изменения одного или нескольких таких указателей.

Достоинствами этого метода являются:

- отсутствие вложенных условных операторов `if` и громоздких операторов `switch`;
- поведение объекта реализуется с помощью его собственных методов, что может значительно сократить количество вводимых сущностей.

Недостатком этого метода является высокая трудоемкость хранения вычислительного состояния, состав и структура которого зависит от текущего управляющего состояния.

Также отметим, что вопросы декомпозиции и структурирования логики автоматных объектов с помощью наследования в работе [47] не рассматриваются.

Метод реализации автоматных объектов на основе виртуальных методов, описанный ниже в настоящей работе, предоставляет аналогичные возможности и, кроме того, обеспечивает декомпозицию и структурирование их логики с помощью наследования.

1.6.3. Расширение поведения конечных автоматов с помощью наследования

Проектирование и реализация сложных автоматных объектов на основе простых имеет большую практическую ценность, так как способствует:

- повторному использованию исходного кода;
- структурированию и декомпозиции исходного кода;

- сокращению дублирования исходного кода.

Вышеперечисленное приводит к сокращению затрат на поддержку и сопровождение программных систем.

Вопрос построения сложных объектов на основе более простых достаточно хорошо разработан в объектно-ориентированном программировании. Основными применяемыми методами являются:

- наследование с использованием механизма виртуальных функций;
- мета-программирование на основе шаблонов.

В работе [51] описан ряд методов, позволяющих расширять логику конечных автоматов. При порождении производных автоматных объектов используются такие методы объектно-ориентированного программирования, как:

- наследование;
- композиция;
- делегирование.

Наиболее интересным для настоящего исследования является расширение поведения автоматных объектов с помощью наследования. В предлагаемом в работе [51] методе реализации автоматных объектов контекст хранит указатель на текущие состояние и переадресует ему все вызовы интерфейса автоматного объекта. Все методы классов состояний возвращают экземпляр следующего состояния. Экземпляры состояний создаются в качестве глобальных переменных или статических переменных класса.

Расширение логики достигается благодаря возможности добавлять новые состояния и *перегружать* переходы между состояниями. *Перегрузка* перехода аналогична перегрузке виртуального метода в объектно-ориентированных языках программирования. В результате перегрузки перехода обычно изменяется целевое состояние.

Возможность реализации *перегрузки* перехода достигается благодаря использованию так называемых «таблиц состояний» (`StateMap`). Таблицы `StateMap` аналогичны механизму таблиц виртуальных функций `vtbl` языка программирования ***C***++ [52]. Каждое состояние автоматного объекта получает собственную таблицу состояний как параметр конструктора. Переход в следующее состояние осуществляется косвенно, через таблицу состояний `StateMap`.

Основным достоинством предлагаемого подхода является возможность повторно использовать независимо разработанные классы состояний, не изменяя исходный код.

Основным недостатком этого подхода является *ручное программирование* таблиц состояний, что приводит к громоздкому и запутанному коду, затрудняющему поддержку и сопровождение программных систем. Метод реализации автоматных объектов на основе виртуальных вложенных классов, описанный ниже в настоящей работе, предоставляет аналогичные возможности и обходится при этом без ручного создания и поддержки таблицы состояний.

Выводы

Из изложенного выше следует:

- существующие методы реализации программных систем на основе ***SWITCH***-технологии обладают рядом недостатков (использование двух операторов `switch`, отсутствие автоматического контроля изменения состояний автоматов, отсутствие средств автоматического протоколирования и т.д.);
- отсутствуют методы графического проектирования автоматных объектов, которые позволяют описывать декомпозицию и структурирование их логики с помощью наследования;

- существующие методы реализации автоматных объектов либо не поддерживают декомпозицию и структурирование их логики с помощью наследования, либо поддерживают их неудовлетворительно.

Настоящая работа направлена на устранение всех вышеперечисленных проблем. Вторая глава содержит описание библиотеки ***STOOL***, устраняющей некоторые недостатки ***SWITCH***-технологии. Третья и четвертая главы содержат описание графической нотации для проектирования автоматных объектов и методов реализации автоматных объектов, обеспечивающих декомпозицию и структурирование их логики с помощью наследования.

Глава 2. Реализация автоматных систем на основе библиотеки *STOOL*

В работах [27–29, 53] предлагается метод построения объектно-ориентированных программ с явным выделением состояний. В указанных работах подробно рассмотрены вопросы проектирования программ этого класса. Однако методы программной реализации, предлагаемые в них, обладают следующими недостатками:

- не выделено состояние системы;
- в функции, реализующей автомат, применяются два оператора `switch`, так как нет механизма различения действий и деятельностей в состояниях. Этот подход снижает удобочитаемость кода и увеличивает вероятность ошибки;
- нет механизма обеспечения повторного использования реализованных автоматов;
- функции протоколирования вручную вводятся в текст программы программистом, что нельзя назвать «автоматическим» построением протокола;
- не предложен механизм обработки ошибок, возникающих при выполнении автоматов;
- не предложен механизм организации параллельных вычислений.

В настоящей работе предлагается способ реализации программных систем рассматриваемого класса, устраняющий перечисленные недостатки. В качестве базы для разработки программ с выделением состояний предлагается библиотека ***STOOL*** - SWITCH-Technology Object Oriented Library [37, 54]. В настоящее время эта библиотека реализована на языке ***C***++. Ниже будет опи-

сана библиотека ***STOOL*** и рассмотрены вопросы разработки программных систем с ее использованием.

2.1. Термины и определения

Класс автоматов – множество автоматов, реализующих один и тот же граф переходов. Например, классом автоматов является множество одинаковых автоматов, осуществляющих подсчет повторений в последовательности.

Автомат – конкретный экземпляр класса автоматов.

Входное воздействие – булева функция, характеризующая состояние внешней среды. Значение входного воздействия не зависит от порядка и количества обращений к нему. Таким образом, обращение к входному воздействию не изменяет состояния внешней среды.

Выходное воздействие – некоторая операция, изменяющая внешнюю среду.

Система автоматов – совокупность автоматов.

Состояние системы – совокупность состояний всех автоматов системы.

Системный переход – переход системы в другое состояние. Системный переход начинается с запуска некоторого автомата A_i. Этот автомат может запускать другие автоматы. После того как автомат A_i и каждый из запущенных им автоматов совершит не более одного перехода, считается, что системный переход осуществлен.

Исключительная ситуация – возникает в случае невозможности опроса входного воздействия или выполнения выходного воздействия.

Этап автомата – набор последовательных запусков, в течение которых автомат сохраняет свое состояние.

Шаг этапа – каждый запуск автомата в течение этапа.

Действие в состоянии – вызов некоторого выходного воздействия, происходящий во время первого шага этапа автомата.

Деятельность в состоянии – вызов некоторого выходного воздействия, происходящий на каждом шаге этапа.

Действие на переходе – вызов некоторого выходного воздействия, происходящий при переходе автомата из одного состояния в другое. При этом сначала выполняются действия на переходе, а затем изменяется состояние автомата.

2.2. Архитектура библиотеки *STOOL*

Предлагаемая архитектура программных систем отличается от архитектуры, предлагаемой в работах [27–29, 53]. Эти отличия состоят в следующем:

- автоматы, входные и выходные воздействия являются объектами;
- явно вводятся понятия класс автоматов и экземпляр класса автоматов;
- каждый автомат не располагает никакой информацией о других автоматах системы. При этом, во-первых, автомат не знает о существовании других автоматов, во-вторых, он не может непосредственно проверять их состояния, как это было предложено, например, в работе [6], а в-третьих – не может непосредственно запускать другие автоматы. Единственная связь автомата с «внешним миром» – это входные и выходные воздействия;
- автоматные объекты содержат только свое управляющее состояние. Вычислительное состояние вынесено за пределы автоматов и хранится в специальном объекте – *окружении*. Автоматы получают доступ к вычислительному состоянию исключительно через входные и выходные воздействия.

Введение понятий *класс автоматов* и *экземпляр класса* автоматов необходимо для обеспечения повторного использования автоматов. Пусть, на-

пример, в системе имеются два одинаковых автомата, отличающиеся только входными и выходными воздействиями. При этом нет необходимости каждый из них проектировать отдельно, а достаточно создать и включить в систему два *экземпляра* одного и того же *класса автоматов*, что соответствует парадигме объектно-ориентированного программирования (ООП). Разработанный класс автоматов может повторно использоваться и в других системах.

Введенные ограничения на взаимодействие автоматов повышает их модульность, и тем самым, увеличивают вероятность их повторного использования.

При этом, если необходимо, например, осуществить переход по номеру состояния другого автомата, то вводится входное воздействие, возвращающее значение `true`, если автомат находится в искомом состоянии. Если же необходимо осуществить запуск другого автомата, то вводится выходное воздействие, запускающее этот автомат.

Поясним теперь, почему в рамках предлагаемого подхода понятие «событие» не применяется. Это связано с переносимостью библиотеки ***STOOL***, на базе которой предлагается реализовывать системы. В разных системах существуют различные модели возникновения и обработки событий, которые весьма трудно обобщить в понятной и краткой форме. Введение в библиотеку какой-нибудь одной модели обработки событий сузило бы область применения библиотеки, а введение нескольких моделей усложнило бы дизайн библиотеки и простоту ее использования. Поэтому в рамках предлагаемой архитектуры реализация механизмов обработки событий возлагается на пользователя. Так, в случае использования операционной системы ***Windows***, пользователь должен самостоятельно создавать и регистрировать обработчики событий этой системы, логика которых, однако, вынесена в автоматы.

Пусть, например, для левой кнопки мыши введено входное воздействие x_0, возвращающее значение `true`, если кнопка нажата. При получении уве-

домления от пользовательского интерфейса о том, что кнопка нажата (например, событие `WM_LBUTTONDOWN` в ОС ***Windows***), пользователь устанавливает соответствующий флаг и запускает автоматы, обрабатывающие это событие. При получении уведомления о том, что кнопка отпущена, пользователь сбрасывает флаг. Таким образом, понятие события вытесняется из системы и заменяется совокупностью входного воздействия и внешнего, по отношению к библиотеке ***STOOL***, обработчика событий.

2.3. Обзор классов библиотеки *STOOL*

Библиотека ***STOOL*** предоставляет абстрактные базовые классы для реализации автоматов, входных и выходных воздействий, а также «инфраструктуру» для организации системы в целом (рис. 17).

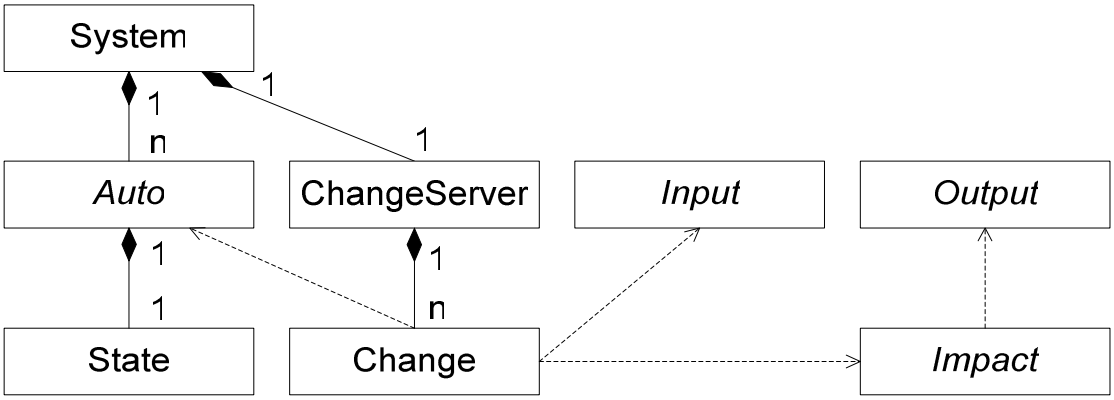


Рис. 17. Диаграмма основных классов библиотеки ***STOOL***

Для реализации автоматов, входных и выходных воздействий необходимы следующие классы библиотеки ***STOOL***:

- `Auto` – базовый класс для разработки классов автоматов. Для определения класса автоматов программист должен переопределить метод `execution()`, реализующий граф переходов автомата;
- `State` – класс, который хранит состояние автомата. Каждый экземпляр класса `Auto` содержит экземпляр класса `State`. Класс `State` обеспечивает также протоколирование любого (даже ошибочного) изменения состояния автомата. Изменять экземпляр

класса `State` программист может только внутри метода `execution()`, реализующего граф переходов;

- `Input` – базовый класс для реализации входных воздействий;
- `Output` – базовый класс для реализации выходных воздействий;
- `Impact` – класс, описывающий процесс выполнения выходного воздействия. Он предоставляет следующие методы: «Выполнить», «Откатить» и «Подтвердить». При каждом выполнении выходного воздействия создается соответствующий ему экземпляр класса `Impact`, который и осуществляет это воздействие.

Ниже приведено описание классов, необходимых для создания «инфраструктуры» системы:

- `System` – класс, управляющий системой автоматов. Содержит экземпляр класса `ChangeServer`, описанного ниже. Он также хранит список всех автоматов системы. Предполагается, что пользователь может создавать потомков класса `System`;
- `Change` – класс, управляющий системным переходом. При возникновении исключительной ситуации во время выполнения системного перехода этот класс отвечает за разворачивание (*unwind*) стека выполненных выходных воздействий. Класс `Change` является абстрактным. Библиотека ***STOOL*** предоставляет два потомка этого класса – `SingleTaskChange` и `MultiTaskChange` (для однопоточной и многопоточной работы). Эти классы могут быть расширены пользователем;
- `ChangeServer` – класс, управляющий созданием и уничтожением переходов. Он разработан для использования в однопоточных и в многопоточных системах.

Дальнейшее описание библиотеки приводится в последующих разделах работы. Так, например, описываются методы `Output::action()`, `Output::activity()` и `Output::jumpAction()`, реализующие различные типы выходных воздействий.

2.4. Основные возможности

2.4.1. Выделение состояния системы в целом

Клиент библиотеки ***STOOL*** может получить список всех автоматов системы, а у каждого автомата – его состояние. Пользователь может также получить информацию о любом выполняющемся в данный момент системном переходе.

Информацию обо всех автоматах системы можно получить с помощью метода `System::enumerate()`, а обо всех выполняющихся переходах – с помощью экземпляра класса `ChangeServer`, возвращаемого методом `System::getChangeServer()`.

Приведем пример кода, распечатывающий список автоматов системы, в котором используется метод `System::enumerate()`:

```
/*...*/
struct AutoReceiver
  : public ItemsReceiver<const Auto&>
{
  virtual bool receiveCount( int _count ) {
    cout << "всего автоматов: " << _count << endl;
    return true;
  }
  virtual bool receiveItem( int _index, const Auto& _item )
  {
    cout << "автомат #" << _index << ": " << endl;
    cout << "  имя класса автоматов: "
         << _item.getInfo().getClassName() << endl;
    cout << "  имя автомата: "
         << _item.getInfo().getInstanceName() << endl;
    cout << "  состояние: "
         << _item.getInfo().getStateName( _item.getState() ) << endl;
    return true;
  }
};
/*...*/
system.enumerate( AutoReceiver() );
```

В строках 2–20 определяется класс `AutoReceiver`, получающий список автоматов. Экземпляр класса `AutoReceiver` передается в метод `enumerate()` в строке 22. Класс `AutoReceiver` перегружает два метода своего базового класса `ItemsReceiver`:

- метод `receiveCount()`, получающий количество автоматов;
- метод `receiveItem()`, последовательно получающий автоматы системы.

2.4.2. Действия и деятельности

Для упрощения шаблона, реализующего автоматы в ***SWITCH***-технологии [29], в котором используются два оператора `switch`, в библиотеке ***STOOL*** имеются средства для различения действий и деятельностей в вершинах.

Деятельности в вершинах реализуется методом `Output::activity()`, а действия – методом `Output::action()`. Объект выходного воздействия сам определяет, находится ли вызывающий автомат в первом шаге *этапа* (разд. 2.1). Для выполнения действия на переходе предназначен метод `Output::jumpAction()`.

При описании поведения автомата в виде графа переходов действие в вершине и действие на переходе будем обозначать символом z_i, а деятельность – символом z_i, взятым в скобки.

Рассмотрим граф переходов, показанный на рис. 18.

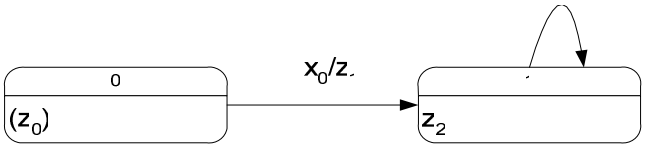


Рис. 18. Действия и деятельности в состоянии

Ниже приведена функция, реализующая граф переходов (рис. 18) с использованием библиотеки ***STOOL***:

```
001 virtual void execution( State& state ) {
002   switch ( state ) {
003   case 0:
004     io.z0().activity( *this );
005     if ( io.x0().is( *this ) ) {
006       io.z1().jumpAction( *this );
007       state = 1;
008     }
009     break;
010   case 1:
011     io.z2().action( *this );
012     break;
013   }
014 }
```

В этом примере действия z_1 и z_2 выполняются не более одного раза.

2.4.3. Повторное использование автоматов

Автоматы предлагается реализовывать на основе паттерна, приведенного ниже:

```
001 class Ai : public Auto {
002 public:
003   struct IO {
004     virtual Input& xk() = 0;
/*...*/
006     virtual Input& xl() = 0;
007
008     virtual Output& zm() = 0;
/*...*/
010     virtual Output& zn() = 0;
011   };
012 private:
013   IO& io;
014 protected:
015   virtual void execution( State& state ) {
016     switch ( state ) {
017     case 0:
018       if ( io.xk().is( *this ) )
019         state = 1;
/*...*/
021       break;
/*...*/
023     }
024   }
025 public:
026   Ai( IO& _io, const string& _instance_name, System& _system )
027     : Auto( _instance_name, "Ai", _system )
028     , io( _io ) {}
029 };
```

Класс `Ai`, являющийся потомком класса `Auto`:

- специфицирует необходимый ему набор входных (`I`) и выходных (`O`) воздействий, определяя вложенный класс `IO`, каждый метод которого задает одно воздействие;
- реализует граф переходов, перегружая метод `execution()` базового класса `Auto`;
- определяет конструктор, принимающий набор входных и выходных воздействий (`_io`), имя автомата (`_instance_name`) и ссылку на систему автоматов (`_system`).

Такая реализация автомата не зависит от окружения, что обеспечивает возможность повторного использования классов автоматов.

2.4.4. Автоматическое протоколирование

Библиотека ***STOOL*** предоставляет средства для организации автоматического протоколирования изменений состояний системы. При разработке системы программист должен создать объект, реализующий детали протоколирования (куда записывается протокол, формат протокола и т.д.). Библиотека поддерживает автоматическое протоколирование:

- изменений состояния каждого автомата системы;
- создания и уничтожения автомата;
- начала и конца выполнения автомата;
- опроса входного воздействия;
- выполнения выходного воздействия;
- возникновения исключительной ситуации.

Гарантируется, что объект, реализующий протоколирование, будет уведомлен обо всех вышеуказанных изменениях.

Ниже приведен пример организации протоколирования:

```
001 class LogSystem : public AutoEventSync {
```

```
002   int change_number;
003 public:
004   LogSystem( Lockable& _locakble )
005     : AutoEventSync( _locakble )
006     , change_number( 0 ) {}
007
008   void preamble() {
009     cout << ++change_number << ".\t";
010   }
011
012   void onEvent( const Event _event,
013                 const AutoEventSync::EventItem& _item ) {
014     Lock lock( *this );
015     switch ( _event ) {
016     case AutoEventSync::E_AFTER_STATE_CHANGED:
017       preamble();
018       const AutoEventSync::StateEventData& data =
019               _item.getStateEventData();
020       const Auto& inst = data.state.getAuto();
021       cout << "автомат " << inst.getInfo().getInstanceName()
022            << "(" <<  inst.getInfo().getClassName() << ")"
023            << " перешел в состояние: " << data.state
024            << " (старое состояние: " << data.old_state << ")"
025            << "." << std::endl;
026       break;
/*...*/
028     }
029   }
030 };
```

Класс `LogSystem` перехватывает сообщения об изменениях, происходящих в системе, перегружая метод `onEvent()` своего базового класса `AutoEventSync`. В методе `onEvent()` выполняется фильтрация по типу и протоколирование поступающих сообщений. Приведенный исходный код класс `LogSystem` протоколирует сообщения об изменении состояний автоматов.

2.4.5. Механизм обработки ошибок

Для обеспечения устойчивости системы к ошибкам (исключительным ситуациям) вводится следующее ограничение: если не удалось осуществить системный переход, то система остается в исходном состоянии. Таким образом, если причина возникновения исключительной ситуации будет устранена, то при повторном запуске системный переход осуществится так, как если бы исключительной ситуации не возникало. Это аналогично транзакциям в

системах баз данных, которые либо выполняются полностью, либо не выполняются вообще.

Для обеспечения устойчивости к ошибкам все выходные воздействия должны иметь возможность отката. Для этого выходные воздействия должны поддерживать следующие три операции:

- «Выполнить». Воздействие выполняется, но при этом сохраняется вся информация, необходимая для отката. Если реализовать откат *сложно*, то захватываются ресурсы, необходимые для выполнения воздействия, а само воздействие выполняется в операции «Подтвердить»;
- «Откатить». Если воздействие было выполнено, то производится откат выполненных изменений. Если были захвачены ресурсы, то они освобождаются;
- «Подтвердить». Если воздействие не было выполнено в операции «Выполнить», то оно выполняется. Ресурсы освобождаются. Вероятность возникновения исключительной ситуации во время выполнения этой операции должна быть сведена к минимуму.

Например, необходимо реализовать выходное воздействие, удаляющее некоторый файл. В операции «Выполнить» блокируется доступ к файлу. Если блокировать файл не удалось (например, он заблокирован другим приложением), то генерируется исключительная ситуация. В операции «Откатить» отменяется блокировка файла, а в операции «Подтвердить» удаляется файл.

Если программист не нуждается в обеспечении устойчивости к ошибкам, то для всех выходных воздействий он может реализовать только операцию «выполнить».

2.4.6. Параллельные вычисления

Для решения многих задач бывает целесообразно организовать несколько потоков выполнения (многопоточные системы). В терминах систем с

явным выделением состояний многопоточная система – это система, в которой в один и тот же момент времени может осуществляться несколько системных переходов.

Библиотека ***STOOL*** может работать как в однопоточном, так и в многопоточном режиме. Переключение между этими режимами осуществляется путем передачи конструктору объекта `System` экземпляра фабрики [49], отвечающего за создание объектов `Change`, `ChangeServer` и `Lockable`. В библиотеке реализованы две фабрики:

- класс `Factory`, настраивающий библиотеку на однопоточный режим;
- класс `AsyncFactory`, настраивающий библиотеку на многопоточный режим. Поддержка многопоточности реализована в библиотеке ***STOOL*** с помощью библиотеки ***boost::thread*** [55].

Любой объект системы может одновременно использоваться только в одном переходе. Рассмотрим следующий пример. Пусть одновременно, в разных потоках, выполняются два системных перехода T_0 и T_1, которые обращаются к входному воздействию x_0. Процесс разделения объекта x_0 между переходами показан на рис. 19.

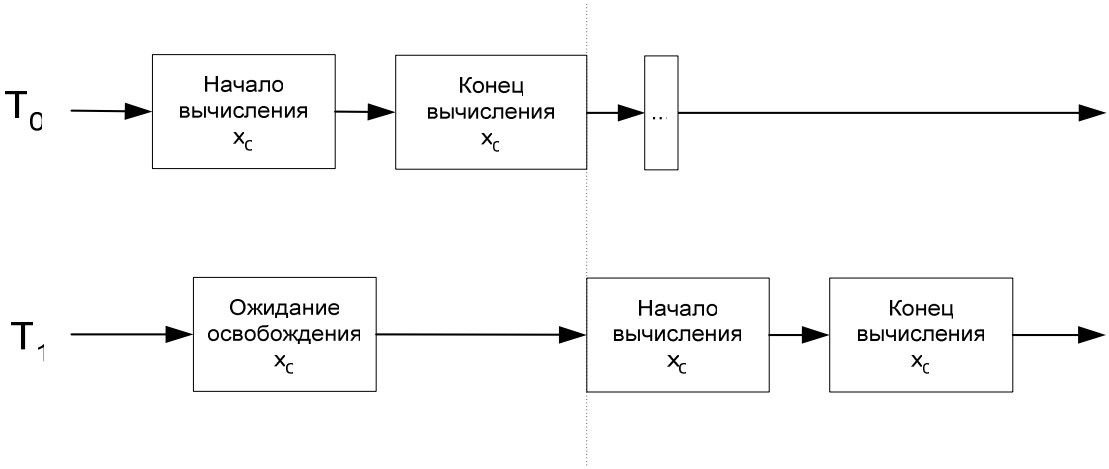


Рис. 19. Разделение объекта между системными переходами

Переход T_0 первым начинает обращение к x_0. Переход T_1 ожидает завершения вычисления значения x_0 переходом T_0 и сразу после этого начинается вычисление значения x_0 переходом T_1.

Автоматы являются единственными объектами библиотеки ***STOOL***, которые остаются «захваченными» до завершения активизирующего их потока. Если системный переход T_0 запускает некоторый автомат i_0, то объект, представляющий этот автомат, остается «захваченным» до завершения этого перехода. Процесс разделения автомата i_0 между переходами показан на рис. 20.

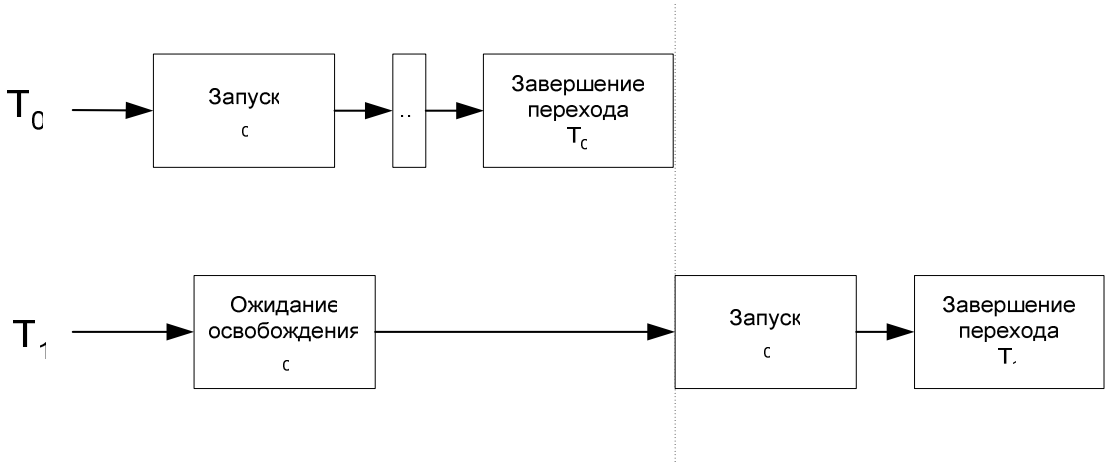


Рис. 20. Разделение автомата между системными переходами

Переход T_0 первым запускает автомат i_0. Переход T_1 ожидает полного завершения перехода T_0 и только после этого запускает автомат i_0.

2.4.7. Реализация входных и выходных воздействий

Для реализации входного воздействия создается потомок класса `Input`, исходный код которого приведен ниже:

```
001 class X0 : public Input {
002   Data& data;
003 protected:
004   virtual bool execution() {
005     return data.isEnd();
006   }
007 public:
008   X0( Data& _data, System& _system )
009     : Input( "X0", _system )
```

```
010     , data( _data ) {}
011 };
```

Класс `X0` является потомком класса Input и переопределяет метод `execution()`. Конструктор класса `X0` принимает указатель на *окружение* `Data`, в котором хранится вычислительное состояние. Также конструктор класса `X0` принимает указатель на систему автоматов.

Для реализации выходного воздействия необходимо выполнить следующие два действия:

- определить потомка класса `Impact`, например класс `IZ0`;
- определить потомка класса `Output`, создающий экземпляр класса `IZ0`.

Ниже приведен каркас реализации потомка класса `Impact` и соответствующего ему потомка класса `Output`:

```
001 class IZ0 : public Impact {
002 public:
003   virtual void execute() {
004     //действие
005   }
006   virtual void rollback(){
007     //откат
008   }
009   virtual void commit() {
010     //подтвеждение
011   }
012 };
013
014 class Z0 : public Output {
015 public:
016   virtual Impact* create() {
017     return new IZ0();
018   }
019 public:
020   Z0( System& _system )
021     : Output( "Z0", _system ) {}
022 };
```

Классы `IZ0` и `Z0` определяют некоторое выходное воздействие z_0. Класс `IZ0` перегружает методы `execute()`, `rollback()` и `commit()` своего базового класса `Impact`, реализуя в них операции «Выполнить», «Откатить» и «Подтвердить» соответственно. Класс `Z0` перегружает метод

`create()` своего базового класса `Output`, создавая в нем экземпляр класса `IZ0`.

Недостатком изложенного является тот факт, что для реализации выходных воздействий программист должен реализовать два класса. Для устранения этого недостатка, библиотека ***STOOL*** предоставляет служебные классы, позволяющие сократить объем кода.

Для реализации выходных воздействий, запускающих другой автомат, предназначен вспомогательный класс `AutoOutput`, принимающий запускаемый автомат в качестве параметра конструктора.

Для того, чтобы не создавать два класса для реализации выходных воздействий, предназначен вспомогательный класс `GOutput`. Например, для создания выходного воздействия на основе класса `IZ0` достаточно следующей строки кода:

```
001 GOutput<IZ0> z0( "Z0", system );
```

В случае, если входные и выходные воздействия представлены набором функций, для упрощенного создания соответствующих объектов можно воспользоваться вспомогательными функциями `makeFInput` и `makeFOutput`:

```
001 bool fx0() {
002   //реализация входного воздействия x0
003   return false;
004 }
005
006 void fz0() {
007   //реализация выходного воздействия z0
008 }
/*...*/
010 Input* x0 = makeFInput( fx0, "x0", system );
011 Output* z0 = makeFOutput( fz0, "z0", system );
```

Функция $fx_0()$ является свободной функцией, реализующей входное воздействие x_0. Функция `fz0()` является свободной функцией, реализующей процесс выполнения выходного воздействия `z0`. В строках 10 и 11 создаются объекты `x0` и `z0`, реализующие соответствующие входное и выходное воздействия через функции `fx0()` и `fz0()`. Данный код заменяет руч-

ное создание потомков классов `Impact` и `Output`, как это выполнялось в предыдущем примере.

Если входные и выходные воздействия представлены методами некоторого класса, то можно воспользоваться функциями `makeFInput` и `makeFOutput` в комбинации с библиотекой `boost::bind` [55]:

```
001 class Ctx {
002 public:
003   bool fx0() {
004     //реализация входного воздействия x0
005     return false;
006   }
007
008   void fz0() {
009     //реализация выходного воздействия z0
010   }
011 };
/*...*/
013 Ctx ctx;
014 Input* x0 = makeFInput( bind( &Ctx::fx0, ref( ctx ) ), "x0", system );
015 Output* z0 = makeFOutput( bind( &Ctx::fz0, ref( ctx ) ), "z0", system );
```

В данном случае воздействия `x0` и `z0` реализуются не свободными функциями, как в предыдущем примере, а методами класса `Ctx`. Использование библиотеки `boost::bind` для связывания входных и выходных воздействий с методами объекта `ctx` продемонстрировано в строках 14 и 15 приведенного исходного кода.

В случае если объекты автоматов, входных и выходных воздействий создаются как члены некоторого класса, то могут быть полезны следующие макросы:

`DECLARE_FUNC_INPUT` – объявляет метод, возвращающий экземпляр входного воздействия, которое вызывает некоторую функцию;

`DECLARE_FUNC_OUTPUT` – объявляет метод, возвращающий экземпляр выходного воздействия, которое вызывает некоторую функцию;

`DECLARE_AUTO_OUTPUT` – объявляет метод, возвращающий экземпляр выходного воздействия, которое запускает некоторый автомат;

`DECLARE_AUTO` – объявляет метод, возвращающий экземпляр класса автоматов.

Использование этих макросов позволяет упростить реализацию объектов системы, что будет показано на примере в следующем разделе.

2.5. Пример использования библиотеки *STOOL*

Пусть на вход подается строка символов, завершающаяся нулем. *Словом* назовем непрерывную последовательность не пробельных символов. *Числом* назовем слово, состоящее только из цифр. Необходимо за один проход перевернуть все числа в строке и подсчитать количество слов. Например, при входной строке `"test 123"`, результирующей строкой будет `"test 321"`, а количество слов будет равно двум.

2.5.1. Проектирование и реализация автоматов

Для решения задачи требуется три класса автоматов:

- A_0 – автомат, управляющий итерированием по строке;
- A_1 – автомат, управляющий переворотом чисел;
- A_2 – автомат, управляющий подсчетом количества слов.

Создадим по одному экземпляру каждого класса автоматов (i_0, i_1 и i_2). На рис. 21 приведена схема взаимодействия автоматов.

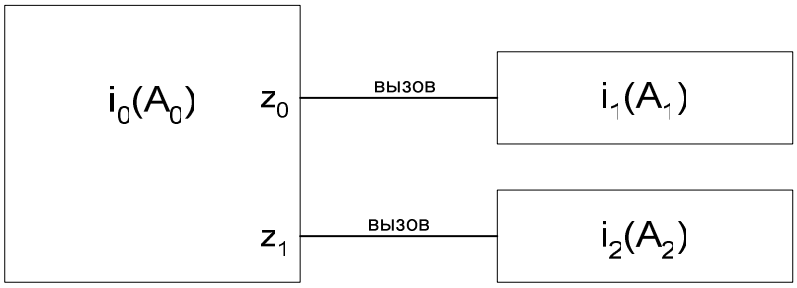


Рис. 21. Схема взаимодействия автоматов

Структурные схемы классов автоматов A_0, A_1 и A_2 приведены на рис. 22.

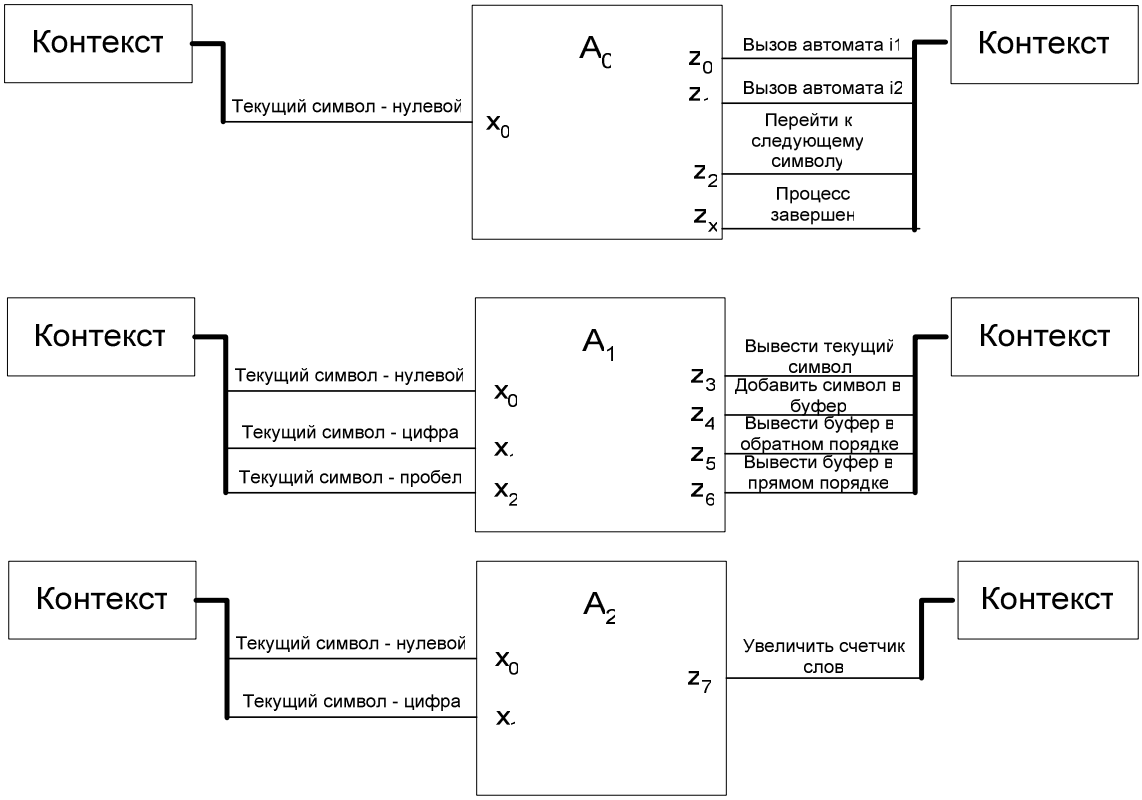


Рис. 22. Структурные схемы классов автоматов A_0, A_1 и A_2

Графы переходов автоматов A_0, A_1 и A_2 приведены на рис. 23.

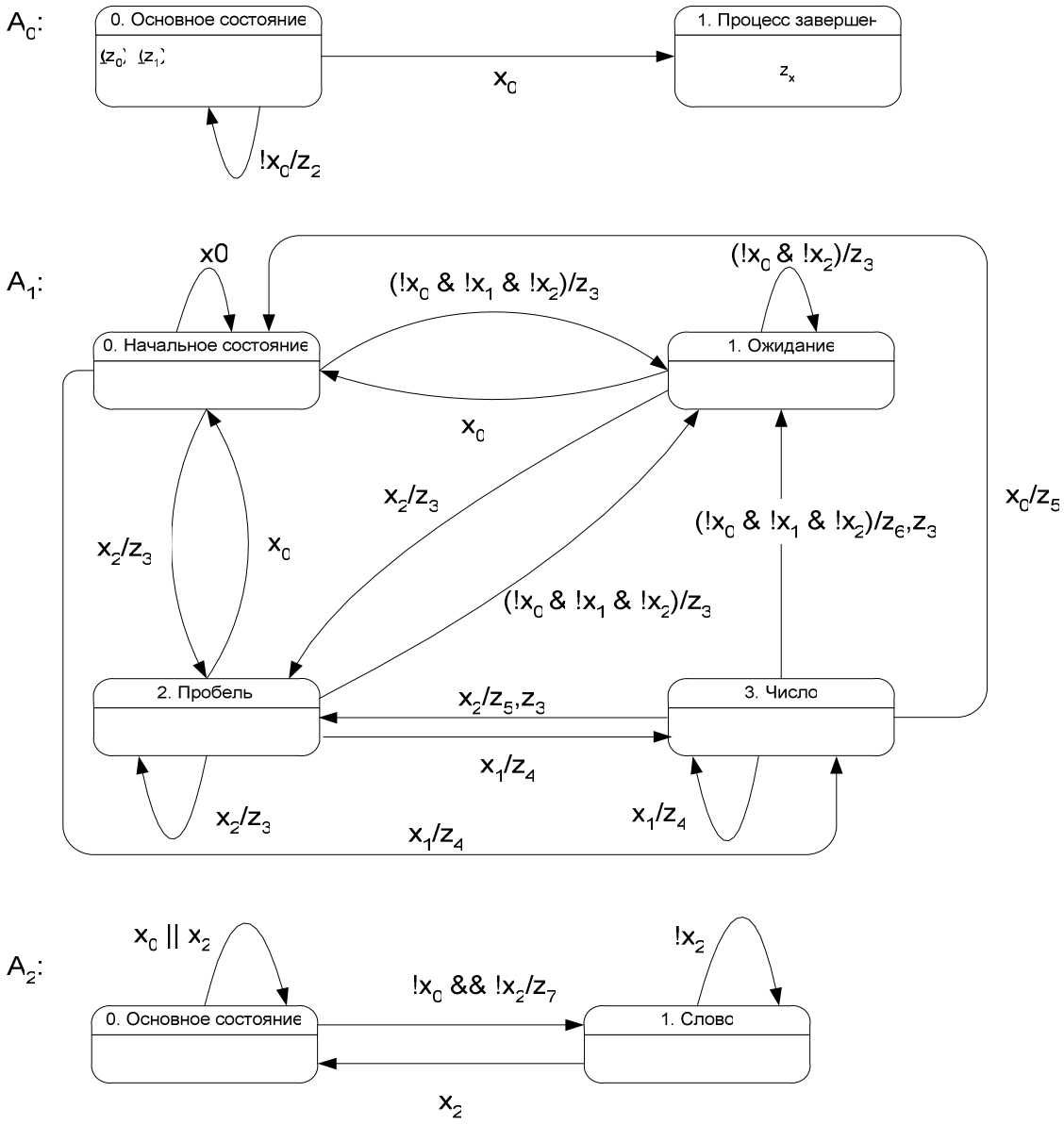


Рис. 23. Графы переходов автоматов A_0, A_1 и A_2

Обратим внимание, что в первом графе переходов использован символ z_x, соответствующий завершению работы системы.

Автоматы A_0, A_1 и A_2 реализуются на основе изложенного подхода следующим образом:

```
001 class A0 : public Auto {
002 public:
003   struct IO {
004     virtual Input& x0() = 0;
005     virtual Output& z0() = 0;
006     virtual Output& z1() = 0;
007     virtual Output& z2() = 0;
008     virtual Output& zx() = 0;
009   };
```

```
010 private:
011   IO& io;
012 protected:
013   virtual void execution( State& state ) {
014     switch ( state ) {
015     case 0:
016       io.z0().activity( *this );
017       io.z1().activity( *this );
018       if ( io.x0().is( *this ) )
019         state = 1;
020       else if ( !io.x0().is( *this ) )
021         io.z2().jumpAction( *this );
022       break;
023     case 1:
024       io.zx().action( *this );
025       break;
026     }
027   }
028 public:
029   A0( IO& _io, const string& _instance_name, System& _system )
030     : Auto( _instance_name, "A0", _system )
031     , io( _io ) {}
032 };
033
034 class A1 : public Auto {
035 public:
036   struct IO {
037     virtual Input& x0() = 0;
038     virtual Input& x1() = 0;
039     virtual Input& x2() = 0;
040     virtual Output& z3() = 0;
041     virtual Output& z4() = 0;
042     virtual Output& z5() = 0;
043     virtual Output& z6() = 0;
044   };
045 private:
046   IO& io;
047 protected:
048   virtual void execution( State& state ) {
049     switch ( state ) {
050     case 0:
051       if ( !io.x0().is( *this ) && !io.x1().is( *this ) &&
052            !io.x2().is( *this ) ) {
053         io.z3().jumpAction( *this );
054         state = 1;
055       } else if ( io.x1().is( *this ) ) {
056         io.z4().jumpAction( *this );
057         state = 3;
058       } else if ( io.x2().is( *this ) ) {
059         io.z3().jumpAction( *this );
060         state = 2;
061       } else if ( io.x0().is( *this ) )
062       {}
063       break;
064     case 1:
065       if ( !io.x0().is( *this ) && !io.x2().is( *this ) )
066         io.z3().jumpAction( *this );
```

```
067       else if ( io.x2().is( *this ) ) {
068         io.z3().jumpAction( *this );
069         state = 2;
070       } else if ( io.x0().is( *this ) )
071         state = 0;
072       break;
073     case 2:
074       if ( io.x1().is( *this ) ) {
075         io.z4().jumpAction( *this );
076         state = 3;
077       } else if ( io.x2().is( *this ) )
078         io.z3().jumpAction( *this );
079       else if ( !io.x0().is( *this ) && !io.x1().is( *this ) &&
080                 !io.x2().is( *this ) ) {
081         io.z3().jumpAction( *this );;
082         state = 1;
083       } else if ( io.x0().is( *this ) )
084         state = 0;
085       break;
086     case 3:
087       if ( io.x1().is( *this ) ) {
088         io.z4().jumpAction( *this );
089       } else if ( io.x2().is( *this ) ) {
090         io.z5().jumpAction( *this );;
091         io.z3().jumpAction( *this );;
092         state = 2;
093       } else if ( io.x0().is( *this ) ) {
094         io.z5().jumpAction( *this );;
095         state = 0;
096       } else if ( !io.x0().is( *this ) && !io.x1().is( *this ) &&
097                   !io.x2().is( *this ) ) {
098         io.z6().jumpAction( *this );;
099         io.z3().jumpAction( *this );;
100         state = 1;
101       }
102       break;
103     }
104   }
105 public:
106   A1( IO& _io, const string& _instance_name, System& _system )
107     : Auto( _instance_name, "A1", _system )
108     , io( _io ) {}
109 };
110
111 class A2 : public Auto {
112 public:
113   struct IO {
114     virtual Input& x0() = 0;
115     virtual Input& x2() = 0;
116     virtual Output& z7() = 0;
117   };
118 private:
119   IO& io;
120 protected:
121   virtual void execution( State& state ) {
122     switch ( state ) {
123     case 0:
```

```
124       if ( !io.x0().is( *this ) && !io.x2().is( *this ) ) {
125         io.z7().jumpAction( *this );
126         state = 1;
127       } else if ( io.x0().is( *this ) || io.x2().is( *this ) )
128       {}
129       break;
130     case 1:
131       if ( io.x2().is( *this ) ) {
132         state = 0;
133       }
134     }
135   }
136 public:
137   A2( IO& _io, const string& _instance_name, System& _system )
138     : Auto( _instance_name, "A2", _system )
139     , io( _io ) {}
140 };
```

Приведенная реализация классов автоматов изоморфна графам переходов, что в значительной мере упрощает кодирование и сопровождение программных систем.

2.5.2. Реализация окружения

Завершив построение автоматной части программы, перейдем к реализации ее *окружения*. Если система автоматов является управляющей частью, то *окружение* является управляемой частью. Окружение хранит вычислительное состояние системы и предоставляет разнообразные вспомогательные функции. Исходный код окружения `Data` приведен ниже.

```
001 class Data {
002   string in;
003   string out;
004   string buffer;
005   string::iterator cursor;
006   int nwords;
007 public:
008   Data( const string& _in )
009     : in( _in )
010     , nwords( 0 )
011     , cursor( in.begin() ) {}
012   bool isEnd() { return in.end() == cursor; }
013   bool isDigit() { return isdigit( *cursor ) != 0; }
014   bool isSpace() { return isspace( *cursor ) != 0; }
015   void forward() { cursor++; }
016   void back() { cursor--; }
017   void output() { out += *cursor; }
018   void push() { buffer += *cursor; }
019   void outBuffer() { out = out + buffer; buffer.clear(); }
```

```
020   void outBufRev()
021    { reverse( buffer.begin(), buffer.end() ); outBuffer(); }
022   void incWords() { nwords++; }
023   string getOutput() const { return out; }
024   int getWordsCount() const { return nwords; }
025 };
```

Класс `Data` содержит следующие поля:

- `in` – входная строка;
- `out` – выходная строка;
- `buffer` – буфер для переворота чисел;
- `cursor` – указатель на текущую позицию во входной строке;
- `nwords` – счетчик количества слов.

Класс `Data` предоставляет также разнообразные служебные методы, которые используются в качестве входных и выходных воздействий.

2.5.3. Связывание системы автоматов

Класс `MySystem` (потомок класса `System`) создает и связывает между собой автоматы, входные и выходные воздействия, а также *окружение*. Исходный код класса `MySystem` приведен ниже.

```
001 class MySystem : public System,
002   public A0::IO, public A1::IO, public A2::IO
003 {
004   Data data;
005   bool stopped;
006
007   DECLARE_AUTO(A0, i0, *this, *this);
008   DECLARE_AUTO(A1, i1, *this, *this);
009   DECLARE_AUTO(A2, i2, *this, *this);
010
011   DECLARE_FUNC_INPUT ( x0, bind( &Data::isEnd,     ref( data) ),*this);
012   DECLARE_FUNC_INPUT ( x1, bind( &Data::isDigit,   ref( data) ),*this);
013   DECLARE_FUNC_INPUT ( x2, bind( &Data::isSpace,   ref( data) ),*this);
014   DECLARE_AUTO_OUTPUT( z0, i1(), *this );
015   DECLARE_AUTO_OUTPUT( z1, i2(), *this );
016   DECLARE_FUNC_OUTPUT( z2, bind( &Data::forward,   ref( data) ),*this);
017   DECLARE_FUNC_OUTPUT( z3, bind( &Data::output,    ref( data) ),*this);
018   DECLARE_FUNC_OUTPUT( z4, bind( &Data::push,      ref( data) ),*this);
019   DECLARE_FUNC_OUTPUT( z5, bind( &Data::outBufRev, ref( data) ),*this);
020   DECLARE_FUNC_OUTPUT( z6, bind( &Data::outBuffer, ref( data) ),*this);
021   DECLARE_FUNC_OUTPUT( z7, bind( &Data::incWords,  ref( data) ),*this);
```

```
022   DECLARE_FUNC_OUTPUT( zx, bind( &MySystem::stop,  ref( *this)),*this);
023 protected:
024   virtual void stop() { stopped = true; }
025 public:
026   MySystem( const string& _in, Factory& _factory )
027     : System( _factory )
028     , data( _in )
029     , stopped( false ) {}
030
031   void run() {
032     while ( !stopped )
033       getChangeServer().start( i0() );
034   }
035
036   const Data& getData() const { return data; }
037 };
```

Класс `MySystem`:

- создает автоматы `i0`, `i1` и `i2` с помощью макроса `DECLARE_AUTO` в строках 7–9;
- связывает входные воздействия `x0`, `x1` и `x2` с методами окружения `isEnd()`, `isDigit()` и `isSpace()` с помощью макроса `DECLARE_FUNC_INPUT` в строках 11–13;
- связывает выходные воздействия `z0` и `z1` с вызовом автоматов `i1` и `i2` с помощью макроса `DECLARE_AUTO_OUTPUT` в строках 14–15;
- связывает выходные воздействия `z2-z7` с методами окружения `forward()`, `output()` и т. д. с помощью макроса `DECLARE_FUNC_OUTPUT` в строках 16–21;
- связывает выходное воздействие `zx` с собственным методом `stop()` с помощью макроса `DECLARE_FUNC_OUTPUT` в строке 22.

Метод `run()` класса `MySystem` в цикле запускает автомат `i0` до тех пор, пока поле `stopped` не будет установлено в `true` с помощью вызова метода `stop()`.

2.5.4. Протоколирование работы системы

Ниже приведена реализация класса `LogSystem`, отвечающего за протоколирование работы системы.

```
001 class LogSystem : public AutoEventSync {
002   int change_number;
003   ostream& stream;
004 public:
005   LogSystem( Lockable& _locakble, ostream& _stream )
006     : AutoEventSync( _locakble )
007     , change_number( 0 )
008     , stream( _stream ) {}
009
010   void preamble() {
011     stream << ++change_number << ".\t";
012   }
013
014   void onEvent(const Event _event,const AutoEventSync::EventItem&_item){
015     Lock lock( *this );
016     switch ( _event ) {
017     case AutoEventSync::E_AFTER_STATE_CHANGED:
018       preamble();
019       stream << "автомат "
020 << _item.getStateEventData().state.getAuto().getInfo().getInstanceName()
021 <<"(" <<
022 _item.getStateEventData().state.getAuto().getInfo().getClassName()
023 << ")" << " перешел в состояние: "
024 << _item.getStateEventData().state
025 << " (старое состояние: " << _item.getStateEventData().old_state << ")"
026 << "." << std::endl;
027       break;
028     case AutoEventSync::E_AFTER_INPUT_CHECKED:
029       preamble();
030       stream << "опрошено входное воздействие "
031         << _item.getInputResultEventData().input.getName()
032         << ", результат: " << _item.getInputResultEventData().result
033         << "." << std::endl;
034       break;
035     case AutoEventSync::E_AFTER_OUTPUT_ACTIVATED:
036       preamble();
037       stream << "выполненно выходное воздействие "
038         << _item.getOutputEventData().output.getName()<<"."<< std::endl;
039       break;
040     case AutoEventSync::E_AFTER_EXCEPTION:
041       preamble();
042       std::exception* ex = _item.getExceptionEventData().exception;
043       if ( ex )
044         stream << "возникло исключение: " << ex->what();
045       else
046         stream << "возникло неизвестное исключение";
047       stream << std::endl;
048       break;
049     }
050   }
051 };
```

Класс `LogSystem` перегружает метод `onEvent()` своего базового класса `AutoEventSync`, фильтрует поступающие сообщения об изменении состояния системы по типу и выводит соответствующие текстовые сообщения в поток протокола.

2.5.5. Запуск системы автоматов

Функция `main`, являющаяся точкой входа демонстрационного приложения, реализуется следующим образом:

```
001 int main()
002 {
003   const int INPUT_SIZE = 256;
004   char input[ INPUT_SIZE + 1 ];
005   cout << "input string: ";
006   cin.getline( input, INPUT_SIZE );
007   Factory factory;
008   ofstream log_stream("log.txt");
009   LogSystem log( factory.makeLockable(), log_stream );
010   MySystem system( input, factory );
011   system.getAutoEventService().inject( log );
012   system.run();
013   cout << "output: " << system.getData().getOutput()
014        << endl;
015   cout << "words count: "
016        << system.getData().getWordsCount() << endl;
017   cout << "press any key...";
018   _getch();
019   return 0;
020 }
```

В строках 3–6 читается входная строка. В строке 7 создается экземпляр класса `Factory`, настраивающий библиотеку ***STOOL*** на однопоточный режим работы. В строках 8 и 9 создается экземпляр класса `LogSystem`, отвечающий за протоколирование. В строке 10 создается экземпляр класса `MySystem`, который создает и связывает между собой все необходимые автоматы, входные и выходные воздействия и окружение. В строке 11 система протоколирования подключается к системе автоматов. В строке 12 осуществляется вызов метода `run()` класса `MySystem`, итеративно запускающий автоматы системы до тех пор, пока не будет выполнено воздействие z_x, завер-

шающее выполнение системы. В строках 13–18 результат работы программы выводится на экран.

Как следует из рассмотренного примера, его реализация практически не содержит никакой «лишней информации» (кроме протоколирования). В реальных системах объем кода, обеспечивающего протоколирование, существенно меньше по сравнению с размером кода остальной части системы.

Выводы

Предложенный в данном разделе подход, основанный на использовании библиотеки ***STOOL***, позволил устранить некоторые недостатки ***SWITCH***-технологии. По сравнению с уже существующими методами реализации автоматных систем, предложенный подход имеет следующие отличительные особенности:

- явно выделено состояние системы;
- реализация графа переходов автомата осуществляется с помощью одного оператора `switch`;
- возможно повторное использование реализованных автоматов;
- протоколирование работы системы выполняется автоматически;
- осуществляется обработка исключительных ситуаций, возникающих при выполнении автоматов;
- возможна организация параллельных вычислений.

Библиотека ***STOOL*** доступна для скачивания по адресу http://sf.net/projects/stool.

Глава 3. Графическая нотация для проектирования автоматных объектов

Предлагаемая в настоящей работе графическая нотация для проектирования автоматных объектов позволяет обобщать, декомпозировать, структурировать и расширять логику автоматных объектов с помощью наследования.

3.1. Термины и определения

Введем следующие термины и определения.

Вычислительное состояние (**Computational State**) – совокупность значений всех атрибутов автоматного объекта.

Управляющее состояние (**State**) – абстракция, характеризующая специфическое поведение объекта.

Переход (**Jump**) – совокупность *начального состояния*, *конечного состояния*, *причины*, *условия* и *действия*, полностью описывающая смену автоматным объектом своего текущего управляющего состояния.

Группа состояний (**State group**) – формально выделенное множество управляющих состояний, имеющих *одинаковые* переходы.

Интерфейс (**Interface**) – объектно-ориентированный интерфейс автоматного объекта.

Факт вызова метода интерфейса рассматривается как событие, являющееся *причиной* перехода. Переходы осуществляются только в моменты вызова методов интерфейса.

Более формально автоматный объект A определяется тройкой $< I, S, J >$, где

- I – множество методов интерфейса автоматного объекта;
- S – множество управляющих состояний автоматного объекта;

- J – множество переходов между состояниями.

Для каждого состояния s $\in S$ определены следующие функции:

- *den(s)* – действие при входе в состояние;
- *dex(s)* – действие при выходе из состояния;
- *dact(s)* – деятельность в состоянии.

На множестве состояний S автоматного объекта определены следующие функции:

- *beg(S)* – начальное состояние;
- *reach(S)* – множество *достижимых* состояний.

Переход $j \in J$ определяется пятеркой $<from, to, ev, cond, do>$, где

- $from(j) \in S$ – начальное состояние перехода;
- $to(j) \in S$ – конечное состояние перехода;
- $ev(j) \in I$ – причина перехода: вызов метода интерфейса, в случае которого *может* произойти переход;
- $cond(j) \in \{true, false\}$ – условие, выполнение которого необходимо для осуществления перехода;
- *do(j)* – действие, выполняемое во время перехода.

Достижимым состоянием называется такое состояние $s_0 \in S$, что существует последовательность переходов $j_0, ..., j_n$, такая что

$$from(j_0) = beg(S),$$
$$\forall k \in [1, n],\ from(j_k) = to(j_{k-1}),$$
$$to(j_n) = s_0.$$

Для осуществления перехода j_0 необходимо выполнение следующих условий:

- текущим состоянием автоматного объекта является состояние $from(j_0)$;

- вызван метод интерфейса автоматного объекта $ev(j_0)$;
- выполнено условие $cond(j_0)$.

В этом случае выполняется следующая последовательность действий:

- выполняется действие $dex(from(j_0))$;
- выполняется действие $do(j_0)$;
- в качестве текущего состояния устанавливается состояние $to(j_0)$;
- выполняется действие $den(to(j_0))$.

После этого переход j_0 считается осуществленным.

3.2. Наследование автоматных объектов

Наследование – это получение свойств или характеристик базового объекта, обычно выполняемое с помощью введения некоторого отношения между базовым и производным объектом [56].

Наследование позволяет определять новые объекты на основе уже существующих объектов. При определении нового объекта указываются только те свойства, которые отличаются от свойств базовых объектов. Остальные свойства добавляются в новый объект *автоматически* [57]. Формально наследование можно записать следующим образом [58–60]:

$$R = P \oplus \Delta R,$$

где

- R – вновь определяемый объект,
- P – набор свойств, наследуемых от уже существующего объекта,
- ΔR – инкрементально добавленные новые свойства, отличающие объект R от объекта P,
- $\oplus$ – некоторая операция, позволяющая скомбинировать P и ΔR.

Одновременное наследование свойств более чем одного объекта называется *множественным наследованием*.

Наследование обычно преследует одну из следующих целей [61]:

- повторное использование некоторой части функциональности базового класса;
- возможность *замещения* базового класса.

Замещение базового класса основано на принципе замещения, предложенного Б. Лисков (Liskov Substitution Principle, ***LSP***). Принцип ***LSP*** гласит: если для каждого объекта b типа B существует объект d типа D такой, что для каждой программы P, определенной в терминах типа B, в случае замены b на d поведение программы P не изменяется, то тип D является подтипом типа B [62].

Наследование классов в большинстве объектно-ориентированных языков программирования не всегда гарантирует возможность замещения типов в соответствии с ***LSP***. В работе [63] предлагается *требование замещения*, гарантирующее соответствие принципу ***LSP***: если утверждение $\varphi(b)$ доказуемо относительно любого объекта b типа B, то утверждение $\varphi(d)$ должно быть также доказуемо для любого объекта d типа D, если тип D является подтипом типа B.

Наследование автоматных объектов основано на перегрузке состояний базового автоматного объекта. Производный объект должен *перегрузить* поведение базового объекта как минимум в одном из его состояний. В производный объект могут быть добавлены новые состояния и переходы между ними.

В настоящей работе предлагается следующий вариант наследования автоматных объектов. Автоматный объект D является потомком автоматного объекта B, $B \leq D$, в случае если:

- множество методов интерфейса I_b, реализуемого автоматным объектом B, входит во множество методов интерфейса I_d, реализуемого автоматным объектом D, $I_b \subseteq I_d$;
- множество состояний S_b автоматного объекта B входит во множество состояний S_d автоматного объекта D, $S_b \subseteq S_d$;
- начальные состояния объектов B и D совпадают, $beg(S_b) = beg(S_d)$;
- для любого перехода j_b, входящего во множество переходов J_b автомата B, существует переход j_d, входящий во множество переходов J_d автомата D, такой, что
  - $from(j_d) = from(j_b)$ – начальные состояния переходов j_d и j_b совпадают;
  - $ev(j_d) = ev(j_b)$ – методы, являющиеся причиной переходов j_d и j_b, совпадают.

Переход j_d автоматного объекта D *перегружает* переход j_b автоматного объекта B, в случае когда:

- $B \leq D$ – автоматный объект D является потомком автоматного объекта B;
- $from(j_d) = from(j_b)$ – начальные состояния переходов j_d и j_b совпадают;
- $ev(j_d) = ev(j_b)$ – методы, являющиеся причиной переходов j_d и j_b, совпадают;
- $to(j_d) \neq to(j_b)$ или $cond(j_d) \neq cond(j_b)$ или $do(j_d) \neq do(j_b)$ – конечные состояния, или условия перехода, или действия на переходе *не совпадают*.

3.3. Декомпозиция и структурирование логики автоматных объектов

Декомпозиция и структурирование логики автоматных объектов осуществляется с помощью групп состояний. Группы состояний используются также в языке ***Statecharts*** [10] и ***SWITCH***-технологии [29]. Группы состояний могут быть вложены друг в друга, образуя иерархию групп состояний.

Для введения понятия групп состояний вводится вспомогательное понятие *луча*. Луч аналогичен *переходу*, за исключением того факта, что для луча не указывается начальное состояние.

Введем множество лучей B. Луч $b \in B$ определяется четверкой $<to, ev, cond, do>$. Для каждого состояния $s \in S$ определена функция $beams(s)$, возвращающая множество лучей, соответствующих данному состоянию. Множество $beams(s)$ эквивалентно подмножеству переходов, имеющих s в качестве начального состояния:

$$beams(s) \equiv j \in J, from(j) = s$$

Группа $g \in G$ определяется тройкой $<gbeams, msub, gsub>$, где

- $gbeams(g) \subset S$ – множество лучей, соответствующих группе g;
- $msub(g) \subset S$ – множество состояний, входящих в группу g;
- $gsub(g) \subset G$ – множество групп, вложенных в группу g.

Для каждой группы $g \in G$ верны следующие утверждения:

- $\forall s \in msub(g), gbeams(g) \subseteq beams(s)$ – множество лучей состояния s, входящего в группу g, является надмножеством множества лучей, соответствующих группе g;
- $\forall g_0 \in gsub(g), gbeams(g) \subseteq gbeams(g_0)$ – множество лучей группы состояний g_0, входящей в группу g, является надмножеством множества лучей, соответствующих группе g;

- $\forall g_0 \in gsub(g)$, $\forall s \in msub(g_0)$, $s \in msub(g)$ – если состояние s вложен в группу g_0, вложенную в группу g, то он вложен также и в группу g;
- $\forall g_0, g_1 \in G$, *если* $g_0 \in gsub(g_1)$ *и* $g \in gsub(g_0)$, *то* $g \in gsub(g_1)$ – если группа g вложена в группу g_0, которая вложена в группу g_1, то группа g вложена также и в группу g_1.

3.4. Диаграммы поведения автоматных объектов

При проектировании автоматных объектов в настоящей работе предлагается использовать диаграммы поведения, являющиеся расширенной версией графов переходов, используемых в ***SWITCH***-технологии. Отличительной особенностью предлагаемых диаграмм поведения является возможность описывать декомпозицию и структурирование логики автоматных объектов с помощью наследования.

Основные элементы графической нотации, используемой для построения диаграмм поведения автоматных объектов, приведены на рис. 24.

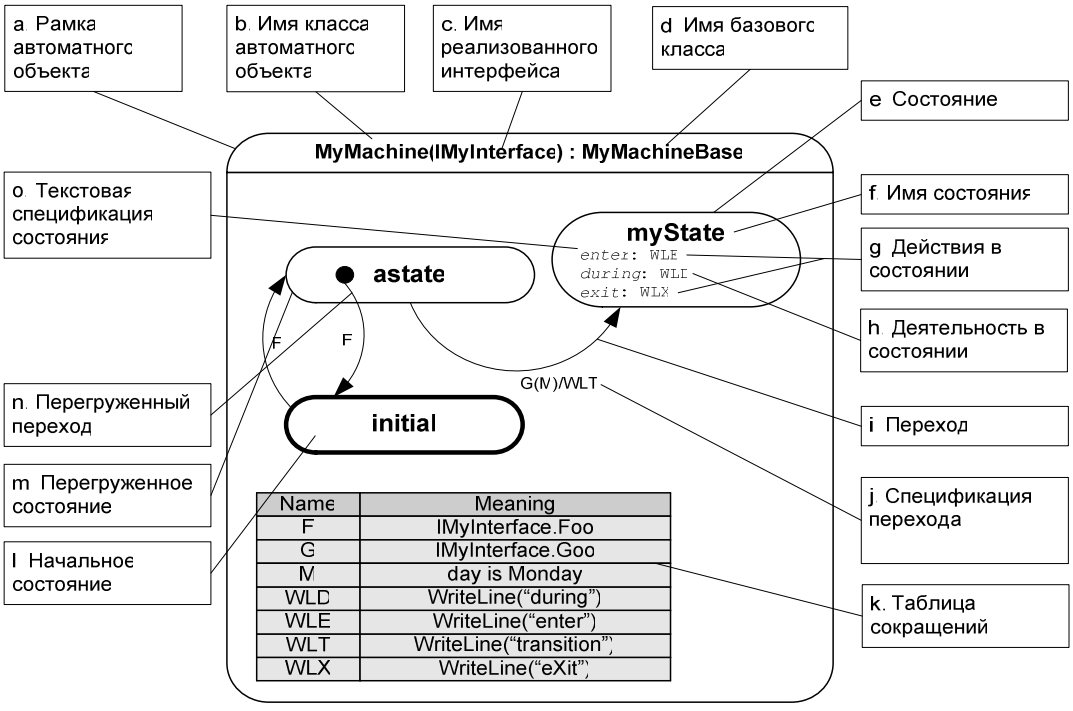


Рис. 24. Основные элементы предлагаемой графической нотации

Рассмотрим основные элементы предлагаемой нотации:

a. Автоматный класс, изображается в виде рамки с заголовком;
b. Имя класса автоматного объекта;
c. Имя реализованного интерфейса;
d. Имя базового класса;
e. Состояние, изображается в виде закругленного прямоугольника внутри рамки автоматного класса;
f. Имя состояния;
g. Действия в состоянии;
h. Деятельность в состоянии;
i. Переход, изображается в виде направленной дуги;
j. Спецификация перехода;
k. Таблица использованных сокращений;
l. Начальное состояние, выделяется жирной рамкой;
m. Перегруженное состояние, выделяется жирной точкой;
n. Перегруженный переход, берет свое начало в жирной точке;
o. Текстовая спецификация состояния.

Таблица использованных сокращений содержит два столбца. В первом столбце записывается сокращение, во втором – расшифровка. Строки таблицы отсортированы по первому столбцу в алфавитном порядке. На диаграмме поведения используются обозначения из первого столбца таблицы сокращений. Использование таблицы сокращений позволяет записывать условия переходов в кратком виде, что может быть критично для многих реальных приложений.

Переход изображается в виде направленной дуги. Рядом с дугой располагается спецификация перехода. Смена состояний происходит если:

- автомат находится в состоянии, *из* которого ведет дуга перехода;
- состояние программной системы удовлетворяет *спецификации перехода*.

Непосредственно перед сменой состояний выполняются действия, указанные в *спецификации перехода*. В процессе смены состояния автомат переходит в состояние, в который ведет дуга перехода.

Спецификация перехода j_0 задается в виде:

$$E(C) / D,$$

где

- E – причина перехода, $ev(j_0)$;
- C – условие перехода, $cond(j_0)$;
- D – действие на переходе, $do(j_0)$.

Причина и условие перехода называются *разрешительной* частью спецификации перехода. Условие и действие являются факультативными составляющими спецификации перехода. Допустимыми являются спецификации без условия, без действия и без условия и без действия одновременно, такие как `e/d`, `e(c)` и `e`.

Рассмотрим один из переходов, изображенный на рис. 24. Его спецификация записана как `G(M)/WLT`, где, в соответствии с таблицей сокращений:

- `G` эквивалентно вызову метода `IMyInterface.Goo`;
- `M` эквивалентно условию `"day is Monday"`;
- `WLT` эквивалентно вызову метода `WriteLine("transition")`.

Поэтому переход в состояние `myState` осуществим только по понедельникам (*Monday*), в случае если вызван метод `Goo()` и автомат находится в состоянии `astate`. При этом перед сменой состояния в стандартный поток выводится строка `"transition"`.

3.4.1. Графическое представление наследования автоматов

Базовый автоматный класс, если он существует, указывается в заголовке рамки автомата после двоеточия. Все состояния и переходы базового класса неявно переходят в производный автоматный класс.

Разрешается перегрузка состояний и переходов базового класса. Перегрузка перехода изображается переходом, имеющим в качестве начала символическое изображение состояния базового класса в виде жирной точки.

В случае множественного *n*-арного наследования изображается по одной пронумерованной *точке* для каждого из базовых классов, в порядке их упоминания. Точка с номером `i` соответствует базовому классу с порядковым номером `i`. Нумерация начинается с нуля. Состояние, помеченное более чем одной точкой, является объединением и расширением одноименных состояний, присутствующих в нескольких базовых классах (рис. 25).

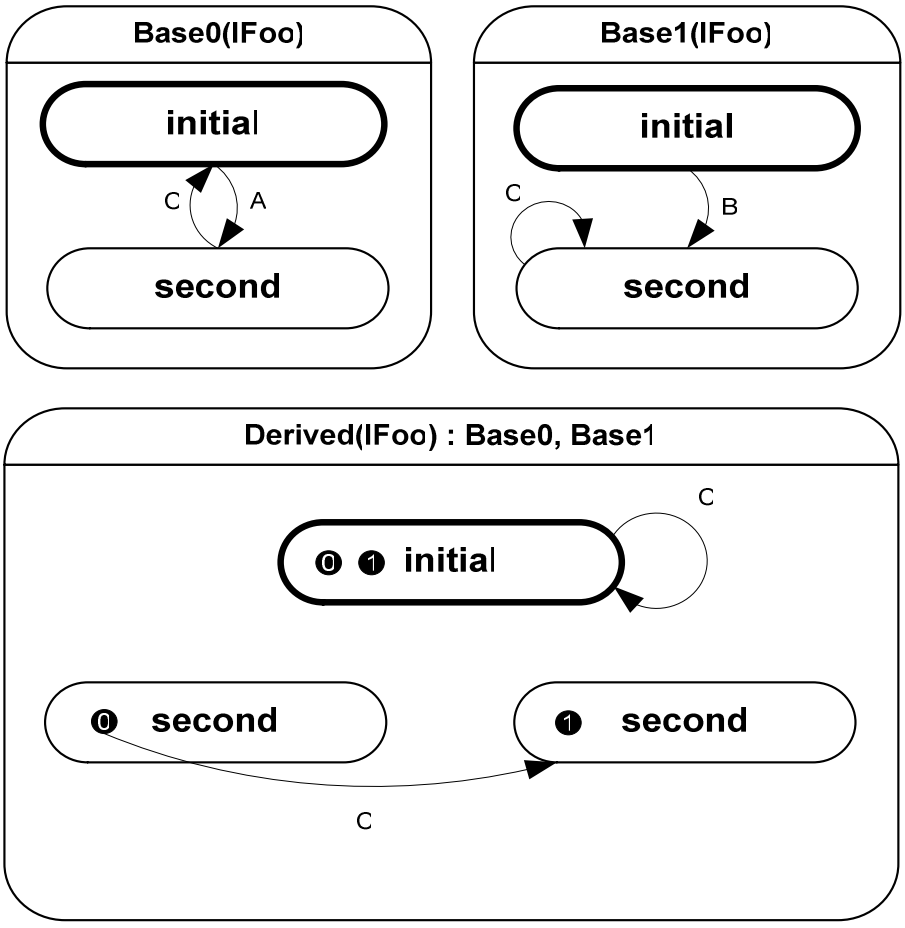


Рис. 25. Множественное наследование автоматных классов

Автоматный класс `Derived` является потомком классов `Base0` и `Base1`. Класс `Derived` *объединяет* состояния `initial` своих базовых классов и добавляет в объединенное состояние петлю `C`. Таким образом, в состоянии `initial` класса `Derived` возможны следующие переходы:

- переход в состояние `second` класса `Base0` по причине `A`;
- переход в состояние `second` класса `Base1` по причине `B`;
- петля по причине `C`.

Перегрузка перехода осуществляется по *разрешительной* части спецификации перехода. В результате перегрузки может быть изменено действие на переходе. Перегруженный переход берет свое начало в точке, изображающей соответствующий базовый класс. Так, на рис. 25, класс `Derived` перегружает переход по причине `C` из состояния `second` базового класса `Base0` и в качестве конечного состояния указывает состояние `second` базового класса `Base1`.

3.4.2. Графическое представление структурирования логики автоматных объектов

Структурирование логики автоматных объектов с помощью наследования позволяет обобщать поведение, общее для нескольких состояний, в результате чего сокращается дублирование переходов при проектировании и реализации автоматных объектов. Подобные технологии структурирования логики автоматных объектов также используются в языке ***Statecharts*** [10] и ***SWITCH***-технологии [29].

Структурирование логики автоматных объектов осуществляется за счет введения групп состояний. На диаграмме поведения состояния, входящие в одну группу, обводятся пунктиром (например, группы состояний `active` и `effective` на рис. 26).

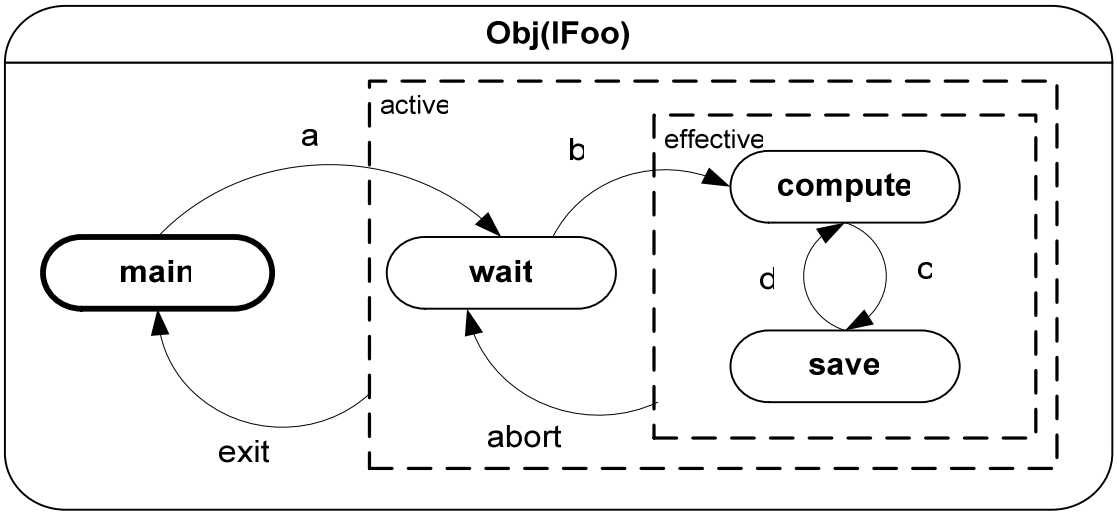


Рис. 26. Структурирование логики автоматных объектов

Группы состояний могут содержать групповые переходы. Групповой переход может быть совершен, когда автоматный объект находится в любом из состояний, входящих в данную группу.

Рассмотрим диаграмму поведения автоматного класса `DirectObj`, аналогичного классу `Obj`, изображенному на рис. 26. В классе `DirectObj` не используется структурирование, поэтому его диаграмма содержит дублирующиеся переходы (рис. 27).

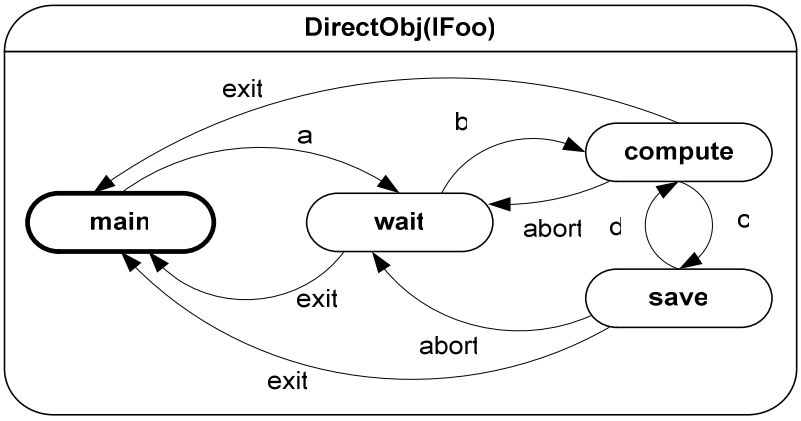


Рис. 27. Автоматный класс без использования группировки состояний

Состояния, входящие в одну группу, являются потомками одного базового класса состояния. Группы состояний могут вкладываться друг в друга, образуя иерархии. Диаграмма классов состояний [64], соответствующая автоматному объекту, изображенному на рис. 26, представлена на рис. 28.

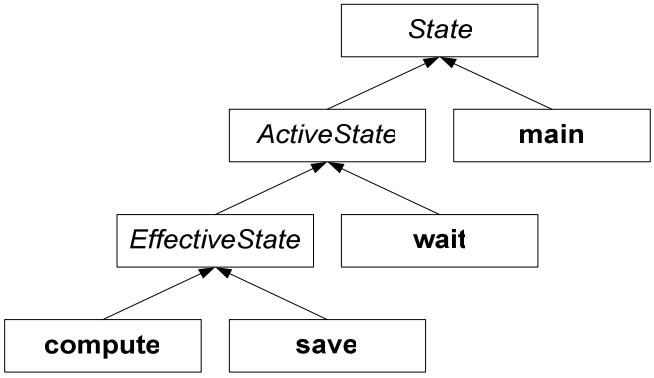


Рис. 28. Иерархия состояний

Состояние `main`, не входящее ни в какую группу, является потомком абстрактного класса `State`. Состояние `wait`, входящий в группу `active`, является потомком абстрактного класса `ActiveState`. Групповой переход из группы `active` в состояние `main` определяется в классе `ActiveState` и неявно переходит во все его производные классы. Состояния `compute` и `save`, входящие в группу `effective`, являются потомками абстрактного класса `EffectiveState`. Эти состояния также являются непрямыми потомками абстрактного класса `ActiveState`, так как группа `effective` вложена в группу `active`.

Структурирование логики может использоваться совместно с наследованием автоматных объектов. Все группы состояний, определенные в базовом классе, неявно переходят в производный класс. Группы состояний базового класса, упоминаемые в производном классе, также как и состояния базового класса, помечаются жирной точкой. Производный автоматный класс может перегружать поведение в группах своего базового класса.

В качестве примера рассмотрим класс `DerivedObj`, являющийся потомком класса `Obj` (рис. 26). Диаграмма поведения класса `DerivedObj` изображена на рис. 29.

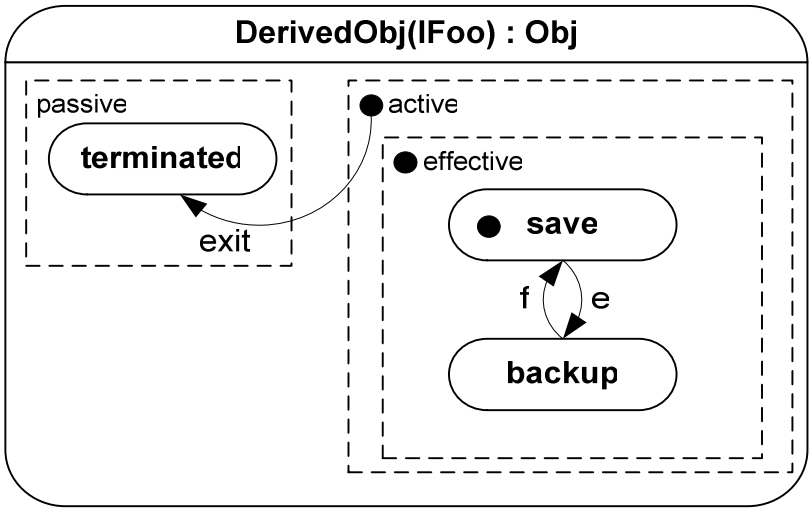


Рис. 29. Совместное использование структурирования и наследования логики автоматных объектов

В автоматном классе `DerivedObj` упоминаются группы состояний `active` и `effective` базового класса `Obj`. Класс `DerivedObj` добавляет также новую группу состояний – `passive`. Кроме этого, автоматный класс `DerivedObj`:

- перегружает переход из группы `active` в состояние `main`, устанавливая в качестве конечного состояния состояние `terminated`, определяемое в классе `DerivedObj`;
- добавляет в группу состояний `effective` состояние `backup` и связывает его переходами с состоянием `save`.

Рассмотрим диаграмму поведения автоматного класса `DirectDerivedObj`, аналогичного классу `DerivedObj` (рис. 26), но построенного без совместного использования структурирования и наследования логики автоматных объектов. Диаграмма поведения класса `DirectDerivedObj` изображена на рис. 30.

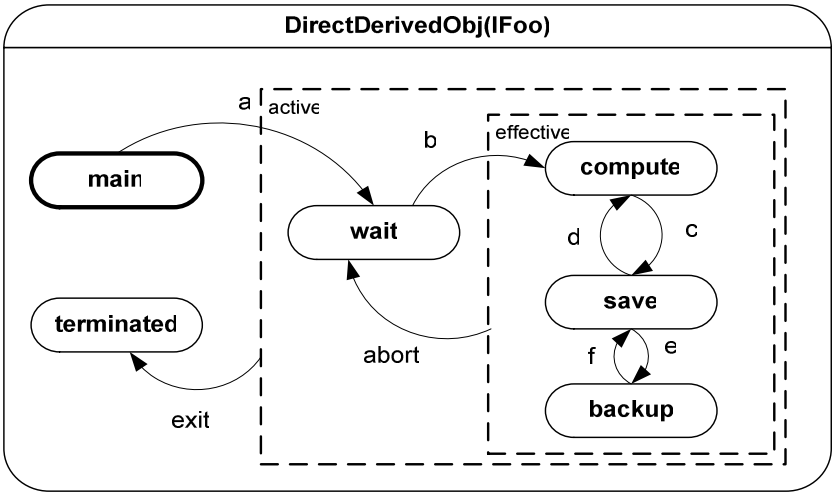


Рис. 30. Автоматный класс без совместного использования структурирования и наследования логики автоматных объектов

Структурирование логики автоматных объектов с помощью группировки состояния позволяет *значительно* сократить дублирование и повысить читаемость, как при проектировании автоматных объектов, так и при их реализации.

Выводы

Предложенная графическая нотация для проектирования автоматных объектов позволяет обобщать, декомпозировать, структурировать и расширять логику автоматных объектов с помощью наследования. Декомпозиция и структурирование логики автоматных объектов значительно сокращают дублирование, как при проектировании, так и при реализации реактивных систем.

При этом отметим, что по предложенным диаграммам поведения с наследованием, текст программы может быть построен формально и изоморфно с помощью методов, описанных в следующей главе.

Глава 4. Реализация автоматных объектов на основе виртуальных методов и виртуальных вложенных классов

В настоящей работе предлагаются методы реализации автоматных объектов на основе:

- виртуальных методов;
- виртуальных вложенных классов.

Оба метода позволяют реализовать автоматный объект в соответствии с основными принципами объектно-ориентированного программирования.

Инкапсуляция. Факт наличия автоматной логики скрывается от клиентов объекта. Экземпляры классов управляющих состояний могут хранить вычислительные состояния.

Полиморфизм. Клиент может взаимодействовать с разными классами автоматных объектов универсальным образом.

Наследование. Поведение автоматного объекта может быть расширено с помощью обычного механизма наследования. Похожее поведение в разных состояниях может быть обобщено и повторно использовано.

4.1. Демонстрационный пример: доступ к файлу

Оба предлагаемых метода реализации автоматных объектов рассматриваются на базе следующего демонстрационного примера. Пусть существует семейство классов, предоставляющих доступ к файлу:

- доступ на чтение (автоматный класс `ReadFile`);
- доступ на запись (автоматный класс `WriteFile`);
- доступ на чтение, запись и чтение/запись (автоматный класс `ReadWriteFile`).

Клиент должен быть изолирован от конкретного типа объекта, и взаимодействовать с ним через обобщенный интерфейс, описанный ниже:

```
001 class IFile
002 {
003 public:
004   virtual void Open(string fname, string mode) = 0;
005   virtual void Close() = 0;
006   virtual bool IsOpened() const = 0;
007
008   virtual void Write(int value) = 0;
009   virtual int Read() = 0;
010
011   class InvalidOperation : public exception {};
012 };
```

Рассматриваемые объекты имеют автоматную природу (с состояниями «Закрыт», «Открыт на чтение» и т.д.). Диаграммы поведения этих автоматных классов приведены на рис. 31.

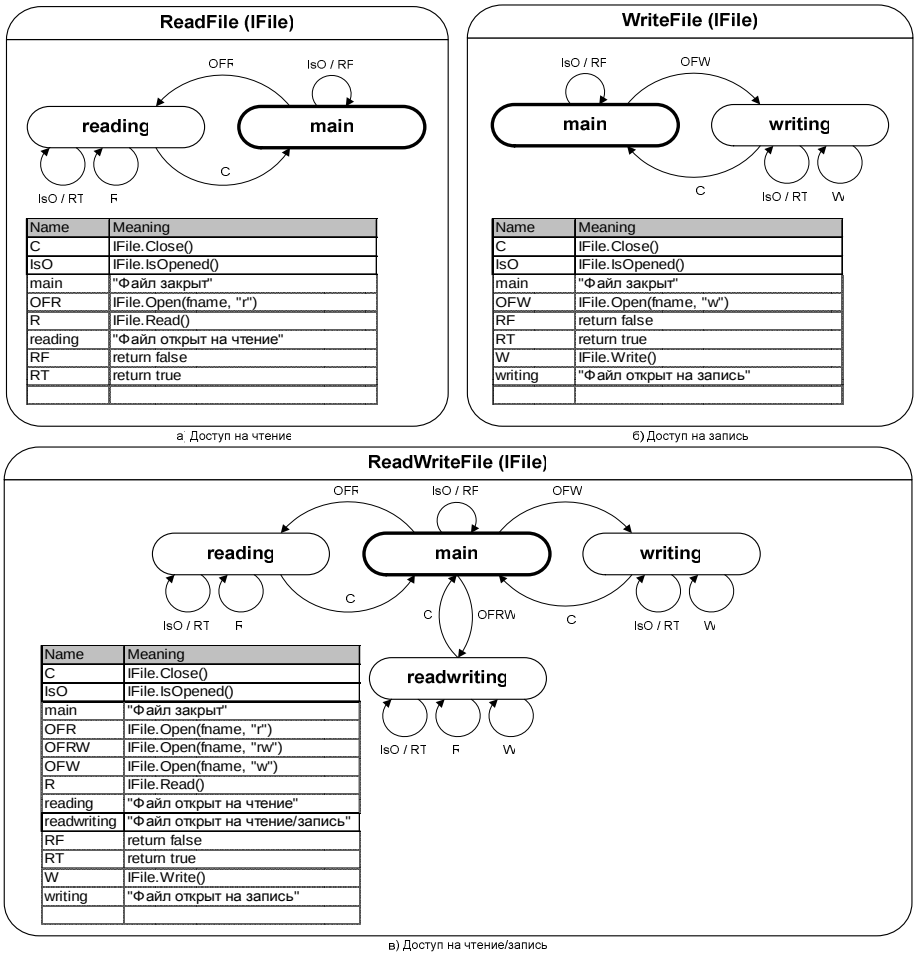


Рис. 31. Диаграммы поведения объектов доступа к файлу без использования наследования

Поведение описанных классов может быть обобщено (выделены одинаковые компоненты) и структурировано с помощью наследования. Эти объекты образуют иерархию, показанную на рис. 32. Корневым элементом предлагаемой иерархии является абстрактный класс, обобщающий некоторые аспекты доступа к файлу.

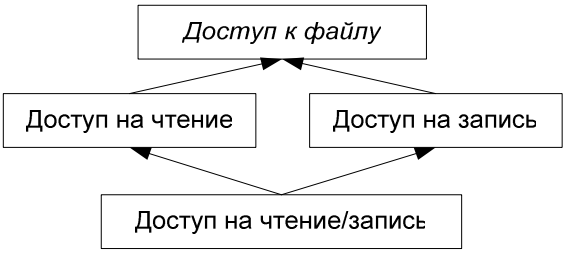


Рис. 32. Иерархия объектов доступа к файлу

Диаграмма поведения этих классов, с использованием наследования, приведена на рис. 33.

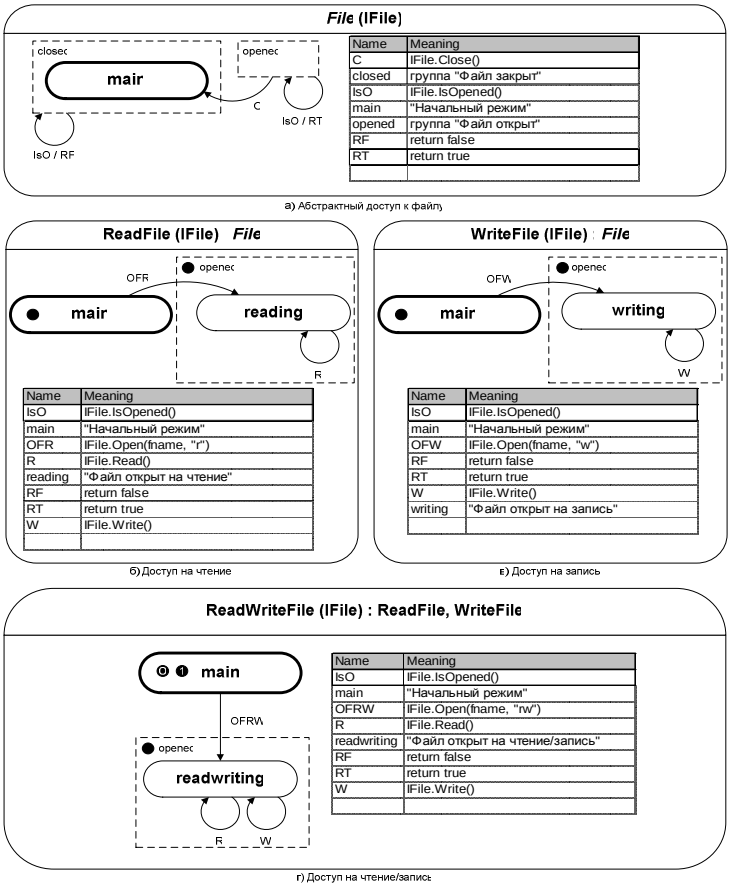


Рис. 33. Диаграммы поведения объектов доступа к файлу с использованием наследования

На диаграмме изображено четыре автоматных класса: `File`, `ReadFile`, `WriteFile` и `ReadWriteFile`. Автоматные классы `ReadFile` и `WriteFile` являются потомками класса `File`. Абстрактный автоматный класс `File` обобщает и структурирует логику доступа к файлу, вводя группы состояний `closed` и `opened`. Автоматный класс `ReadWriteFile` является потомком классов `ReadFile` и `WriteFile`. Все состояния и переходы, определенные в базовых классах, неявно переходят в класс `ReadWriteFile`.

Поведение любого из автоматных классов может быть расширено с помощью наследования. На рис. 34 приведена диаграмма поведения автоматного класса `AppendFile`, расширяющего логику автоматного класса `ReadWriteFile`, добавляя в него еще одно состояние – `appending`. Расширение происходит *инкрементально*, без изменения уже существующих автоматных классов.

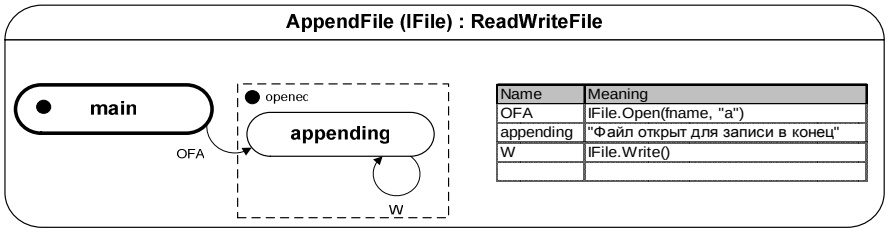
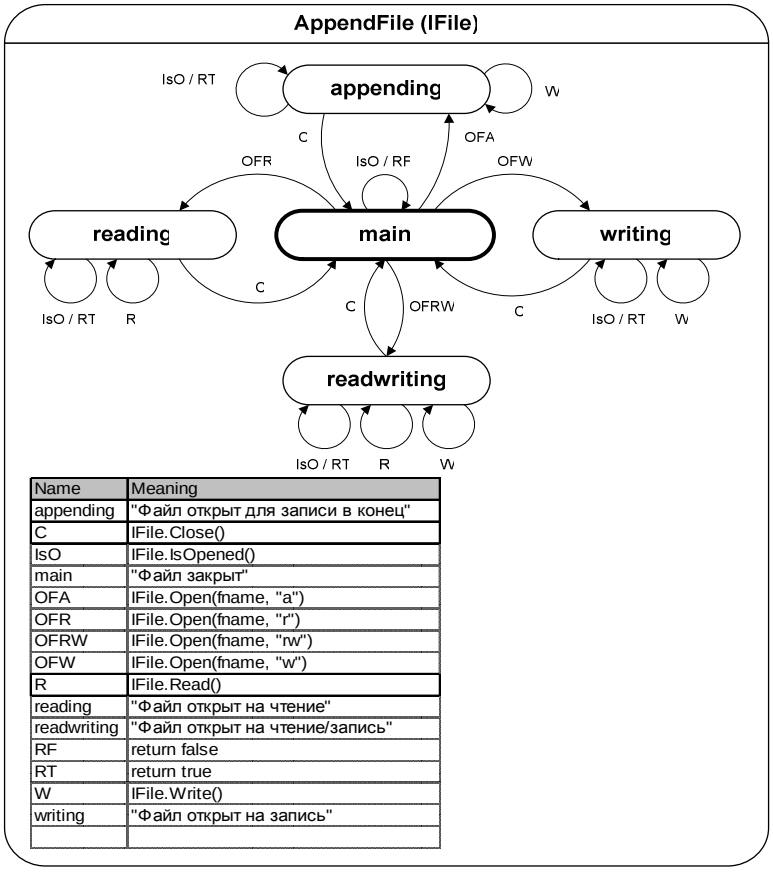


Рис. 34. Диаграмма поведения класса `AppendFile` с использованием наследования

Диаграмма поведения класса `AppendFile`, приведенная на рис. 34, эквивалентна диаграмме поведения класса `AppendFile` (рис. 35), которая построена без использования наследования. Отметим, что использование наследования *значительно* сокращает дублирование состояний и переходов.



Name	Meaning
appending	"Файл открыт для записи в конец"
C	IFile.Close()
IsO	IFile.IsOpened()
main	"Файл закрыт"
OFA	IFile.Open(fname, "a")
OFR	IFile.Open(fname, "r")
OFRW	IFile.Open(fname, "rw")
OFW	IFile.Open(fname, "w")
R	IFile.Read()
reading	"Файл открыт на чтение"
readwriting	"Файл открыт на чтение/запись"
RF	return false
RT	return true
W	IFile.Write()
writing	"Файл открыт на запись"

Рис. 35. Диаграмма поведения класса `AppendFile` без использования наследования

Рассмотрим пример использования описанных автоматных объектов доступа к файлу:

```
001 void ReadingClient(IFile& file) {
002   file.Open("data.txt", "r");
003   int value = file.Read();
004   file.Close();
005 }
006
007 int main() {
008   ReadFile rfile;
009   ReadWriteFile rwfile;
010
011   ReadingClient(rfile);
012   ReadingClient(rwfile);
013
014   return 0;
015 }
```

Функция `ReadingClient()` ожидает в качестве параметра ссылку на интерфейс `IFile`. Эта функция открывает файл *на чтение*, читает целое число и закрывает файл. Функция `main()` создает два различных объекта

доступа к файлу (`ReadFile` и `ReadWriteFile`), с которыми клиент (`ReadingClient`) работает *полиморфно* или *универсально*.

4.2. Реализация автоматных объектов на основе виртуальных методов

В методе реализации автоматных объектов на основе виртуальных методов (далее по тексту – ***VM***-метод) автоматный объект реализуется следующей трехуровневой структурой [65]:

- *традиционный* объектно-ориентированный интерфейс;
- промежуточный уровень, конвертирующий методы в сообщения;
- контекст, реализующий автоматную логику.

Реализация контекста основана на стандартном механизме виртуальных функций. Множество состояний автомата отображается на множество виртуальных методов, так что каждому состоянию соответствует один и только один метод, называемый *методом состояния*.

4.2.1. Термины и определения

Для описания ***VM***-метода вводятся следующие термины.

Посредник (Proxy) – класс, конвертирующий вызовы методов интерфейса в события и переадресовывающий эти события контексту.

Контекст (Context) – класс, реализующий поведение автоматного объекта.

Метод состояния (**State method**) – метод контекста автоматного объекта, представляющий его поведение в некотором состоянии.

Метод группы состояний (**State group method**) – метод контекста автоматного объекта, обобщающий поведение объекта, неизменное в каком-то подмножестве состояний.

4.2.2. Метод реализации автоматных объектов на основе виртуальных методов

Метод реализации автоматных объектов на основе виртуальных методов состоит из следующих этапов.

1. Автоматный объект разделяется на три части: интерфейс, посредник и контекст.
2. Посредник реализует интерфейс, преобразуя вызовы его методов в сообщения, которые затем передаются для обработки контексту.
3. Контекст реализуется следующим образом:
   3.1. Поведение автоматного объекта в каждой из его групп состояний реализуется с помощью защищенного метода группы состояний [52].
   3.2. Поведение автоматного объекта в каждом из его состояний реализуется с помощью защищенного метода состояния.
   3.3. Текущее состояние хранится в виде закрытого указателя на соответствующий метод состояния.
   3.4. Доступ к контексту предоставляется в виде открытого метода, принимающего сообщение и переадресовывающего его текущему методу состояния.
4. Методы состояния и группы состояний реализуются следующим образом:
   4.1. Методы состояния и группы состояний реализуются в виде защищенных методов контекста. Все методы имеют одинаковую сигнатуру.
   4.2. В качестве единственного параметра методы принимают сообщение, соответствующее вызову метода интерфейса.
   4.3. Метод состояния или группы состояний последовательно, для каждого перехода, возможного в данном состоянии или группе состояний, проверяет выполнение разрешительной части специ-

фикации перехода и в случае успеха совершает этот переход, возвращая значение `true`.

4.4. В случаях, когда переход не был выполнен:

4.4.1. Если состояние или группа состояний вложены в некоторую группу, то управление передается методу этой группы.

4.4.2. В противном случае возвращается значение `false`.

4.2.3. Отношения и взаимодействия

На рис. 40 показана иерархия классов, возникающая при реализации описанного демонстрационного примера на основе ***VM***-метода.

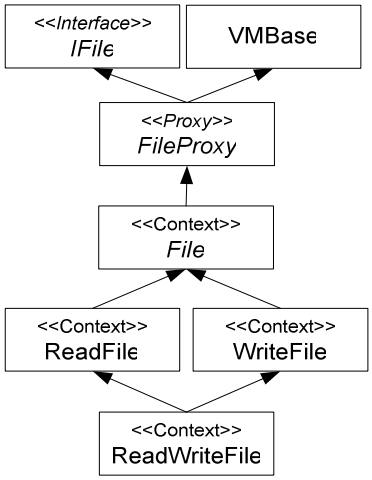


Рис. 36. Иерархия классов доступа к файлу

Класс `IFile` играет роль интерфейса, класс `FileProxy` – посредника. На рисунке изображены четыре различных контекста: `File`, `ReadFile`, `WriteFile` и `ReadWriteFile`, причем последний является потомком двух предыдущих. Класс `VMBase` является каркасом предлагаемого подхода и все посредники должны от него наследоваться.

На интерфейс автоматного объекта не накладывается никаких дополнительных ограничений. Интерфейс может содержать любое количество методов. Метод может принимать произвольное число параметров и иметь возвращаемое значение. Некоторые методы могут быть объявлены константными.

Множество состояний автоматного объекта отображается на множество методов состояния. Каждое состояние соответствует одному и только одному методу состояния.

Метод состояния реализует поведение объекта в соответствующем состоянии. Все методы состояния имеют одинаковую сигнатуру. В качестве единственного параметра методы состояния принимают объект-событие, содержащий информацию о вызванном методе интерфейса.

Посредник не содержит какой-либо логики, связанной с поведением объекта, и отвечает исключительно за связь между интерфейсом и контекстом.

Контекст является потомком посредника (возможно непрямым) и предоставляет набор методов состояния. Контексты могут наследоваться от других контекстов. Контекст-потомок может перегружать виртуальные методы состояния контекста-предка. Так обеспечивается возможность расширения логики автоматных объектов путем наследования.

Рассмотрим структуру ***VM***-метода, схематически изображенную на рис. 37.

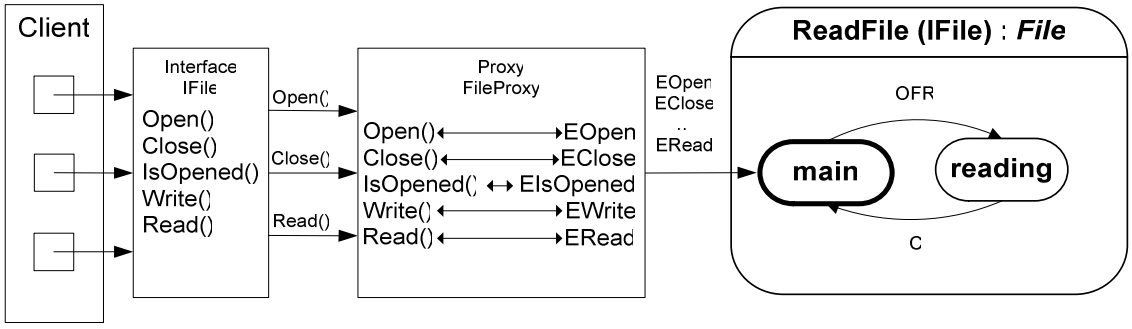


Рис. 37. Структура ***VM***-метода

Клиенты автоматного объекта взаимодействуют с ним через посредника, реализующего интерфейс (`IFile`) автоматного объекта (`ReadFile`). При вызове метода интерфейса выполняется следующая последовательность действий:

- посредник (`FileProxy`) конвертирует вызовы методов интерфейса в сообщения, содержащие информацию о вызванном методе и переданных параметрах;
- посредник передает объект-событие (`EOpen`, `EClose`, ..., `ERead`) в текущий метод состояния автоматного объекта;
- текущий метод состояния (`main`, `reading`) автоматного объекта совершает все необходимые действия на основе переданной информации, сохраняет возвращаемое значение и выходные параметры в объекте-событии и передает его посреднику;
- посредник передает клиенту возвращаемое значение и выходные параметры вызванного метода.

4.2.4. Реализация посредника

Класс-посредник `FileProxy` должен быть унаследован от интерфейса `IFile` и каркасного класса `VMBase`, как показано на рис. 38.

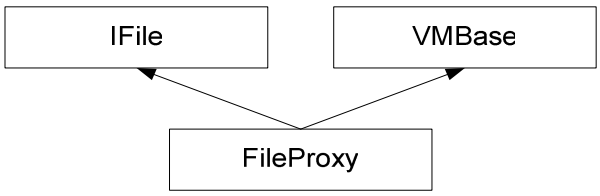


Рис. 38. Схема наследования посредника

Рассмотрим реализацию посредника. Во-первых, для каждого метода интерфейса должен быть определен специализированный тип объекта-сообщения. Эти типы должны быть унаследованы от класса `VMBase::Event` и могут быть вложены в класс `FileProxy`:

```
001 class FileProxy : public virtual IFile,
002                   protected virtual VMBase {
003 protected:
004   struct EOpen : public Event {
005     EOpen(string const& _fname, string const& _mode)
006       : fname(_fname), mode(_mode) {}
007
008     string const& GetFileName() const {
```

```
009       return fname;
010     }
011
012     string const& GetMode() const {
013       return mode;
014     }
015
016   private:
017     EOpen& operator=(EOpen const&);
018     string const fname;
019     string const mode;
020   };
021
022   struct EClose : public Event {};
023
024   struct EIsOpened : public Event {
025     EIsOpened() : result(false) {}
026
027     bool GetResult() const {
028       return result;
029     }
030
031     bool SetResult(bool _result) {
032       result = _result;
033       return true;
034     }
035   private:
036     bool result;
037   };
/*...*/
094 };
```

В строках 4 – 14 класс `FileProxy` объявляет вложенный класс объекта-сообщения `EOpen`, соответствующего вызову метода `Open()` интерфейса `IFile`. Класс `EOpen` предоставляет методы для получения параметров вызова метода `Open()`интерфейса `IFile`:

- `GetFileName()`– для получения имени открываемого файла;
- `GetMode()` – для получения требуемого режима доступа к файлу.

В строке 22 объявляется вложенный класс объекта-сообщения `EClose`, соответствующего вызову метода `Close()` интерфейса `IFile`. Класс `EClose` не предоставляет дополнительных методов, так как метод `Close()` не имеет параметров и возвращаемого значения.

В строках 24–37 объявляется вложенный класс объекта-сообщения `EIsOpened`, соответствующего вызову метода `IsOpened()` интерфейса `IFile`. Класс `EIsOpened` предоставляет методы `GetResult()` и `SetResult()` для получения и установки возвращаемого значения метода `IsOpened()`.

Во-вторых, в посреднике должны быть реализованы все методы интерфейса. Их реализации имеют одинаковую структуру – на стеке создается соответствующий объект-событие и передается методу `VMBase::Execute()`:

```
001 class FileProxy : public virtual IFile,
002                   protected virtual VMBase {
/*...*/
067 public:
068   virtual void Open(string fname, string mode) {
069     EOpen event(fname, mode);
070     Execute(event);
071   }
072
073   virtual void Close() {
074     EClose event;
075     Execute(event);
076   }
077
078   virtual bool IsOpened() const {
079     EIsOpened event;
080     Execute(event);
081     return event.GetResult();
082   }
/*...*/
094 };
```

Класс `FileProxy` перегружает в строках 68–71 метод `Open()` интерфейса `IFile`. На стеке создается экземпляр `event` сообщения `EOpen`, в конструктор которого передаются параметры метода `Open()`. Затем сообщение `event` передается методу `Execute()` базового класса `VMBase`. Метод `Execute()` передает сообщение текущему методу состояния.

В строках 73–76 аналогичным образом перегружается метод `Close()`. В строках 78–82 перегружается метод `IsOpened()`. В качестве возвращаемого значения метода `IsOpened()` используется значение, возвращаемое методом `GetResult()` объекта-сообщения `event`.

4.2.5. Реализация контекста и структурирование логики

Контекст содержит набор методов состояний и групп состояний. Методы групп состояний являются средством структурирования логики автоматных объектов. Реализация методов состояний и групп состояний осуществляется по шаблону, приведенному ниже.

```
001 virtual bool <состояние>(Event& _event) {
002   if (<причина> [&& <условие>]) {
003     <действие на переходе>;
004     return SetState(*this, &<контекст>::<новое состояние>);
005   } else if (<причина> [&& <условие>]) {
006     return EventCast< тип сообщения >(_event)->SetResult(<значение>);
007   } else if (<причина> [&& <условие>]) {
008     <действие на переходе>;
009     return true;
010   } else
011     return <группа>(_event);
012 }
013
014 virtual bool <группа>(Event& _event) {
015   if (<причина> [&& <условие>]) {
016     <действие на переходе>;
017     return SetState(*this, &<контекст>::<новое состояние>);
018   } else if (<причина> [&& <условие>]) {
019     return EventCast< тип сообщения >(_event)->SetResult(<значение>);
020   } else if (<причина> [&& <условие>]) {
021     <действие на переходе>;
022     return true;
023   } else
024     return false;
025 }
```

Прокомментируем некоторые обозначения, использованные в приведенном шаблоне:

- `<причина>` – причина перехода, задается в виде `EventCast<тип сообщения>(_event)`, что истинно в том случае, когда полученное сообщение `_event` соответствует вызову метода, являющегося причиной данного перехода;
- `<условие>` – необязательное условие, истинность которого необходима для осуществления данного перехода.

Метод состояния состоит из набора операторов ветвления `if-else`. Каждый условный оператор задает один из возможных в данном состоянии переходов. Оператор `if` в качестве *выражения* получает разрешительную

часть спецификации перехода, состоящую из причины и необязательного условия. В *теле* оператора `if` могут выполняться следующие операции:

- действие на переходе;
- смена состояния, выполняется с помощью метода `SetState()` каркасного класса `VMBase`;
- установка возвращаемых значений, выполняется с помощью метода объекта-сообщения `SetResult()`.

Если был осуществлен один из переходов, то метод состояния возвращает значение `true`. В противном случае управление передается в группу состояний, если она есть. Если группа отсутствует, то возвращается значение `false`. Метод группы состояний реализуется аналогичным образом. Если в процессе обработки события не было совершено ни одного перехода, то каркасный класс `VMBase` автоматически обеспечивает генерацию исключения `UnexpectedOperation`.

Рассмотрим реализацию контекста `File`, граф переходов которого изображен на рис. 33, а. Контекст `File` определяет группы состояний `closed` и `opened`. Исходный код контекста `File` приведен ниже:

```
001 class File : public virtual FileProxy {
002 protected:
003   virtual bool main(Event& _event) = 0;
004 
005   virtual bool closed(Event& _event) {
006     if (EIsOpened* e = EventCast< EIsOpened >(_event)) {
007       return e->SetResult(false);
008     } else
009       return false;
010   }
011 
012   virtual bool opened(Event& _event) {
013     if (EIsOpened* e = EventCast< EIsOpened >(_event)) {
014       return e->SetResult(true);
015     } else if (EClose* e = EventCast< EClose >(_event)) {
016       closeFile();
017       return SetState(*this, &File::main);
018     } else
019       return false;
020   }
021 
022   virtual void closeFile() = 0;
023 };
```

Рассмотрим реализацию группы состояний `closed`. Оператор `if`, задающий петлю в группе `closed`, проверяет поступление сообщения `EIsOpened`, соответствующего вызову метода `IsOpened()` интерфейса `IFile` (строка 6). В случае выполнения этого условия в качестве возвращаемого значения метода `IsOpened()` устанавливается значение `false`. В противном случае метод `closed` возвращает значение `false` (строка 9), что приводит к генерации каркасом `VMBase` исключения `UnexpectedOperation`.

В первом операторе `if` группы состояний `opened` также обрабатывается вызов метода `IsOpened()` интерфейса `IFile` (строки 13, 14). Второй оператор `if` проверяет поступление сообщения `EClose`, соответствующего вызову метода `Close()` интерфейса `IFile` (строка 15). В случае выполнения этого условия, вызывается абстрактный метод `closeFile()` (строка 16) и в качестве следующего состояния устанавливается состояние `main` (строка 17). В противном случае метод `opened` возвращает значение `false` (строка 19).

Рассмотрим реализацию контекста `ReadFile`, граф переходов которого изображен на рис. 33, б.

```
001 class ReadFile : public virtual File {
002   ifstream* ifile;
003 protected:
004   virtual bool main(Event& _event) {
005     if (EventCast< EOpen >(_event) &&
006         EventCast< EOpen >(_event)->GetMode() == "r") {
007       EOpen* e = EventCast< EOpen >(_event);
008       ifile = new ifstream(e->GetFileName().c_str());
009       return SetState(*this, &ReadFile::reading);
010     } else
011       return closed(_event);
012   }
013
014   virtual bool reading(Event& _event) {
015     if (ERead* e = EventCast< ERead >(_event)) {
016       int res = 0;
017       (*ifile) >> res;
018       return e->SetResult(res);
019     } else
020       return opened(_event);
021   }
022
```

```
023   virtual void closeFile() {
024     delete ifile;
025     ifile = 0;
026   }
027 };
```

Контекст `ReadFile` является потомком контекста `File` и наследует у него группы состояний `closed` и `opened`. Контекст `ReadFile` перегружает метод состояния `main`, входящий в группу `closed` и добавляет новое состояние `reading`, входящий в группу `opened`.

Рассмотрим реализацию метода состояния `main`. Первый оператор `if`, задающий переход из состояния `main` в состояние `reading`, проверяет поступление сообщения `EOpen`, соответствующего вызову метода `Open()` интерфейса `IFile`, с параметром `mode` равным `"r"` (строки 5, 6). В случае выполнения этого условия файл открывается на чтение (строка 8) и в качестве следующего состояния устанавливается состояние `reading` (строка 9). В противном случае управление передается в группу `closed` (строка 11).

Метод состояния `reading` контекста `ReadFile` обрабатывает вызов метода `Read()` интерфейса `IFile` (строки 15–18) и передает управление в группу `opened`.

Класс `ReadFile` перегружает также абстрактный метод `File::closeFile()`, реализуя в нем закрытие потока ввода `ifile` (строки 23–26).

4.2.6. Расширение логики с помощью наследования

Одно из основных достоинств предлагаемого метода – возможность расширения логики с помощью наследования. Поведение объекта в каком-то конкретном состоянии может быть изменено. В автомат могут быть добавлены новые состояния.

Например, рассмотрим класс `ReadWriteFile` (рис. 33, г), являющийся потомком классов `ReadFile` и `WriteFile`. Кроме состояний `reading` и `writing`, класс `ReadWriteFile` имеет дополнительное сме-

шанное состояние readwriting. Исходный код класса ReadWriteFile приведен ниже:

```
001 class ReadWriteFile : public virtual ReadFile,
002                       public virtual WriteFile {
003   fstream* file;
004 protected:
005   virtual bool main(Event& _event) {
006     if (ReadFile::main(_event)) {
007       return true;
008     } else if (WriteFile::main(_event)) {
009       return true;
010     } else if (EventCast< EOpen >(_event) &&
011                EventCast< EOpen >(_event)->GetMode() == "rw") {
012       EOpen* e = EventCast< EOpen >(_event);
013       file = new fstream(e->GetFileName().c_str());
014       return SetState(*this, &ReadWriteFile::readwriting);
015     } else
016       return closed(_event);
017   }
018
019   virtual bool readwriting(Event& _event) {
020     if (EWrite* e = EventCast< EWrite >(_event)) {
021       (*file) << e->GetValue() << endl;
022       return true;
023     } else if (ERead* e = EventCast< ERead >(_event)) {
024       int res = 0;
025       (*file) >> res;
026       return e->SetResult(res);
027     } else
028       return opened(_event);
029   }
030
031   virtual void closeFile() {
032     delete file;
033     file = 0;
034   }
035 };
```

Рассмотрим реализацию метода состояния main контекста ReadWriteFile. На диаграмме поведения (рис. 33, г) состояние main отмечено двумя жирными точками с номерами 0 и 1. Это означает, что состояние main контекста ReadWriteFile является объединением и расширением состояний main своих базовых классов ReadFile и WriteFile.

Поведение в состояние main контекста ReadWriteFile можно описать следующим образом:

- делается попытка выполнить переходы из состояния main контекста ReadFile;

- в случае неуспеха делается попытка выполнить переходы из состояния `main` контекста `WriteFile`;
- в случае неуспеха делается попытка выполнить переходы, добавленные в состояние `main` в контексте `ReadWriteFile`;
- в случае неуспеха управление передается в группу `closed`.

Метод состояния `ReadWriteFile::main` вызывает метод состояния `main` базового класса `ReadFile` (строка 6). Если в методе `ReadFile::main` был осуществлен переход и возвращено значение `true`, то метод `ReadWriteFile::main` также возвращает `true`. Аналогичным образом вызывается метод состояния `main` базового класса `WriteFile` (строки 8, 9). Если в методах main обоих базовых классов не был осуществлен переход, то проверяется факт вызова метода `Open()` интерфейса `IFile` с параметром mode равным `"rw"` и осуществляется переход в состояние `readwriting`.

По причине использования множественного наследования, вводятся дополнительные, необязательные в случае одиночного наследования, требования:

- интерфейс объекта и служебный класс `VMBase` во всех классах должны наследоваться виртуально;
- в классе автомата, являющегося потомком нескольких автоматов, должен быть переопределен метод `main`, что необходимо для устранения неоднозначности вызова метода `main` результирующего автомата.

4.2.7. Каркас `VMBase`

При использовании ***VM***-метода, все автоматные объекты должны наследоваться от служебного класса `VMBase`. Этот класс предоставляет:

- базовый для объектов-событий класс `VMBase::Event`;
- константную и неконстантную версии метода `Execute()`, который используется на промежуточном уровне для обращения к контексту;
- метод `SetState()`, применяемый для изменения состояния объекта;
- шаблонный метод `EventCast()`, используемый для выяснения конкретного типа объекта-события.

4.2.7.1. Реализация константных методов интерфейса

Константным методом объекта *a* называется метод, не модифицирующий объект *a*. В языке программирования ***C++*** константные методы помечаются ключевым словом `const` [52]. Примером константного метода является метод `IsOpened()` интерфейса `IFile` (разд. 4.2.2).

К сожалению, механизм автоматического контроля константности объекта во время выполнения *метода состояния* не доступен. Это объясняется тем, что в предлагаемом подходе в процессе конструирования автоматного объекта запоминается неконстантная ссылка на текущий экземпляр. В процессе вызова константного метода интерфейса используется неконстантная ссылка, что не позволяет применять встроенный механизм контроля константности объекта.

Для устранения этого недостатка класс `VMBase` автоматически контролирует константность состояния объекта во время выполнения. Если вызван константный метод интерфейса, и он пытается вызвать метод `SetState()`, то классом `VMBase` гарантируется генерация исключения `VMBase::ReadOnlyViolation`.

Константность вспомогательных данных должна контролироваться вручную. Если метод состояния пытается изменить данные и метод

`VMBase::IsImmutable()` возвращает `true`, то программистом должно быть сгенерированно указанное выше исключение.

В качестве примера рассмотрим исходный код класса `ReadFileViolator` (рис. 39), иллюстрирующего оба случая нарушения константности объекта:

- попытка изменения состояния объекта во время обработки константного метода интерфейса (переход из группы `closed` в состояние `reading` на рис. 39);
- попытка изменения вспомогательных данных во время обработки константного метода интерфейса (петля в группе `opened` на рис. 34).

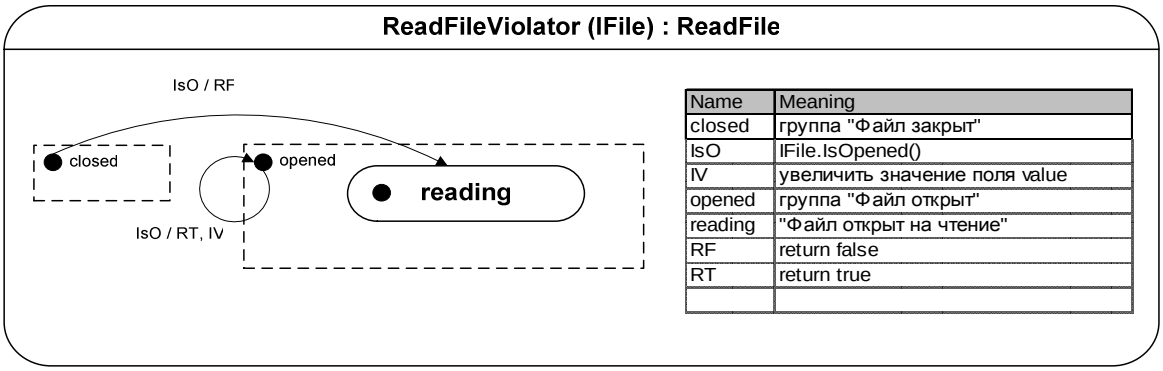


Рис. 39. Диаграмма поведения автоматного класса `ReadFileViolator`

Исходный код класса `ReadFileViolator` приведен ниже.

```
001 class ReadFileViolator : public virtual ReadFile {
002   int value;
003
004   void incValue()
005   {
006     if ( IsImmutable() )
007       throw ReadOnlyViolation();
008     value++;
009   }
010
011 protected:
012
013   virtual bool closed(Event& _event) {
014     if (EIsOpened* e = EventCast< EIsOpened >(_event)) {
015       e->SetResult(false);
016       return SetState(*this, &ReadFileViolator::reading);
017     } else
```

```
018       return ReadFile::closed(_event);
019   }
020
021   virtual bool opened(Event& _event) {
022     if (EIsOpened* e = EventCast< EIsOpened >(_event)) {
023       incValue();
024       return e->SetResult(true);
025     } else
026       return ReadFile::opened(_event);
027   }
028 };
```

Для демонстрации попытки изменения состояния объекта во время обработки константного метода интерфейса достаточно вызвать метод `IsOpened()` в состоянии `main`. В строке 16 приведенного исходного кода осуществляется попытка перевести объект в состояние `reading`, в то время как метод `IsOpened()` является константным и состояние объекта меняться не должен.

Для демонстрации попытки изменения вспомогательных данных во время обработки константного метода интерфейса достаточно вызвать метод `IsOpened()` в состоянии `reading`. В строке 23 приведенного исходного кода осуществляется попытка увеличить значение поля `value`, в то время как метод `IsOpened()` является константным и вспомогательные данные объекта меняться не должны. Доступ к полю `value` осуществляется через метод `incValue()`, который перед каждым изменением поля `value` проверяет значение, возвращаемое методом `IsImmutable()`, и, в случае возвращения значения `true`, генерирует исключение `ReadOnlyViolation`.

4.2.8. Недостатки *VM*-метода

Недостатками подхода являются:

- высокая трудоемкость хранения *вычислительного состояния*, состав которого может зависеть от текущего *управляющего состояния* автоматного объекта;
- высокая трудоемкость обеспечения константности данных (разд. 4.2.7.1);

- высокая трудоемкость преобразования методов в сообщения и обратно (разд. 4.2.4).

Основным недостатком является высокая трудоемкость хранения *вычислительного состояния*. Это обусловлено тем фактом, что состояния объекта представлены методами, а методы не могут *хранить* информацию. Поэтому вычислительное состояние, необходимое объекту во всех его управляющих состояния, хранится в контексте, что

- нарушает инкапсуляцию управляющего состояния;
- засоряет контекст (в случае большого количества управляющих состояний).

Например, поток ввода `ifile` в контексте `ReadFile` (разд. 4.2.5) хранится непосредственно в контексте, тогда как он нужен только в состоянии `reading`, так как в состоянии `main` файл закрыт и никакого потока ввода не существует. Однако при предлагаемом подходе нет возможности хранить поток ввода `ifile` только в состоянии `reading`.

4.3. Реализация автоматных объектов на основе виртуальных вложенных классов

Предлагаемый подход, основанный на виртуальных вложенных классах (далее по тексту – ***VIC***-метод) [66, 67], сохраняет возможность расширения логики с помощью наследования и избавляет от недостатков указанных в разд. 4.2.8.

Если механизм виртуальных методов позволяет перегружать *методы* базового класса в его потомках, то механизм виртуальных вложенных классов позволяет перегружать вложенные *классы* базового класса в его потомках. Например, если класс `Vector` содержит вложенный виртуальный класс `Item`, то класс `IntVector` может перегрузить его вложенным классом `IntItem`. Подобная возможность предоставляется, например, в языке про-

граммирования ***Python*** [68]. Многие статически типизируемые объектно-ориентированные языки программирования не поддерживают виртуальные вложенные классы, хотя в большинстве из них существует возможность эмулировать подобную функциональность. В данном разделе будет использоваться язык программирования ***C++***, позволяющий эмулировать виртуальные вложенные классы с помощью виртуальных методов, являющихся *виртуальными* конструкторами вложенных классов.

4.3.1. Термины и определения

Для описания ***VIC***-метода вводятся следующие термины.

Посредник (**Proxy**) – класс, реализующий интерфейс и переадресовывающий все методы текущему экземпляру класса состояния.

Контекст (**Context**) – класс, предоставляющий множество фабрик состояний.

Фабрика состояния (**State factory**) – метод контекста, используемый для создания экземпляра класса состояния.

Класс состояния (**State class**) – класс, реализующий интерфейс и представляющий поведение объекта в конкретном состоянии.

Класс группы состояний (**State group class**) – класс, реализующий интерфейс и обобщающий поведение объекта, неизменное в каком-то подмножестве состояний.

Групповой метод контекста (**Group method**) – виртуальный метод контекста, реализующий поведение в некоторой группе состояний.

4.3.2. Метод реализации автоматных объектов на основе виртуальных вложенных классов

Метод реализации автоматных объектов на основе виртуальных вложенных классов состоит из следующих этапов.

1. Автоматный объект разделяется на три части: *интерфейс*, *посредник* и *контекст*.

2. Посредник реализует интерфейс, переадресовывая вызовы его методов контексту.
3. Контекст реализуется следующим образом:
   3.1. Поведение автоматного объекта в каждой из его групп состояний реализуется с помощью вложенного защищенного класса группы состояний, реализующего интерфейс автоматного объекта.
   3.2. Поведение автоматного объекта в каждом из его состояний реализуется с помощью вложенного защищенного класса состояния, реализующего интерфейс автоматного объекта.
   3.3. Текущее состояние хранится в виде закрытого указателя на экземпляр соответствующего класса состояния.
   3.4. Переключение состояний осуществляется с помощью фабрик состояний.
   3.5. Фабрика состояния реализуется в виде защищенного метода контекста, возвращающего экземпляр класса состояния.
   3.6. Доступ к контексту предоставляется в виде открытого метода, возвращающего текущий экземпляр класса состояния.
4. Классы состояния и группы состояний реализуются в виде вложенных защищенных классов следующим образом:
   4.1. Классы состояния и группы состояния реализуют интерфейс автоматного объекта.
   4.2. Каждый метод класса состояния или группы состояний реализует переходы, возможные в данном состоянии или группе состояний и имеющие в качестве своей причины вызов соответствующего метода интерфейса.
   4.3. Метод класса группы состояний может быть вынесен в контекст, для обеспечения возможности дальнейшего расширения поведения.

4.4. Каждый метод класса состояния или группы состояний последовательно, для каждого из возможных состояний, проверяет выполнение условия перехода и в случае успеха совершает этот переход, завершая свое выполнение.

4.5. В случаях, когда переход не был выполнен:

4.5.1. Если тип возвращаемого значения позволяет вернуть значение δ (например: `false`, `null`), идентифицирующее *неуспех*, то возвращается значение δ.

4.5.2. Если тип возвращаемого значения (например, `void`) метода m не позволяет вернуть значение δ, идентифицирующее *неуспех*, то в случае необходимости метод m заменяется методом m', тип возвращаемого значения которого позволяет вернуть δ (без изменения интерфейса автоматного объекта).

4.3.3. Отношения и взаимодействия

В качестве демонстрационного примера для описания ***VIC***-метода будет повторно использован пример доступа к файлу (разд. 4.2.2). На рис. 40 изображена иерархия классов, возникающая при реализации демонстрационного примера на основе ***VIC***-метода.

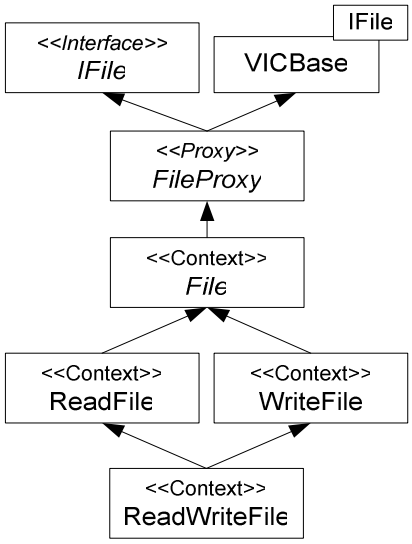


Рис. 40. Иерархия классов доступа к файлу

Класс `IFile` играет роль интерфейса, класс `FileProxy` – посредника. На рис. 40 изображены также четыре различных контекста: `File`, `ReadFile`, `WriteFile` и `ReadWriteFile`. Контексты `ReadFile` и `WriteFile` являются потомками контекста `File`. Контекст `ReadWriteFile` является потомком контекстов `ReadFile` и `WriteFile`.

Промежуточный класс `File` используется для хранения классов групп состояний `ClosedGroup` и `OpenedGroup`.

Класс `VICBase` является каркасом предлагаемого подхода и все посредники должны от него наследоваться.

Пространство состояний автоматного объекта отображается на пространство фабрик состояния. Каждое состояние соответствует одной и только одной фабрике состояния.

Классы группы состояний служат средством структурирования управляющих состояний (разд. 3.4.2). Классы группы состояния позволяют обобщать, вынося в базовый класс, следующие аспекты:

- переходы;
- зависящие от управляющего состояния данные;
- зависящие от управляющего состояния вспомогательные методы.

Фабрика состояния является *виртуальным конструктором* соответствующего класса состояния и может иметь произвольный набор параметров. Виртуальная фабрика состояния может быть перегружена в потомках, что позволяет изменять поведение объекта в соответствующем состоянии. Предопределенная абстрактная фабрика состояния `main()` используется для создания начального состояния. Каждый конкретный контекст должен перегрузить эту фабрику.

Посредник хранит экземпляр текущего класса состояния и переадресует к нему все вызовы методов интерфейса. Экземпляр класса состояния отве-

чает за изменение состояния объекта, когда это необходимо. Другими словами, логика поведения автоматного объекта распределена между классами его управляющих состояний.

Для переключения между состояниями используются указатели на методы. Текущий экземпляр класса состояния передает указатель на фабрику состояния и все необходимые этой фабрике параметры методу `NextState()` каркаса `VICBase`. Таким образом, в соответствии с механизмом вызова виртуальных методов по указателю [52], если данная фабрика состояния была перегружена, то будет вызвана версия, наиболее подходящая в текущем контексте.

Поясним изложенное на примере. Пусть контекст `ACtx` имеет две фабрики состояний: `foo()` и `goo()`. Причем фабрика `goo()` – виртуальная. Контекст `ACtx` содержит два класса состояния – `FooState` и `GooState`. Класс `FooState` переключается в состояние `goo()`, используя указатель на метод `ACtx::goo()`. Контекст `BCtx` является потомком контекста `ACtx` и перегружает фабрику состояния `goo()`. Фабрика `BCtx::goo()` создает экземпляр состояния `ExtendedGooState`. Таким образом, если будет создан экземпляр класса `BCtx` и из класса `ACtx::FooState` будет запрошено переключение в состояние `goo`, то будет вызван метод `BCtx::goo()` и создан экземпляр состояния `ExtendedGooState`.

Рассмотрим структуру ***VIC***-метода, схематически изображенную на рис. 41.

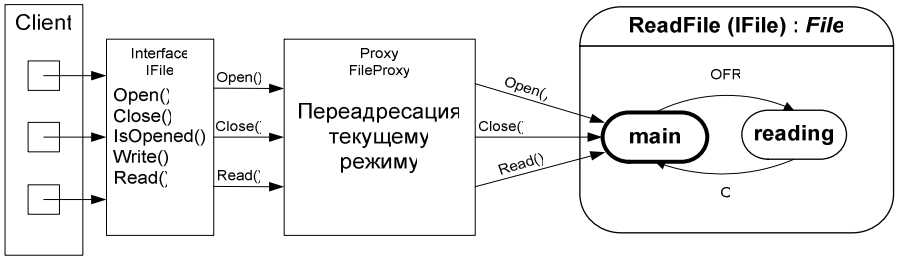


Рис. 41. Структура ***VIC***-метода

Клиенты автоматного объекта взаимодействуют с ним через посредника, реализующего интерфейс (`IFile`) автоматного объекта (`ReadFile`). При вызове метода интерфейса выполняется следующая последовательность действий:

- посредник (`FileProxy`) переадресует вызовы методов интерфейса текущему экземпляру класса состояния, предоставляющему реализацию методов интерфейса, соответствующую поведению в данном состоянии;
- текущий экземпляр класса состояния выполняет все необходимые действия (включая возможное изменения текущего состояния) и передает возвращаемое значение и выходные параметры непосредственно клиенту.

4.3.4. Реализация посредника

Реализация класса `FileProxy` (рис. 40) состоит только в переадресации всех методов интерфейса текущему экземпляру класса состояния. Класс `FileProxy` является потомком классов `IFile` и `VICBase<IFile>`. Исходный код класса `FileProxy` приведен ниже.

```
001 class FileProxy
002   : public IFile
003   , public VICBase<IFile>
004 {
005 public:
006   virtual bool Open(string fname, string mode) {
007     return CurrentState()->Open(fname, mode);
008   }
009
010   virtual void Close() {
011     CurrentState()->Close();
012   }
013
014   /*...и так далее...*/
015 };
```

Класс `FileProxy` перегружает все методы интерфейса `IFile`. Реализация каждого метода переадресует вызов текущему состоянию, возвращаемому методом `CurrentState()` (например, строка 7).

4.3.5. Реализация контекста и структурирование логики

Контекст содержит набор фабрик состояний, каждая из которых возвращает объект, реализующий интерфейс автоматного объекта и описывающий его поведение в данном состоянии. В качестве примера рассмотрим реализацию контекста `ReadFile`.

```
001 class ReadFile : public virtual File
002 {
003 protected:
/*...*/
020   virtual IFile* main() {
/*...*/
022   }
023
024   virtual IFile* reading(string fname) {
/*...*/
041   }
042 };
```

Контекст `ReadFile` объявляет фабрики состояний `main` и `reading`, соответствующие одноименным состояниям автоматного класса `ReadFile` (рис. 33, б). Фабрики состояний возвращают указатель на объект, реализующий интерфейс автоматного объекта – `IFile`.

Фабрики состояний возвращают экземпляры классов состояний. Классы состояний могут быть реализованы как:

- вложенный класс контекста;
- локальный класс фабрики состояния.

Достоинством вложенного класса состояния является возможность его использования в потомках контекста. Достоинством локального класса является более высокий уровень инкапсуляции.

Продемонстрируем объявление классов состояний на примере контекста `ReadFile`. На приведенном ниже листинге класс `MainState` объявлен как вложенный класса контекста `ReadFile`, а класс `ReadingState` – как локальный класс фабрики состояния `reading()`.

```
001 class ReadFile : public virtual File
002 {
003 protected:
```

```
004
005   struct MainState : public virtual ClosedGroup
006   {
/*...*/
012     virtual void Open(string fname, string mode) {
/*...*/
019     }
020   };
021
022   virtual IFile* main() {
023     return new MainState( *this );
024   }
025
026   virtual IFile* reading(string fname)
027   {
028     struct ReadingState : public OpenedGroup {
/*...*/
033       virtual int Read() {
/*...*/
037       }
/*...*/
041     };
042
043     return new ReadingState(fname, *this);
044   }
045 };
```

Класс состояния `MainState` объявляется как вложенный класс состояния. Класс `MainState` является потомком класса `ClosedGroup`, так как состояние `main` входит в группу `closed`. Класс `MainState` перегружает метод `Open()`, реализуя переход в состояние `reading`.

Класс состояния `ReadingState` является локальным классом фабрики состояния `reading`, так как объявлен внутри метода `reading()`. Класс `ReadingState` является потомком класса `OpenedGroup`, так как состояние `reading` входит в группу `opened`. Класс `ReadingState` перегружает метод `Read()`, реализуя петлю в состоянии `reading`.

Классы группы состояний реализуются как вложенные классы контекста. Рассмотрим контекст `File` (рис. 40), реализующий классы группы состояний `ClosedGroup` и `OpenedGroup`, которые соответствуют группам `opened` и `closed` на рис. 33, а. Эти группы обобщают поведение объекта, когда файл открыт или закрыт соответственно. Например, они перегружают метод `IsOpened()`, возвращающий значение `false` в классе `ClosedGroup` и значение `true` в классе `OpenedGroup`.

Реализация классов `ClosedGroup` и `OpenedGroup` приведена ниже.

```
001 class File : public FileProxy {
002 protected:
003   struct ClosedGroup : public State {
/*...*/
009     virtual bool Open(string fname, string mode) {
010       throw IFile::InvalidOperation();
011     }
012
013     virtual void Close() {
014       throw IFile::InvalidOperation();
015     }
016
017     virtual bool IsOpened() const {
018       return false;
019     }
020
021     virtual void Write(int value) {
022       throw IFile::InvalidOperation();
023     }
024
025     virtual int Read() {
026       throw IFile::InvalidOperation();
027     }
028   };
029
030   struct OpenedGroup : public State
031   {
/*...*/
037       virtual bool Open(string fname, string mode) {
038         throw IFile::InvalidOperation();
039       }
040
041       virtual void Close() {
042         NextState(&File::main);
043       }
044
045       virtual bool IsOpened() const {
046         return true;
047       }
048
049       virtual void Write(int value) {
050         throw IFile::InvalidOperation();
051       }
052
053       virtual int Read() {
054         throw IFile::InvalidOperation();
055       }
056   };
057 };
```

Класс `ClosedGroup` является потомком класса `State`, вложенного в каркасный класс `VICBase`. В методе `IsOpened()` класс `ClosedGroup` реализует петлю из группы `closed` (рис. 33, а), возвращая значение `false` (строки 17–19). Класс `ClosedGroup` перегружает также методы `Open()`,

`Close()`, `Write()` и `Read()`, генерируя исключение `InvalidOperation`, так как вызов этих методов в группе `closed` запрещен (нет переходов или петлей с соответствующими причинами).

Класс `OpenedGroup` в методе `IsOpened()`реализует петлю из группы `opened`, возвращая значение `true` (строки 45–47). В методе `Close()` класс `OpenedGroup` реализует переход из группы `opened` в состояние `main` (строки 41–43). В отличие от ***VM***-метода (разд. 4.2.5), здесь нет необходимости специальным образом заботится о закрытии файла, так как за это отвечает деструктор соответствующего состояния. Все остальные методы класса `OpenedGroup` генерируют исключения, так как соответствующие им петли или переходы отсутствуют в группе `opened`.

Рассмотрим более подробно реализацию методов классов состояния и группы состояний. Этот метод обрабатывает переходы, возможные в текущем состоянии, причиной которых является вызов соответствующего метода интерфейса. Например, метод `Read()` класса `ReadingState` обрабатывает петлю в состоянии `reading`, причиной которой является вызов метода `Read()` интерфейса `IFile`. В случае наличия переходов с условиями метод класса состояния реализуется по следующему шаблону:

```
001 virtual <тип > <метод>(<параметры>) {
002   if (<условие перехода>) {
003     <действие на переходе>;
004     NextState(&<контекст>::<следующее состояние>);
005     return <значение>;
006   } else if (<условие перехода>) {
007     NextState(&<контекст>::<следующее состояние>);
008     return <значение>;
009   } else
010     <завершение>;
011 }
```

Метод класса состояния последовательно проверяет выполнение условий перехода и в случае успеха совершает соответствующий переход, выполняя следующие операции:

- действие на переходе;

- смена состояния, выполняется с помощью вызова каркасного метода `NextState()`;
- установка возвращаемого значения.

Выполнение метода состояния завершается сразу после того, как был осуществлен хотя бы один переход. В случае, если не было совершено ни одного перехода, выполняется одно из следующих действий:

- вызывается одноименный метод базового класса, если он есть;
- возвращается значение, идентифицирующее неудачу (например, `false`, `null`).

В качестве примера рассмотрим реализацию класса состояния `MainState` контекста `ReadFile`:

```
001 struct MainState : public virtual ClosedGroup
002 {
003   MainState( TVirtualState& _outer )
004   : ClosedGroup( _outer ) {}
005
006   virtual ~MainState() {}
007
008   virtual bool OpenImpl(string fname, string mode) {
009     if ( mode == "r" ) {
010       NextState( &ReadFile::reading, fname );
011       return true;
012     } else
013       return false;
014   }
015
016   virtual void Open(string fname, string mode) {
017     if (!OpenImpl(fname, mode))
018       ClosedGroup::Open(fname, mode);
019   }
020 };
```

Так как метод `Open()` интерфейса `IFile` возвращает значения типа `void`, то в классе состояния `MainState` этот метод заменяется методом `OpenImpl()`, возвращающим значение типа `bool` (строки 16–19). Метод OpenImpl() обрабатывает переход из состояния `main` в состояние `reading` по причине вызова метода `Open()` интерфейса `IFile` с параметром `mode` равным `"r"`. Метод Open() проверяет выполнение упомянутого условия в

строке 9 и в случае успеха осуществляет переход в состояние `reading` (строка 10). В противном случае возвращается значение `false`.

4.3.6. Расширение логики с помощью наследования

Предлагаемый метод реализации автоматных объектов на основе виртуальных вложенных классов, также так и описанный выше ***VM***-метод, предоставляет возможность расширения их логики с помощью наследования.

Проиллюстрируем расширение логики автоматного объекта с помощью наследования на примере контекста `ReadWriteFile`, диаграмма поведения которого приведена на рис. 33, г. Вложенный класс состояния `MainState` контекста `ReadWriteFile` является потомком классов состояния `MainState` контекстов `ReadFile` и `WriteFile`. Поведение класса `ReadWriteFile::MainState` является объединением и расширением поведения его предков. Исходный код контекста `ReadWriteFile` приведен ниже:

```
001 class ReadWriteFile
002   : public virtual ReadFile
003   , public virtual WriteFile {
004 protected:
005
006   struct MainState
007     : public virtual ReadFile::MainState
008     , public virtual WriteFile::MainState
009   {
010       MainState( TVirtualState& _outer )
011       : ReadFile::MainState( _outer )
012       , WriteFile::MainState( _outer )
013       , ClosedGroup( _outer ) {}
014
015       virtual bool OpenImpl(string fname, string mode) {
016         if ( mode == "rw" ) {
017           NextState( &ReadWriteFile::readwriting, fname );
018           return true;
019         } else if (ReadFile::MainState::OpenImpl(fname, mode)) {
020           return true;
021         } else if (WriteFile::MainState::OpenImpl(fname, mode)) {
022           return true;
023         } else
024           return false;
025       }
026
027       virtual void Open(string fname, string mode) {
028         if (!OpenImpl(fname, mode))
029           ClosedGroup::Open(fname, mode);
```

```
030       }
031   };
032
033   virtual IFile* main() {
034     return new MainState(*this);
035   }
036
037   virtual IFile* readwriting(string fname)
038   {
039       struct ReadWritingState : public OpenedGroup
040       {
041         ReadWritingState(string fname, TVirtualState& _outer)
042           : OpenedGroup( _outer )
043           , file( fname.c_str() ) {}
044
045         virtual void Write(int value) {
046           file << value << endl;
047         }
048
049         virtual int Read() {
050           int res = 0;
051           file >> res;
052           return res;
053         }
054
055       private:
056         fstream file;
057       };
058
059       return new ReadWritingState(fname, *this);
060   }
061 };
```

Класс `MainState` является потомком классов `ReadFile::MainState` и `WriteFile::MainState`. Класс `MainState` перегружает метод `OpenImpl()`, реализующий метода `Open()` интерфейса `IFile`, пытаясь выполнить в нем переход из состояния `main` в состояние `readwriting` (строки 16–18). В случае если упомянутый переход не совершается, то управление последовательно передается в методы базовых классов:

- сначала в метод `Open()` класса состояния `MainState` контекста `ReadFile` (строки 19–20);
- затем в метод `Open()` класса состояния `MainState` контекста `WriteFile` (строки 21–22).

В противном случае возвращается значение `false`.

Класс состояния `ReadWritingState`, является локальным классом фабрики состояния `readwriting` (строки 39–57). Класс `ReadWritingState` является потомком класса `OpenedState`, так как состояние `readwriting` входит в группу `opened`. Класс состояния `ReadWritingState` перегружает методы `Write()` (строки 45–47) и `Read()` (строки 49–53), реализуя в них петли из состояния `readwriting`.

Класс состояния `ReadWritingState` объявляет поле `file` (строка 56), являющееся *вычислительным состоянием* автоматного объекта в управляющем состоянии `readwriting`. Отметим, что вычислительное состояние инкапсулировано в классе состояния и недоступно другим состояниям.

4.3.7. Реализация расширения поведения в группе состояний

В качестве примера рассмотрим следующую задачу: пусть требуется реализовать автоматный класс `LoggedRWFile`, являющийся потомком автоматного класса `ReadWriteFile` и протоколирующий закрытие файла в любом из его состояний. Диаграмма поведения автоматного класса `LoggedRWFile` приведена на рис. 42.

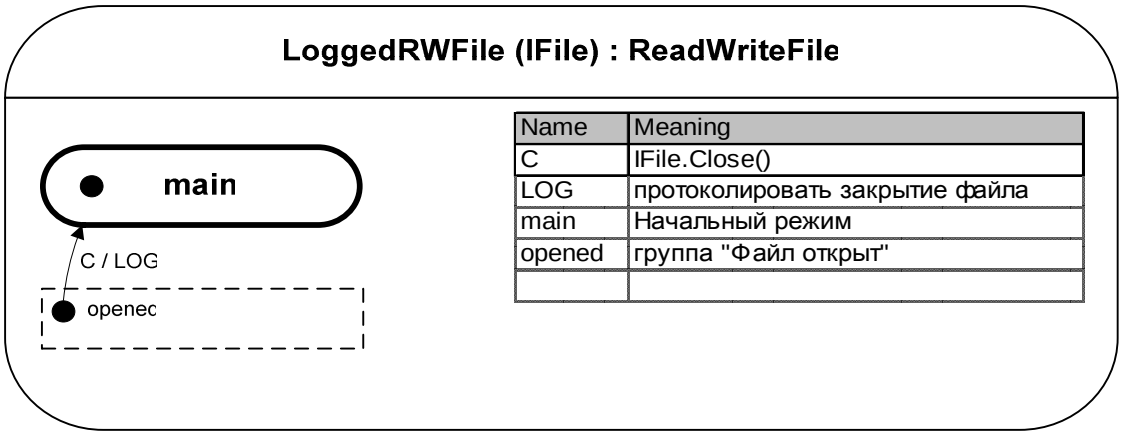


Рис. 42. Диаграмма поведения автоматного класса `LoggedRWFile`

При реализации логики автоматного объекта на основе ***VIC***-метода, обработка вызова метода `Close()` интерфейса `IFile` осуществляется в методе `Close()` класса группы состояний `OpenedGroup` контекста `File`

(рис. 33, а). Иерархия «открытых» классов состояний контекста `ReadWriteFile` приведена на рис. 43.

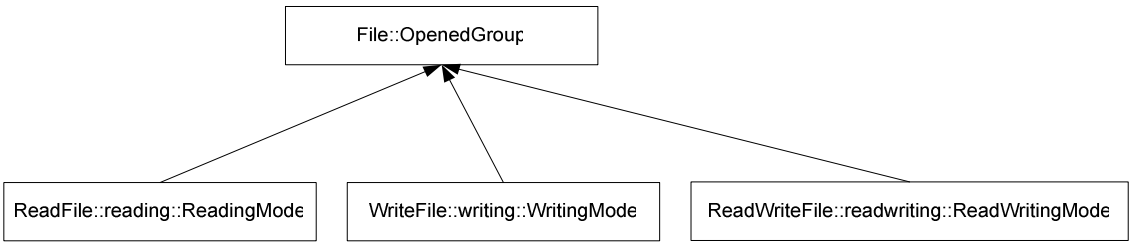


Рис. 43. Иерархия классов состояния и группы состояний контекста `ReadWriteFile`

Поведение, реализованное в классе `OpenedGroup` контекста `File`, наследуется тремя классами: `ReadingState`, `WritingState` и `ReadWritingState`. При текущем варианте реализации, изменить это поведение можно следующими способами:

- перегрузив метод `Close()` во всех трех состояниях, входящих в группу `OpenedGroup`, что приводит к массовому дублированию кода;
- изменив поведение метода `Close()` в классе `OpenedGroup`, что приводит к изменению поведения других контекстов (`ReadWriteFile` и т.д.).

Предлагается следующий способ реализации методов класса группы состояний, позволяющий расширять групповое поведение в потомках контекста. Метод класса группы состояний, предназначенный для расширения, выносится в контекст, что позволяет изменять поведение в этом методе без массового дублирования исходного кода. Пример использования этого приема приведен ниже.

```
001 class File : public FileProxy {
002 protected:
003   /*...*/
004   struct OpenedGroup : public State
005   {
006       OpenedGroup(File& _outer)
007         : State(_outer), context(_outer) {}
```

```
008
009       /*...*/
010
011       virtual void Close() {
012         context.openedClose(*this);
013       }
014
015       /*...*/
016
017   protected:
018       File& context;
019   };
020
021   virtual void openedClose(OpenedGroup& mode) {
022     NextState(&File::main);
023   }
024 };
025
026 /*...*/
027
028 class LoggedRWFile : public ReadWriteFile {
029 protected:
030   virtual void openedClose(OpenedGroup& mode) {
031     cout << "log: file is closed" << endl;
032     ReadWriteFile::openedClose(mode);
033   }
034 };
035
```

Класс группы состояния `OpenedGroup` контекста `File` реализуется метод `Close()` (строки 11–13) через виртуальный метод контекста `openedClose()` (строки 21–23). Для того, чтобы реализовать протоколирование закрытия файла, контекст `LoggedRWFile` перегружает виртуальный метод `openedClose()` контекста `File` (строки 30–33). В итоге для перегрузки одного перехода в группе состояний автоматного класса необходимо перегрузить только один метод, независимо от числа состояний, входящих в эту группу.

4.3.8. Каркас `VICBase`

Рассмотрим каркас, представленный шаблонным классом `VICBase` (рис. 40), который можно рассматривать в качестве библиотечного класса. Единственным параметром этого *шаблонного* класса является интерфейс.

Класс `VICBase` предоставляет два основных семейства методов: `CurrentState()` и `NextState()`. Константная и неконстантная версии метода `CurrentState()` возвращают текущий экземпляр класса состояния.

Основная функциональность класса `VICBase` сосредоточена в семействе шаблонных методов `NextState()`. Каждый из этих методов получает указатель на фабрику состояния и ее параметры. Метод `NextState()` создает экземпляр соответствующего класса `StateCtorX`, где `X` – число параметров вызываемой фабрики состояния. Все классы `StateCtorX` реализуют интерфейс `IStateCtor`. Этот интерфейс содержит единственный метод `Create()`, возвращающий экземпляр класса состояния.

Созданный экземпляр класса `StateCtorX` запоминается классом `VICBase` в виде указателя на `IStateCtor`. Когда текущая транзакция обращения к объекту завершается, класс `VICBase` проверяет присутствие экземпляра `IStateCtor`. Если указатель ненулевой, то с помощью экземпляра `IStateCtor` создается новый экземпляр класса состояния. Уничтожение текущего экземпляра класса состояния является отложенным процессом, поэтому экземпляр не будет разрушен, пока не завершится текущий вызов.

Выводы

Оба предложенных метода реализации автоматных объектов предоставляют возможность декомпозиции и структурирования их логики с помощью наследования. Предложенные методы реализации автоматных объектов полностью покрывают возможности графической нотации для проектирования автоматных объектов, предложенной в настоящей работе. Использование декомпозиции и структурирования логики автоматных объектов приводит к значительному сокращению дублирования исходного кода при реализации реактивных систем.

Метод реализации автоматных объектов на основе виртуальных методов (***VM***-метод) хорошо применим при следующих условиях:

- существует множество автоматных объектов с одинаковым интерфейсом, но с разным поведением;
- логика этих объектов поддается структуризации в виде иерархии;
- в объектах мало вспомогательных данных (полное отсутствие вспомогательных данных, или *вычислительных состояний*, является идеальным случаем).

Метод реализации автоматных объектов на основе виртуальных вложенных классов (***VIC***-метод) является развитием ***VM***–метода. ***VIC***-метод применим при тех же условиях, что и ***VM***-метод, но при этом:

- отсутствует необходимость трудоемкого преобразования методов интерфейса в события;
- имеется возможность хранить *вычислительные состояния* в классах управляющих состояний.

Глава 5. Внедрение предложенных методов проектирования и реализации автоматных объектов в практику разработки реактивных систем

Объектно-ориентированные методы проектирования и реализации реактивных систем, предложенные в данной работе, используются при разработке программного обеспечения телекоммуникационных систем в группе компаний *Транзас* [69].

Группа компаний *Транзас*, основанная в 1990 в Санкт-Петербурге, является одним из мировых лидеров в таких областях, как:

- береговые системы безопасности судоходства;
- морское и авиационное бортовое оборудование;
- интегрированные навигационные комплексы;
- морские и авиационные тренажеры;
- аэронавигационное обеспечение.

Результаты, полученные в диссертации, используются при разработке следующих программных телекоммуникационных систем, производимых компанией *Транзас:*

- коммуникационный сервер системы мониторинга мобильных объектов ***Navi-Manager***;
- каркас редакторов пространственных данных ***Iris***, используемый в системе мониторинга мобильных объектов ***Navi-Manager***.

Планируется повторное использование коммуникационного сервера системы ***Navi-Manager*** в системе online-мониторинга ***Fleet View Online*** и других системах.

5.1. Эмуляция систем связи *Inmarsat-C* и *Inmarsat-D*$^{+}$

5.1.1. Область внедрения

Группа компаний *Transas* является производителем следующих систем мониторинга мобильных объектов:

- ***Navi-Manager*** – система мониторинга и управления мобильными объектами, предоставляющая комплексное организационно-техническое решение для управленческих звеньев организаций, занимающихся эксплуатацией мобильных объектов, обеспечивающих их безопасность и осуществляющих контроль их перемещения;
- ***Fleet View Online*** – интернет-сервис, предназначенный для спутникового мониторинга мобильных объектов.

Основными сферами применения рассматриваемых систем мониторинга мобильных объектов являются:

- мониторинг (в том числе и государственный) различных типов морских, речных и наземных мобильных объектов;
- сопровождение технологических процессов на береговых объектах ГМССБ (глобальные морские системы связи при бедствии);
- поддержка мероприятий по борьбе с чрезвычайными ситуациями;
- охрана исключительной экономической зоны;
- обеспечение безопасности на море.

5.1.2. Использование глобальных систем спутниковой связи

Одной из отличительных особенностей вышеперечисленных систем мониторинга является возможность работы с глобальной системой спутниковой связи ***Inmarsat*** [70]. На данный момент, в продуктах компании *Transas*,

активно используются системы ***Inmarsat-C*** и ***Inmarsat-D+***. Также, идет внедрение системы ***Inmarsat-F77***. Структурная схема использования глобальных систем спутниковой связи в системах мониторинга изображена на рис. 44.

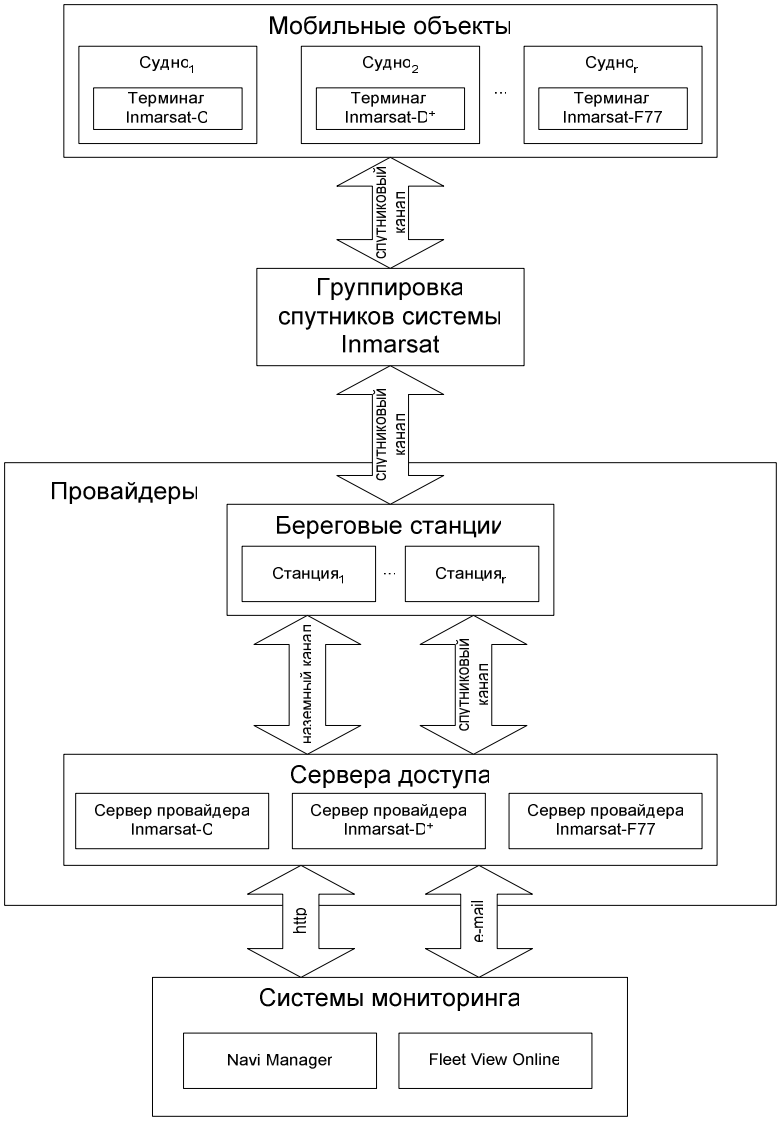


Рис. 44. Структурная схема системы мониторинга

Компонентами схемы являются:

- мобильные объекты – морские и речные суда, оснащенные терминалами той или иной системы ***Inmarsat***;
- группировка спутников системы ***Inmarsat*** – спутниковая система, обеспечивающая глобальное покрытие поверхности земли;

- провайдеры – независимые поставщики услуг доступа к системе ***Inmarsat***;
- береговые станции – стационарные узлы спутниковой связи;
- сервера доступа – сервисы, предоставляющие доступ конечным пользователям к системе ***Inmarsat*** по некоторому протоколу;
- системы мониторинга – программные системы, обеспечивающие прием, отправку и обработку данных.

Одной из основных проблем, возникающих при разработке систем, активно взаимодействующих с системой ***Inmarsat***, является сложность тестирования таких приложений. Источником проблем является невозможность использования реальной системы в интересах тестирования в течение всего жизненного цикла разработки системы. Это связано с тем, что

- компонентами системы являются *реальные* суда;
- услуги спутниковой связи весьма дороги.

Поэтому тестирование на реальной системе осуществляется исключительно редко, в основном на этапе окончательного тестирования перед выпуском очередной версии продукта.

5.1.3. Тестирование взаимодействия с системами глобальной спутниковой связи

Для тестирования взаимодействия с системами глобальной спутниковой связи целесообразно применять программные системы, эмулирующие поведения реальной системы связи. Современные системы мониторинга взаимодействуют с системой ***Inmarsat*** через сервера доступа независимых провайдеров. Вся сложность системы инкапсулируется сервером доступа. Пользователь частично изолируется от факта наличия береговых станций и спутниковых каналов. Создается иллюзия, что в системе присутствует только сервер доступа и связанные с ним мобильные объекты. Это позволяет срав-

нительно легко реализовать эмулятор, имитирующий поведение реальной системы.

Можно выделить следующие варианты тестирования:

- ручное стендовое тестирование;
- автоматическое модульное тестирование.

В случае стендового тестирования эмулятор должен предоставлять интерфейс, неотличимый от интерфейса реальной системы. Так, в случае ***Inmarsat-C***, общение с системой должно происходить посредством отправки и получения электронных писем [71]. В случае модульного тестирования эмулятор должен предоставлять упрощенный интерфейс доступа, избавляющий от асинхронности и других сложностей реальных протоколов связи.

Как следует из рассмотренной структурной схемы, изображенной на рис. 45, эмулятор не содержит таких объектов, как «спутник», «спутниковый канал» и «береговая станция». Взаимодействие сервера доступа и мобильных объектов осуществляется по каналу, свойства которого похожи на реальный многоступенчатый канал связи с мобильным объектом.

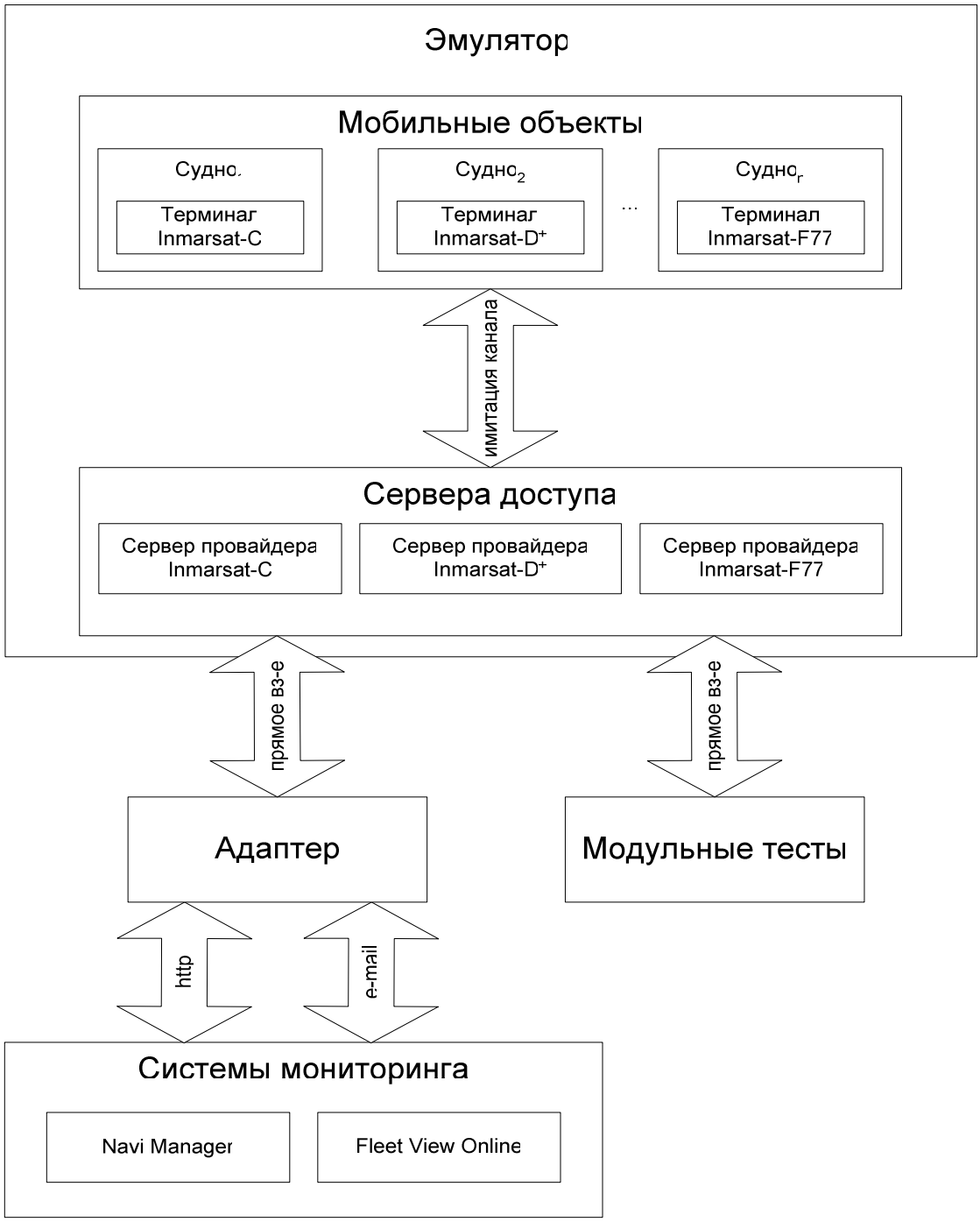


Рис. 45. Структурная схема тестирования

Система глобальной спутниковой связи является типичным представителем реактивных систем. Поведение системы на всех уровнях определяется посредством обмена сообщениями между участниками системы. Эмулятор системы также является реактивной системой и может быть спроектирован и реализован на основе совместного использования объектно-ориентированного и автоматного программирования.

5.1.4. Постановка задачи

Требуется спроектировать и реализовать программную систему-эмулятор, эмулирующую поведение реальной системы глобальной спутниковой связи ***Inmarsat***. Эмулятор должен обеспечивать следующие основные виды функциональности:

- имитация серверов доступа систем ***Inmarsat-C*** и ***Inmarsat-D+*** с интерфейсами идентичными интерфейсам реальных серверов (`http`, `e-mail` и т. д.);
- возможность использования упрощенного интерфейса для облегчения автоматического модульного тестирования;
- имитация поведения набора мобильных объектов, оснащенных терминалами той или иной системы;
- возможность расширения или модификации логики поведения серверов и мобильных объектов с целью тестирования отдельных аспектов системы;
- возможность изменения масштаба времени.

5.1.5. Применимость автоматного программирования

Общая архитектура системы соответствует объектно-ориентированной технологии проектирования и программирования. Объекты системы являются аналогами объектов реальной системы, таких как *сервер доступа*, *терминал* и т.д. С другой стороны, многие объекты системы имеют ярко выраженную реактивную природу, так как имеет место взаимодействие с окружающей средой посредством обмена сообщениями в темпе, задаваемом средой. Такие объекты проектируются и реализуются в виде автоматных объектов. Использование методов, предложенных в данной работе, позволяет использовать наследование для декомпозиции и структурирования логики таких объектов.

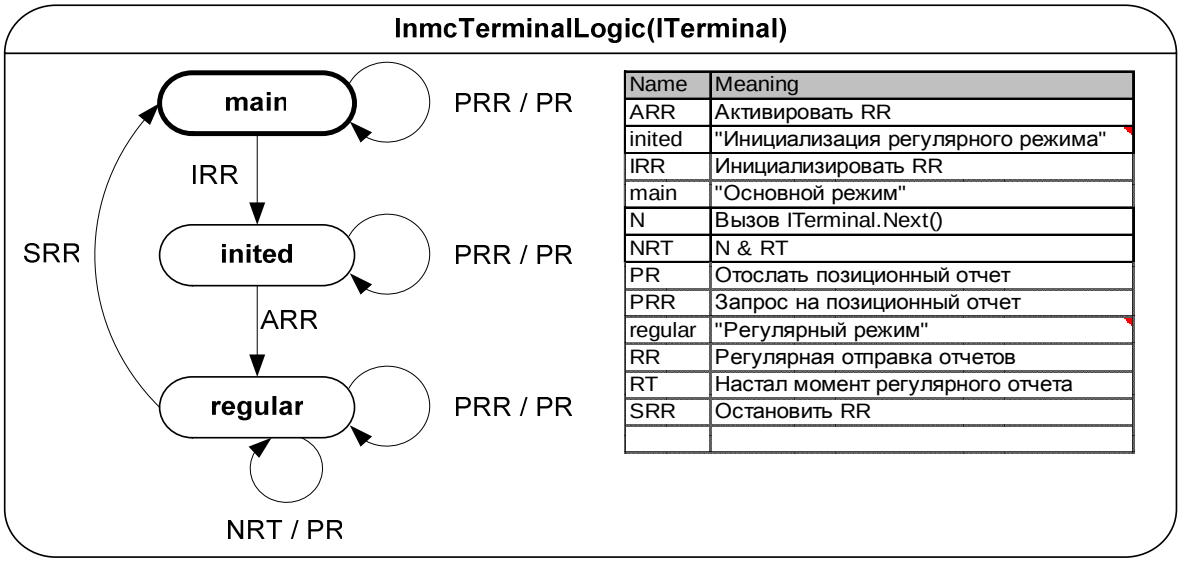
5.1.6. Проектирование поведения терминала системы *Inmarsat*

Эмулятор терминала спутниковой связи реализует интерфейс `ITerminal`, приведенный на листинге ниже:

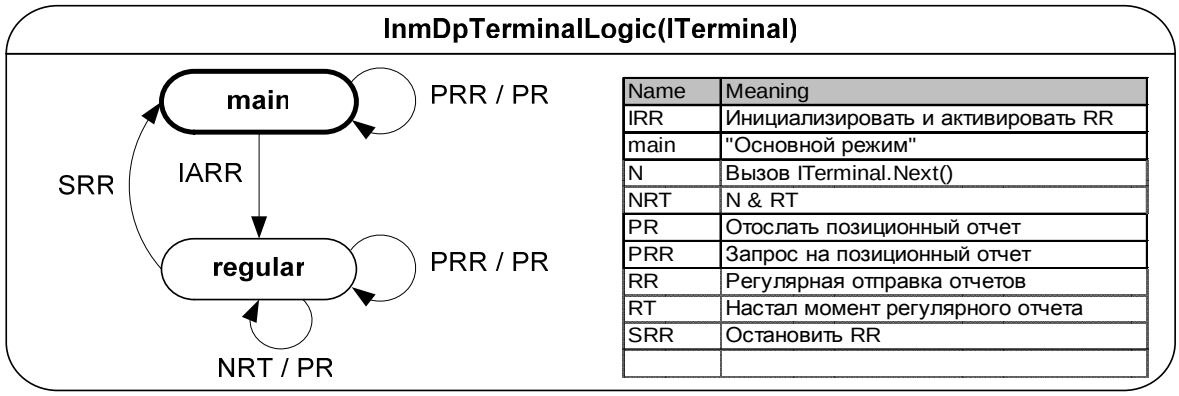
```
001 public interface ITerminal
002 {
/*...*/
006     void Next(DateTime now, TimeSpan step);
007 }
```

Метод `ITerminal.Next()` вынуждает эмулятор терминала обработать свое текущее состояние и вновь поступившие сообщения и перейти в следующее состояние, возможно выполнив при этом некоторые дополнительные действия.

Штатное поведение терминала системы ***Inmarsat-C*** обеспечевается автоматным классом `InmcTerminalLogic`, который реализует интерфейс `ITerminal`. Упрощенная диаграмма поведения автоматного класса `InmcTerminalLogic`, построенная без использования наследования и структурирования, приведена на рис. 46, а.



а) Поведение терминала системы *Inmarsat-C*



б) Поведение терминала системы *Inmarsat-D+*

Рис. 46. Поведение терминалов ***Inmarsat*** без использования декомпозиции и структурирования логики

Терминал системы ***Inmarsat-C*** имеет следующие состояния:

- `main` – основное состояние;
- `inited` – терминал запрограммирован на регулярную отправку отчетов;
- `regular` – терминал находится в состоянии регулярной отправки отчетов.

Терминал рассматриваемой системы характеризуется двухэтапным процессом программирования регулярной отправки позиционных отчетов. На первом этапе задаются параметры регулярной отправки позиционных отчетов. После этого терминал переходит из состояния `main` в состояние

`inited`. На втором этапе состояние регулярной отправки отчетов явно активируется. После этого терминал переходит в состояние `regular`. Регулярная отправка отчетов осуществляется по петле в состоянии `regular`.

В каждом из состояний возможна отправка дополнительного позиционного отчета по явному запросу, представленная одинаковыми петлями в состояниях `main`, `inited` и `regular`.

Штатное поведение терминала системы ***Inmarsat-D***$^{+}$ реализуется автоматным классом `InmDpTerminalLogic`. Диаграмма поведения автоматного класса `InmDpTerminalLogic`, построенная без использования наследования и структурирования, приведена на рис. 46, б. Терминал рассматриваемой системы характеризуется одноэтапным процессом программирования регулярной отправки позиционных отчетов. Передача параметров регулярной отправки позиционных отчетов и активизация состояния регулярной отправки отчетов осуществляются одновременно. После этого терминал переходит из состояния `main` в состояние `regular`.

В каждом из состояний возможна отправка дополнительного позиционного отчета по явному запросу, представленная одинаковыми петлями в состояниях `main` и `regular`.

Поведение терминалов систем ***Inmarsat-C*** и ***Inmarsat-D***$^{+}$ характеризуется следующим общим свойством: терминал может находиться в одном из двух базовых состояний:

- *основное* состояние – позиционный отчет отсылается только по запросу;
- *регулярное* состояние – позиционный отчет отсылается по запросу и автоматически, в заданные моменты времени.

Поведение терминалов может быть обобщено и структурировано абстрактным автоматным классом `InmarsatTerminalLogic`, граф переходов которого приведен на рис. 47, а.

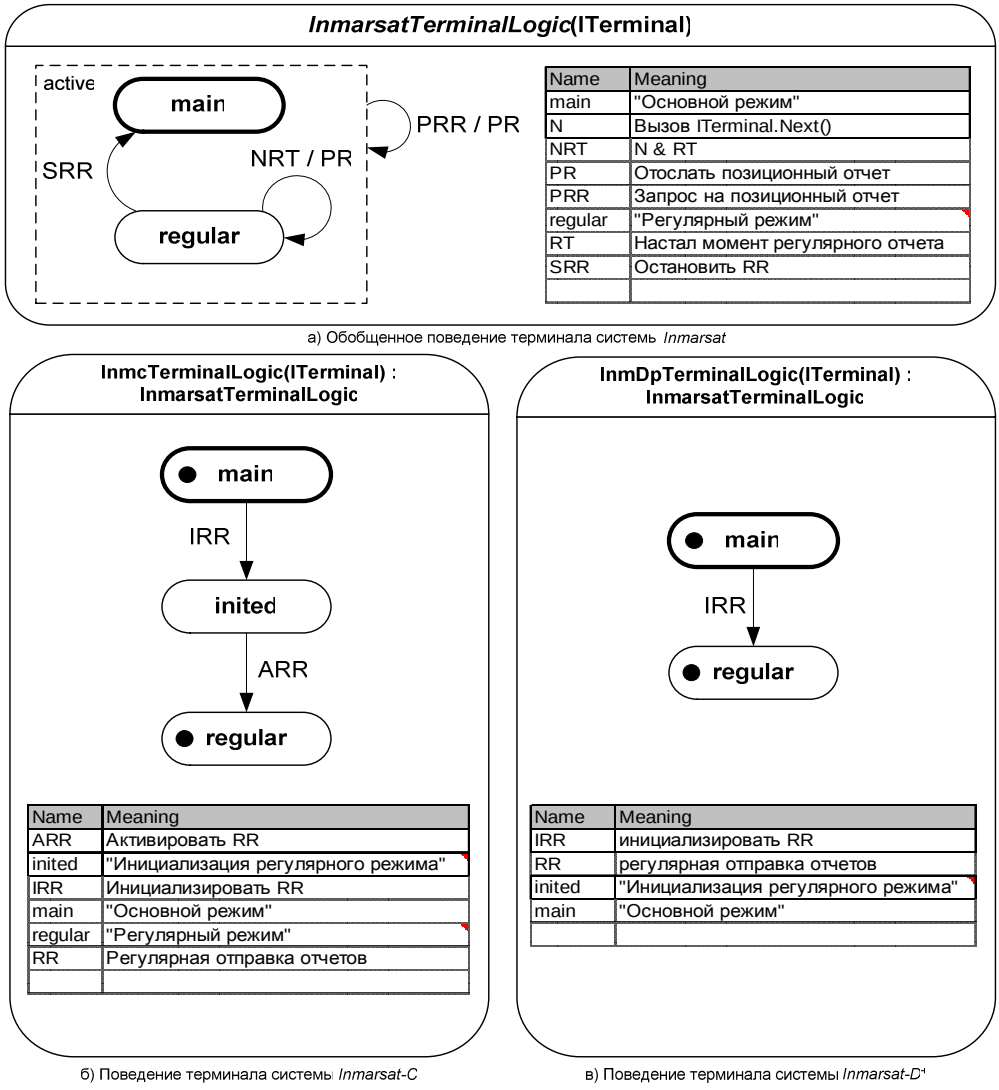


Рис. 47. Поведение терминалов ***Inmarsat*** с использованием декомпозиции и структурирования логики

Абстрактный автоматный класс `InmarsatTerminalLogic` имеет состояния `main` и `regular`. Способ перехода из состояния `main` в состояние `regular` в классе `InmarsatTerminalLogic` не специфицирован. Регулярная отправка позиционных отчетов осуществляется по петле в состоянии `regular`. Прекращение регулярной отправки позиционных отчетов происходит после перехода из состояния `regular` в состояние `main`.

Состояния `main` и `regular` входят в группу состояний `active`, для которой определена петля отправки позиционного отчета по запросу.

Расширяя логику абстрактного автоматного класса InmarsatTerminalLogic, можно в компактном виде реализовать автоматные классы InmcTerminalLogic и InmDpTerminalLogic. Диаграмма поведения автоматного класса InmcTerminalLogic, с использованием наследования и структурирования, приведена на рис. 47, б. Автоматный класс InmcTerminalLogic является потомком абстрактного автоматного класса InmarsatTerminalLogic. Класс InmcTerminalLogic перегружает поведение своего базового класса в состояниях main и regular и добавляет новое состояние – inited.

Дополняя логику класса InmarsatTerminalLogic, автоматный класс InmcTerminalLogic специфицирует двухэтапный переход из состояния main в состояние regular, через состояние inited.

Диаграмма поведения автоматного класса InmDpTerminalLogic, с использованием наследования и структурирования, приведена на рис. 47, в. Автоматный класс InmDpTerminalLogic является потомком абстрактного автоматного класса InmarsatTerminalLogic. Класс InmDpTerminalLogic перегружает поведение своего базового класса в состояниях main и regular.

Дополняя логику класса InmarsatTerminalLogic, автоматный класс InmDpTerminalLogic специфицирует одноэтапный переход из состояния main в состояние regular.

Для сравнения способов проектирования поведения автоматных объектов с использованием декомпозиции и структурирования их логики с помощью наследования и без этого, произведем подсчет использованных состояний, групп состояний и переходов на рис. 46, 47. Результаты подсчета приведены в табл. 1.

Таблица 1. Количественные характеристики способов проектирования

	Без использования наследования и структурирования логики	С использованием наследования и структурирования логики
Количество состояний	5	3
Количество перегруженных состояний	–	4
Количество групп состояний	–	1
Количество переходов	12	6

Как следует из приведенной таблицы, предложенный в настоящей работе способ декомпозиции и структурирования логики автоматных объектов значительно сокращает количество используемых переходов, что происходит за счет устранения их дублирования. При этом в диаграмму добавляются новые сущности, такие как перегруженные состояния и группы состояний.

5.1.7. Реализация терминала системы *Inmarsat*

При программировании эмуляторов терминала используется адаптированный для языка программирования ***C#*** вариант ***VIC***-метода, описанного в разд. 4.3.

Рассмотрим реализацию абстрактного автоматного класса `InmarsatTerminalLogic`, диаграмма поведения которого приведена на рис. 47, а. Данный класс обобщает и структурирует логику поведения терминалов систем ***Inmarsat-C*** и ***Inmarsat-D+***. Исходный код автоматного класса `InmarsatTerminalLogic` приведен ниже.

```
001 public abstract class InmarsatTerminalLogic : IInmarsatTerminal
002 {
```

```
/*...*/
029     protected virtual void ToMainState()
030     {
031         state = new MainState(this);
032     }
033
034     protected abstract void ToRegularState(IExchangeItem item);
/*...*/
044     protected abstract class ActiveGroup : State
045     {
/*...*/
051         public override bool Next(IExchangeItem item, DateTime now,
052                                   TimeSpan step)
053         {
054             if (Context.IsPositionReportRequested(item))
055             {
056                 Context.SendPositionReport(item, now);
057                 return true;
058             } else
059                 return false;
060         }
061     }
062
063     protected class MainState : ActiveGroup
064     {
065         public MainState(InmarsatTerminalLogic context)
066             : base(context)
067         {
068         }
069     }
070
071     protected abstract class RegularState : ActiveGroup
072     {
/*...*/
078         protected abstract bool IsReportTime(DateTime now);
079         protected abstract void SendPositionReport(DateTime now);
080
081         public override bool Next(IExchangeItem item, DateTime now,
082                                   TimeSpan step)
083         {
084             if (base.Next(item, now, step))
085                 return true;
086             else if (IsReportTime(now))
087             {
088                 SendPositionReport(now);
089                 return true;
090             } else if (Context.IsReportingStopped(item))
091             {
092                 Context.ToMainState();
093                 return true;
094             }
095                 return false;
096         }
097     }
098
099     protected abstract class State : IInmarsatTerminal
100     {
/*...*/
124
125         public abstract bool Next(IExchangeItem item, DateTime now,
126                                   TimeSpan step);
127
```

```
128         void ITerminal.Next(DateTime now, TimeSpan step)
129         {
130             IExchangeItem item = Context.PopMessage();
131             Next(item, now, step);
132         }
133     }
/*...*/
151 }
```

Контекст `InmarsatTerminalLogic` объявляет класс группы состояний `ActiveGroup` для группы состояний `active` и классы состояния, входящие в эту группу, `MainState` и `RegularState` для состояний `main` и `regular` соответственно.

Контекст `InmarsatTerminalLogic` объявляет фабрики состояний `ToMainState()` (строки 29–32) и `ToRegularState()` (строка 34), соответствующие состояниям `main` и `regular`. Виртуальная фабрика состояния `ToMainState()` устанавливает в качестве следующего состояния экземпляр класса `MainState`. Так как класс состояния `RegularState` объявлен абстрактным и в контексте `InmarsatTerminalLogic` нет возможности создать его экземпляр, то и фабрика `ToRegularState()` объявлена абстрактной и должна быть обязательно перегружена в потомках.

Все классы состояния контекста `InmarsatTerminalLogic` являются потомками каркасного класса `State`. В отличие от языка программирования ***C++***, при использовании языка программирования ***C#***, класс `State` реализуется вручную. Тип возвращаемого значения метода `void Next()` интерфейса `IInmarsatTerminal` не позволяет идентифицировать осуществление перехода (разд. 4.3.2), поэтому вызов метода `void Next()` интерфейса `IInmarsatTerminal` при реализации преобразуется в вызов метода `bool Next()` класса `State`. Классы состояния перегружают метод `bool Next()` возвращая значение `true`, если переход был осуществлен и значение `false` в противном случае.

Метод `Next()` класса группы состояний `ActiveGroup` реализует петлю в группе `active` (строки 45–54). Класс состояния `MainState` явля-

ется потомком класса ActiveGroup, но при этом не перегружает метод Next(), так как состояние main абстрактного автоматного класса InmarsatTerminalLogic не содержит исходящих из него переходов. Метод Next() класса состояния RegularState вызывает метод Next() своего базового класса ActiveGroup (строка 78), и если в базовом классе не было совершено переходов, то реализует петлю в состоянии regular (строки 86–89). Далее реализуется переход из состояния regular в состояние main (строки 90–93). В противном случае возвращается значение false (строка 85). Кроме этого, класс состояния RegularState объявляет абстрактные *вычислительные* методы, реализация которых выносится в потомки класса RegularState (строки 78, 79).

Рассмотрим реализацию автоматного класса InmDpTerminalLogic, диаграмма поведения которого приведена на рис. 47, в. Данный класс конкретизирует логику поведения терминалов системы ***Inmarsat-D+***, расширяя логику абстрактного автоматного класса InmarsatTerminalLogic. Исходный код автоматного класса InmDpTerminalLogic приведен ниже:

```
001 public class InmDpTerminalLogic : InmarsatTerminalLogic
002 {
017
018     protected override void ToMainState()
019     {
020         NextState = new MainState(this);
021     }
022
023     protected override void ToRegularState(IExchangeItem item)
024     {
025         NextState = new RegularState(this, item);
026     }
027
195
196     protected new class MainState : InmarsatTerminalLogic.MainState
197     {
198         public MainState(InmDpTerminalLogic context) : base(context)
199         {
200         }
201
202         public override bool Next(IExchangeItem item, DateTime now,
203                                   TimeSpan step)
204         {
205             if (base.Next(item, now, step))
206                 return true;
207             else if (Context.IsReportingInitiated(item))
```

```
208             {
209                 Context.ToRegularState(item);
210                 return true;
211             } else
212                 return false;
213         }
214 
215         public new InmDpTerminalLogic Context
216         {
217             get { return (InmDpTerminalLogic)base.Context; }
218         }
219     }
220 
279 }
```

Контекст `InmDpTerminalLogic` перегружает фабрики состояний `ToMainState` и `ToRegularState`, создавая в них экземпляры классов состояния `MainState` и `RegularState` соответственно.

Контекст `InmDpTerminalLogic` перегружает классы состояний `MainState` (строки 176–219) и `RegularState` (строки 221–278), реализующие поведение в состояниях `main` и `regular` соответственно.

Метод `Next()` класса состояния `MainState` вызывает метод `Next()` своего базового класса (строка 205), и если в базовом классе не было осуществлено переходов, реализует переход из состояния `main` в состояние `regular` (строки 207–210).

Класс состояния `RegularState` контекста `InmDpTerminalLogic` не изменяет логику поведения автоматного объекта в состоянии `regular`. Перегрузка класса состояния `RegularState` необходима для реализации абстрактных *вычислительных* методов его базового класса.

5.2. Каркас для построения графических редакторов *Iris*

Визуальное редактирование пространственных данных (электронных карт, пользовательских объектов и т.д.) является важной частью многих приложений в сфере навигационных, береговых и тренажерных систем. Большинство современных программных систем, разрабатываемых в этих областях, являются сложными, распределенными программно-аппаратными комплексами. Поэтому актуальной задачей является создание повторно исполь-

зуемого каркаса, облегчающего реализацию визуального редактирования пространственных данных.

Под термином *каркас* здесь понимается библиотека, определяющая архитектуру решения некоторой задачи в прикладных программных системах. Каркас ***Iris***, разрабатываемый в компании *Транзас*, предназначен для построения редакторов пространственных (*spatial*) данных. Архитектура каркаса ***Iris*** основана на принципах и идеях, предложенных Дж. Влиссидесом и М. Линтоном в каркасе для построения прикладных графических редакторов ***Unidraw*** [72], но с учетом специфики работы с пространственными данными в условиях распределенной программной системы. Ниже приведен краткий обзор каркаса ***Unidraw***, являющегося прообразом каркаса ***Iris***.

Одной из основных составляющих каркаса ***Iris*** являются манипуляторы редактирования. Манипуляторы редактирования являются типичными представителями реактивных систем и для их разработки применяются методы проектирования и реализации автоматных объектов, предложенные в настоящей работе.

5.2.1. Обзор каркаса *Unidraw*

Каркас ***Undraw*** занимает нишу между прикладным графическим редактором и оконной системой, обеспечивая редактор необходимой для его работы функциональностью. Объектная модель, являющаяся основой каркаса ***Undraw***, состоит из следующих основных иерархий [72]:

- *компоненты* – представляют элементы прикладной области, определяя их внешний вид и семантику. Примерами компонентов являются электронные детали в редакторе проектирования печатных плат, ноты в музыкальном редакторе и т.д.;
- *инструменты* – обеспечивают интерактивное редактирование компонентов, отвечая, в том числе, за анимацию и прочие графические эффекты, необходимые для обратной связи с пользовате-

лем. Примерами инструментов являются инструмент выделения, инструмент переноса объектов и т.д.;

- *команды* – определяют операции над компонентами и другими объектами. Примерами команд являются команды изменения атрибута объекта, поворота объекта и т.д.;
- *внешние представления* – отвечают за зависящее от конкретной предметной области представление компонентов вне редактора. Примерами внешних представлений являются звуковой файл в музыкальном редакторе, исходный код в редакторе языка ***UML*** и т.д.

Каркас ***Unidraw*** предоставляет базовые классы для компонентов, инструментов, команд и внешних представлений. Объекты прикладного графического редактора наследуются от этих базовых классов и определяют свое поведение через заданный ими протокол.

В каркасе ***Unidraw*** используется вариант паттерна проектирования пользовательского интерфейса *Model-View-Controller* (*MVC*) [73]. Компоненты в каркасе ***Unidraw*** явно разделяются на *субъект* (*subject*) и *вид* (*view*). *Субъект* – это контекстно-независимое состояние компонента и операции применимые к нему. *Вид* – это контекстно-зависимое представление компонента. Субъект компонента может иметь один или более видов, каждый из которых представляет данный компонент в разных контекстах. *Субъект* уведомляет связанные с ним *виды* об изменении своего состояния.

5.2.2. Механизмы редактирования в каркасе *Unidraw*

Механизм интерактивного редактирования в каркасе ***Unidraw*** состоит из тройки объектов:

- *инструмент* (*tool*);
- *манипулятор* (*manipulator*);

- *команда* (*command*).

Большая часть работы в течение процесса интерактивного редактирования выполняется *манипулятором*. Манипулятор отвечает за обработку сообщений пользовательского интерфейса и визуализацию процесса редактирования. Интерфейс манипулятора, приведен ниже:

```
001 class Manipulator {
002 public:
003     virtual ~Manipulator();
004 
005     virtual void Grasp(Event&);
006     virtual boolean Manipulating(Event&);
007     virtual void Effect(Event&);
008 
009     virtual void SetViewer(Viewer*);
010     virtual void SetTool(Tool*);
011 
012     virtual Viewer* GetViewer();
013     virtual Tool* GetTool();
014 protected:
015     Manipulator();
016 };
```

Наибольший интерес представляет тройка методов `Grasp()`–`Manipulating()`–`Effect()`, через которую осуществляется управление процессом интерактивного редактирования.

Рассмотрим процесс редактирования в каркасе ***Unidraw*** [74]. Управляют процессом редактирования, как показано диаграмме деятельности на рис. 48, *вид* и *инструмент*.

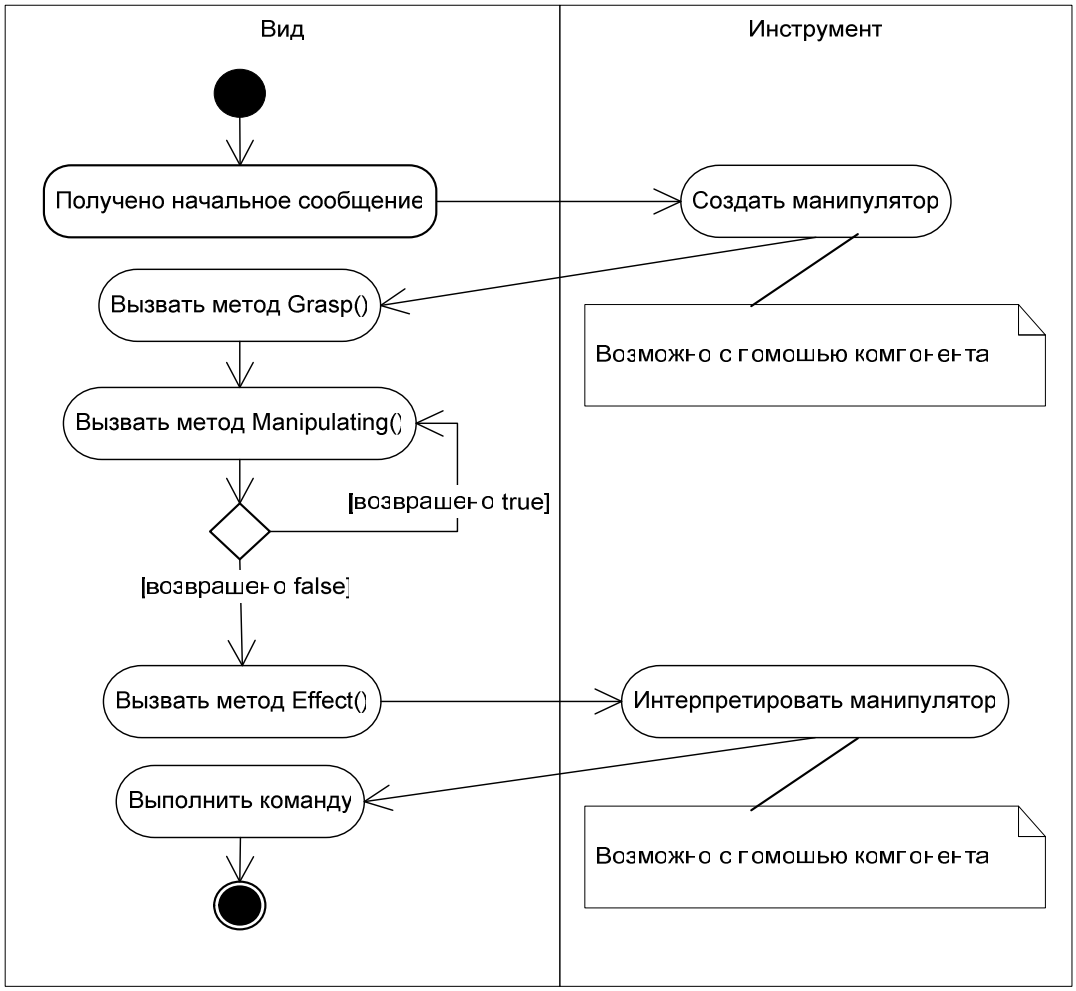


Рис. 48. Процесс редактирования в каркасе ***Unidraw***

В течение цикла редактирования выполняются следующие действия:

1. *Вид* получает пользовательское сообщение, например, нажатие клавиши мыши.
2. *Вид* запрашивает у текущего *инструмента* создание манипулятора, соответствующего полученному сообщению.
3. Для создания манипулятора инструмент выполняет одно из следующих действий:

   3.1. Создает манипулятор самостоятельно.

   3.2. Запрашивает создание манипулятора у *компонента*.
4. В течение процесса манипуляции *вид* выполняет следующие действия:

4.1. Вызывает метод манипулятора `Grasp()`, передавая в качестве параметра начальное сообщение.

4.2. Ожидает вновь поступающие сообщения и передает их методу манипулятора `Manipulating()`, до тех пор, пока `Manipulating()` не вернет false.

4.3. Вызывает метод манипулятора `Effect()`, передавая в качестве параметра последнее полученное сообщение.

5. *Вид* запрашивает у текущего *инструмента* интерпретацию манипулятора.

6. Для интерпретации манипулятора *инструмент* выполняет одно из следующих действий:

6.1. Интерпретирует манипулятор самостоятельно, создавая соответствующую *команду*.

6.2. Запрашивает интерпретацию манипулятора и создание *команды* у *компонента*.

7. Вид выполняет созданную команду.

8. Выполнение команды завершает процесс редактирования.

Каркасом ***Unidraw*** *неявно (*через интерфейс манипулятора) предопределяется набор состояний, в которых может находиться манипулятор. В явном виде конечные автоматы для построения и реализации манипуляторов в каркасе ***Unidraw*** не используются.

Построим граф переходов конечного автомата, соответствующего поведению манипуляторов в каркасе ***Unidraw*** (рис. 49).

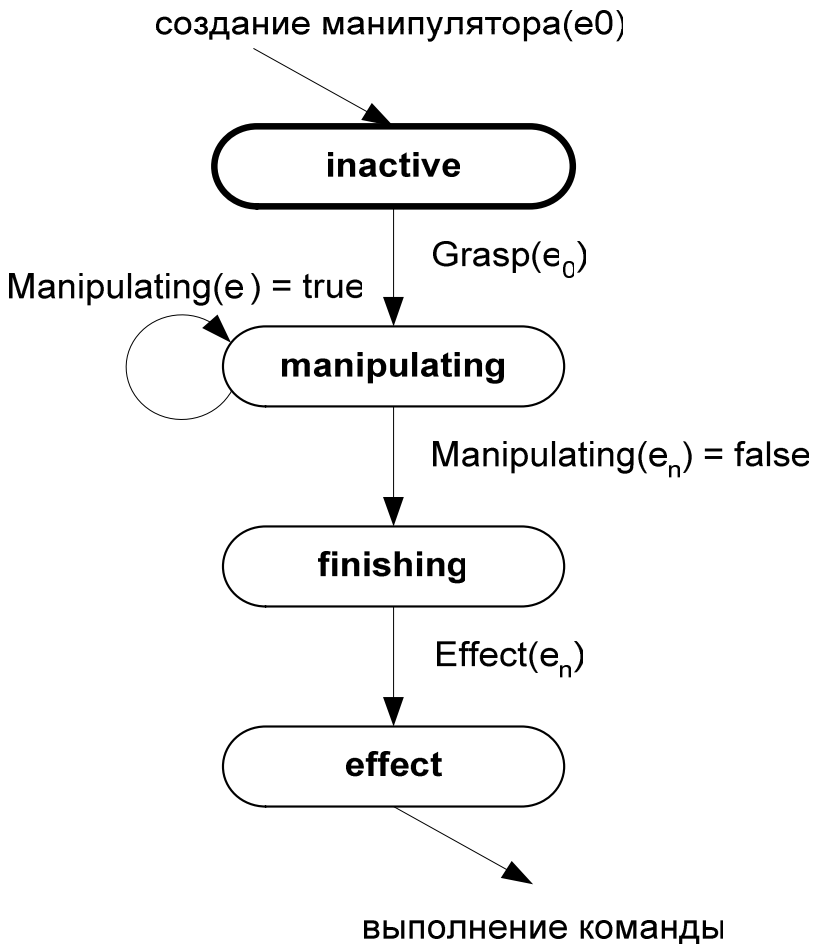


Рис. 49. Граф переходов манипулятора в каркасе ***Unidraw***

Отметим, что состояние `manipulating` может иметь вложенные состояния, которые каркасом не специфицируются.

Инструмент отвечает за создание подходящего манипулятора на основе первого полученного сообщения ввода (например, нажатие кнопки мыши). Это сообщение передается манипулятору в метод `Grasp()`. После этого манипулятор переходит из состояния `inactive` в состояние `manipulating`. В течение манипуляции манипулятор обрабатывает все последующие сообщения, получая их через метод `Manipulating()`, до тех пор, пока не перейдет в состояние `finishing`. После этого последнее сообщение повторно передается манипулятору в метод `Effect()`. После этого манипулятор переходит в состояние `effect`. На этом процесс манипуляции завершается. Манипулятор передается инструменту для интерпретации совершенной ма-

нипуляции. Инструмент создает команду, с помощью которой выполняется соответствующее манипуляции действие, например перенос объекта.

В предложенном подходе к организации процесса интерактивного редактирования были выявлены следующие недостатки:

- предлагаемая архитектура не предполагает возможности расширения и структурирования логики поведения манипуляторов;
- существуют (и широко используются) манипуляторы, которые требуют выполнения промежуточных команд в течение времени своей манипуляции. Например, смешанный манипулятор выделения/переноса, часто используемый в современных графических редакторах в качестве *манипулятора по умолчанию*, должен иметь возможность выполнить команду выделения для переносимого объекта. Архитектура, предлагаемая в каркасе ***Unidraw,*** не позволяет выполнять несколько команд в течение одной манипуляции;
- предлагаемая архитектура предполагает многократную обработку некоторых сообщений. Как показано на рис. 49, сообщения e_0 и e_n обрабатываются дважды. Это провоцирует либо массовое дублирование кода, либо значительное его усложнение и децентрализацию;
- некоторые манипуляции не укладываются в предлагаемую жесткую схему. Например, некоторые манипуляции, скажем «*удаление выделенного объекта*», имеют вырожденное состояние `manipulating`, что приводит к понижению читабельности кода;
- предлагаемая архитектура провоцирует нарушение инкапсуляции, так как логика манипуляции распределяется между *манипулятором* и *инструментом*. Как показано на рис. 49, первое со-

общение частично обрабатывается как инструментом, так и манипулятором. После завершения манипуляции, ее результат интерпретируется инструментом.

В методе построения манипуляторов на основе автоматных объектов, используемом в каркасе ***Iris***, указанные недостатки устранены. В следующих разделах приводится краткое описание общей архитектуры каркаса ***Iris*** и подробное описание реализации манипуляторов на его основе.

5.2.3. Архитектура каркаса *Iris*

Архитектура каркаса ***Iris*** похожа на архитектуру каркаса ***Unidraw***. Каркас ***Iris*** занимает нишу между прикладным редактором пространственных данных и оконной системой, обеспечивая редактор необходимой для его работы функциональностью. Многоуровневая структура прикладного графического редактора пространственных данных на основе каркаса ***Iris*** показана на рис. 50.

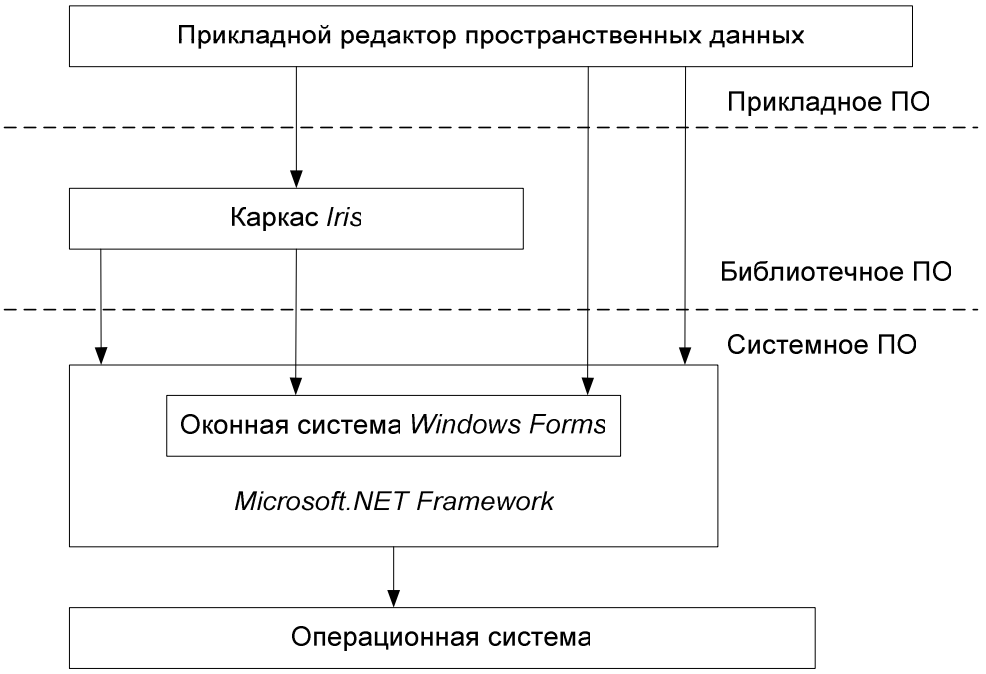


Рис. 50. Структура графического редактора на основе ***Iris***

На самом нижнем уровне находится операционная система. Следующим уровнем является каркас ***Microsoft.NET Framework***, абстрагирующий под-

робности конкретных операционных систем и предоставляющий объектно-ориентированный, *управляемый* интерфейс. Составной частью ***Microsoft.NET Framework*** является оконная система ***Windows Forms***, предоставляющая объектно-ориентированный, управляемый каркас для построения оконного интерфейса. Следующим уровнем является каркас ***Iris***, предоставляющий набор абстракций, необходимых для построения редакторов пространственных данных.

В каркасе ***Iris*** учитываются многие специфические моменты, связанные с редактированием пространственных данных. Редактируемые данные произвольным образом проецируются на плоскость поверхности экрана, и редактирование осуществляется в экранных координатах. Не делается никаких предположений об исходном формате данных и способе проецирования. Могут использоваться как двух-, так и трехкомпонентные системы географических координат. Каркас также ***Iris*** учитывает особенности работы с циклическими системами координат, часто используемыми при работе с пространственными данными.

Каркас ***Iris*** предоставляет традиционный для современных графических редакторов набор функциональности: развитые механизмы селекции и редактирования объектов, откат и повторное выполнение команд, произвольные ограничения на геометрию редактируемых объектов, групповые операции и т.д.

5.2.4. Механизмы в каркасе *Iris*

Одним из основных понятий каркаса ***Iris*** является *механизм* (*gear*). *Механизм* полностью специфицирует выполнение одной или нескольких семантических операции над объектами. Примерами механизмов являются механизм выделения объектов, механизм переноса объектов и т.д. Различаются:

- *механизмы выделения* (например, механизм «выделение прямоугольником»);

- *механизмы редактирования* (например, механизм «перенос объекта»);
- *механизмы* управления проекцией (например, механизм «центрирования»).

Механизм является совокупностью следующих объектов:

- *инструмент* – является организующим звеном редактирования;
- *манипулятор* – отвечает за процесс выполнения операции;
- *метка* – помечает пользовательский объект как «пригодный» для данного механизма;
- *призрак* – отвечает за визуализацию процесса манипулирования пользовательским объектом;
- *команда* – хранит *всю* необходимую информацию для выполнения одного из действий механизма.

В некоторых механизмах, одна или несколько из перечисленных составных частей отсутствуют. Например, механизм удаления объектов не содержит призрака, так как процесс удаления объекта не нуждается в визуализации.

Все инструменты являются фабриками манипуляторов. Инструмент также может выполнять вспомогательную деятельность вроде инициализации механизма и выполнения команд.

Призрак обычно создается пользовательским объектом для визуализации известного типа манипуляции. Таким образом, призрак имеет знание о конкретных типах текущего механизма и редактируемого пользовательского объекта. Информация о текущем процессе манипуляции передается призраку посредством команды.

В качестве примера рассмотрим механизм вращения объектов. Он состоит из следующих классов:

- `RotateTool` – инструмент, является фабрикой манипулятора `RotateManipulator`;
- `RotateManipulator` – манипулятор, отвечает за протокол механизма вращения, инкапсулируя процесс «установка центра – задание угла – генерация команды». Визуализирует процесс манипулирования, и, в частности, рисует «центр вращения»;
- `IRotatable` – метка, помечает «вращаемые» объекты;
- `PolygonRotateGhost` – призрак, визуализирует процесс поворота полигонального объекта;
- `RotateCmd` – команда, содержит ссылку на вращаемый объект, центр вращения и угол поворота.

Рассмотрим типичный жизненный цикл механизма вращения:

- пользователь устанавливает инструмент `RotateTool` в качестве текущего инструмента;
- координатор процесса редактирования вызывает метод `CreateManipulator()`. В результате создается экземпляр `RotateManipulator`;
- манипулятор `RotateManipulator` ожидает одиночного нажатия кнопки мыши, устанавливающего центр вращения. После этого установленный центр начинает визуализироваться;
- манипулятор `RotateManipulator` ожидает наступления комплексного события «нажали кнопку мыши и потащили», в результате которого для всех *выделенных* объектов создаются призраки;
- после каждого события «перемещение мыши» для каждого призрака создается команда, содержащая текущие центр и угол по-

ворота. В результате выполнения этих команд призраки получают информацию о текущем состоянии манипуляции. Центр вращения, как и прежде, визуализируется манипулятором. Процесс вращения объектов визуализируется соответствующими призраками;

- после события «отпущена кнопка мыши» для каждого выделенного объекта создается и выполняется команда, содержащая центр и окончательный угол поворота.

Манипулятор является типичной реактивной системой, так как его поведение полностью определяется потоком поступающих сообщений пользовательского интерфейса. Причем результат *манипуляции* может зависеть как от порядка, так и от скорости поступления сообщений. Воздействие на окружающую среду манипулятор также осуществляет с помощью посылки сообщений.

5.2.5. Реализация манипуляторов в каркасе *Iris*

В каркасе ***Iris*** манипуляторы являются автоматными объектами. Проектирование манипуляторов осуществляется с помощью графической нотации, описанной в разд. 3.4. Манипуляторы реализуются на основе ***VIC***-метода, описанного в разд. 4.3.

Интерфейс манипулятора `IManipulator` предоставляет только два метода:

- `Manipulating()`, предназначенный для обработки поступающих сообщений;
- `Draw()`, предназначенный для визуализации текущего состояния манипуляции.

Интерфейс `IManipulator` определяется следующим образом:

```
001 public interface IManipulator : IDrawable
002 {
```

```
003     ICommand Manipulating(UserEvent userEvent);
004 }
```

Отметим, что метод `Draw()` определен в интерфейсе `IDrawable`.

Манипулятор создается в момент активации инструмента. Инструмент не участвует ни в обработке сообщений, ни в создании результирующей команды. Вся логика, связанная с осуществлением манипуляции, *полностью* сконцентрирована внутри манипулятора. Внешние по отношению к манипулятору объекты даже не имеют представления о текущем состоянии манипуляции, вплоть до того, что неизвестно, началась ли она вообще. Это позволяет универсально реализовывать манипуляторы как с очень простым, вырожденным циклом манипуляции, так и с очень сложным, многоступенчатым циклом.

При получении каждого следующего сообщения вызывается метод `Manipulating()`, которому в качестве единственного параметра передается полученное сообщение. Любая ненулевая команда, возвращенная методом `Manipulating()`, немедленно выполняется, что позволяет выполнять несколько команд в течение одной манипуляции. При таком подходе каждое событие обрабатывается не более одного раза, что значительно уменьшает дублирование и повышает читабельность кода.

Часть иерархии манипуляторов, предопределенных каркасом ***Iris***, приведена на рис. 51.

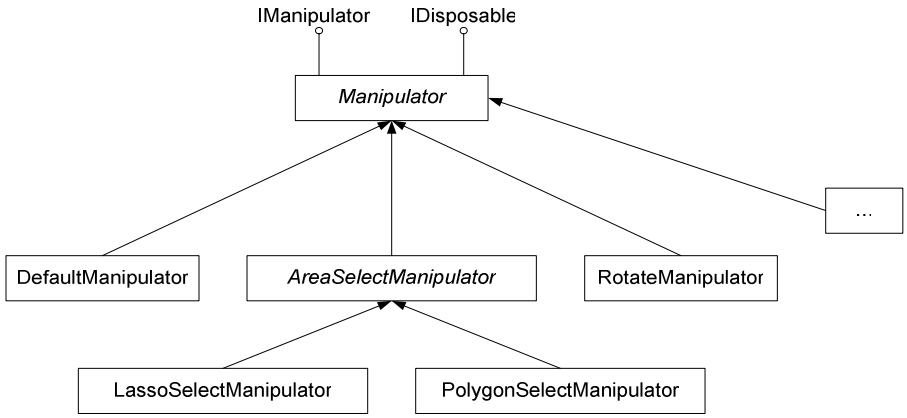


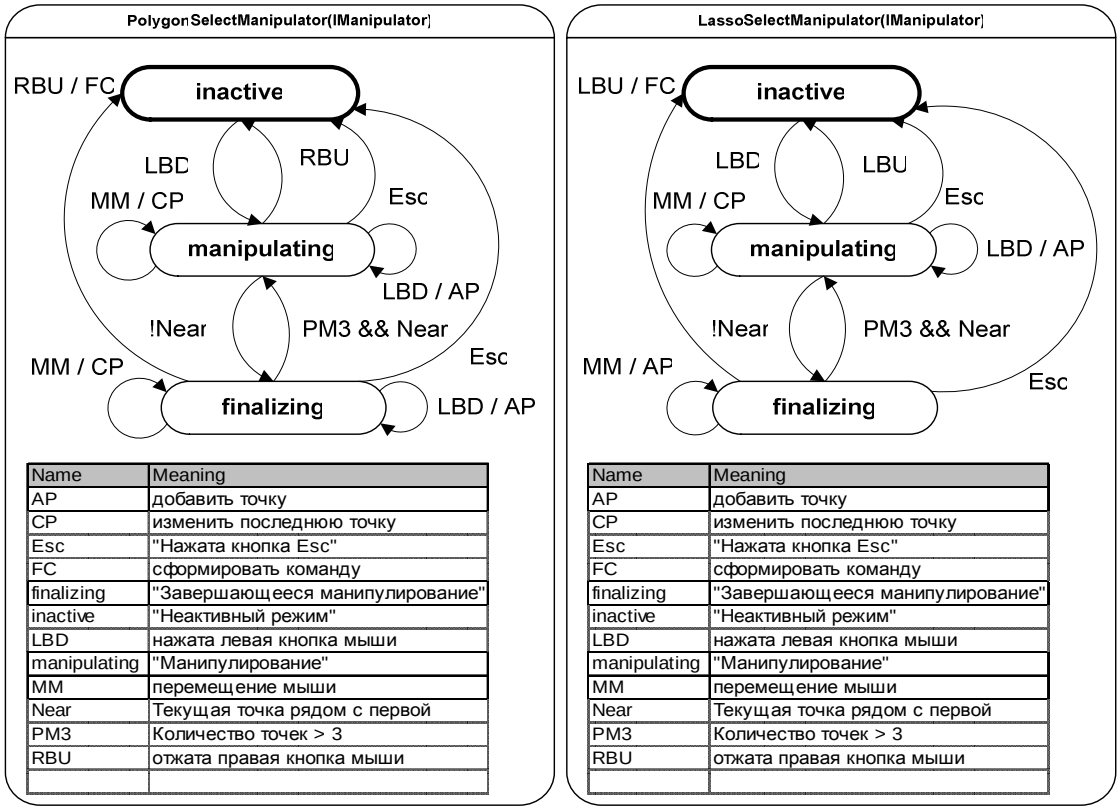
Рис. 51. Иерархия манипуляторов, предопределенных в каркасе ***Iris***

Корневым элементом иерархии является абстрактный автоматный класс `Manipulator`, реализующий интерфейсы `IManipulator` и `IDisposable`. Автоматный класс `Manipulator` одновременно является *посредником* и *контекстом* (разд. 4.3.1). В качестве посредника он переадресует все вызовы методов интерфейса методам текущего экземпляра класса состояния. Автоматный класс `Manipulator` в качестве *контекста* будет рассмотрен ниже.

Рассмотрим декомпозицию и структурирование логики автоматных объектов с помощью наследования на примере площадных манипуляторов выделения каркаса ***Iris***. Каркас ***Iris*** содержит следующие площадные манипуляторы выделения:

- *выделение прямоугольником*;
- *выделение многоугольником*;
- *выделение лассо*.

Диаграммы поведения манипуляторов выделения *многоугольником* и *лассо* приведены на рис. 52.



Name	Meaning
AP	добавить точку
CP	изменить последнюю точку
Esc	"Нажата кнопка Esc"
FC	сформировать команду
finalizing	"Завершающееся манипулирование"
inactive	"Неактивный режим"
LBD	нажата левая кнопка мыши
manipulating	"Манипулирование"
MM	перемещение мыши
Near	Текущая точка рядом с первой
PM3	Количество точек > 3
RBU	отжата правая кнопка мыши

а) Выделение многоугольником

Name	Meaning
AP	добавить точку
CP	изменить последнюю точку
Esc	"Нажата кнопка Esc"
FC	сформировать команду
finalizing	"Завершающееся манипулирование"
inactive	"Неактивный режим"
LBD	нажата левая кнопка мыши
manipulating	"Манипулирование"
MM	перемещение мыши
Near	Текущая точка рядом с первой
PM3	Количество точек > 3
RBU	отжата правая кнопка мыши

б) Выделение лассо

Рис. 52. Диаграммы поведения манипуляторов выделения

Как следует из приведенной диаграммы, автоматные классы `PolygonSelectManipulator` и `LassoSelectManipulator`, соответствующие выделению *многоугольником* и *лассо*, имеют *похожее* поведение, которое может быть обобщено и структурировано с помощью наследования так, как показано на рис. 53.

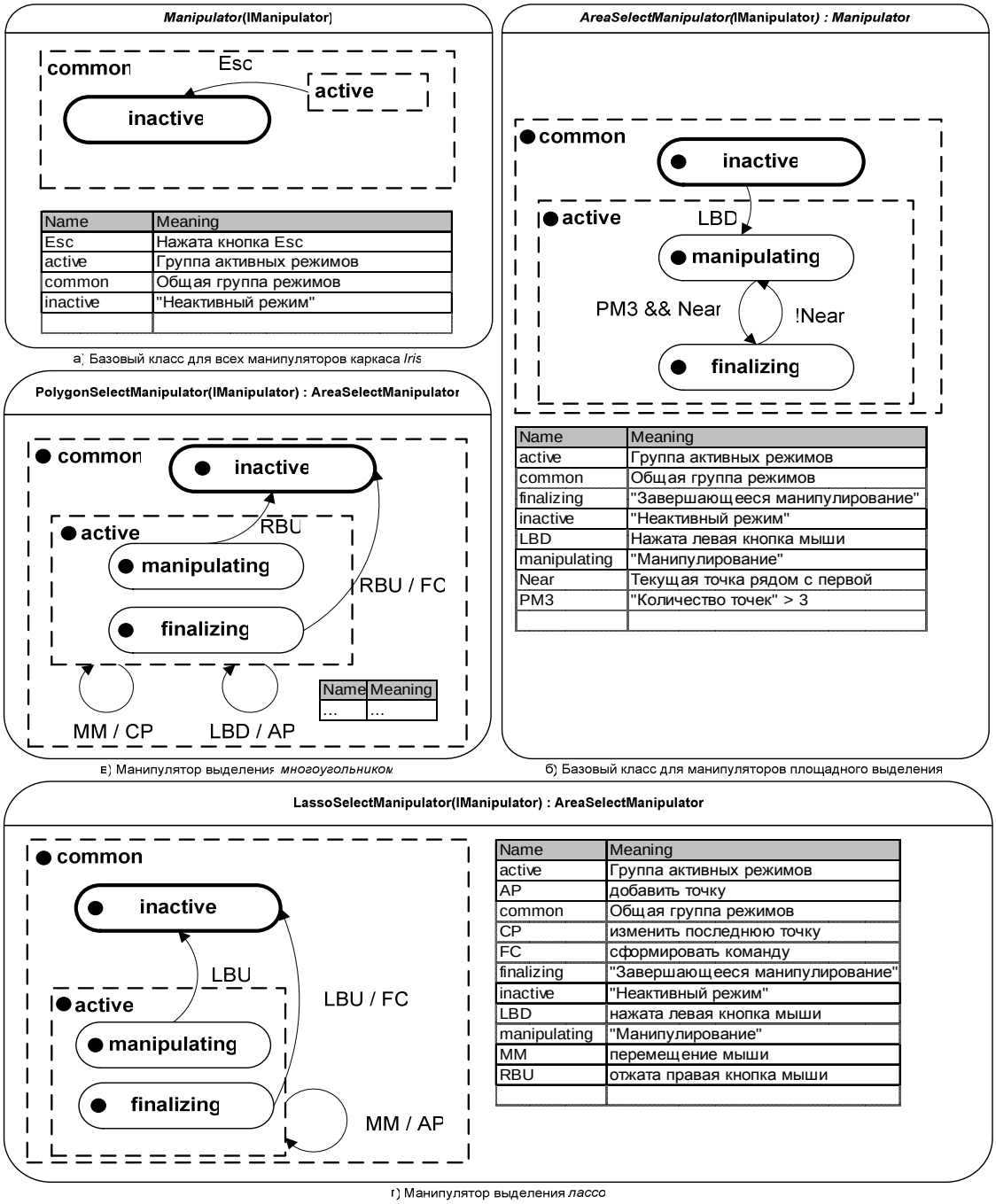


Рис. 53. Диаграммы поведения манипуляторов выделения с использованием наследования

Как показано на рис. 53, автоматные классы `PolygonSelectManipulator` и `LassoSelectManipulator`, соответствующие *выделению многоугольником* и *лассо*, имеют общий базовый автоматный класс – `AreaSelectManipulator`. Конкретные манипуляторы расширяют логику своего базового общего класса, перегружая его поведение

в состояниях и группах состояний. Автоматный класс `AreaSelectManipulator` в свою очередь является потомком автоматного класса `Manipulator`.

Абстрактный автоматный класс `Manipulator` (рис. 53, а) задает общую схему поведения манипуляторов каркаса ***Iris***, определяя состояние `inactive` и группы состояний `common` и `active`. Группа `common` необходима для дальнейшего расширения поведения манипулятора сразу во всех состояниях. Переход из группы `active` в состояние `inactive` специфицирует завершение любой манипуляции на любом этапе по нажатию клавиши `Esc`.

Абстрактный автоматный класс `AreaSelectManipulator` (рис. 53, б) задает общую схему поведения площадного манипулятора. По нажатию левой кнопки мыши манипулятор переходит в активное состояние (группа состояний `active`). В границу области выделения добавляется точка под курсором мыши. Манипулятор имеет два «активных» состояния:

- `manipulating` – обычное «активное» состояние;
- `finalizing` – завершающее «активное» состояние.

Переходы между состояниями `manipulating` и `finalizing` совершаются в зависимости от того, насколько близко курсор мыши находится к первой точке границы области выделения. Если курсор находится «достаточно близко», то манипулятор находится в состоянии `finalizing`. Граница области выделения в этом состоянии рисуется жирной линией. Прекращение манипуляции в состоянии `finalizing` приводит к возвращению команды на выделение из метода `Manipulating()`. Прекращение манипуляции в состоянии `manipulating` равнозначно отмене манипуляции (команда выделения не возвращается и манипуляция заканчивается *ничем*).

Автоматный класс `PolygonSelectManipulator` (рис. 53, в) конкретизирует логику поведения класса `AreaSelectManipulator` следую-

щим образом. Добавление точки в границу области выделения осуществляется по нажатию левой клавиши мыши. Изменение последней точки границы осуществляется по перемещению мыши. Завершение манипуляции осуществляется по нажатию правой клавиши мыши.

Автоматный класс `LassoSelectManipulator` (рис. 53, г) конкретизирует логику поведения класса `AreaSelectManipulator` следующим образом. Добавление точки в границу области выделения осуществляется по перемещению мыши. Завершение манипуляции осуществляется по отпусканию левой клавиши мыши.

Для демонстрации преимуществ использования декомпозиции и структурирования логики автоматных объектов с помощью наследования, сравним количество переходов в классе `PolygonSelectManipulator` с использованием наследования (рис. 52, а) и без него (рис. 53, в). Количество переходов на диаграмме поведения автоматного класса `PolygonSelectManipulator` на рис. 52 равно 11, в то время как диаграмма поведения автоматного класса `PolygonSelectManipulator` на рис. 53 содержит всего четыре перехода.

Рассмотрим фрагменты реализации абстрактного автоматного класса `AreaSelectManipulator`.

```
001 public abstract class AreaSelectManipulator : Manipulator
002 {
/*...*/
015     protected override void ToInactiveState()
016     {
017         State = new InactiveState(this);
018     }
019 
020     protected abstract void ToManipulatingState(IPinPoint[] points);
021 
022     protected abstract void ToFinalizingState(IPinPoint[] points);
023 
024     protected class InactiveState : CommonGroup
025     {
026         public InactiveState(AreaSelectManipulator context) : base(
027                                 context)
028         {
029         }
030 
031         public override ICommand Manipulating(UserEvent userEvent)
032         {
```

```
033             if (userEvent.Type == UserEvent.EventType.
034                 Mouse_LButtonDown)
035             {
036                 MouseUserEvent mue = (MouseUserEvent) userEvent;
037                 IPinPoint point = Context.pinBox.Create(mue.Point);
038                 Context.ToManipulatingState(new IPinPoint[]{ point,
039                                             point });
040                 return null;
041             }
042             return null;
043         }
/*...*/
056     }
057
058     protected new abstract class ActiveGroup : Manipulator.
059     ActiveGroup
060     {
/*...*/
069         public ArrayList Points
070         {
071             get { return points; }
072         }
073
074         public override void Draw(Graphics graphics, IStyleProvider
075                                   provider)
076         {
077             Color color = SelectCtx.GetPenColor(provider);
078
079             using (Pen pen = new Pen(color, GetPenWidth(provider)))
080                 graphics.DrawLines(pen, PinUtils.PinArrayToScreen((
081                                    IPinPoint[])points.ToArray(
082                                    typeof(IPinPoint))));
083         }
084
085         protected abstract float GetPenWidth(IStyleProvider
086                                              provider);
/*...*/
098     }
099
100     protected abstract class ManipulatingState : ActiveGroup
101     {
/*...*/
108         public override ICommand Manipulating(UserEvent userEvent)
109         {
110             MouseUserEvent mue = userEvent as MouseUserEvent;
111
112             if (mue != null && points.Count > 3 && Closing(mue.
113                 Point))
114             {
115                 Context.ToFinalizingState((IPinPoint[])points.
116                                           ToArray(typeof (IPinPoint))
117                                           );
118                 return null;
119             } else
120                 return base.Manipulating(userEvent);
121         }
/*...*/
131         protected override float GetPenWidth(IStyleProvider
132                                              provider)
133         {
134             return SelectManipulationStartCtx.GetPenWidth(provider);
135         }
```

```
136     }
137
138     protected abstract class FinalizingState : ActiveGroup
139     {
140         public FinalizingState(AreaSelectManipulator context,
141                                IPinPoint[] points)
142             : base(context, points)
143         {
144         }
145
146         public override ICommand Manipulating(UserEvent userEvent)
147         {
148             MouseUserEvent mue = userEvent as MouseUserEvent;
149             if (mue != null && !Closing(mue.Point))
150             {
151                 Context.ToManipulatingState((IPinPoint[])points.
152                                             ToArray(typeof (
153                                             IPinPoint)));
154                 return null;
155             } else
156                 return base.Manipulating(mue);
157         }
/*...*/
167         protected override float GetPenWidth(IStyleProvider
168                                              provider)
169         {
170             return SelectManipulationFinishCtx.GetPenWidth(provider)
171                                                                 ;
172         }
/*...*/
202     }
203 }
```

Автоматный класс `AreaSelectManipulator` перегружает фабрику состояния `ToInactiveState` (строки 15–18), соответствующую состоянию `inactive`, и добавляет новые абстрактные фабрики `ToManipulatingState` (строка 20) и `ToFinalizingState` (строка 22), соответствующие состояниям `manipulating` и `finalizing`. Класс `AreaSelectManipulator` переопределяет класс группы состояний `ActiveGroup`, добавляя в него общее для состояний этой группы поле, содержащее точки границы создаваемой области выделения (строки 69–72);

Также автоматный класс `AreaSelectManipulator` определяет абстрактные классы состояния `ManipulatingState` (строки 100–136) и `FinalizingState` (строки 138–202), соответствующие состояниям `manipulating` и `finalizing`. Метод `Manipulating()` класса состояния `ManipulatingState` реализует переход из состояния

`manipulating` в режим `finalizing` (строки 112–119). В противном случае вызывается метод `Manipulating()` базового класса. Класс состояния `FinalizingState` реализуется аналогичным способом.

Рассмотрим фрагменты реализации автоматного класса `PolygonSelectManipulator`.

```
001 public class PolygonSelectManipulator : AreaSelectManipulator
002 {
003     public PolygonSelectManipulator(ITool tool, IPinBox pinBox,
004                                     IPanel panel)
005         : base(tool, pinBox, panel)
006     {
007     }
008
009     protected override void ToManipulatingState(IPinPoint[] points)
010     {
011         State = new ManipulatingState(this, points);
012     }
013
014     protected override void ToFinalizingState(IPinPoint[] points)
015     {
016         State = new FinalizingState(this, points);
017     }
018
019     protected override ICommand activeGroup(Manipulator.ActiveGroup
020                                             raw, UserEvent
021                                             userEvent)
022     {
023         MouseUserEvent mue = userEvent as MouseUserEvent;
024         ActiveGroup state = raw as ActiveGroup;
025         if (userEvent.Type == UserEvent.EventType.Mouse_LButtonDown
026             && state != null)
027         {
028             state.Points.Add(pinBox.Create(mue.Point));
029             panel.Invalidate(true);
030             return null;
031         } else if (userEvent.Type == MouseUserEvent.EventType.
032                    Mouse_Move && state != null)
033         {
034             state.Points[state.Points.Count - 1] = pinBox.Create(mue.
035                                                    Point);
036             panel.Invalidate(true);
037             return null;
038         } else
039             return base.activeGroup(state, userEvent);
040     }
041
042     protected new class ManipulatingState : AreaSelectManipulator.
043     ManipulatingState
044     {
045         public ManipulatingState(AreaSelectManipulator context,
046                                  IPinPoint[] points)
047             : base(context, points)
048         {
049         }
050
051         public override ICommand Manipulating(UserEvent userEvent)
```

```
052         {
053             if (userEvent.Type == UserEvent.EventType.
054                 Mouse_RButtonUp)
055             {
056                 Context.ToInactiveState();
057                 Context.panel.Invalidate(true);
058                 return null;
059             } else
060                 return base.Manipulating(userEvent);
061         }
/*...*/
070     }
071
072     protected new class FinalizingState : AreaSelectManipulator.
073     FinalizingState
074     {
075         public FinalizingState(AreaSelectManipulator context,
076                                IPinPoint[] points) : base(context,
077                                points)
078         {
079         }
080
081         public override ICommand Manipulating(UserEvent userEvent)
082         {
083             MouseUserEvent mue = userEvent as MouseUserEvent;
084             if (userEvent.Type == UserEvent.EventType.
085                 Mouse_RButtonUp)
086             {
087                 points.Add(Context.pinBox.Create(mue.Point));
088                 Context.ToInactiveState();
089                 return CreateCommand(mue);
090             } else
091                 return base.Manipulating(userEvent);
092         }
/*...*/
101     }
102 }
```

Автоматный класс `PolygonSelectManipulator` перегружает фабрики состояния `ToManipulatingState` (строки 9–12) и `ToFinalizingState` (строки 14 – 17), создавая в них экземпляры классов состояния `ManipulatingState` и `FinalizingState` соответственно. Классы состояния `ManipulatingState` и `FinalizingState` являются потомками одноименных классов состояния автоматного класса `AreaSelectManipulator`. Метод `Manipulating` класса состояния `ManipulatingState` реализует переход из состояния `manipulating` в состояние `inactive` по нажатию правой клавиши мыши (строки 53–59). В противном случае вызывается метод базового класса (строка 60). Класс состояния `FinalizingState` реализуется аналогичным образом.

Автоматный класс `PolygonSelectManipulator` перегружает также групповой метод `activeGroup()`, реализуя петли в группе `active`. В случае нажатия левой кнопки мыши к границе области выделения добавляется еще одна точка (строки 25–31). В случае перемещения мыши изменяется последняя точка границы области выделения (строки 31–38). В противном случае вызывается групповой метод `activeGroup()` автоматного класса `AreaSelectManipulator` (строка 39).

Автоматный класс `LassoSelectManipulator` реализуется аналогичным образом.

5.3. Сравнение методов реализации автоматных объектов

В настоящее время неизвестны способы исчерпывающего и непротиворечивого сравнения методов программирования. Однако широко распространены методы получения метрик исходного кода, которые неявно позволяют получить представление о том, какой метод лучше, а какой хуже.

В данной работе, для получения метрик использованы следующие программные пакеты:

- ***Resource Standard Metrics*** производства фирмы ***M Squared Technologies LLC™*** [75];
- ***SourceMonitor*** производства фирмы ***Campwood Software*** [76].

Из набора метрик, вычисляемых этими инструментами, выбрано подмножество метрик, приведенное в табл. 2. Все выбранные метрики обладают следующим свойством: чем больше значение метрики, тем «хуже» оцениваемая программа.

Таблица 2. Выбранные метрики оценки исходного кода

№	Имя метрики	Инструмент	Описание
1	*S*	***SourceMonitor***	Количество блоков кода, разделенных точкой с запятой
2	*ASM*	***SourceMonitor***	Среднее количество выражений в методе
3	*MMC*	***SourceMonitor***	Максимальная сложность метода [77]
4	*AD*	***SourceMonitor***	Средняя глубина блока
5	*TTC*	***Resource Standard Metrics***	Полная цикломатическая сложность (по методам)
6	*ACC*	***Resource Standard Metrics***	Средняя цикломатическая сложность (по методам)
7	*AIC*	***Resource Standard Metrics***	Средняя сложность интерфейса (по методам)
8	*ARP*	***Resource Standard Metrics***	Среднее количество точек возврата (по методам)

Сравнение методов реализации автоматных объектов выполнено на примере площадных манипуляторов выделения каркаса ***Iris*** (разд. 5.2). В сравнении участвует исходный код, необходимый для реализации следующих манипуляторов выделения:

- *выделение многоугольником*;
- *выделение лассо*.

В случае использования декомпозиции и структурирования логики автоматных объектов с помощью наследования в анализе участвуют *все* базовые классы этих манипуляторов.

Анализируются следующие варианты реализации манипуляторов:

- без явного использования конечных автоматов – поведение манипуляторов реализует *неформально*, в объектно-ориентированном стиле;
- без использования декомпозиции и структурирования логики поведения – поведение манипуляторов реализуется с помощью паттерна проектирования ***State***;
- ***VM***-метод – поведение манипуляторов реализуется на основе ***VM***-метода;
- ***VIC***-метод – поведение манипуляторов реализуется на основе ***VIC***-метода.

Результаты вычисления метрик для этих вариантов реализации манипуляторов приведены в табл. 3.

Таблица 3. Метрики для разных вариантов реализации манипуляторов

	NoFsm	***NoInh***	***VIC***	***VM***
S	134	177	163	163
ASM	6.71	3.96	1.86	4.96
MMC	8	8	3	4
AD	1.96	2.54	2.23	2.04
TTC	36	48	51	45
ACC	3.60	2.00	1.38	1.96
AIC	4.20	3.13	2.76	3.09
ARP	2.80	1.58	1.30	1.78

Как следует из приведенной таблицы, методы, предложенные в данной работе, *превосходят* (по выбранным метрикам) неформальный метод реализации автоматных объектов и метод реализации на основе конечных автоматов без использования наследования. ***VIC***-метод превосходит ***VM***-метод по пяти метрикам из восьми.

Выводы

Практическая ценность предложенных методов проектирования и реализации автоматных объектов в области телекоммуникационных систем подтверждается использованием этих методов при разработке программного обеспечения в компании *Транзас*.

На основе предложенных в настоящей работе методов выполнено проектирование и реализация:

- автоматных объектов эмулятора систем ***Inmarsat-C*** и ***Inmarsat-D***$^{+}$;
- манипуляторов интерактивного редактирования каркаса ***Iris***.

Для проектирования автоматных объектов использована графическая нотация, предложенная в настоящей работе. Для реализации автоматных объектов использован метод реализации автоматных объектов на основе виртуальных вложенных классов (***VIC***-метод). Обоснованность использования ***VIC***-метода при реализации автоматных объектов подтверждает сравнительным анализом получаемого исходного кода и исходного кода получаемого на основе других подходов к реализации автоматных объектов.

В настоящее время редактирование пользовательских объектов в системе мониторинга ***Navi-Manager*** реализовано на основе каркаса ***Iris***. Поведение манипуляторов, предопределенных в каркасе ***Iris***, расширено с учетом особенностей системы ***Navi-Manager***. Это позволяет сделать вывод о практической ценности и достоверности полученных результатов в плане декомпозиции и структурирования логики автоматных объектов с помощью наследования.

Отметим, что некоторые другие результаты, полученные в работе, также внедрены в практику. Библиотека ***STOOL*** используется в учебном процессе на кафедре «Компьютерные технологии» СПбГУ ИТМО при чтении лекций по курсу «Теория автоматов в программировании».

Заключение

В диссертации получены следующие научные результаты:

1. Предложен метод реализации автоматных систем на основе библиотеки ***STOOL***, устраняющий ряд недостатков ***SWITCH***-технологии.
2. Разработана графическая нотация, которая позволяет описывать расширение и структурирование логики автоматных объектов с помощью наследования.
3. Разработан метод реализации автоматных объектов на основе виртуальных методов, обеспечивающий декомпозицию и структурирование логики автоматных объектов с помощью наследования.
4. Разработан метод реализации автоматных объектов на основе виртуальных вложенных классов, обеспечивающий декомпозицию и структурирование логики автоматных объектов с помощью наследования.

Результаты работы внедрены при разработке системы мониторинга мобильных объектов ***Navi-Manager*** и каркаса для построения редакторов пространственных данных ***Iris*** в компании *Transas*, а также в учебном процессе на кафедре «Компьютерные технологии» СПбГУ ИТМО при чтении лекций по курсу «Теория автоматов в программировании». Акты внедрения прилагаются.

Все основные результаты, полученные в работе, опубликованы. Приведем информацию о личном участии автора.

В работах [37, 54, 78] автором предложена идея усовершенствования методов реализации автоматных систем в ***SWITCH***-технологии. Основные идеи и принципы, предложенные в этих работах, реализованы в библиотеке ***STOOL***.

В работе [16] автором, впервые на русском языке, выполнен обзор синхронного программирования вообще и языков ***Esterel***, ***Lustre*** и ***Argos*** в частности.

В работе [65] автором предложен метод реализации автоматных объектов на основе виртуальных методов.

В работах [66, 67] автором предложен метод реализации автоматных объектов на основе виртуальных вложенных классов.

Синтаксис графической нотации для проектирования автоматных объектов был предложен автором в работах [65, 66].

Литература

1. **Harel D., Pnueli A.** On the development of reactive systems //In "Logic and Models of Concurrent Systems". NATO Advanced Study Institute on Logic and Models for Verification and Specification of Concurrent Systems. Springer Verlag, 1985. pp. 477–498.
2. **Specification and Description Language** (SDL), International Engineering Consortium, http://www.iec.org/acrobat.asp?filecode=125
3. **Буч Г., Рамбо Дж., Джекобсон А.**. UML. Руководство пользователя. М. ДМК 2000. – 432 с.
4. **Benveniste A.** The Synchronous Languages 12 Years Later. Proceedings of the IEEE, vol. 91, 2003, no. 1, pp. 64-83.
5. **André C.** SyncCharts: A Visual Representation of Reactive Behaviors /Tech. Rep. RR 95-52, I3S, Sophia-Antipolis, France, 1995.
6. **Шалыто А.А.** SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998. – 628 с.
7. **Гольдштейн Б. С.** Сигнализация в сетях связи. Том 1. М.: Радио с связь, 2001. – 439 с.
8. **Specification and Description Language** (SDL), рекомендации ITU-T серии Z.100, http://www.itu.int/rec/recommendation.asp?type=folders&lang=e&parent=T-REC-Z.100
9. **Encontre V.** SDL: a standard language for Ada real-time applications /Proceedings of the conference on TRI-Ada '91, pp.45–53, October 21-25, 1991, San Jose, California, United States.
10. **Harel D.** Statecharts: A visual formalism for complex systems //Sci. Comput. Program. 1987. Vol. 8, pp. 231–274.

11. **Automata Studies** / Shannon C.E., McCarthy J. Princeton University Press, 1956.

12. **Harel D., Naamad A.** The Statemate Semantics of Statecharts // ACM Trans. Softw. Eng. Methodology, vol. 5, October 1996. pp. 293–333.

13. **Mikk E., Lakhnech Y., Petersohn C., Siegel M.** On Formal Semantics of Statecharts as Supported by STATEMATE // Proceedings of the 2nd BCS-FACS Northern Formal Methods Workshop, Ilkley, 14-15 July 1997.

14. **Хопкрофт Д., Мотвани Р., Ульман Дж.** Введение в теорию автоматов, языков и вычислений. М.: Вильямс, 2001. – 528 с.

15. **OMG Unified Modeling Language Specification**, Version 1.5, March 2003.

16. **Шалыто А.А., Шопырин Д.Г.** Синхронное программирование //Информационно-управляющие системы. 2004. № 3, с. 35–42.

17. **Апериодические** автоматы /Астановский А.Г., Варшавский В.И., Мараховский В.Б. и др. М.: Наука, 1976. – 423 с.

18. **Palshikar G.K.** An Introduction to Esterel //Embedded Systems Programming. 2001. http://www.embedded.com/story/OEG20011018S0090

19. **IEEE** Standard VHDL Language Reference Manual. IEEE Press. Piscataway, NJ, 1994, pp. 1076–1993.

20. **Caspi P., Pilaud D., Halbwachs N., Plaice J. A.** LUSTRE: A declarative language for programming synchronous systems. In ACM Symp. Principles Program. Lang. (POPL), Munich, Germany, 1987, pp. 178–188.

21. **Berry G., Gonthier G.** The Esterel synchronous programming language: Design, semantics, implementation //Sci. Comput. Program., vol. 19, Nov., 1992. pp. 87–152.

22. **Benveniste A., Guemic P.** Hybrid dynamical systems theory and the SIGNAL language //IEEE Trans. Automat. Contr., vol. AC-35, May 1990. pp. 535–546.

23. **Maraninchi F.** The Argos language: Graphical representation of automata and description of reactive systems /Presented at the IEEE Workshop Visual Lang., Kobe, Japan, 1991.

24. **André C.** Representation and Analysis of Reactive Behaviors: A Synchronous Approach /CESA'96, Lille, France, IEEE-SMC, July 1996.

25. **Rapicault P., Mallet F.** Behavioral specification of Java component using SyncCharts /ECOOP2000. Workshop on Pervasive Component Systems, June 2000, Cannes, France.

26. **Berry G.** Preemption in Concurent Systems //Proceedings of FSTTCS 93. Springer Verlag, LNCS 761, 1993. pp. 72–93.

27. **Туккель Н.И., Шалыто А.А.** Система управления танком для игры Robocode. Объектно-ориентированное программирование с явным выделением состояний. http://is.ifmo.ru (раздел «Проекты»).

28. **Шалыто А.А., Туккель Н.И.** Программирование с явным выделением состояний // Мир ПК. 2001. № 8, 9, с. 116–121, с. 132–138.

29. **Шалыто А.А., Туккель Н.И.** Реализация автоматов при программировании событийных систем. //Программист. 2002. № 4, с. 74–80.

30. **Шалыто А.А.** Технология автоматного программирования //Мир ПК. 2003. № 10, с. 74–78.

31. **Шалыто А.А.** Программная реализация управляющих автоматов // Судостроительная промышленность. Серия «Автоматика и телемеханика». 1991. № 13, с. 41–42.

32. **Шалыто А.А.** Автоматное проектирование программ. Алгоритмизация и программирование задач логического управления //Известия РАН. Теория и системы управления. 2000. № 6, с. 63–81.

33. **Шалыто А.А., Туккель Н.И.** SWITCH-технология – автоматный подход к созданию программного обеспечения «реактивных» систем //Программирование. 2001. № 5, с. 45–62.

34. **Шалыто А.А.** Алгоритмизация и программирование для систем логического управления и «реактивных» систем //Автоматика и телемеханика. 2001. № 1, с. 3–39.

35. **Шалыто А.А., Туккель Н.И.** Проектирование программного обеспечения системы управления дизель-генераторами на основе автоматного подхода // Системы управления и обработки информации: СПб.: ФГУП "НПО "Аврора". 2002. № 5, с. 66–82.

36. **Гуров В.С., Мазин М.А., Нарвский А.С., Шалыто А.А.** UML. SWITCH-технология. Eclipse //Информационно-управляющие системы. 2004. № 6, с. 12–17.

37. **Шопырин Д.Г., Шалыто А.А.** Объектно-ориентированный подход к автоматному программированию // Информационно-управляющие системы, 2003, № 5, с. 29–39.

38. **Adamczyk P.** The Anthology of the Finite State Machine Design Patterns // The 10th Conference on Pattern Languages of Programs, 2003.

39. **Palfinger G.** State Action Mapper, // PLoP 1997, Writer's Workshop. http://jerry.cs.uiuc.edu/~plop/plop97/Proceedings.ps/palfinger.ps

40. **van Gurp, J., Bosch J.** On the Implementation of Finite State Machines // Proceedings of the IASTED International Conference, 1999. http://www.xs4all.nl/~jgurp/homepage/publications/fsm-sea99.pdf

41. **Jones D. W.** How (not) to code a finite state machine // *SIGPLAN Not.* 1988. vol. 23, num. 8, pp. 19–22.

42. **Cargill T.** C++ Programming Style. Addison Wesley, 1992.

43. **Samek M.** Practical Statecharts in C/C++. CMP Books, 2002.

44. **Шалыто А.А., Наумов Л.А.** Методы объектно-ориентированной реализации реактивных агентов на основе конечных автоматов // Искусственный интеллект. 2004. № 4, с. 756–762.

45. **Adamczyk P.** Selected Patterns for Implementing Finite State Machines /The 11th Conference on Pattern Languages of Programs, 2004.

46. **Odrowski J., Sogaard P.** Pattern Integration - Variations of State /Proceedings of PLoP96. http://www.cs.wustl.edu/~schmidt/PLoP-96/odrowski.ps.gz.

47. **Henney K.** Methods for states. A pattern for realizing object lifecycles //Proceedings of the First Nordic Conference on Pattern Languages of Programs, 2002, September.

48. **Шалыто А. А., Туккель Н. И.** От тьюрингова программирования к автоматному //Мир ПК. 2002. № 2, с. 144–149.

49. **Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.** Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб.: Питер, 2001. – 368 с.

50. **Шамгунов Н.Н., Корнеев Г.А., Шалыто А.А.** State Machine - новый паттерн объектно-ориентированного проектирования //Информационно-управляющие системы. 2004. № 5, с. 13–25.

51. **Sane A., Campbell R.** Object-Oriented State Machines: Subclassing, Composition, Delegation, and Genericity /OOPSLA '95. http://choices.cs.uiuc.edu/sane/home.html.

52. **Страуструп Б.** Язык программирования C++. 3-е издание. СПб.: Невский диалект, 1999. – 991 с.

53. **Шалыто А.А., Туккель Н.И.** Танки и автоматы. //BYTE/Россия. 2003. №2, с. 69–73.

54. **Шопырин Д.Г., Шалыто А.А.** Применение класса «STATE» в объектно-ориентированном программировании с явным выделением состояний //Труды X Всероссийской научно-методической конференции "Телематика-2003". СПб.: СПбГИТМО (ТУ). 2003, т. 1, с. 284, 285.

55. **Boost C++** library (http://www.boost.org).

56. **Danforth S., Tomlinson C.** Type theories and object-oriented programming //ACM Comput. Surv. vol. 20, № 1, 1988. pp. 29–72.

57. **Taivalsaari A.** On the notion of inheritance //*ACM Comput. Surv.* vol. 28, № 3, 1996, pp. 438–479.

58. **Bracha G., Cook W.** Mixin-based inheritance //In OOPSLA/ECOOP'90 Conference Proceedings, ACM SIGPLAN Not. vol. 25, № 10, 1990. pp. 303–311.

59. **Cook W.** A denotational semantics of inheritance. //Ph.D. thesis, Brown University, Tech. Rep. CS-89-33, 1989.

60. **Wegner P., Zdonik S.** Inheritance as an incremental modification mechanism or what Like is and isn't like. //In ECOOP'88: European Conference on Object-Oriented Programming. Lecture Notes in Computer Sci. 276, Springer-Verlag. 1988, pp. 55–77.

61. **Lee J., Xue N., Kuei T.** A note on state modeling through inheritance // *SIGSOFT Softw. Eng. Notes* vol. 23, 1998, № 1.

62. **Liskov B.** Data Abstraction and Hierarchy // SIGPLAN Notices. Vol. 23. № 5, 1988, pp. 17–34.

63. **Liskov, B., Wing J.** A Behavioral Notion of Subtyping // ACM Transactions on Programmi ng Languages and Systems, Vol. 16, № 6. 1994, pp. 1811–1841.

64. **Заякин Е.А., Шалыто А.А.** Метод устранения повторных фрагментов кода при реализации конечных автоматов // Мир ПК (диск). 2005. № 8. http://is.ifmo.ru, раздел «Проекты».

65. **Шопырин Д.Г.** Объектно-ориентированная реализация конечных автоматов на основе виртуальных методов // Информационно-управляющие системы. 2005, № 3, с. 36–40.

66. **Шопырин Д.Г.** Метод проектирования и реализации конечных автоматов на основе виртуальных вложенных классов //Информационные технологии моделирования и управления. 2005, № 1(19) с. 87–97.

67. **Шопырин Д.Г.** Программирование с явным выделением состояний на платформе .Net //Труды XII Всероссийской научно-методической конференции "Телематика-2005". СПб.: СПбГИТМО (ТУ). 2005, т. 1, с. 86, 87.

68. **Лутц М.** Программирование на Python. СПб.: Символ-Плюс, 2002. – 1136 с.

69. **Группа компаний Транзас**, http://transas.ru/

70. **Inmarsat Global Ltd**, http://www.inmarsat.com/

71. **Xantic** Inmarsat-C Data Reporting and Polling User Manual, http://www.xantic.net/default.asp?FILE=items/3189/108&rnd=2.396339E-02

72. **Vlissides J. M., Linton M. A.** Unidraw: a framework for building domain-specific graphical editors //*ACM Trans. Inf. Syst.* 1990. vol. 8, num. 3, pp. 237–268.

73. **Krasner G. E., Pope S. T.** A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80 // Object-Oriented Program. 1, 3 (Aug./Sept. 1988), pp. 26–49.

74. **Vlissides, J.M.** Generalized graphical object editing. Ph.D. dissertation, Stanford Univ., 1990.

75. **M Squared Technologies LLC™**, http://msquaredtechnologies.com/

76. **Campwood Software**, http://campwoodsw.com/

77. **Макконнелл С.** Совершенный код. М.: Русская редакция, 2005. – 867с.

78. **Шопырин Д.Г.** Разработка промежуточного языка представления конечных автоматов //Труды XI Всероссийской научно-методической конференции "Телематика-2004". СПб.: СПбГИТМО (ТУ). 2004, т. 1, с. 195–197.