МИНИСТЕРСТВО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ

Факультет **Информационных технологий и программирования**

Направление **Прикладная математика и информатика** Специализация : **Математическое и программное обеспечение вычислительных машин**

Академическая степень **магистр математики**

Кафедра **Компьютерных технологий** Группа **6538**

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

на тему

Автоматный подход к реализации элементов графического пользовательского интерфейса

Автор магистерской диссертации Корниенко А.А. (подпись)
(Фамилия, И., О.)

Научный руководитель Казаков М.А. (подпись)
(Фамилия, И., О.)

Руководитель магистерской программы (подпись)
(Фамилия, И., О.)
:

К з а щ и т е д о п у с т и т ь
Зав. кафедрой ВАСИЛЬЕВ В.Н. (подпись)
(Фамилия, И., О.)

“ 10 ” июня 2006г.

Санкт-Петербург, 2006 г.

ОГЛАВЛЕНИЕ

1. ВВЕДЕНИЕ

Одной из важных функций программного обеспечения является взаимодействие с пользователем – предоставление пользовательского интерфейса. Наиболее часто применяемым пользовательским интерфейсом является графический оконный интерфейс. Существует множество готовых библиотек для реализации графического пользовательского интерфейса приложений. Однако эти библиотеки обычно в высокой степени зависят от среды исполнения и операционной системы в частности. Кроме того, логика работы элементов пользовательского интерфейса может существенно различаться, что затрудняет написание переносимых приложений с графическим пользовательским интерфейсом.

Также, в ряде приложений (например, во многих игровых и мультимедийных программах) использование для реализации пользовательского интерфейса стандартных графических компонентов операционной системы (ОС) невозможно. Поэтому реализовать в таком приложении даже простой пользовательский интерфейс штатными средствами ОС либо с использованием стандартных библиотек не представляется возможным. В подобной ситуации единственным выходом является разработка и реализация собственной библиотеки пользовательского интерфейса, которая не будет использовать стандартные графические компоненты ОС. Однако традиционный подход к

разработке логики пользовательского интерфейса, основанный на использовании флагов, обладает существенным недостатком, состоящим в децентрализованности логики работы, что приводит к трудно понимаемому коду, а, следственно, появлению большого количества ошибок. Также, в существующих реализациях, созданных на основе традиционного подхода, обычно сложно отделить код, отвечающий за поведение системы от низкоуровневого системно-зависимого кода, что усложняет перенос реализации на другие платформы. Таким образом, весьма актуальна задача разработки эффективного подхода к реализации различных элементов пользовательского интерфейса, лишенного описанного недостатка.

В настоящей работе предлагается общий подход к построению элементов пользовательского интерфейса произвольной сложности с использованием системы взаимодействующих конечных автоматов. Основным достоинством автоматного подхода является централизация логики работы элементов управления, что позволяет создавать реализации с более четкой и понятной структурой, а также отделять поведение от системно-зависимого кода. Рассматриваемый подход не накладывает никаких ограничений на используемые средства ввода данных от пользователя и графическую подсистему, что делает предлагаемый подход платформо-независимым.

2. АНАЛИЗ ЗАДАЧИ И СУЩЕСТВУЮЩИХ ПОДХОДОВ

2.1. Обзор предметной области

Графический пользовательский интерфейс приложения обычно представляет собой несколько элементов управления, организованных в окна, диалоги, панели, группы и т.д. Каждый элемент управления может отображать какую-либо информацию:

- текстовую (статические текстовые метки, поля ввода, заголовки окон, подписи на кнопках и флажках и т.д.);
- графическую (графические изображения и пиктограммы на кнопках, различные индикаторы и т.д.);
- числовую (поля числового ввода, ползунки, различные индикаторы прогресса);
- бинарную (флажки, индикаторы, радио-кнопки);

и реагировать на действия пользователя, произведенные с помощью устройств ввода, таких как:

- клавиатура;
- мышь;
- сенсорный экран
- и другие.

Каждый элемент принадлежит одному из небольшого количества типов. Пример окон с различными элементами графического пользовательского интерфейса приведен на рис. 1.

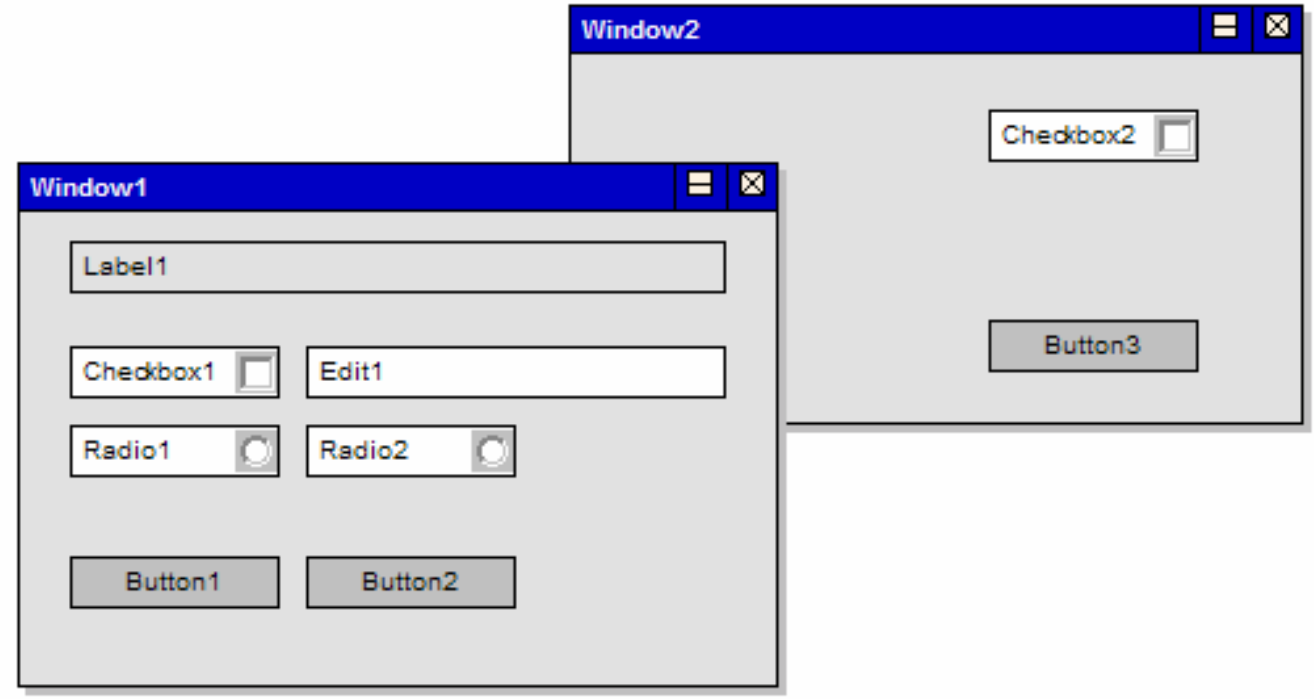

Рис. 1. Примеры окон с различными элементами пользовательского интерфейса

Поведение элементов управления каждого типа обычно четко определено и пользователь (на основе личного опыта или интуиции) имеет представление о том, как каждый конкретный элемент управления может реагировать на различные действия (например, неотмеченный флажок при нажатии станет отмеченным, и наоборот). Организация интерфейса в виде набора элементов нескольких типов, имеющих свой характерный вид и поведение, помогает пользователю эффективно и быстро выполнять действия, и позволяет сделать

работу с данным интерфейсом более комфортной для пользователя, что также важно [1].

Одной из основных характеристик элемента пользовательского интерфейса является его состояние. В каждом состоянии элемент управления обычно отображается и реагирует на события пользовательского ввода особым образом. Например, многие элементы интерфейса могут быть запрещены. Обычно, при этом они отображаются серым цветом и не реагируют на ввод пользователя. Или, например, элемент может быть активным, при этом он отображается с более яркой рамкой и ввод с клавиатуры передается в первую очередь этому элементу. У элемента управления "кнопка" в ОС *Windows* есть 6 состояний (рис. 2):

- нейтральное;
- нажатое;
- нейтральное с установленным фокусом ввода;
- состояние кнопки по умолчанию;
- кнопка по умолчанию с установленным фокусом ввода;
- заблокированное состояние.

Рис. 2. Состояния кнопки в графическом интерфейсе *ОС Windows*

Концепция состояния легка для понимания, поскольку пользователь уже после недолгого времени работы с интерфейсом четко ассоциирует внешний вид элемента управления с его состоянием, а состояние дает информацию о данных,

которые элемент управления отображает и о наборе возможных действий по отношению к элементу.

Поэтому концепция состояний позволяет сделать работу с пользовательским интерфейсом более удобной для пользователей и предоставляет разработчикам возможность создавать приложения с легким для использования интерфейсом.

Для описания пользовательского интерфейса удобно использовать объектно-ориентированный подход. Представляя каждый элемент пользовательского интерфейса в виде объекта, соединяющего в себе ассоциированные с элементом данные и его поведение, можно удобно описать отношения обобщения (наследования) и принадлежности между разными элементами, а также взаимодействие между ними путем передачи сообщений. В том или ином виде объектно-ориентированное программирование используется во многих системах, реализующих пользовательский интерфейс. В некоторых из них (например, ОС *Windows*) объектно-ориентированный подход реализован в неявном виде на языках, не имеющих поддержки ООП, что, тем не менее, не мешает использовать все его преимущества.

Таким образом, говоря о реализации пользовательского интерфейса, обычно подразумевают систему классов, реализующих различные элементы интерфейса, способ взаимодействия между этими классами, способ получения этой системой входных воздействий от устройств ввода (обычно, с помощью соответствующих средств операционной системы), способ отображения информации (также с

помощью средств ОС) и способ взаимодействия с программным кодом, использующим данную систему классов.

Взаимодействие с пользовательским кодом – одна из основных функций системы элементов пользовательского интерфейса. Набор элементов пользовательского интерфейса должен предоставлять клиентскому приложению возможность создавать свой интерфейс произвольной сложности, отображать данные о своем состоянии и получать от пользователя команды и данные. Таким образом, система пользовательского интерфейса должна предоставлять возможность создавать отдельные элементы, объединять их в иерархическую структуру (например, окно → панель → кнопка), устанавливать и читать данные, ассоциированные с каждым элементом (например, текст, введенный пользователем в поле ввода), изменять и получать состояние элементов, а также получать уведомления о действиях пользователя, произведенных с каждым элементом.

2.2. Существующие реализации и их недостатки

Практически все существующие в настоящий момент реализации пользовательского интерфейса используют для кодирования состояний элементов интерфейса флаги, то есть разбивают состояние на несколько логических признаков. При этом логика работы элементов получается "размазанной" по коду

библиотеки и порой очень тесно переплетается с функциональностью, не имеющей к поведению элемента никакого отношения.

Проиллюстрируем это утверждение на примере стандартной библиотеки пользовательского интерфейса платформы *Java 2 SE — AWT* (Abstract Window Toolkit). *AWT* является одной из немногих многоплатформенных библиотек, реализующих компоненты пользовательского интерфейса. За более чем 10 лет ее существования, использования во множестве приложений и постоянного развития *AWT* многократно изменялась с учетом найденных ошибок проектирования и реализации. На ее базе была также построена библиотека *Swing*, обладающая более широкими возможностями по настройке визуального представления элементов пользовательского интерфейса. В данный момент *AWT* можно считать примером одной из лучших реализаций классического подхода к пользовательскому интерфейсу. Недостатки *AWT* во многом обуславливаются самим подходом и присущи практически всем другим современным реализациям.

Рассмотрим, каким образом в *AWT* реализуются состояния компонентов интерфейса. В этой библиотеке базовым классом для всех элементов пользовательского интерфейса является `java.awt.Component`. Для кодирования состояния элементов используются несколько булевских полей класса:

```
boolean visible;
boolean enabled;
```

```
boolean valid;
boolean focusable;
```

и несколько неявно заданных значений, которые можно получить методами

```
boolean isFocusOwner();
boolean isShowing();
```

Поле `visible` используется в восьми методах класса `java.awt.Component` и в пяти классах наследниках `java.awt.Component` в явном виде и в гораздо большем количестве мест через методы `isVisible` и `setVisible`. Аналогично обстоят дела с другими переменными, кодирующими состояние. Для выделения состояния элемента необходимо писать код примерно такого вида (*JRE 1.5.0_03*, `java\awt\Component.java`, строка 6604) :

```
if (!(toTest.isDisplayable()
    && toTest.isVisible()
    && (toTest.isEnabled()
        || toTest.isLightweight())))
{…}
```

Или, например, часть кода отображения кнопки в библиотеке *Swing* (*JRE 1.5.0_03*, `com\sun\java\swing\plaf\windows\WindowsButtonUI.java`):

```
if (model.isArmed()
        && model.isPressed() || model.isSelected()) {
    index = 2;
} else if (!model.isEnabled()) {
    index = 3;
} else if (model.isRollover() || model.isPressed()) {
    index = 1;
} else if (b instanceof JButton
```

```
                && ((JButton)b).isDefaultButton()) {
        index = 4;
    } else if (c.hasFocus()) {
        index = 1;
    }
```

Вторая проблема состоит в том, что в таких реализациях отделить логику работы от системно-зависимого кода очень сложно и, как следствие, большинство имеющихся в настоящий момент реализаций пользовательского интерфейса жестко привязаны к операционной системе или среде исполнения. В случае *AWT* переносимость обеспечивается частично за счет самой платформы *Java 2 SE*, частично за счет того, что логика работы конкретных элементов управления реализована для каждой операционной системы отдельно. Для *Windows* реализация основной части *AWT* написана на языке *C* и находится в динамической библиотеке `awt.dll`. При этом, например, логика работы элемента управления «Флажок» (`Checkbox`) разбита на несколько классов: `java.awt.Checkbox`, `sun.awt.WComponentPeer` и `sun.awt.WCheckboxPeer`. Два последних из них реализованы большей частью в указанной выше библиотеке `awt.dll`. Таким образом, в *AWT* логика поведения не только децентрализована, но и написана на разных языках.

Другим примером может служить оконная библиотека платформы *.Net*: *Windows Forms*. Базовый класс всех элементов пользовательского интерфейса в ней — `System.Windows.Forms.Control` — содержит поля

```
Int32 state;
Int32 state2;
```

Но эти поля являются всего лишь упакованной формой для нескольких флагов и используются так же, как если бы были просто набором булевых переменных. Например, для определения, является ли элемент управления видимым, используется свойство `Visible`, метод `get` которого в свою очередь вызывает метод `GetVisibleCore`:

```
public bool Visible
{
    get
    {
        return GetVisibleCore();
    }
    set
    {
        SetVisibleCore(value);
    }
}
```

Метод `GetVisibleCore` делает вызов метода `GetState(2)`, где цифра *2* – это битовая маска признака `visible`:

```
internal bool GetState(int flag)
{
    return ((this.state & flag) != 0);
}
```

Кроме того, библиотека *Windows Forms* опирается на графическую подсистему операционной системы *Windows*. Каждому элементу управления

соответствует объект «окно» операционной системы, доступный только через дескриптор. Это ведет к тому, что признаки `visible`, `enabled` и т.д. продублированы как в виде указанных выше флагов, так и во внутренних структурах ОС *Windows*, а логика работы рассредоточена по коду библиотеки *Windows Forms* и частично находится в системных библиотеках операционной системы.

Стоит также заметить, что есть успешные попытки создания переносимой альтернативы *.Net Framework* — *Mono* и *Portable .Net*. И среди небольшого количества нереализованных в них частей стандартной библиотеки — *Windows Forms*. Одной из главных причин этого является запутанная логика работы пользовательского интерфейса и сложность отделения переносимого кода, отвечающего за поведение элементов пользовательского интерфейса от кода, специфичного для операционной системы *MS Windows*.

Следующий пример — графический интерфейс операционной системы *MS Windows*. В нем логика работы элементов пользовательского интерфейса перемешивается даже с кодом, не имеющим отношения к графике и пользовательскому интерфейсу вообще (системные уведомления, межпроцессное взаимодействие, управление жизненным циклом приложения и т.п.).

Еще две широко распространенные реализации пользовательского интерфейса – *MFC* и *VCL* обладают описанными недостатками в еще большей степени, поскольку с самого начала не создавались с расчетом на переносимость и

задачи отделения логической составляющей от системно-зависимого кода при их создании не ставилось.

Также следует отметить, что во многих реализациях пользовательского интерфейса различается поведение одинаковых элементов управления. Например, в большинстве реализаций элемент управления (кнопка, флажок, радио-кнопка) считается нажатым, если пользователь отпускает кнопку мыши, курсор мыши находится на элементе управления и находился на нем тогда, когда пользователь нажимал кнопку мыши. Однако в некоторых реализациях элемент управления считается нажатым, если пользователь нажал кнопку мыши в момент, когда курсор находился на этом элементе. Различия в поведении одинаковых элементов управления уменьшает эффективность работы с интерфейсом, субъективное удовлетворение пользователя, а также увеличивает время обучения и количество ошибок пользователя.

Из приведенных примеров можно сделать вывод, что серьезной проблемой существующего подхода к реализации пользовательского интерфейса является децентрализованность логики и связанная с этим сложность отделения поведения системы от реализации ввода данных, графического вывода и другого платформо-зависимого кода. С децентрализованностью логики связана проблема переносимости кода библиотек, реализующих пользовательский интерфейс.

И отдельной проблемой является то, что внешнее представление элементов управления не соответствует внутренней структуре логики их работы. Когда пользователь четко видит одно состояние элемента, в программе одновременно

существует лишь набор логических признаков. Это является трудностью в основном для разработчиков, вынужденных работать с запутанным кодом, не отражающим логическую суть поведения компонентов пользовательского интерфейса.

Для решения этих проблем разработки компонентов пользовательского интерфейса целесообразно использовать программирование с явным выделением состояний, также называемое автоматным программированием. В работе [2] было предложено проектировать программы для систем логического управления на основе использования конечных автоматов. Такой способ построения программ был назван «Switch-технология» или «автоматное программирование». Автоматное программирование также описано в работах [3 – 5]. Данный подход позволяет централизовать логику работы системы и отделить поведение системы от функций сопряжения с низкоуровневым кодом. Автоматный подход также подразумевает изначальную документированность поведения системы, что в свою очередь переносит многие логические ошибки на стадию проектирования, чем способствует уменьшению их количества.

3. ОПИСАНИЕ АВТОМАТНОГО ПОДХОДА К РЕАЛИЗАЦИИ ПОЛЬЗОВАТЕЛЬСКОГО ИНТЕРФЕЙСА

В предыдущей главе было обосновано применение автоматного подхода к реализации элементов пользовательского интерфейса. В рамках данного подхода логику работы системы пользовательского интерфейса можно описать в виде набора взаимодействующих автоматов.

Диаграммы состояний автоматов, описывающие логику работы системы, могут быть реализованы на любом языке способом, который не зависит от среды исполнения. При таком способе реализации логики работы можно использовать естественное для элементов пользовательского интерфейса определение понятия состояния. Обычно, отображаемое состояние будет соответствовать состоянию автомата, реализующего данный элемент управления. Это облегчает разработку и отладку программной реализации и позволяет сделать код изоморфным логической структуре программы. Кроме того, в любой реализации предлагаемого метода поведение одних и тех же элементов управления будет совершенно одинаковым, что немаловажно для обеспечения интуитивности интерфейса и удобства его использования. А с учетом отсутствия каких-либо явных привязок логики к системе отображения, становится возможным реализовать множество различных графических представлений одних и тех же элементов, что может быть полезно при разработке игр и программ с оригинальным интерфейсом.

Для каждой операционной системы или отдельной реализации системы будет отличаться слой сопряжения, обеспечивающий получение входных сообщений для системы автоматов и выполнение выходных воздействий. В дальнейшем при описании будет использоваться терминология языка *C++*, реализованного для многих платформ. Для реализации коллекций и строк предлагается использовать стандартную библиотеку шаблонов языка *C++* (*STL*), также имеющую множество реализаций. Однако стоит повторить, что предложенный метод может быть использован для реализации пользовательского интерфейса на любом современном языке и для любой среды исполнения, представляющей возможности ввода данных от пользователя и графического отображения.

3.1. Связь с объектно-ориентированным программированием

Как отмечалось выше, для описания элементов пользовательского интерфейса удобно и естественно использовать объектный подход. При этом имеет смысл выделить базовый класс для всех элементов пользовательского интерфейса. При разработке автоматного подхода к реализации элементов пользовательского интерфейса мы также будем использовать принципы объектно-ориентированного программирования.

В целях демонстрации предлагаемого подхода будет спроектирована система классов, реализующих несколько широко применяемых элементов пользовательского интерфейса и средства их взаимодействия с внешним относительно системы автоматов миром. Поскольку, целью данного примера является демонстрация автоматного подхода, то мы постараемся ограничиться базовой функциональностью элементов управления. В частности, список обрабатываемых событий пользовательского ввода в нашей системе будет включать в себя всего 4 события: передвижение курсора мыши (`EMMove`), нажатие кнопки мыши (`EMDown`), отпускание кнопки мыши (`EMUp`) и ввод символа с клавиатуры (`EChar`).

Диаграмма классов предлагаемой системы элементов пользовательского интерфейса показана на рис. 3.

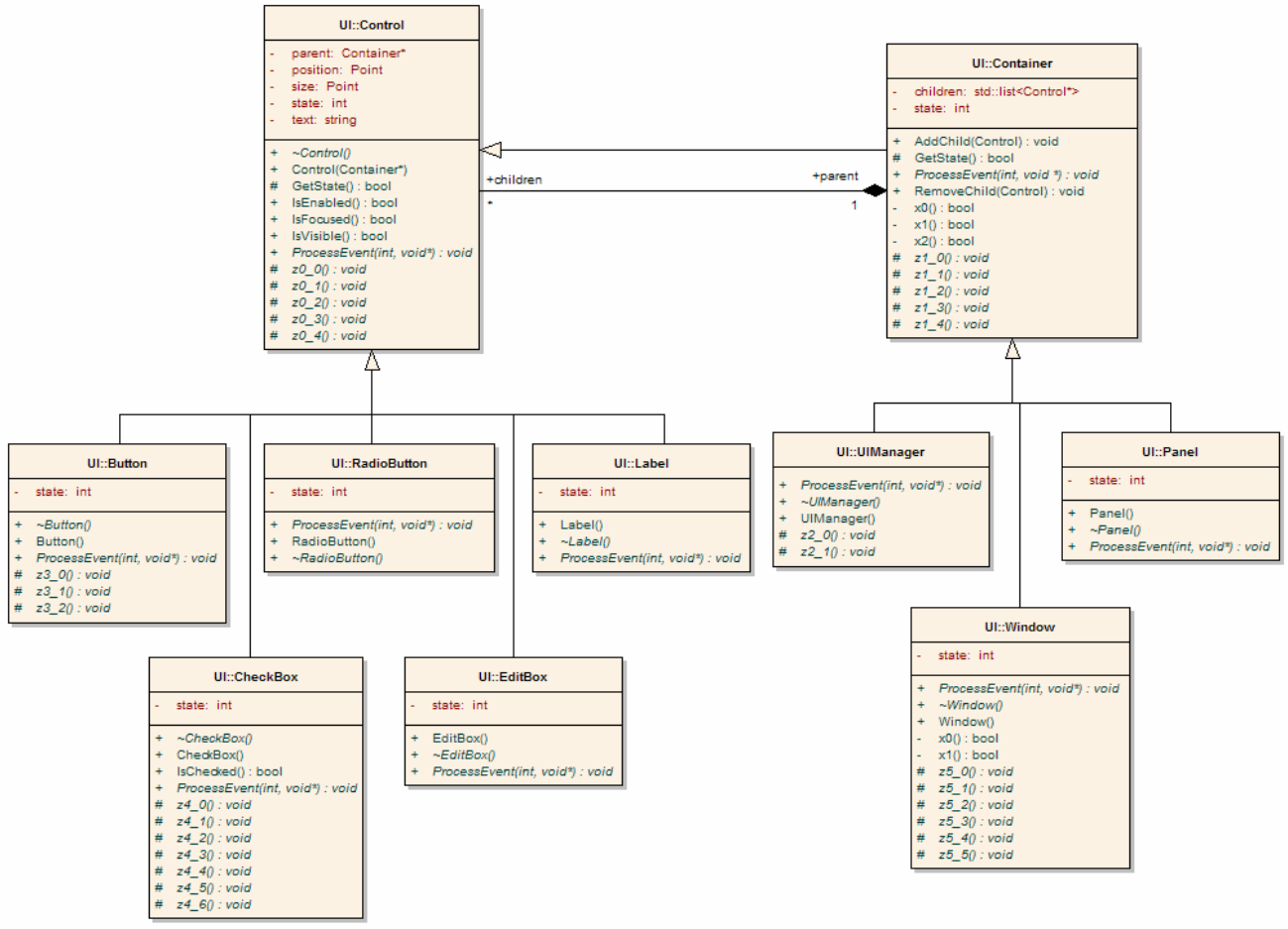

Рис. 3. Диаграмма классов системы элементов пользовательского интерфейса

Каждый элемент пользовательского интерфейса реализован в виде класса. Классы элементов пользовательского интерфейса разделяются на элементы управления (`Control`) и контейнеры (`Container`), которые соответственно наследуются от классов `UI::Control` и `UI::Container`. Класс `UI::UIManager` реализует слой сопряжения системы элементов пользовательского интерфейса со средствами среды исполнения и содержит в себе элементы интерфейса верхнего уровня, например, окна (`UI::Window`).

Каждый класс, представленный на диаграмме, содержит конечный автомат. Обработка событий в автомате производится в методе

`ProcessEvent(int event, void *arg)`, принимающем в аргументах идентификатор события и указатель на связанные с ним дополнительные данные, тип которых зависит от события. Будем называть эти дополнительные данные аргументами события.

Каждый класс в иерархии наследования является отдельным автоматом, имеет свое собственное состояние и свой собственный граф переходов. Метод `ProcessEvent` является виртуальным и реализован во всех классах. Поэтому событие, в первую очередь, обрабатывается конечным классом в иерархии наследования. При необходимости класс может передать полученное сообщение своему непосредственному родителю или вызвать автоматный метод родителя с любым другим сообщением. События поступают в систему автоматов через класс `UI::UIManager`, преобразующий сообщения операционной системы в события для автоматов. Объект класса `UI::UIManager` в свою очередь с использованием механизма класса `UI::Container` передает сообщение содержащимся в нем элементам интерфейса верхнего уровня, для которых это сообщение актуально (например, при щелчке мыши событие должно быть передано окну, находящемуся под курсором и окну, которое было активно в предыдущий момент времени). Те из них, которые являются контейнерами, передают событие ниже по иерархии принадлежности объектам, для которых это событие актуально.

Диаграмма последовательности, отвечающая такому образу взаимодействия, представлена на рис. 4.

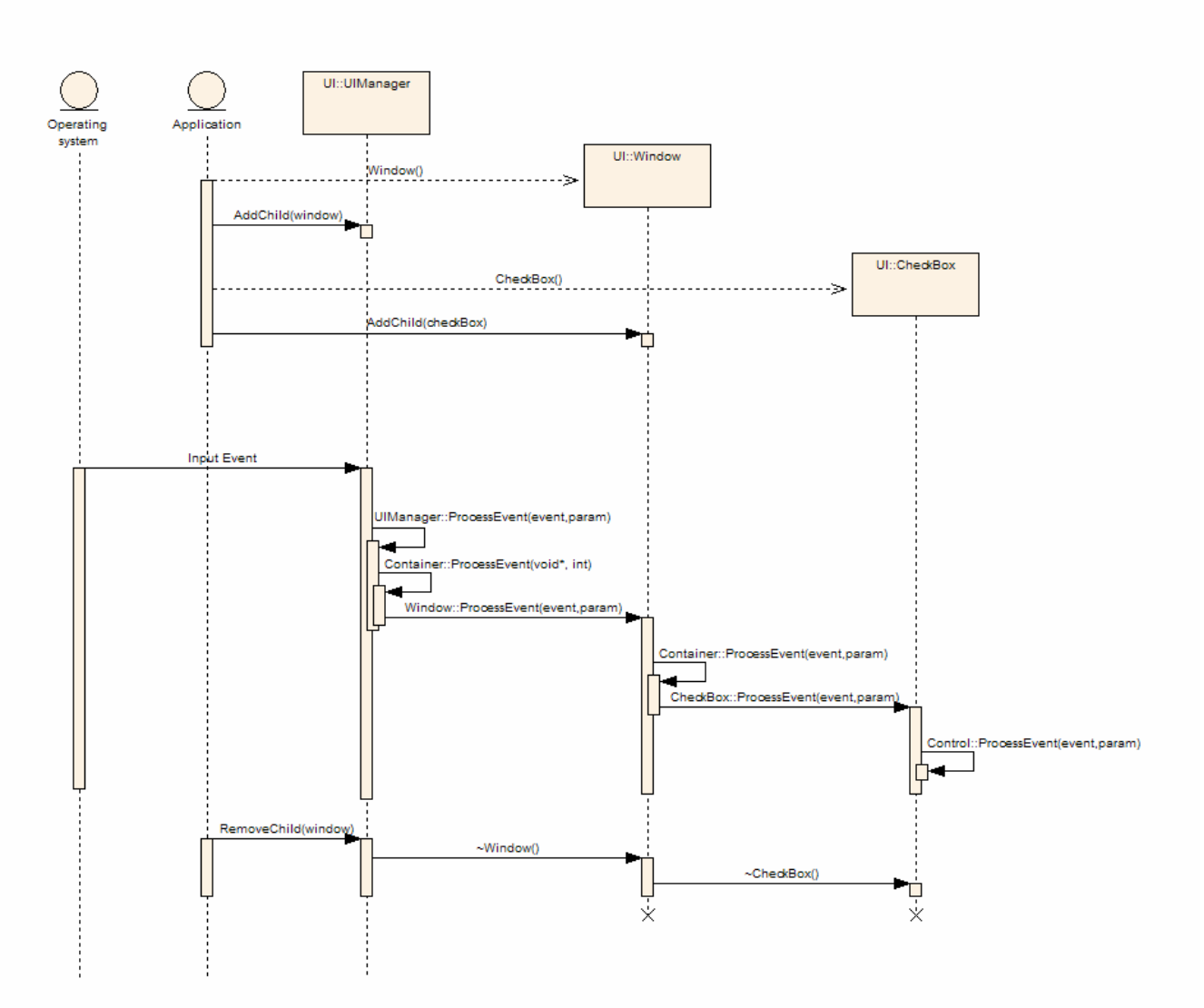

Рис. 4. Диаграмма последовательности вызовов при создании окна с элементом управления и передаче сообщения пользовательского ввода от операционной системы

На представленной диаграмме продемонстрирован процесс создания окна с элементом управления «Флажок», передача события пользовательского ввода и удаление окна. Опишем последовательность производимых при этом действий.

1. Приложение (`Application`) создает объект «Окно» (объект класса `UI::Window`).

2. Для того, чтобы созданное окно смогло взаимодействовать с другими элементами пользовательского интерфейса путем приема и передачи событий, его необходимо включить в систему автоматов. Для этого приложение добавляет окно в менеджер пользовательского интерфейса с помощью метода `UI::UIManager::AddChild`.
3. Приложение создает элемент управления «Флажок» (объект класса `UI::CheckBox`).
4. Приложение добавляет флажок в окно с помощью метода `UI::Window::AddChild`.
5. В процессе работы от операционной системы (среды исполнения, библиотеки ввода) поступает событие пользовательского ввода (`Input Event`). Для определенности будем считать, что это, например, событие «Нажатие кнопки мыши» (`EMDown`).
6. Событие пользовательского ввода поступает в объект «Менеджер пользовательского интерфейса» (`UI::UIManager`). Способ получения этого события зависит от конкретной среды исполнения. Для *Windows* приложения, например, это оконная функция.
7. Менеджер пользовательского интерфейса решает, какому событию системы автоматов соответствует данное событие пользовательского ввода, формирует нужное событие и его аргументы. Затем менеджер пользовательского интерфейса передает событие содержащемуся в

нем автомату «Менеджер пользовательского интерфейса» путем вызова метода `UI::UIManager::ProcessEvent`.

8. Автомат «Менеджер пользовательского интерфейса» проверяет, может ли он обработать событие, с которым он был вызван. Событие `EMDown` в этом автомате не обрабатывается, поэтому проверка не проходит, и событие должно быть передано вверх по иерархии наследования.
9. Далее, автомат «Менеджер пользовательского интерфейса» передает событие автомату своего базового класса (`UI::Container`) — автомату «Контейнер».
10. Автомат «Контейнер» может обработать событие `EMDown`. Он определяет, какому из элементов управления, содержащихся в нем, это событие должно быть перенаправлено.
11. Автомат «Контейнер» передает сообщение автомату найденного элемента управления (в данном случае, это окно), путем вызова метода `UI::Window::ProcessEvent`.
12. Автомат «Окно», просмотрев аргументы события `EMDown`, определяет, что нажатие кнопки мыши произошло в клиентской зоне окна, и передает событие автомату своего базового класса — `UI::Container`.

13. Далее, аналогично пп. 10 и 11 событие передается автомату «Флажок» элемента управления, содержащегося в окне.
14. В самой простой реализации автомат «Флажок» осуществляет переход между состояниями «Неотмеченный» и «Отмеченный», после чего передает автомату «Элемент управления» своего базового класса `UI::Control` событие «Перерисовка».
15. Происходит перерисовка флажка в отмеченном состоянии.
16. Когда работа с окном закончена, приложение удаляет окно из системы автоматов, вызывая метод `UI::UIManager::RemoveChild`, который, в свою очередь, удаляет окно и все содержащиеся в нем элементы управления.

Описанная схема взаимодействия достаточно гибка и не привязана к конкретному способу передачи сообщений между автоматами. В рассматриваемом примере реализации используется прямой вызов автоматных методов объектов. Однако такой способ взаимодействия может быть с легкостью реализован и в рамках существующих библиотек автоматного программирования [6, 6].

3.2. Автомат «Элемент управления»

Итак, основным «строительным блоком» систем пользовательского интерфейса является *элемент управления*. Элементы управления, как уже отмечалось ранее, предназначены для отображения какой-либо информации (текстовой, графической, числовой, булевой и т.п.) и получения ввода от пользователя. Опишем общую часть логики работы различных элементов управления с помощью конечного автомата «Элемент управления» (*A0*).

При реализации логики работы элементов управления классическим способом – с использованием флагов, обычно выделяют такие свойства, как:

- видимость (`visible/invisible`);
- заблокированность (`disabled/enabled`);
- активность (наличие фокуса ввода в данном элементе, `focused/unfocused`).

При этом полный набор включает восемь состояний. Если рассмотреть те состояния, которые реально возможны в работающем элементе управления, то останется всего четыре состояния (рис. 6):

- **невидимый** — элемент управления не отображается на экране;
- **заблокированный** — элемент управления отображается как заблокированный (обычно серым цветом), и не доступен для пользователя;
- **нейтральный** — элемент отображается обычным способом;

- **активный** — элементу принадлежит фокус ввода, при этом ввод с клавиатуры направляется в первую очередь данному элементу.

Переходы между этими состояниями происходят по управляющим событиям `EShow`, `EHide`, `EFocusEnter`, `EFocusLeave`, `EDisable`, `EEnable` (рис. 5).

Рис. 5. Схема взаимодействия автомата «Элемент управления» (*A0*)

Событие `ERepaint` используется для инициации родительским контейнером перерисовки элемента управления. Это необходимо, когда изменяется видимая на экране область объекта (например, при перемещении

одного элемента поверх другого). События EMove, EResize используются для изменения соответственно, положения и размера элемента управления. Новое положение (или размер) передаются в аргументах события.

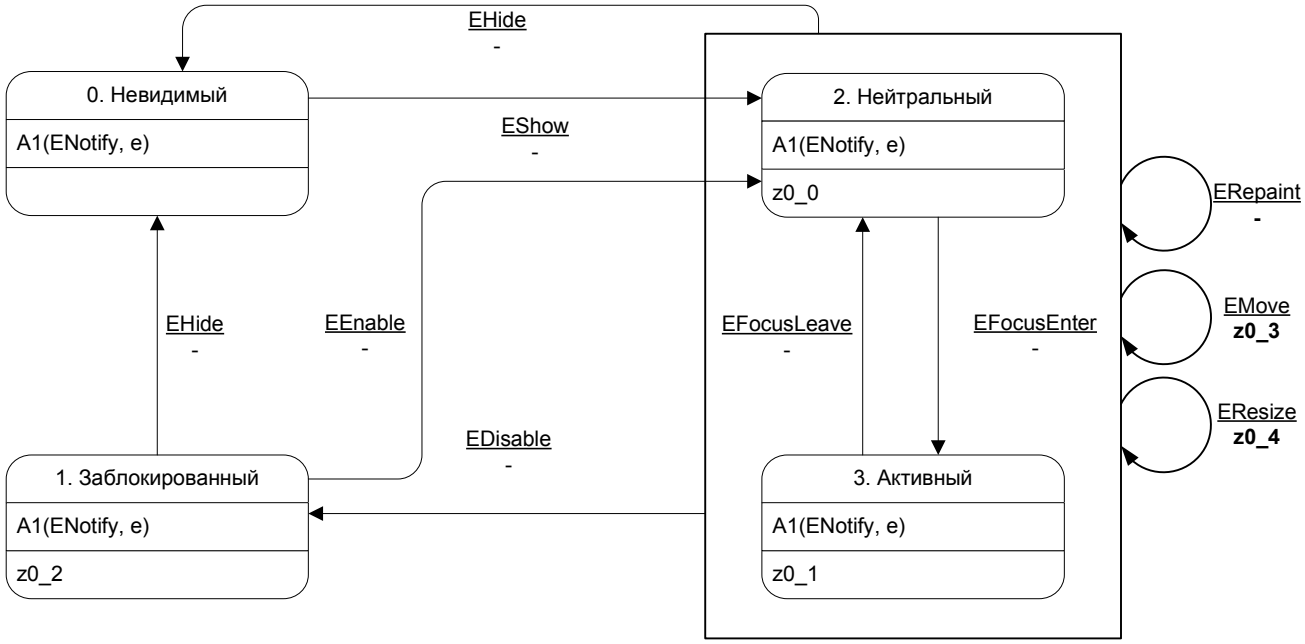

Рис. 6. Диаграмма состояний автомата «Элемент управления» (*A0*)

Элемент управления также принимает от автомата «Контейнер» события пользовательского ввода: EMDown, EMUp, EMMove, EChar. События ввода не обрабатываются в автомате «Элемент управления», но могут быть обработаны в автоматах элементов управления, классы которых перекрывают наследованный метод ProcessEvent. События пользовательского ввода перенаправляются автоматом «Контейнер» без изменения. Есть также два события ввода, генерируемые непосредственно автоматом «Контейнер»: EMHover и EMOut, оповещающие элемент управления о том, что курсор мыши вошел в область элемента либо вышел из нее.

Элементу управления также необходимо оповещать свой контейнер об изменении своего состояния, положения и размера. Для оповещения родительского контейнера автомат «Элемент управления» использует событие `ENotify`, в аргументах которого передаются данные о событии, которое вызвало изменение состояния автомата.

Автомат "Элемент управления" реализован в методе `ProcessEvent` класса `UI::Control`. Этот класс является базовым для всех классов элементов пользовательского интерфейса.

Класс `UI::Control` также содержит основные свойства, характерные для элементов управления. Эти свойства включают в себя:

- ссылку на родительский контейнер (`parent`);
- положение в системе координат родительского контейнера (`position`);
- размер прямоугольника, описывающего элемент управления (`size`);

Также часто с элементом управления ассоциируется текст. Для окна это текст в заголовке, для кнопки – надпись на ней, для всплывающей подсказки – текст подсказки. Поэтому в список полей класса `UI::Control` также включено строковое поле `text`.

3.3. Автомат «Контейнер»

Для удобства разработки и использования пользовательского интерфейса элементы управления обычно группируются с помощью окон, диалогов, панелей, групп, закладок, и других *контейнеров*. Контейнеры, по сути, также являются элементами управления, но, кроме того, могут содержать в себе другие элементы управления, организовывать их размеры, расположение, передавать им события пользовательского ввода.

Автомат «Контейнер» (*A1*), схема взаимодействия которого представлена на рис. 7, реализован в методе `ProcessEvent` класса `UI::Container`. Класс `UI::Container` наследован от класса `UI::Control`, поскольку контейнеры удобно представлять в виде элементов управления, которые могут содержать другие элементы управления. Для многих контейнеров также характерны основные состояния, в которых может находиться элемент управления.

Основными функциями автомата «Контейнер» являются поддержание списка находящихся в нем элементов управления, перенаправление им событий ввода, отслеживание изменений фокуса и координация перерисовки элементов управления. Для выполнения последней функции автомат «Контейнер» должен получать уведомления об изменении состояния, размера и положения от содержащихся в нем элементов управления. Этой цели служит сообщение `ENotify`.

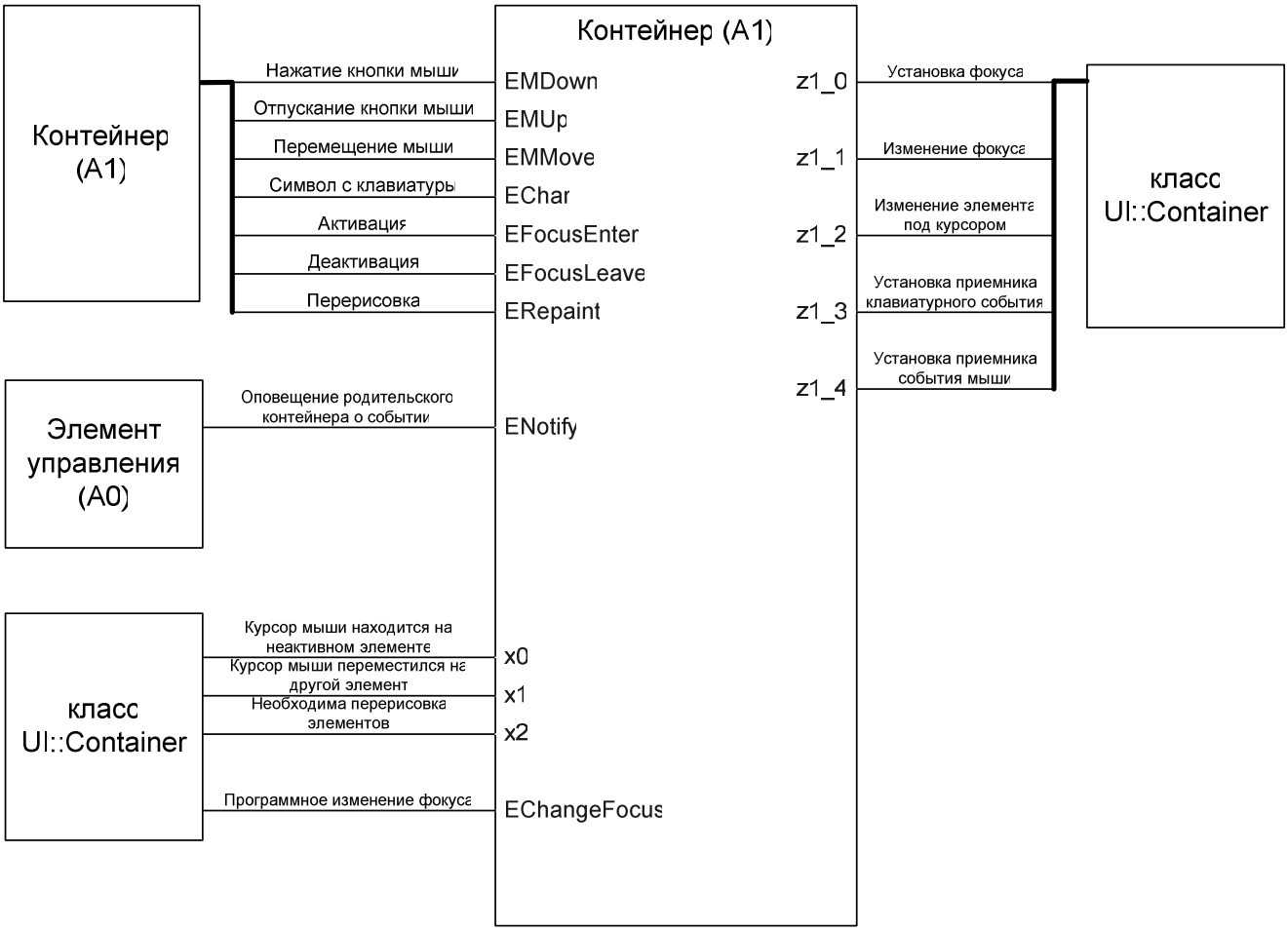

Рис. 7. Схема взаимодействия автомата «Контейнер» (*A1*)

Прорисовка также может быть запущена событием `ERepaint`, полученным от контейнера, содержащего данный либо от автомата «Менеджер пользовательского интерфейса».

Диаграмма состояний автомата «Контейнер» представлена на рис. 8.

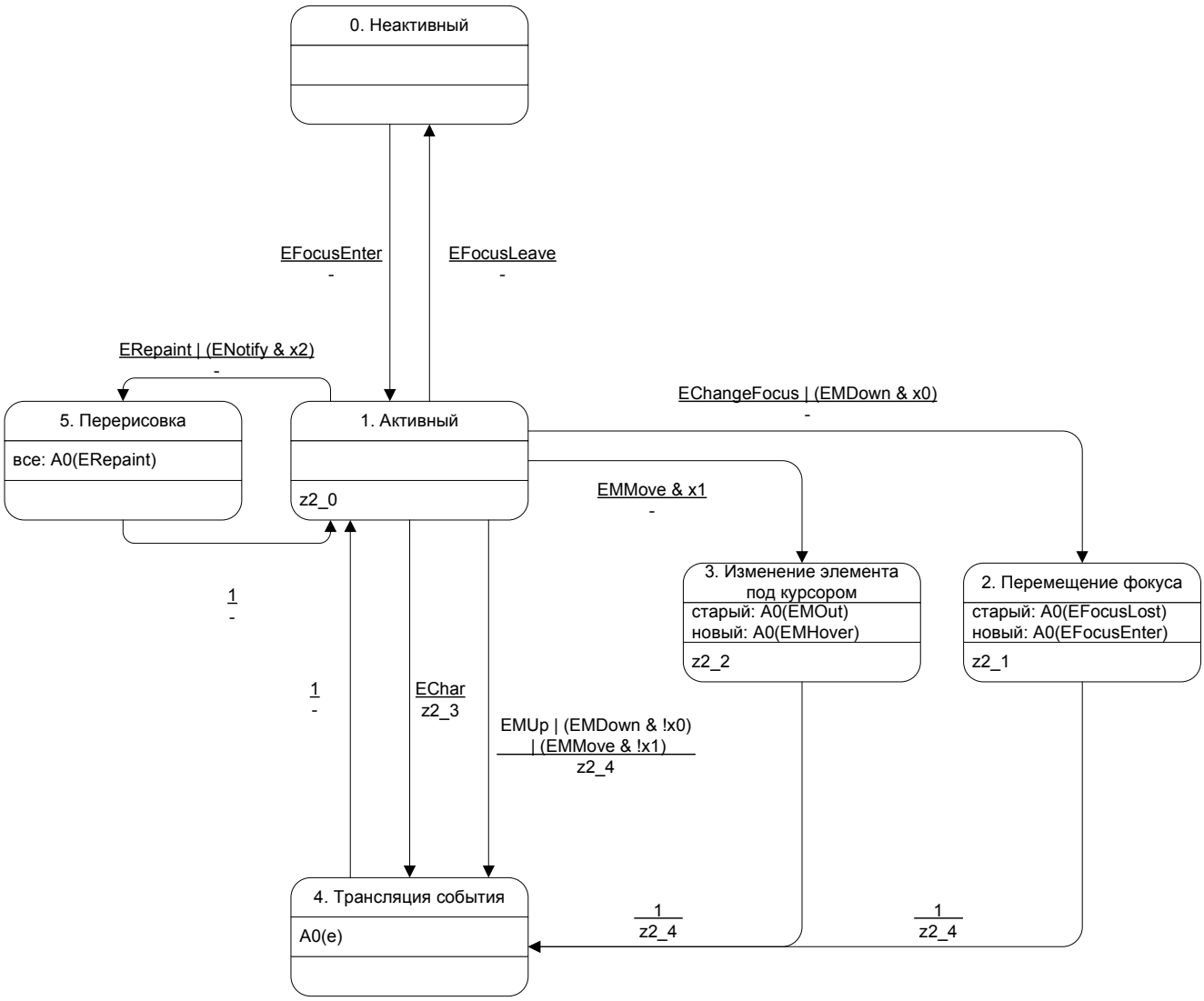

Рис. 8. Диаграмма состояний автомата «Контейнер» (*A1*)

События ввода перенаправляются автомату «Элемент управления». Объект класса `UI::Control`, которому должно быть направлено событие ввода, определяется следующим образом:

- клавиатурное событие направляется содержащемуся элементу управления, находящемуся в активном состоянии (имеющему фокус ввода), либо автомату «Элемент управления» данного контейнера, если в нем нет активных элементов управления;

- событие ввода от мыши перенаправляется находящемуся под курсором мыши элементу управления, либо автомату «Элемент управления» данного контейнера, если курсор мыши находится вне всех содержащихся в данном контейнере элементов управления.

При перенаправлении события перемещения курсора мыши также производится проверка, не пересек ли курсор границы элемента управления. При необходимости генерируются события `EMHover` и `EMOut`.

При нажатии кнопки мыши на неактивном элементе либо при поступлении события `EChangeFocus` происходит перемещение фокуса ввода и генерируются соответствующие события для элемента, бывшего активным до этого и для нового активного элемента.

3.4. Автомат «Менеджер пользовательского интерфейса»

Для того, чтобы события пользовательского ввода поступали в систему автоматов, и чтобы система автоматов могла производить различные действия во внешней среде, необходим слой сопряжения. Таким слоем сопряжения в разрабатываемой системе является класс `UI::UIManager`, содержащий автомат «Менеджер пользовательского интерфейса» (*A2*).

Класс `UI::UIManager` является наиболее зависимым от среды исполнения. Он получает оповещения о событиях пользовательского ввода от операционной системы, библиотеки пользовательского ввода или среды

исполнения. Также, класс `UI::UIManager` является контейнером для элементов управления верхнего уровня, поэтому он наследован от класса `UI::Container`.

Реализация автомата «Менеджер пользовательского интерфейса» во многом также определяется конкретной средой исполнения. В случаях, когда система событий операционной системы (или библиотеки пользовательского ввода) мало отличается от событий системы автоматов пользовательского интерфейса, реализация автомата «Менеджер пользовательского интерфейса» будет тривиальной: автомат будет всего лишь перенаправлять события автомату «Контейнер» своего базового класса. В других случаях, автомат «Менеджер пользовательского интерфейса» должен поддерживать состояние, необходимое для преобразования входных событий в события системы автоматов. Например, если операционная система присылает оповещения только о нажатии и отпускании кнопок клавиатуры, но не присылает событий о повторении символа при длительном удержании кнопки, то эту возможность следует реализовать в автомате «Менеджер пользовательского интерфейса».

Таким образом, для операционной системы с богатым набором событий ввода (например, ОС *Windows*) реализация описываемого автомата будет тривиальной. Схема связи автомата «Менеджер пользовательского интерфейса» приведена на рис. 9.

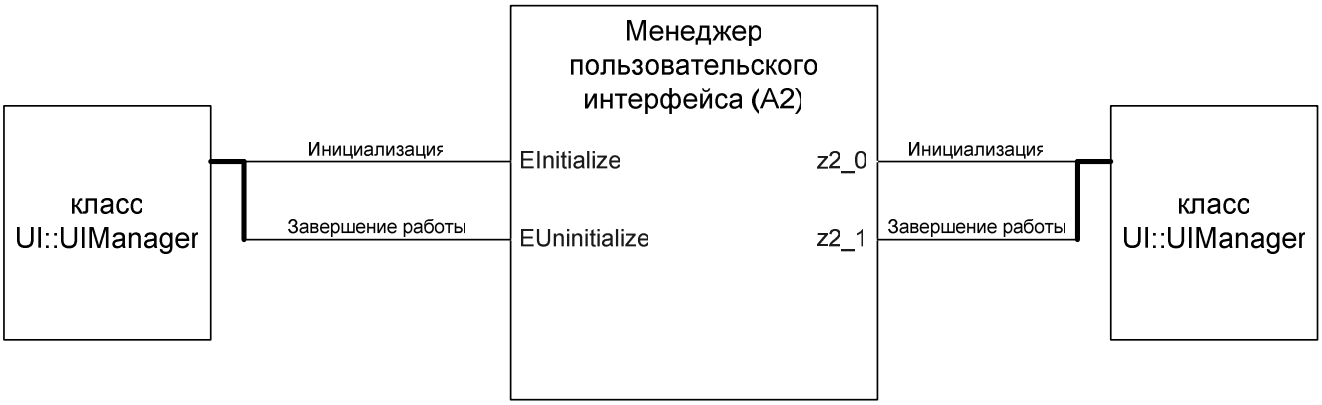

Рис. 9. Схема взаимодействия автомата «Менеджер пользовательского интерфейса» (*A2*)

Диаграмма состояний автомата «Менеджер пользовательского интерфейса» приведена на рис. 10.

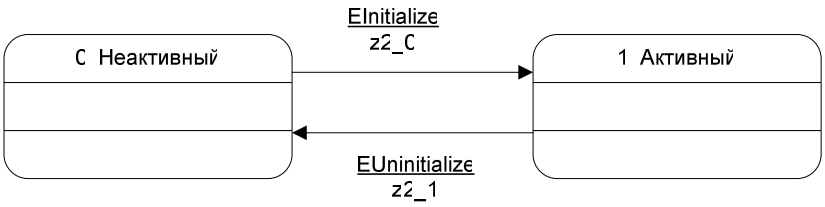

Рис. 10. Диаграмма состояний автомата «Менеджер пользовательского интерфейса» (*A2*)

Описанные в данной главе автоматы «Элемент управления», «Контейнер», «Менеджер пользовательского интерфейса» и соответствующие им классы `UI::Control`, `UI::Container` и `UI::UIManager`, являются базой для построения системы конкретных элементов управления и контейнеров.

4. РЕАЛИЗАЦИЯ КОНКРЕТНЫХ ЭЛЕМЕНТОВ УПРАВЛЕНИЯ И КОНТЕЙНЕРОВ

В предыдущей главе был описан метод реализации базовых классов и автоматов системы элементов пользовательского интерфейса. Реализации конкретных элементов управления будут опираться на описанные классы и автоматы.

Для построения диаграмм состояний конкретных элементов пользовательского интерфейса необходимо:

- рассмотреть сценарии взаимодействия пользователя с элементом управления посредством устройств ввода;
- рассмотреть взаимодействие элемента управления с программным кодом приложения;
- выявить основные визуально-различимые состояния элемента управления;
- определить дополнительные состояния, необходимые для реализации сценариев взаимодействия с пользователем и кодом приложения;
- определить события пользовательского ввода, приводящие к изменению состояния и действия, производимые при этом;
- определить программные события, приводящие к изменению состояния;

- для визуально-различимых состояний добавить в качестве выходных воздействий отображение элемента управления в данном состоянии;
- добавить дополнительные выходные воздействия, необходимые для реализации сценариев взаимодействия элемента управления с пользователем и кодом приложения.

Далее приведем примеры реализации нескольких элементов управления и контейнеров. Стоит заметить, что реализация пассивных элементов управления (таких как текстовая метка, изображение, панель, группа, и т.д.) — тривиальна, и состоит лишь в реализации метода прорисовки элемента в нейтральном состоянии.

4.1. Элемент управления «Кнопка»

Кнопка (рис. 11) является самым простым из активных элементов пользовательского интерфейса.

Рис. 11. Элемент управления «Кнопка»

Взаимодействие этого элемента управления с пользователем ограничивается возможностью нажать кнопку. Приложение при этом должно получить оповещение о нажатии. Обычно, считается, что кнопка нажата, когда была отпущена кнопка мыши, при этом курсор находился над кнопкой, и, когда

кнопка мыши нажималась, курсор тоже находился над кнопкой. Будем реализовывать именно такое поведение кнопки.

Вопрос оповещения приложения о нажатии кнопки можно решить множеством элементарных способов, поэтому в данной работе мы не будем останавливаться на этом подробнее. Будем считать, что выходным воздействием автомата может являться оповещение приложения о каком-либо событии.

Итак, кроме стандартных состояний пользовательского элемента, кнопка имеет два особых состояния:

- кнопка нажата;
- кнопка была нажата, но курсор мыши выведен за ее пределы.

Эти два состояния возможны только когда состояние автомата «Элемент управления» базового класса — либо «нейтральный» (2), либо «активный» (3).

Опишем реализацию элемента управления «Кнопка» с помощью автомата «Кнопка» и класса `UI::Button`.

Отображение кнопки в двух новых состояниях выполняется в методах `z3_0` и `z3_1` класса `UI::Button`. Отображение же кнопки в состояниях «нейтральное», «активное», «заблокированное» (принадлежащих автомату «Элемент управления») производится в переопределенных защищенных виртуальных методах `z0_0`, `z0_1`, `z0_2`, описанных в классе `UI::Control`, а реализованных в классе `UI::Button`.

В результате, диаграмма взаимодействия автомата «Кнопка» приобретает вид, показанный на рис. 12.

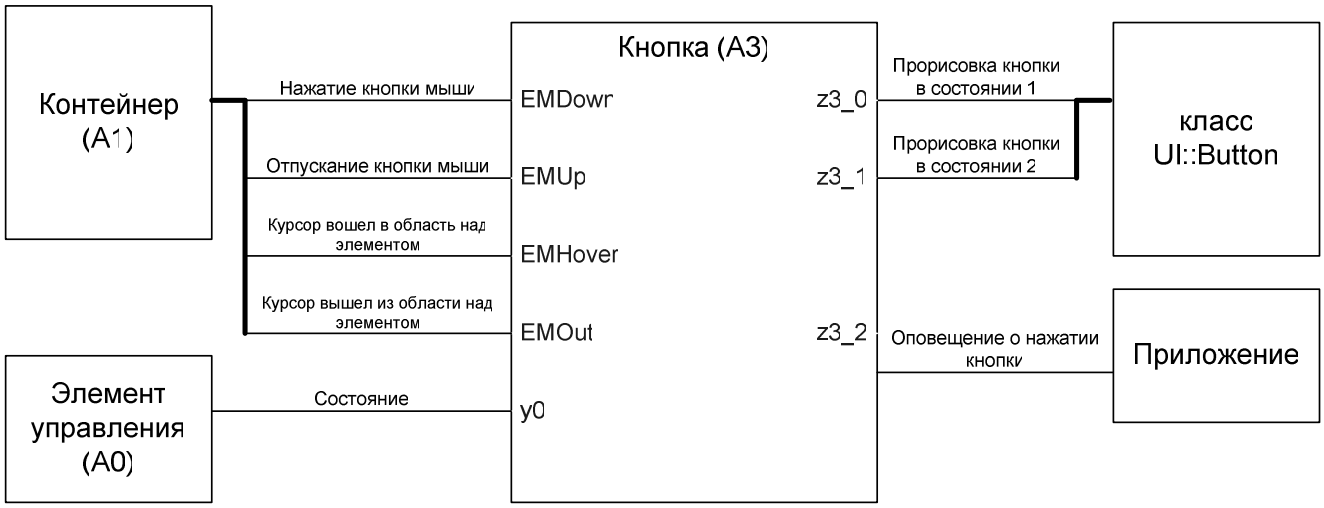

Рис. 12. Схема взаимодействия автомата «Кнопка» (*A3*)

Диаграмма состояний автомата «Кнопка» представлена на рис. 13.

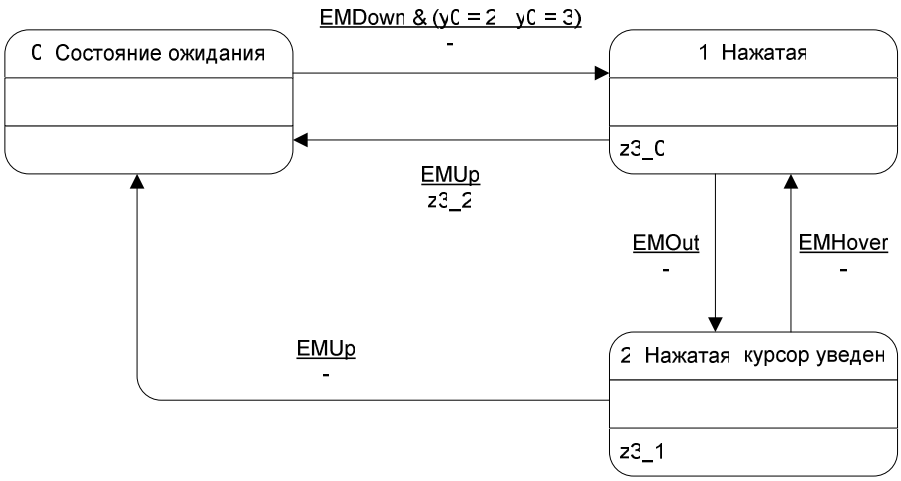

Рис. 13. Диаграмма состояний автомата «Кнопка» (*A3*)

На примере этого простого элемента управления уже можно проиллюстрировать более четкое отделение логики поведения от системно-зависимых функций. Например, в библиотеке *AWT* среды исполнения *Java 1.5.0_03* для ОС *Windows* реализация логики обработки событий

пользовательского ввода элемента управления «Кнопка» находится в 3х классах: `java.awt.Button`, `sun.awt.windows.WComponentPeer` и `sun.awt.windows.WButtonPeer`, последний из которых реализован большей частью на языке `C` в платформо-зависимой библиотеке `awt.dll` и выполняет также функции отображения.

В нашем же случае вся логика обработки сообщений пользовательского ввода, специфичная для элемента управления «Кнопка», расположена в одном методе `ProcessEvent` класса `UI::Button`. А код, отвечающий за прорисовку, расположен в виртуальных методах `z3_0` и `z3_1`, которые можно реализовать как в классе `UI::Button`, так и в наследованном от него классе, если потребуется.

4.2. Элемент управления «Флажок»

Элемент управления «Флажок» (рис. 14) представляет собой переключатель, который может находиться в отмеченном и неотмеченном состоянии.

Рис. 14. Элемент управления «Флажок» в отмеченном и неотмеченном состоянии

Кроме этого, если реализовывать поведение, аналогичное поведению кнопки при нажатии, описанному в предыдущем параграфе, то список состояний будет такой:

- неотмеченное;
- неотмеченное-нажатое;
- неотмеченное-нажатое, курсор вне области флажка;
- отмеченное;
- отмеченное-нажатое;
- отмеченное-нажатое, курсор над флажком.

Список обрабатываемых событий включает все события, обрабатываемые элементом управления «Кнопка», плюс событие программного изменения отметки – `EToggleCheck`. Список выходных воздействий состоит из функций прорисовки флажка в различных состояниях и функции оповещения приложения об изменении отметки. Итак, можно построить диаграмму взаимодействия (рис. 15) автомата «Флажок» (*A4*), реализующего элемент управления «флажок».

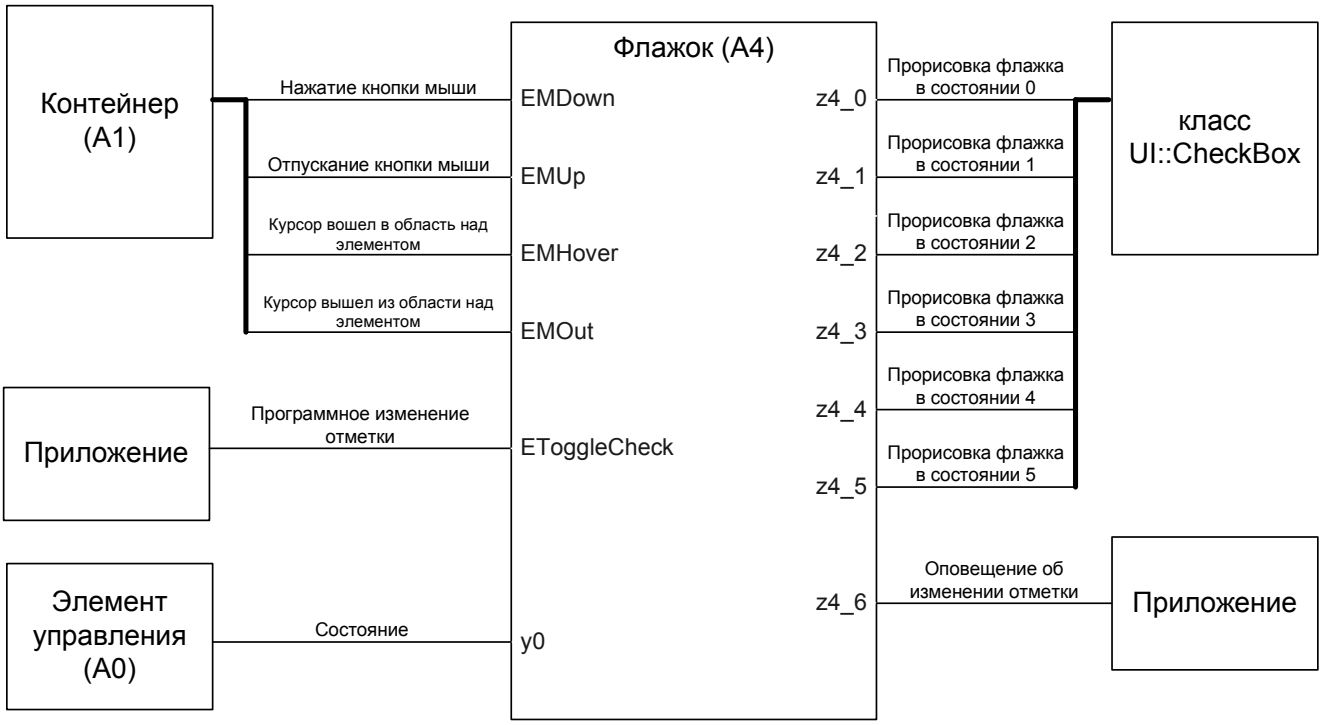

Рис. 15. Схема взаимодействия автомата «Флажок» (*A4*)

Диаграмма состояний автомата «Флажок» (A4) представлена на рис. 16.

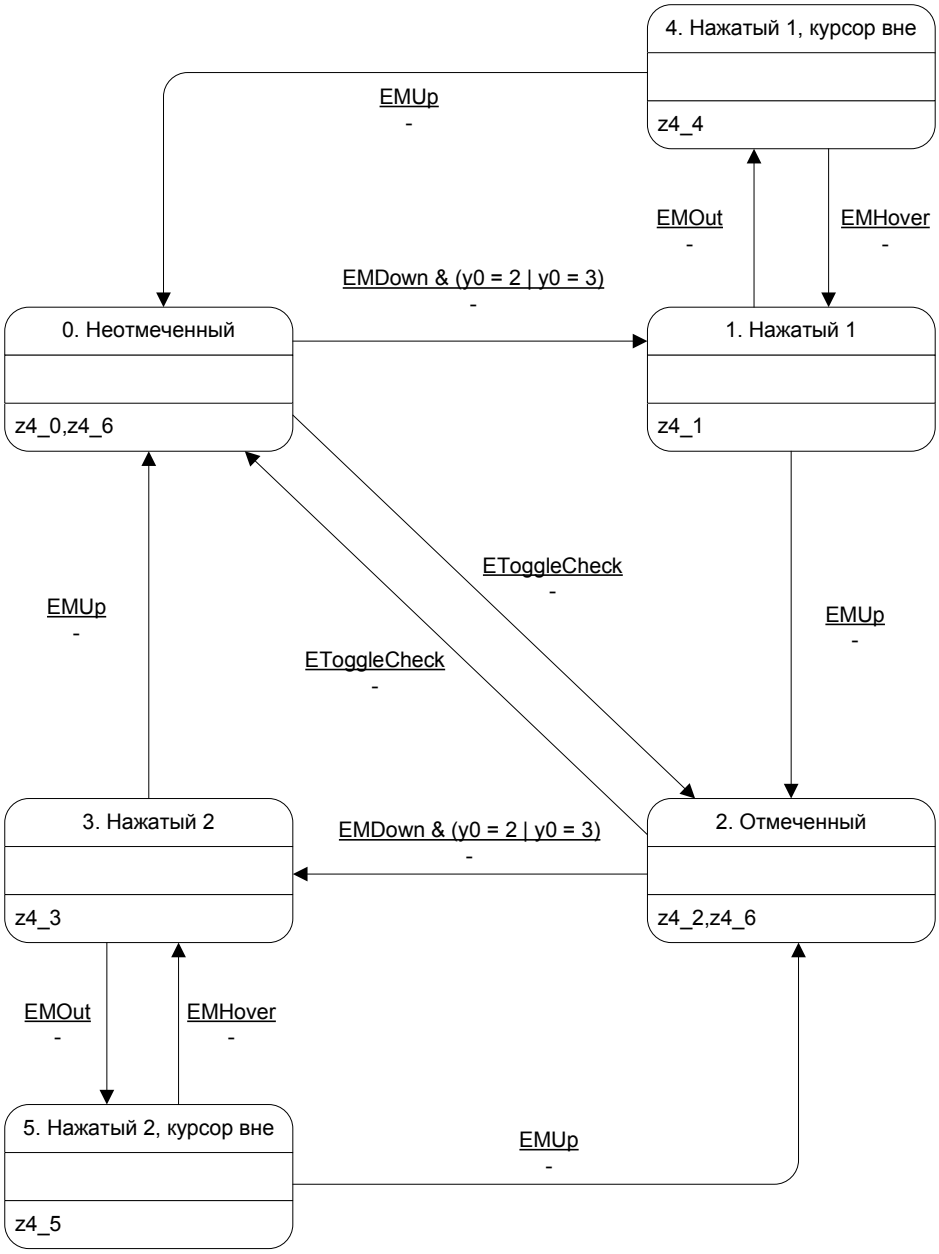

Рис. 16. Диаграмма состояний автомата «Флажок» (*A4*)

Так же как и в элементе управления «Кнопка» логика работы элемента управления «Флажок» полностью реализована в автоматном методе класса `UI::CheckBox`, а не «размазана» по нескольким классам и более чем десяти методам, как при использовании традиционного подхода.

4.3. Контейнер «Окно»

Окно является контейнером верхнего уровня. Окна бывают достаточно сложные, с поддержкой различных визуальных эффектов, прозрачности, и т.п. В качестве примера рассмотрим простой, но вполне функциональный вид окна (рис. 17).

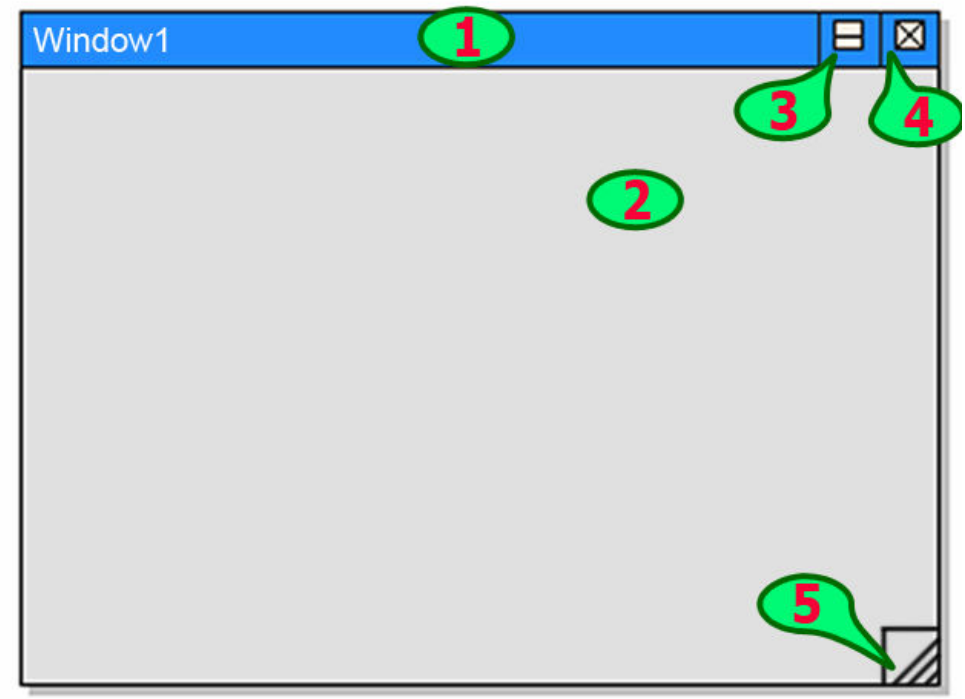

Рис. 17. Контейнер «Окно»

В представленном окне можно выделить следующие области:

1. Заголовок окна. В нем отображается название окна и за него окно можно перетаскивать по экрану.
2. Клиентская область окна. Служит для расположения содержащихся в окне элементов управления.

3. Кнопка минимизации окна. При ее нажатии клиентская область окна перестает отображаться. При повторном нажатии окно возвращается в нормальный вид.
4. Кнопка закрытия окна. Генерирует оповещение для приложения. Приложение само должно определить, каким образом его обрабатывать.
5. Зона изменения размера. При перетаскивании ее мышью изменяется размер окна.

Для реализации кнопок закрытия и минимизации можно воспользоваться описанным выше элементом управления «Кнопка». Эти две кнопки должны быть добавлены в окно при его создании. Обработчики событий их нажатия должны генерировать автоматные события `EMinimize` и `EClose`. Другие обрабатываемые события перечислены на схеме взаимодействия автомата «Окно» (А5) (рис. 18).

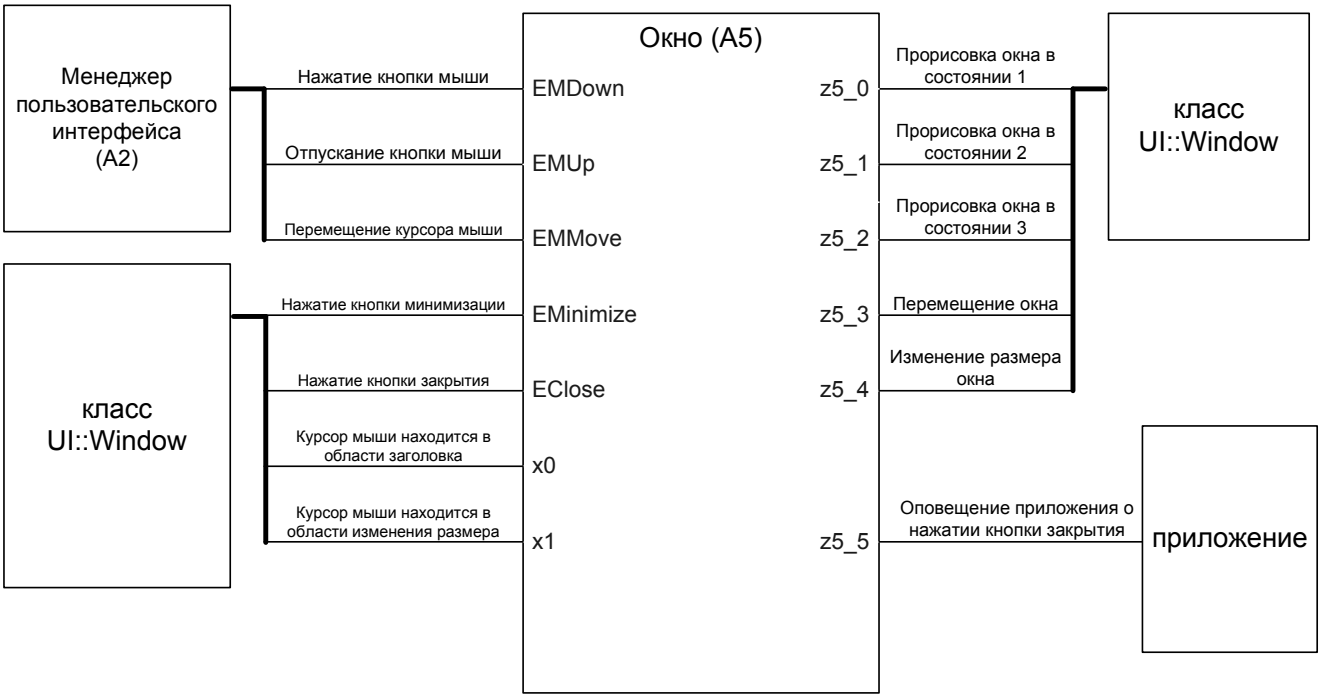

Рис. 18. Схема взаимодействия автомата «Окно» (*A5*)

Диаграмма состояний автомата представлена на рис. 19.

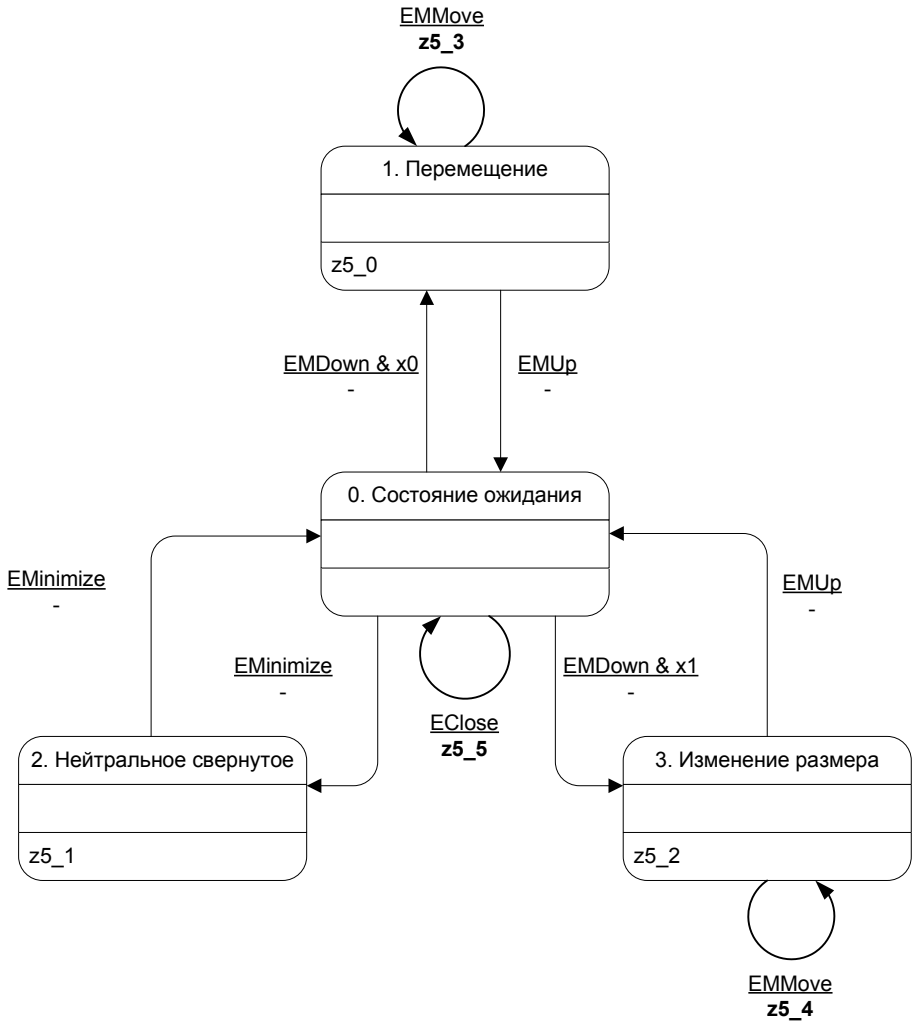

Рис. 19. Диаграмма состояний автомата «Окно» (*A5*)

Как видно из представленных диаграмм, логика работы элементов управления достаточно проста и прямолинейна. Использование базовых классов, содержащих автоматы, реализующие базовую часть логики работы элементов управления, позволяет создавать простые и понятные диаграммы состояний конкретных элементов управления и контейнеров.

ЗАКЛЮЧЕНИЕ

В настоящей работе были рассмотрены основные принципы создания элементов пользовательского интерфейса и проанализированы некоторые широко используемые реализации пользовательского интерфейса. В результате был сделан вывод, что существующий подход к реализации элементов пользовательского интерфейса обладает рядом недостатков. Среди них:

- состояние элементов пользовательского интерфейса явно не выделяется, а для его кодирования используются флаги;
- логика работы элементов пользовательского интерфейса децентрализована и трудно отделима от системно-зависимого кода;
- поведение элементов пользовательского интерфейса в различных реализациях может различаться.

Был разработан автоматный подход к реализации элементов пользовательского интерфейса. Разработанный подход устраняет перечисленные недостатки, а также приносит некоторые преимущества:

- централизация логики работы;
- соответствие отображаемого состояния элементов управления внутреннему представлению и логике работы;
- высокая эффективность разработки и отладки систем пользовательского интерфейса;

- высокая переносимость реализаций, разработанных в соответствии с автоматным подходом, обусловленная выделением системно-зависимого кода в отдельный слой сопряжения;
- изначальная документированность поведения системы элементов пользовательского интерфейса;
- выявление многих ошибок разработки на этапе проектирования, за счет возможности подробного описания поведения системы автоматов.

Список литературы

1. *Головач В.В.* Дизайн пользовательского интерфейса. http://www.uibook1.ru/
2. *Шалыто А.А.* SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998. 628с. http://is.ifmo.ru/books/switch/1
3. *Шалыто А.А., Туккель Н.И.* Программирование с явным выделением состояний. //Мир ПК. 2001. № 8, с. 116–121, № 9, с. 132–138. http://is.ifmo.ru/works/mirk/
4. *Шалыто А.А., Туккель Н.И.* Танки и автоматы //BYTE/Россия. 2003. № 2. http://is.ifmo.ru/works/tanks_new/
5. *Шалыто А.А.* Технология автоматного программирования //Мир ПК. 2003. № 10. http://is.ifmo.ru/works/tech_aut_prog/
6. *Наумов А.С.* Объектно-ориентированное программирование с явным выделением состояний. СПб.: СПбГУ ИТМО, 2004. http://is.ifmo.ru/papers/anoop/
7. *Шопырин Д.Г., Шалыто А.А.* Объектно-ориентированный подход к автоматному программированию //Информационно-управляющие системы. 2003. № 5, с. 29–39. http://is.ifmo.ru/works/ooaut/