Статья опубликована в журнале «Технология клиент-сервер», 2004, № 3, с. 3-11.

ICQ и автоматы

Вадим Гуров
Андрей Нарвский
Компания eVelopers Corporation, Санкт-Петербург

Анатолий Шалыто
Санкт-Петербургский государственный университет информационных технологий, механики и оптики

В статье описан процесс разработки простого клиент-серверного приложения с использованием автоматно-ориентированного подхода. В качестве инструмента для моделирования конечных автоматов применяется, разработанный при участии авторов, программный пакет с открытым кодом UniMod, который основан на SWITCH-технологии и позволяет создавать схемы связей и графы переходов автоматов, используя UML-нотацию диаграмм классов и диаграмм состояний соответственно.

Введение

Опишем процесс разработки простого клиент-серверного приложения на примере разработки системы мгновенного обмена сообщениями между любым количеством пользователей (простой аналог *ICQ – Я Ищу Тебя*).

Система состоит из сервера сообщений и однотипных клиентских приложений. После запуска клиентское приложение присоединяется к серверу сообщений. Затем оно может получить список пользователей уже присоединенных к серверу и обмениваться сообщениями с этими пользователями. При выходе какого-либо пользователя из системы, остальные пользователи должны быть уведомлены об этом. При завершении работы сервера, все подсоединенные пользователи должны отключится от него.

При взаимодействии между серверным и клиентским приложениями в качестве транспортного протокола используется протокол *TCP,* который позволяет *пересылать массивы байт*. Для *обмена сообщениями* на более высоком (прикладном) уровне необходимо создать другой протокол, что является одной из составных частей процесса разработки. Будем говорить, что клиентское и серверное приложения обмениваются ***сообщениями***, в то время как пользователи системы – ***репликами***.

В соответствии с известными методологиями объектно-ориентированного проектирования [1] первым шагом при создании системы является описание требований, предъявляемых к системе, и ее вариантов использования (*use cases*).

Требования

Перечислим требования к системе. Должна быть обеспечена возможность:

- авторизованного входа пользователей в систему;

- обмена репликами между пользователями;
- получения по запросу списка других присоединенных пользователей.

Варианты использования

Варианты использования можно описывать как с помощью *UML*-диаграмм соответствующего типа, так и в текстовом виде [2]. Авторы применили второй подход:

1. *Вход в систему*. Клиентское приложение посылает сообщение серверу, содержащее имя и пароль пользователя. Сервер проверяет правильность этих параметров на соответствие внутренней базе данных о пользователях. После этого он посылает клиенту либо подтверждение о входе, либо отказ. При подтверждении входа все уже присоединенные клиенты уведомляются о появлении нового пользователя.
2. *Обмен репликами*. Клиент посылает сообщение серверу, содержащее имя получателя и реплику. Сервер пересылает сообщение адресату.
3. *Запрос списка пользователей*. Клиент посылает сообщение с запросом списка пользователей. Сервер формирует этот список и возвращает его клиенту.
4. *Выход клиента из системы*. Клиент посылает уведомление серверу о своем выходе. Сервер уведомляет всех остальных присоединенных клиентов об этом.
5. *Завершение работы сервера*. Сервер рассылает всем присоединенным клиентам уведомление о завершении своей работы. Все клиенты отсоединяются от сервера.

Следующим шагом, в соответствии с известными методологиями, является создание модели предметной области в виде *UML*-диаграммы классов.

Модель предметной области

Модель предметной области системы приведена на рис. 1, 2.

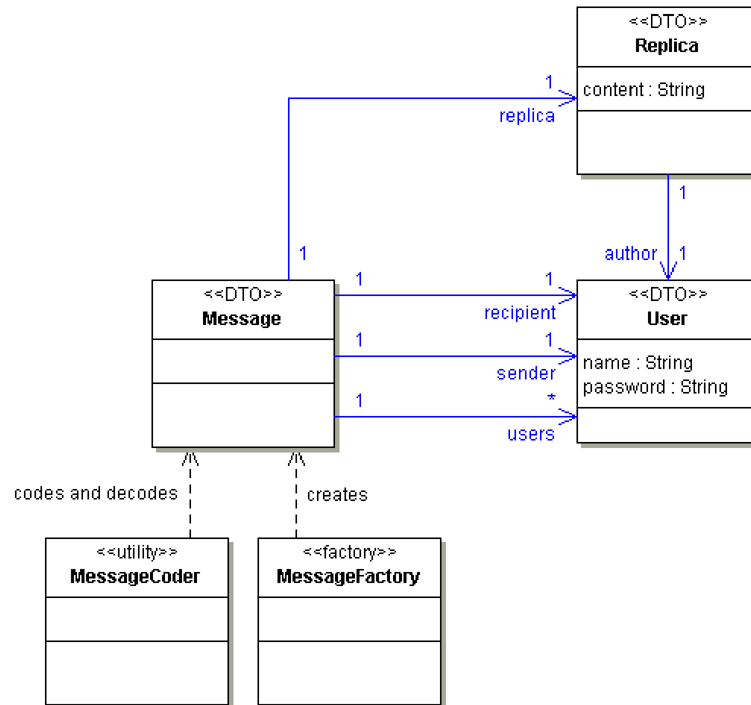

Рис. 1. Диаграмма классов. Часть 1

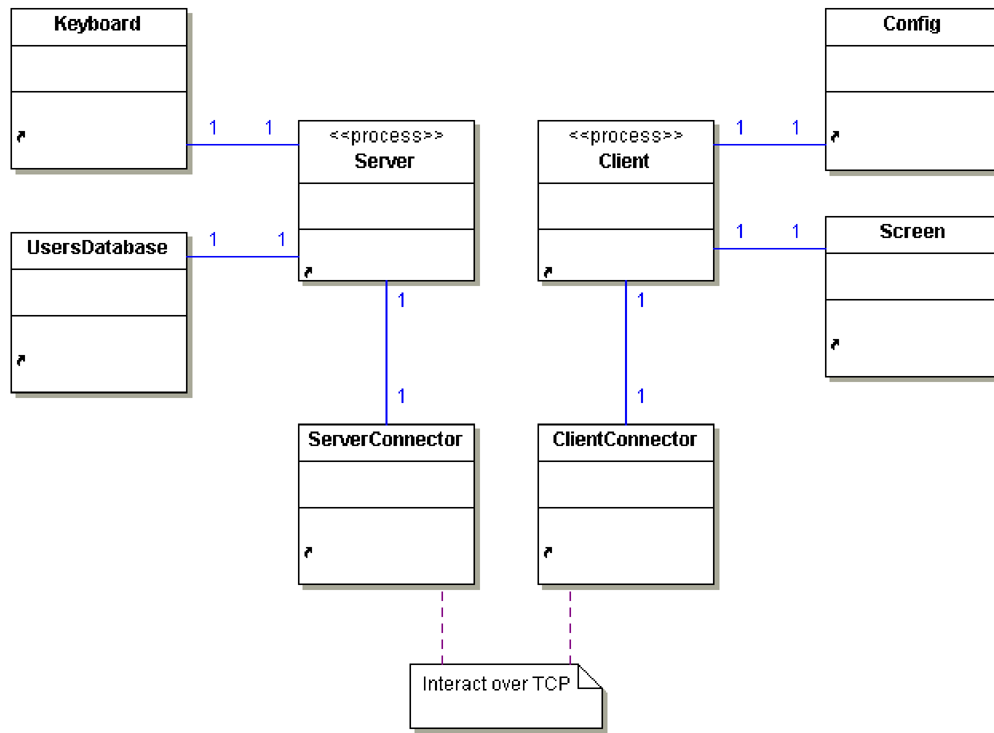

Рис. 2. Диаграмма классов. Часть 2

Диаграмма на рис. 1 содержит две группы классов:

- группа классов для передачи данных – *DTO* (*Data Transfer Objects*). Эти классы не обладают поведением и используются только как контейнеры для данных;
- группа вспомогательных классов – *Utility* и *Factory*. Они реализуют утилитные функции;

Диаграмма на рис. 2 содержит еще две группы классов:

- классы, являющиеся «входными точками» для серверного и клиентского приложений – *Process*;
- классы, реализующие функциональность подсистем серверного и клиентского приложений.

Опишем каждую из этих групп с учетом связей на приведенных выше диаграммах.

Приведем описание классов, предназначенных для передачи данных:

- класс *User* содержит информацию о пользователе системы. Сервер также может выступать в роли пользователя – инициатора сообщений;
- класс *Replica* содержит реплики, которыми пользователи обмениваются друг с другом. Реплика всегда имеет автора;
- класс *Message* содержит сообщения, которыми обмениваются клиентское и серверное приложения. Эти сообщения всегда имеют отправителя и получателя. Сообщения (в зависимости от типа) могут содержать или не содержать реплику или список присоединенных пользователей. Описание типов сообщение приведено в табл. 1.

Таблица 1. Типы сообщений

Тип сообщения	Инициатор сообщения	Описание
LOGIN	Клиент	Запрос входа в систему. Сообщение должно содержать имя и пароль пользователя
LOGIN	Сервер	Подтверждение входа
LOGOUT	Клиент	Сообщение о выходе из системы
LOGOUT	Сервер	Отказ во входе в систему либо уведомление о завершении работы сервера
REPLICA	Клиент	Посылка реплики пользователем другому пользователю
REPLICA	Сервер	Пересылка сервером реплики адресату
USERS	Клиент	Запрос списка присоединенных пользователей
USERS	Сервер	Сообщение, содержащее список присоединенных пользователей

Группа вспомогательных классов содержит:

- класс *MessageCoder*, который реализует преобразование сообщений в массив байт и обратное преобразование, обеспечивая взаимодействие между прикладным протоколом обмена сообщениями и низкоуровневым транспортным протоколом *TCP*;
- класс *MessageFactory*, реализующий фабрику [3] сообщений. Он позволяет создавать сообщения всех необходимых типов.

Запуск серверного и клиентского приложений обеспечивают классы *Server* и *Client* соответственно.

Классы, реализующие функциональность подсистем:

- класс *ServerConnector* предоставляет клиентам сетевой интерфейс для взаимодействия с сервером сообщений;
- класс *UserDatabase* представляет базу данных зарегистрированных пользователей;
- класс *Keyboard* обеспечивает управление сервером с клавиатуры;
- класс *ClientConnector* отвечает за сетевое взаимодействие клиента с сервером сообщений;
- класс *Screen* является ответственным за графический интерфейс взаимодействия с пользователем;
- класс *Config* отвечает за настройку таких параметров, как имя и пароль пользователя, адрес сервера сообщений.

После создания модели предметной области системы, воспользуемся автоматно-ориентированным подходом, который в явном виде не применяется в известных методологиях.

Автоматно-ориентированный подход

В работе [4] предложен метод проектирования событийных объектно-ориентированных программ с явным выделением состояний, названный «*SWITCH*-технологией». Особенность этого подхода состоит в том, что поведение в таких программах описывается с помощью графов переходов структурных конечных автоматов с нотацией, предложенной в работе [5].

SWITCH-технология определяет для каждого автомата два типа диаграмм (схему связей и граф переходов) и их операционную семантику. При наличии нескольких автоматов, кроме того, строится схема их взаимодействия. *SWITCH*-технология задает свою нотацию диаграмм. Предлагается, сохранив автоматный подход, использовать *UML*-нотацию при построении диаграмм в рамках *SWITCH*-технологии. При этом, применяя нотацию диаграмм классов языка *UML*, строятся схемы связей автоматов, определяющих их интерфейс, а графы переходов строятся с помощью *UML*-нотации диаграммы состояний.

Применение автоматно-ориентированного подхода для моделирования систем состоит в следующем:

1. На основе анализа предметной области разрабатывается концептуальная модель системы, определяющая сущности и отношения между ними (диаграммы классов). Для разрабатываемой системы обмена сообщениями этот этап уже завершен и описан выше.
2. В отличие от традиционных для объектно-ориентированного программирования подходов [1], из числа сущностей выделяются источники событий, объекты управления и автоматы. Источники событий активны — они по собственной инициативе воздействуют на автомат. Объекты управления пассивны — они выполняют действия по команде от автомата. Объекты управления также формируют значения входных переменных для автомата. Автомат активируется источниками событий и на основании значений входных переменных и текущего состояния воздействует на объекты управления, переходя в новое состояние.
3. Используя нотацию диаграммы классов, строится схема связей автомата, задающая его интерфейс. На этой схеме слева отображаются источники событий, в центре — автоматы, а справа — объекты управления. Источники событий с помощью *UML*-ассоциаций связываются с автоматами, события которым они поставляют. Автоматы связываются с объектами, которыми они управляют.
4. Схема связей, кроме задания интерфейса автомата, выполняет функцию, характерную для диаграммы классов — задает объектно-ориентированную структуру программы.
5. Каждый объект управления содержит два типа методов, реализующих входные переменные (`xj`) и выходные воздействия (`zk`).
6. Для каждого автомата с помощью нотации диаграммы состояний строится граф переходов типа *Мура-Мили*, в котором дуги могут быть помечены событием (`ei`), булевой формулой из входных переменных и формируемыми на переходах выходными воздействиями. В вершинах могут указываться выходные воздействия и имена вложенных автоматов. Каждый автомат имеет одно начальное и произвольное количество конечных состояний.
7. Состояния на графе переходов могут быть простыми и сложными. Если в состояние вложено другое состояние, то оно называется сложным. В противном случае

состояние простое. Основной особенностью сложных состояний является то, что наличие дуги, исходящей из такого состояния, заменяет однотипные дуги из каждого вложенного состояния.

8. Каждая входная переменная и каждое выходное воздействие являются методами соответствующего объекта управления, которые реализуются вручную на целевом языке программирования. Классы, соответствующие источникам событий, также реализуются вручную.
9. Использование символьных обозначений в графах переходов позволяет весьма компактно описывать сложное поведение проектируемых систем. Смысл таких символов задает схема связей.

Программный пакет с открытым кодом *UniMod* (**http://unimod.sf.net**), созданный компанией *eVelopers* (**http://www.evelopers.com**), обеспечивает разработку и выполнение автоматно-ориентированных программ. Он выполнен как встраиваемый модуль (*plug-in*) для платформы *Eclipse* (**http://www.eclipse.org**) и позволяет создавать и редактировать *UML*-диаграммы классов и состояний, которые соответствуют схеме связей и графу переходов. Отметим, что платформа *Eclipse* разработана компанией *IBM*, в состав которой входит подразделение, создающее инструменты на основе UML-диаграмм.

Пакет *UniMod*, после создания диаграмм, предоставляет возможность:

- преобразовывать диаграммы в формат *XML;*
- исполнять полученное *XML*-описание с помощью интерпретатора, входящего в пакет *UniMod*;
- запускать диаграммы прямо из среды разработки *Eclipse* «в одно нажатие». При этом диаграммы сначала преобразуются в *XML*-формат, а затем запускается интерпретатор.

Проектирование программ с использованием пакета *UniMod* состоит в следующем:

- логика приложения описывается структурным конечным автоматом, заданным в виде набора указанных выше диаграмм, которые построены с применением *UML*-нотации;
- источники событий и объекты управления задаются кодом на целевом языке программирования.

В проекте используется язык *Java,* поэтому источникам событий и объектам управления соответствуют классы. Для этого языка в пакете *UniMod* реализован интерпретатор *XML*-описаний, которые пакет строит на основе указанных диаграмм. При запуске программы интерпретатор загружает в оперативную память *XML*-описание и создает экземпляры источников событий и объектов управления. В процессе работы источники формируют события и направляют их интерпретатору, который обрабатывает события в соответствии с логикой, описываемой диаграммой состояний. При этом он вызывает методы объектов управления, реализующие входные переменные и выходные воздействия. Все действия выполняемые интерпретатором заносятся в протокол (лог), в котором отображаются также изменения состояний автомата.

Далее, в соответствии с описанным выше автоматно-ориентированным подходом, и, используя пакет *UniMod,* спроектируем серверную и клиентскую части разрабатываемого приложения.

Серверное приложение

Схема связей автомата, реализующего серверное приложение, созданная с помощью пакета *UniMod*, приведена на рис. 3. Она построена следующим образом:

- классу *Server*, изображенному на диаграмме классов (рис. 2), соответствует автомат (*statemachine*) *A1*;
- класс *ServerConnector*, показанный на рис. 2, преобразован в два класса – источник событий (*eventprovider*) *ConnectorEventProvider,* который поставляет автомату *A1* сообщения, пришедшие от клиентских приложений, и объект управления *ServerConnector,* который реализует методы для посылки сообщений клиентским приложениям;
- класс *UserDatabase* перенесен с диаграммы классов (рис. 2) на схему связей без изменений, так как он пассивен и не может воздействовать на автомат;
- классу *Keyboard*, заданному на рис. 2, на схеме связей соответствует источник событий *Keyboard,* который поставляет автомату *A1* события о нажатии клавиш на клавиатуре.

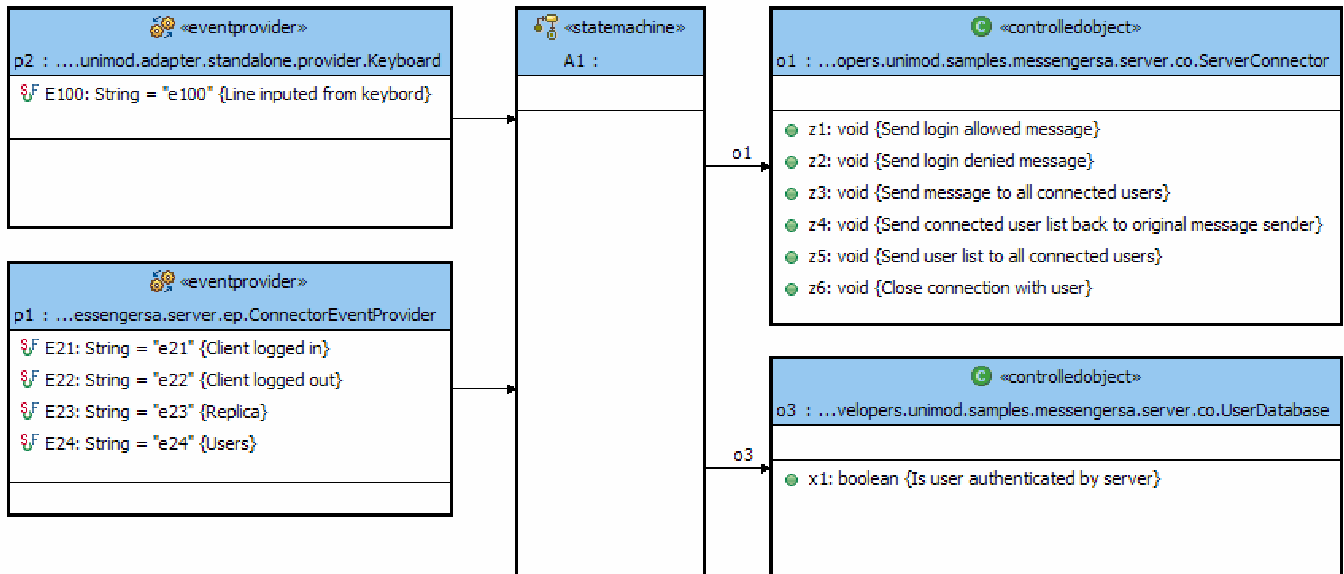

Рис. 3. Схема связей серверного автомата *A1*

Для реализации входных переменных и выходных воздействий необходимо знать их функциональность и данные, которые они получают или заносят в общий поток данных [6, 7]. В данной работе этот поток управляется диаграммой состояний. При этом отметим, что в дальнейшем будем различать входные переменные и входные данные, выходные воздействия и выходные данные. Первые из них предназначены для управления, а вторые – для описания потока данных.

Поясним, как выполняется управление потоком данных. Модель поведения серверного приложения является событийной. Поэтому первоначальный источник данных – событие. Далее потребителями и поставщиками данных могут быть только методы объектов управления, соответствующие входным переменным и выходным воздействиям. Поток начинается с данных, пришедших с событием (выходные данные события), и завершается данными последнего воздействия (входные данные воздействия), выполненного автоматом при обработке события.

Общее *правило для формирования корректного потока данных* может быть сформулировать следующим образом: если для каждого перехода сформировать последовательность из события, входных переменных и выходных воздействий, записанных на переходе, то множество входных данных каждого элемента последовательности должно включаться в объединение множеств выходных данных всех предшествующих элементов. Это правило является ограничением при проектировании.

Отметим, что для класса *A1* код на языке *Java* отсутствует, но для него генерируется *XML*-описание, которое в дальнейшем интерпретируется.

В табл. 2 приведены описания событий, а в табл. 3 – описания входных переменных и выходных воздействий серверного приложения. Отметим, что в колонках, соответствующих входным и выходным данным, в качестве типов данных используются стандартный *Java* класс *String* и контейнеры данных *Message*, *User*, определенные на диаграмме классов (рис. 1).

Таблица 2. События для серверного приложения

Событие	Описание	Аргументы (выходные данные) в формате имя:тип
ConnectorEventProvider		
e21	Запрос входа в систему	MESSAGE: Message
e22	Уведомление о выходе из системы	MESSAGE: Message
e23	Получен текст	MESSAGE: Message
e24	Запрос на получение списка пользователей	USER: User
Keyboard		
e100	Введена строка с клавиатуры	INPUT: String

Таблица 3. Входные переменные и выходные воздействия серверного приложения

Воздействия	Описание	Входные данные в формате имя:тип	Выходные данные в формате имя:тип
ServerConnector (o1)			
z1	Послать сообщение, подтверждающее разрешение входа в систему	MESSAGE: Message	
z2	Послать сообщение, запрещающее вход в систему	MESSAGE: Message REASON: String	
z3	Переслать текст всем присоединенным пользователям	MESSAGE: Message	
z4	Послать список пользователей инициатору сообщения	MESSAGE: Message	
z5	Послать список пользователей всем присоединенным пользователям		
z6	Закрыть соединение с клиентом	MESSAGE: Message	
UserDatabase (o3)			
x1	Возвращает `true` в случае успешной авторизации пользователя, и `false` в противном случае. Если авторизация не удалась, также возвращает причину ошибки в виде выходного данного	MESSAGE: Message	REASON: String

Применяя эти обозначения, построим диаграмму состояний для автомата *A1* (рис. 4). При этом звездочка на диаграмме означает «любое событие».

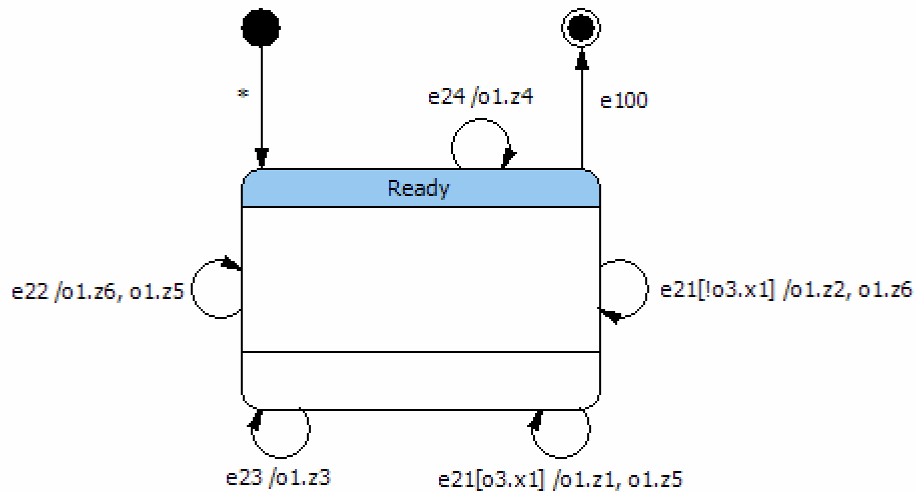

Рис. 4. Граф переходов серверного автомата *A1*

Можно показать, что все переходы этого графа удовлетворяют правилу построения потока данных, сформулированного выше. Например, для дуги, помеченной выражением `e21[!o3.x1]/o1.z2,o1.z6`, это правило справедливо. Это иллюстрируется следующим образом:

Действие над потоком данных	Содержимое потока данных после выполнения действия
Событие `e21` поставляет в поток данное `MESSAGE:Message`	`MESSAGE:Message`
Входное воздействие `o3.x1` получает из потока данное `MESSAGE:Message` и поставляет `REASON:Reason`	`MESSAGE:Message` `REASON:Reason`
Выходное воздействие `o1.z2` получает из потока данное `MESSAGE:Message` и `REASON:Reason`	`MESSAGE:Message` `REASON:Reason`
Выходное воздействие `o1.z2` получает из потока данное `MESSAGE:Message`	`MESSAGE:Message` `REASON:Reason`

Отметим, что автомат *A1* является формальным описанием протокола взаимодействия клиента и сервера. Он имеет три состояния (рис. 4), из которых только одно рабочее.

Если бы разрабатывался протокол с большим числом рабочих состояний, то пришлось бы на серверной стороне иметь по одному экземпляру автомата для каждого присоединенного клиента. Это объясняется тем, что в любой момент времени каждый из присоединенных клиентов может находиться в состоянии, отличном от состояния другого клиента. При этом можно говорить не об отдельном экземпляре автомата для каждого присоединенного клиента, а об отдельной конфигурации автомата, определяемой состоянием, в котором автомат в данный момент находится. В этом случае при появлении события от клиента серверному автомату помимо этого события передается также и состояние клиента.

С помощью пакета *UniMod* на основе диаграмм на рис. 3, 4 можно автоматически построить *XML*-описание автомата *A1* (Листинг 1), которое в дальнейшем передается интерпретатору.

Листинг 1. *XML*-описание серверного автомата *A1*

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE model PUBLIC "-//eVelopers Corp.//DTD State machine model V1.0//EN"
       "http://www.evelopers.com/dtd/unimod/statemachine.dtd">
<model name="A2_model">
       <controlledObject name="o1"
              class="com.evelopers.unimod.samples.messengersa.server.co.ServerConnector"/>
       <controlledObject name="o3"
              class="com.evelopers.unimod.samples.messengersa.server.co.UserDatabase"/>

       <eventProvider name="p1"
       class="com.evelopers.unimod.samples.messengersa.server.ep.ConnectorEventProvider">
              <association targetRef="A1"/>
       </eventProvider>
       <eventProvider name="p2"
              class="com.evelopers.unimod.adapter.standalone.provider.Keyboard">
              <association targetRef="A1"/>
       </eventProvider>

       <rootStateMachine>
              <stateMachineRef name="A1"/>
       </rootStateMachine>

       <stateMachine name="A1">
              <association targetRef="o1" supplierRole="o1"/>
              <association targetRef="o3" supplierRole="o3"/>

              <state name="TOP" type="NORMAL">
                     <state name="Ready" type="NORMAL"/>
                     <state name="EndState1" type="FINAL"/>
                     <state name="StartState1" type="INITIAL"/>
              </state>

              <transition name="" sourceRef="Ready" targetRef="Ready" event="e22">
                     <outputAction ident="o1.z6"/>
                     <outputAction ident="o1.z5"/>
              </transition>

              <transition name="" sourceRef="Ready" targetRef="Ready" event="e21"
                     guard="!o3.x1">
                     <outputAction ident="o1.z2"/>
                     <outputAction ident="o1.z6"/>
              </transition>

              <transition name="" sourceRef="Ready" targetRef="EndState1"
                     event="e100"/>

              <transition name="" sourceRef="Ready" targetRef="Ready" event="e23">
                     <outputAction ident="o1.z3"/>

              </transition>
              <transition name="" sourceRef="Ready" targetRef="Ready" event="e24">
                     <outputAction ident="o1.z4"/>
              </transition>

              <transition name="" sourceRef="Ready" targetRef="Ready" event="e21"
                     guard="o3.x1">
                     <outputAction ident="o1.z1"/>
                     <outputAction ident="o1.z5"/>
              </transition>
```

```
            <transition name="" sourceRef="StartState1" targetRef="Ready"/>
        </stateMachine>
</model>
```

Это описание изоморфно указанным диаграммам. В нем сначала описаны объекты управления, затем источники событий, после этого имя головного автомата (в данном случае – *A1*), а в конце приведено описание графа переходов автомата *A1*.

Этот листинг является описанием **поведения** серверного приложения. Поэтому о нем, можно говорить, как об **автоматной программе**, которой поставляются события от источников событий и из которой вызываются методы объектов управления. В силу изоморфизма и благодаря автоматической генерации *XML*-описания, диаграммы, изображенные на рис. 3, 4, можно считать **автоматной графической программой**.

В случае если *Java*-классы, соответствующие источникам событий и объектам управления, уже реализованы, то существует возможность «запустить» диаграмму состояний прямо из среды разработки. На рис. 5 показана среда разработки, в которой запущено серверное приложение.

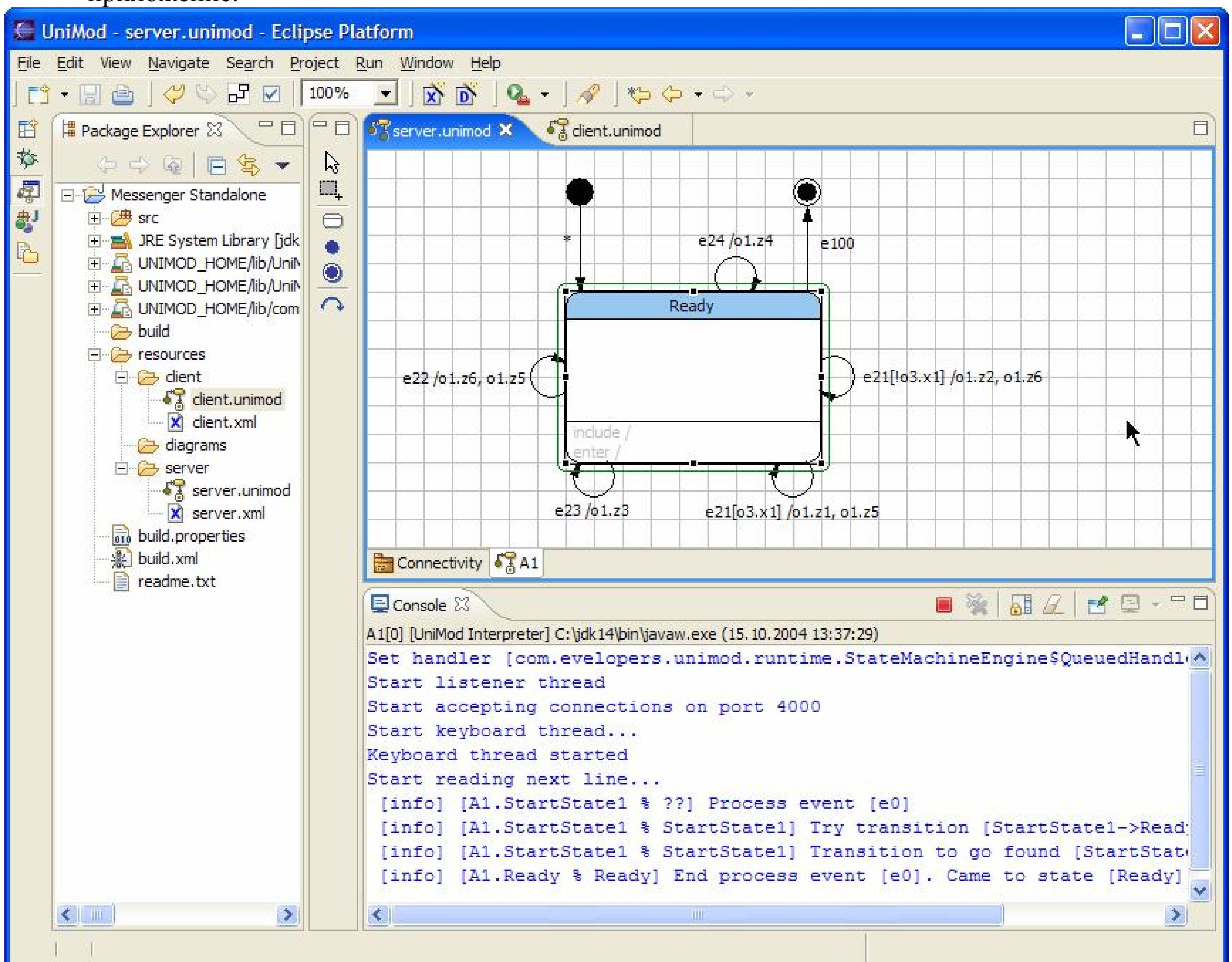

Рис. 5. Запуск серверного приложения из среды разработки

В нижней части этого рисунка показан лог работы автомата *A1*. Протокол имеет следующий формат:

```
[StateMachine.ActiveState % StateBeingProcessed] Message
```

В этом формате применяются следующие обозначения:

- `StateMachine` – название автомата;
- `ActiveState` – текущее состояние автомата;
- `StateBeingProcessed` – состояние, переход из которого в данный момент пытается осуществить автомат. Это состояние может не совпадать с текущим, если из состояния, в котором находится автомат, выполнен переход в сложное состояние.

Из анализа протокола следует, что после запуска интерпретатор передал автомату событие `e0`, которое уведомляет о начале работы. После этого автомат перешел в состояние `Ready`. Это активное состояние автоматически выделяется на диаграмме состояний *рамкой*.

Далее описан процесс проектирования клиентского приложения.

Клиентское приложение

Окно пользовательского интерфейса представлено на рис. 6. Ссылками показаны события, порождаемые кнопками *Send* и *Connect*.

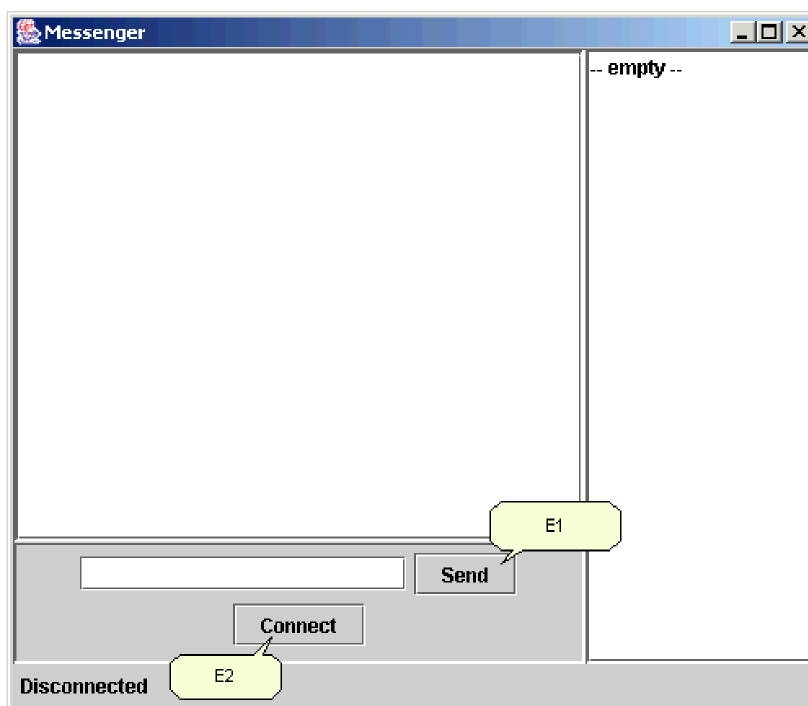

Рис. 6. Пользовательский интерфейс клиентского приложения

Схема связей автомата *A2*, реализующего поведение клиентского приложения, представлена на рис. 7. Она построена следующим образом:

- классу *Client*, приведенному на рис. 2, соответствует автомат *A2*;
- класс *ClientConnector*, определенный на рис. 2, разбит на два класса – источник событий *ConnectorEventProvider,* поставляющий автомату *A2* сообщения, пришедшие от сервера, и объект управления *ClientConnector,* реализующий методы для посылки сообщений серверу;
- класс *Screen* также разбит на два класса – источник событий *ScreenEventProvider,* который поставляет автомату *A2* события (нажатие кнопок) от пользовательского интерфейса, и объект управления *Screen,* реализующий методы для управления окном пользовательского интерфейса;
- класс *Config* перенесен из диаграммы классов (рис. 2) на схему связей без изменений, так как он пассивен и не может воздействовать на автомат.

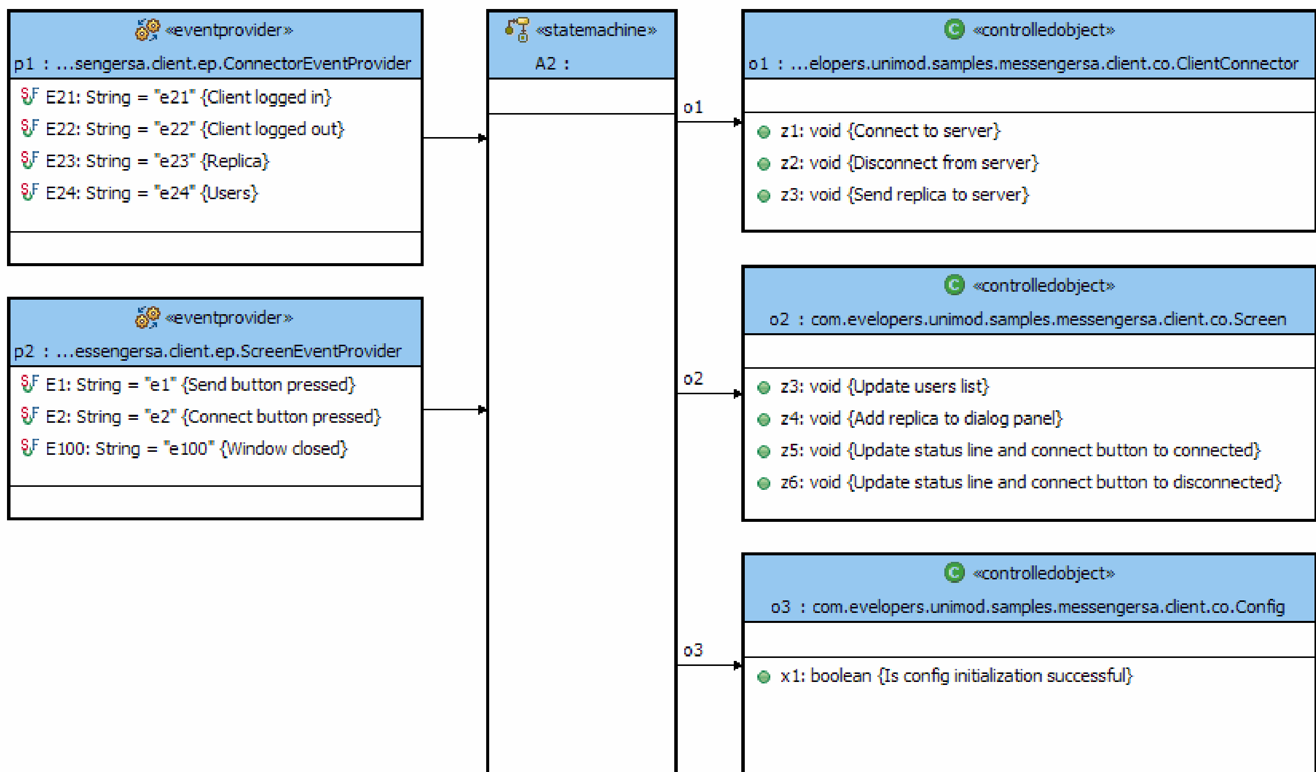

Рис. 7. Схема связей клиентского автомата *A2*

В табл. 4 приведено описание событий, воздействующих на автомат *A2*, а в табл. 5 – описание входных переменных и выходных воздействий, реализуемых объектами управления, связанными с этим автоматом. Также как и в описании серверного приложения, в колонках, описывающих входные и выходные данные, в качестве типов данных используются стандартный *Java* класс *String* и контейнеры данных *Replica*, *User*, определенные на рис. 1.

Таблица 4. События для клиентского приложения

Событие	Описание	Аргументы (выходные данные) в формате имя:тип
ConnectorEventProvider		
e21	Получено сообщение, разрешающее вход в систему	
e22	Получено сообщение, запрещающее вход в систему или уведомление о выходе сервера из системы	REASON: String
e23	Получен текст	REPLICA: Replica
e24	Получен список пользователей	USERS: User []
ScreenEventProvider		
e1	Нажата кнопка *Send*	REPLICA: String
e2	Нажата кнопка *Connect*	
e100	Запрос на закрытие окна	

Таблица 5. Входные переменные и выходные воздействия клиентского приложения

Воздействия	Описание	Входные данные в формате имя:тип	Выходные данные в формате имя:тип
ClientConnector (o1)			
z1	Присоединится к серверу		
z2	Отсоединится от сервера		
z3	Послать текст всем пользователям	REPLICA: String	
Screen (o2)			
z3	Обновить список подключенных пользователей	USERS: User []	
z4	Отобразить текст сообщения	REPLICA: Replica	
z5	Изменить надпись *Connect* на *Disconnect* на кнопке. Обновить строку состояния окна		
z6	Изменить надпись *Disconnect* на *Connect* на кнопке. Обновить строку состояния окна	REASON: String	
Config (o3)			
x1	Выполняет конфигурирование клиентского приложения. Возвращает `true` ,если конфигурирование произведено успешно и `false` в противном случае		

На рис. 8 показан граф переходов автомата *A2*.

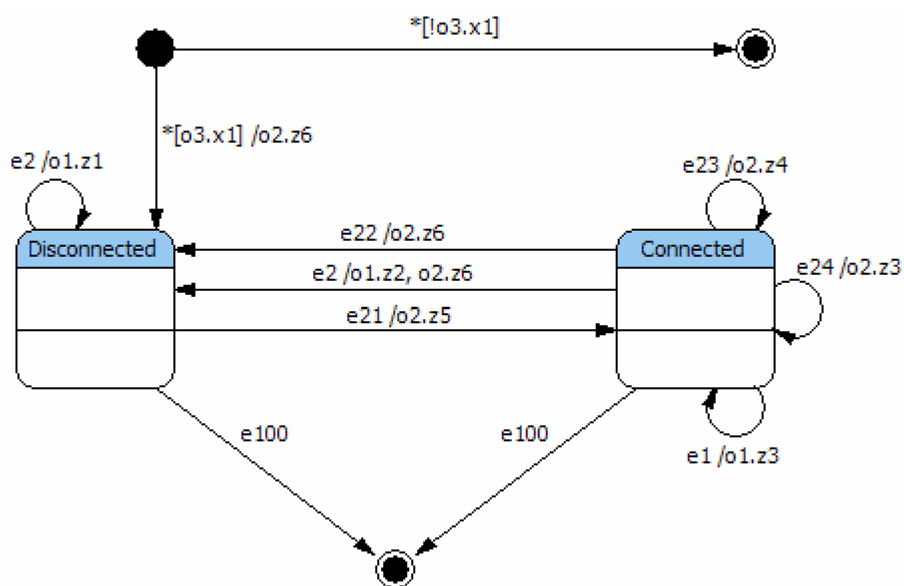

Рис. 8. Граф переходов клиентского автомата *A2*

Листинг 2 содержит *XML*-описание диаграмм, изображенных на рис. 7, 8. Данное описание изоморфно указанным диаграммам.

Листинг 2. *XML*-описание клиентского автомата *A2*

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE model PUBLIC "-//eVelopers Corp.//DTD State machine model V1.0//EN"
       "http://www.evelopers.com/dtd/unimod/statemachine.dtd">
<model name="A1_model">
       <controlledObject name="o1"
              class="com.evelopers.unimod.samples.messengersa.client.co.ClientConnector"/>
       <controlledObject name="o2"
              class="com.evelopers.unimod.samples.messengersa.client.co.Screen"/>
       <controlledObject name="o3"
              class="com.evelopers.unimod.samples.messengersa.client.co.Config"/>

       <eventProvider name="p1"
       class="com.evelopers.unimod.samples.messengersa.client.ep.ConnectorEventProvider">
              <association targetRef="A2"/>
       </eventProvider>
       <eventProvider name="p2"
       class="com.evelopers.unimod.samples.messengersa.client.ep.ScreenEventProvider">
              <association targetRef="A2"/>
       </eventProvider>

       <rootStateMachine>
              <stateMachineRef name="A2"/>
       </rootStateMachine>

       <stateMachine name="A2">
              <association targetRef="o1" supplierRole="o1"/>
              <association targetRef="o2" supplierRole="o2"/>
              <association targetRef="o3" supplierRole="o3"/>

              <state name="TOP" type="NORMAL">
                     <state name="Disconnected" type="NORMAL"/>
                     <state name="Connected" type="NORMAL"/>
                     <state name="Start" type="INITIAL"/>
                     <state name="Final" type="FINAL"/>
                     <state name="s1" type="FINAL"/>
              </state>

              <transition name="" sourceRef="Disconnected" targetRef="Final"
                     event="e100"/>
              <transition name="" sourceRef="Disconnected" targetRef="Disconnected"
                     event="e1">
                     <outputAction ident="o1.z1"/>
                     <outputAction ident="o1.z3"/>
              </transition>
              <transition name="" sourceRef="Disconnected" targetRef="Disconnected"
                     event="e2">
                     <outputAction ident="o1.z1"/>
              </transition>
              <transition name="" sourceRef="Disconnected" targetRef="Connected"
                     event="e21">
                     <outputAction ident="o2.z5"/>
              </transition>
              <transition name="" sourceRef="Connected" targetRef="Final"
                     event="e100"/>
              <transition name="" sourceRef="Connected" targetRef="Connected"
                     event="e24">
                     <outputAction ident="o2.z3"/>
              </transition>
              <transition name="" sourceRef="Connected" targetRef="Connected"
                     event="e23">
```

```
                <outputAction ident="o2.z4"/>
            </transition>
            <transition sourceRef="Connected" targetRef="Connected" event="e1">
                <outputAction ident="o1.z3"/>
            </transition>
            <transition name="" sourceRef="Connected" targetRef="Disconnected"
                event="e22">
                <outputAction ident="o2.z6"/>
            </transition>
            <transition name="" sourceRef="Connected" targetRef="Disconnected"
                event="e2">
                <outputAction ident="o1.z2"/>
                <outputAction ident="o2.z6"/>
            </transition>
            <transition name="" sourceRef="Start" targetRef="s1" guard="!o3.x1"/>
            <transition name="" sourceRef="Start" targetRef="Disconnected"
                guard="o3.x1">
                <outputAction ident="o2.z6"/>
            </transition>
        </stateMachine>
</model>
```

На рис. 9 показана среда разработки с запущенным клиентским приложением. Из анализа протокола следует, что после запуска приложение перешло в состояние *Disconnected.* Это состояние автоматически выделено *рамкой* на диаграмме состояний.

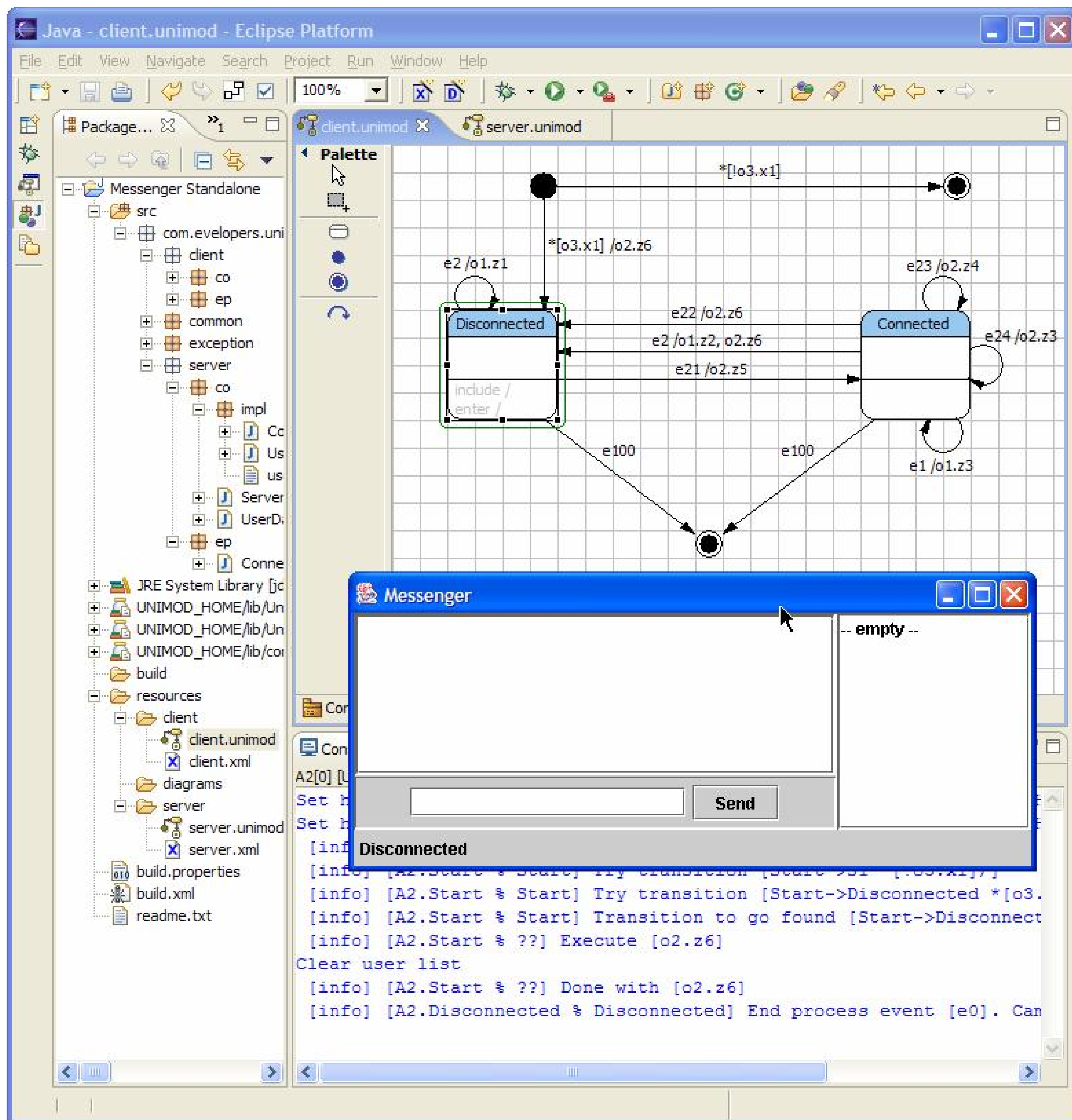

Рис. 9. Клиентское приложение, запущено из среды разработки

На рис. 10 показано два запущенных из среды разработки клиентских приложения и протокол серверного приложения. Клиентские приложения присоединены к серверу и обмениваются сообщениями. Из анализа серверного протокола следует, что последним было обработано событие `e23` – получен текст. В протоколе также отображено, что при обработке этого события выполнено выходное воздействие `o1.z3` – пересылка текста присоединенным пользователям. В окнах клиентских приложений видны реплики, которыми обменивались пользователи, пересылая сообщения друг другу через сервер. На диаграмме состояний также видно, что клиентские приложения находятся в состоянии *Connected* – оно выделено *рамкой*.

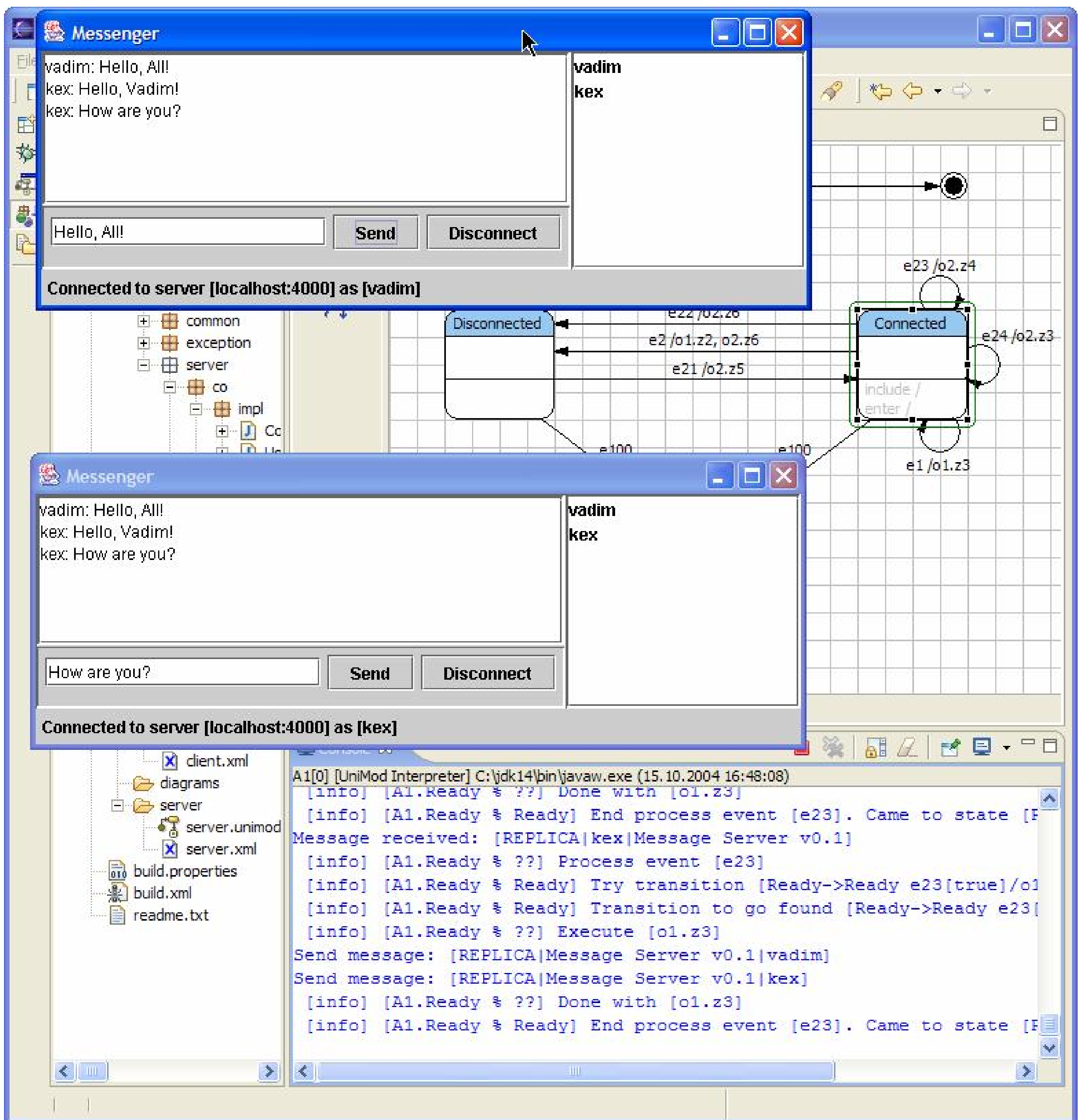

Рис. 10. Одна из рабочих ситуаций с двумя запущенными клиентскими приложениями

Обеспечение возможности модификации системы

Опишем, как модифицируется система, при необходимости. *Предположим, что весь код, реализующий источники событий и объекты управления, уже написан и отлажен*. Предположим также, что появилась необходимость внести следующие изменения:

- если клиент не подсоединен к серверу, то после нажатия кнопки *Send* сначала автоматически должно выполняться подсоединение к серверу, а потом осуществлена отправка сообщения;
- нельзя закрыть клиентское окно, пока клиент подсоединен к серверу.

Проанализировав табл. 4 и 5, можно сделать вывод, что написанный код изменять не следует, а требуется изменить лишь граф переходов автомата *A2*. При этом в состоянии *Disconnected* необходимо добавить петлю, помеченную символами `e1/o1.z1,o1.z3` (при получении события «послать сообщение», необходимо присоединиться к серверу и послать сообщение), и убрать переход из состояния *Connected* в финальное состояние (запретить завершение работы приложения при получении события «запрос на закрытие окна»).

На рис. 11 представлена модифицированная схема графа переходов клиентского автомата *A2*.

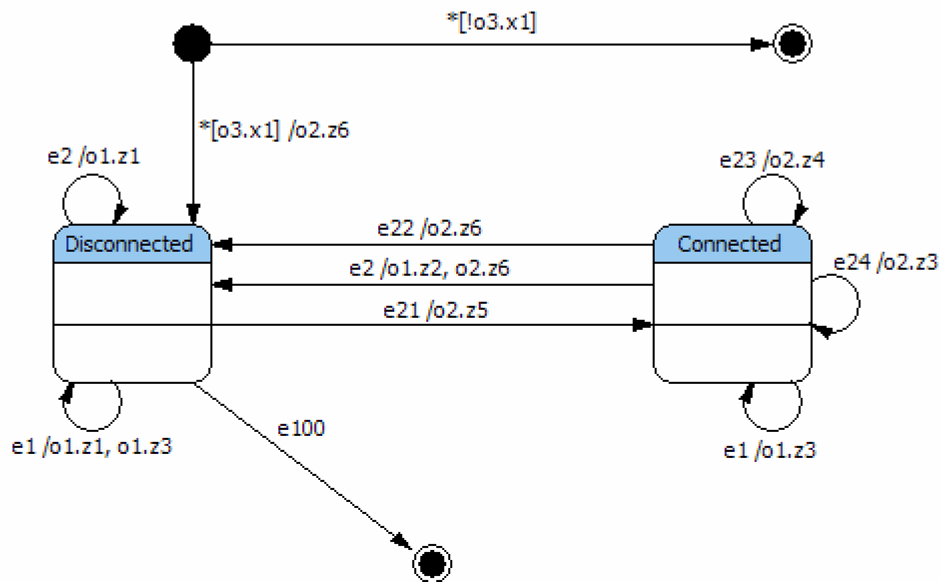

Рис. 11. Преобразованный граф переходов автомата *A2*

После этого достаточно сгенерировать новое *XML*-описание автомата *A2*, и система готова к работе.

Заключение

В работе приведен пример построения простого клиент-серверного приложения на основе автоматного подхода. При этом основой для разработки такого приложения является описание протокола взаимодействия между клиентами и сервером. Используя этот протокол, строятся графы переходов рассматриваемых приложений, а также описываются потоки данных.

Применяемый подход базируется на совместном использовании *SWITCH*-технологии и *UML*-диаграмм. Диаграммы создаются в свободно распространяемом пакете для автоматно-ориентированного программирования *UniMod*.

На основе анализа предметной области строится диаграмма классов системы, выделяются объекты и управляющие ими автоматы.

Схема связей каждого автомата изображается с помощью нотации диаграммы классов, а его граф переходов – в виде диаграммы состояний. С помощью пакета *UniMod* эти диаграммы преобразуются в *XML*-описание, которое интерпретируется после «ручной реализации» источников событий и объектов управления.

Правильность работы приложения демонстрируется с помощью протоколирования, выполняемого в автоматной терминологии, и подсветки активных состояний на графе переходов, осуществляемой автоматически.

Как показано в разделе «Обеспечение модификации системы», при наличии библиотеки источников событий и объектов управления для некоторой предметной области, можно обойтись без ручного программирования.

Исходные тексты описанного клиент-серверного приложения и пакет для автоматно-ориентированного программирования *UniMod* доступны по адресу **http://unimod.sf.net/**.

В заключении отметим, что рассмотренный пример является весьма простым, однако у авторов есть основания предполагать, что он может быть распространен и на более сложные системы.

Источники

1. *Грэхем И.* Объектно-ориентированные методы. Принципы и практика. М.: Вильямс, 2004. 768 с.
2. *Коберн А.* Современные методы описания требований к системам. М.: Лори, 2002. 262 с.
3. *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Приемы объектно-ориентированного программирования. Паттерны проектирования. СПб.: Питер, 2001. 324 с.
4. *Шалыто А.А., Туккель Н.И.* Танки и автоматы //BYTE/Россия. 2003. № 2, с. 69-73. **http://is.ifmo.ru/** (раздел «Статьи»).
5. *Шалыто А.А., Туккель Н.И.* SWITCH-технология — автоматный подход к созданию программного обеспечения "реактивных" систем //Программирование. 2001. № 5, с. 45-62. **http://is.ifmo.ru/** (раздел «Статьи»).
6. *Гайсарян С. С.* Объектно-ориентированные технологии проектирования прикладных программных систем. (**http://www.citforum.ru/programming/oop_rsis/glava4_2.shtml**, **http://www.citforum.ru/programming/oop_rsis/glava2_6_1.shtml**)
7. *Yourdon E.* Modern Structured Analysis. Prentice Hall PTR, 1988. 688 p.