

Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

«ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ» (ТУСУР)

Кафедра автоматизированных систем управления (АСУ)

УДК 004.75

В.Г. Резник

РАСПРЕДЕЛЕННЫЕ ВЫЧИСЛИТЕЛЬНЫЕ СЕТИ

Учебное пособие

Томск - 2019

УДК 004.75

Резник В.Г.

Распределенные вычислительные сети. Учебное пособие / В.Г. Резник. – Томск, ТУСУР, 2019. – 223 с.

В учебном пособии рассмотрены основные современные технологии организации распределенных вычислительных сетей, которые уже получили достаточно широкое распространение и подкреплены соответствующими инструментальными средствами реализации распределенных приложений. Представлены основные подходы к распределенной обработке информации. Проводится обзор организации распределенных вычислительных систем: методы удалённых вызовов процедур, многослойные клиент-серверные системы, технологии гетерогенных структур и одноранговых вычислений. Приводится описание концепции GRID-вычислений и сервис-ориентированный подход к построению распределенных вычислительных систем. Рассматриваемые технологии подкрепляется описанием инструментальных средств разработки программного обеспечения, реализованных на платформе языка Java.

Пособие предназначено для студентов бакалавриата по направлению 09.03.01 «Информатика и вычислительная техника» при изучении курсов «Вычислительные системы и сети» и «Распределенные вычислительные системы».

УДК 004.75

Оглавление

Введение.....	6
1 Тема 1. Введение в теорию вычислительных сетей.....	8
1.1 Общая классификация систем обработки данных.....	11
1.1.1 Сосредоточенные системы.....	12
1.1.2 Распределенные системы.....	14
1.1.3 Распределенные вычислительные сети.....	18
1.2 Сетевые объектные системы.....	20
1.2.1 Классические приложения модели OSI.....	21
1.2.2 Распределенная вычислительная среда (DCE).....	22
1.2.3 Технология CORBA.....	24
1.2.4 Удалённый вызов методов.....	25
1.3 Сервис-ориентированные технологии.....	26
1.3.1 Функции и сервисы.....	27
1.3.2 Системы <i>middleware</i>	28
1.3.3 Сервисные шины предприятий.....	29
1.4 Виртуальные системы.....	30
1.4.1 Виртуальные машины.....	30
1.4.2 Виртуализация вычислительных комплексов на уровне ОС...31	
1.4.2 Виртуализация ПО на уровне языка.....	32
1.4.3 Виртуальная машина языка <i>Java</i>	33
1.5 Итоги теоретических построений.....	35
Вопросы для самопроверки.....	37
2 Тема 2. Инструментальные средства языка <i>Java</i>.....	38
2.1 Общее описание инструментальных средств языка.....	40
2.1.1 Инструментальные средства командной строки.....	41
2.1.2 Пакетная организация языка <i>Java</i>	43
2.1.3 Инструментальные средства <i>Eclipse</i>	47
2.2 Классы и простые типы данных.....	51
2.2.1 Операторы и простые типы данных.....	52
2.2.2 Синтаксис определения классов.....	53
2.2.3 Синтаксис и семантика методов.....	54
2.2.4 Синтаксис определения интерфейсов.....	56
2.2.5 Объекты и переменные.....	56
2.3 Управляющие операторы языка.....	59
2.4 Потоки ввода-вывода.....	61
2.4.1 Стандартный ввод-вывод.....	61
2.4.2 Классы потоков ввода.....	64
2.4.3 Классы потоков вывода.....	66
2.5 Управление сетевыми соединениями.....	69
2.5.1 Адресация на базе класса <i>InetAddress</i>	69
2.5.2 Адресация на базе <i>URL</i> и <i>URLConnection</i>	71
2.5.3 Сокеты протокола <i>TCP</i>	73

2.5.4 Сокеты протокола UDP.....	74
2.5.5 Простейшая задача технологии клиент-сервер.....	75
2.6 Организация доступа к базам данных.....	82
2.6.1 Инструментальные средства СУБД Apache Derby.....	82
2.6.2 SQL-запросы и драйверы баз данных.....	86
2.6.3 Типовой пример выборки данных.....	89
Вопросы для самопроверки.....	97
3 Тема 3. Объектные распределенные системы.....	98
3.1 Брокерные архитектуры.....	99
3.1.1 Вызов удалённых процедур.....	101
3.1.2 Использование удалённых объектов.....	103
3.2 Технология CORBA.....	105
3.2.1 Брокерная архитектура CORBA.....	105
3.2.2 Проект серверной части приложения NotePad.....	108
3.2.3 Проект клиентской части приложения Example12.....	114
3.2.4 Генерация распределенного объекта OrbPad.....	118
3.2.5 Реализация серверной части ORB-приложения.....	125
3.2.6 Реализация клиентской части ORB-приложения.....	130
3.3 Технология RMI.....	136
3.3.1 Интерфейсы удалённых объектов.....	137
3.3.2 Реализация RMI-сервера.....	138
3.3.3 Реализация RMI-клиента.....	144
3.3.4 Завершение реализации RMI-проекта.....	148
Вопросы для самопроверки.....	151
4 Тема 4. Web-технологии распределенных систем.....	152
4.1 Описание технологии web.....	153
4.1.1 Унифицированный идентификатор ресурсов (URI).....	153
4.1.2 Общее представление ресурсов (HTML).....	155
4.1.3 Протокол передачи гипертекста (HTTP).....	156
4.2 Модели «Клиент-сервер».....	158
4.2.1 Распределение приложений по уровням.....	159
4.2.2 Типы клиент-серверной архитектуры.....	160
4.3 Технология Java-сервлетов.....	162
4.3.1 Классы Servlet и HttpServlet.....	165
4.3.2 Контейнер сервлетов Apache Tomcat.....	167
4.3.3 Диспетчер запросов - RequestDispatcher.....	176
4.3.4 Технология JSP-страниц.....	179
4.3.5 Модель MVC.....	188
Вопросы для самопроверки.....	197
5 Тема 5. Сервис-ориентированные архитектуры.....	198
5.1 Концепция SOA.....	199
5.1.1 Связывание распределённых программных систем.....	200
5.1.2 Web-сервисы первого и второго поколений.....	201
5.1.3 Брокерные архитектуры web-сервисов.....	203
5.2 Частные подходы к реализации сервисных технологий.....	206

5.2.1 Технология одноранговых сетей.....	206
5.2.2 Технологии GRID.....	208
5.2.3 Облачные вычисления и «виртуализация».....	209
Вопросы для самопроверки.....	211
Заключение.....	212
Список использованных источников.....	215
Алфавитный указатель.....	219

Введение

В современном обществе сохраняется устойчивая тенденция к интенсивному развитию информационных технологий. В этот процесс вовлекается все большее количество материальных ресурсов и жителей планеты. Компьютеры и их программное обеспечение стали универсальным инструментом во многих сферах человеческой деятельности. Постоянно внедряются все более сложные автоматизированные комплексы и системы. Непрерывно растёт потребность в специалистах направления «Информатика и вычислительная техника».

Настоящее учебное пособие предназначено для студентов старших курсов бакалавриата, которые уже освоили базовые курсы по основам вычислительной техники и программированию, изучили дисциплины, связанные с хранением информации в базах данных и информационной работы в сетях ЭВМ. В этих условиях особую актуальность приобретают знания о сложных интегрированных системах, включающих в себя достижения уже изученных отраслей науки и создающих новый уровень концептуальных представлений указанного научного направления. Такие знания и идеи заложены в концепцию систем, именуемую автором пособия как «*Распределенные вычислительные сети*» (РВ-сети). К ним относятся различные подходы к построению распределенных приложений, среди которых следует выделить: объектные распределенные системы, GRID-вычисления, сервис-ориентированные архитектуры и облачные вычисления.

В предлагаемом учебном пособии учебный материал изложен последовательно и подкреплён примерами реализации элементов РВ-сетей, обеспечивающих студенту начальные навыки их создания. Все практические примеры созданы с применением единых инструментальных средств программирования, основанных на языке Java. Это, по мнению автора, создаёт наиболее простое и целостное представление бакалавра о предмете обучения и ориентирует его на последующее практическое закрепление изученного материала.

Общая структура учебного пособия включает пять глав, раскрывающих следующую тематику:

Тема 1. Введение в теорию распределенных вычислительных сетей.

Тема 2. Инструментальные средства языка Java.

Тема 3. Объектные распределенные системы.

Тема 4. Web-технологии распределенных систем.

Тема 5. Сервис-ориентированные архитектуры.

Первая глава посвящена общим теоретическим вопросам построения распределенных вычислительных сетей. Здесь обсуждаются базовые концепции, изложенные в источниках [1-5], приводится анализ классификации систем обработки данных (СОД), вводятся необходимые определения и даётся начальный обзор рассматриваемых в учебном пособии технологий.

Глава 2 полностью посвящена базовым основам языка Java, включая средства управления сетевыми соединениями и организацией доступа к базам данных.

Уровень изложения учебного материала по данной теме выбран с учётом знания обучающимся основ программирования на языках C/C++, а также опыт работы с сетями и базами данных, полученного при изучении соответствующих дисциплин. Практические примеры данной главы ориентированы на последующее применение полученных навыков в процессе изучения материала последующих глав.

Учебный материал, изложенный в темах 3-4, опирается на теоретические представления обозначенные в теме 1. Каждая из последних трёх тем раскрывает соответствующий набор технологий, обеспечивающих построение распределенных приложений.

Глава 3 раскрывает тему объектных распределенных систем, заложивших теоретическую и практическую основу изучаемой дисциплины, отделив ее от набора классических технологий и обеспечив начальную стандартизацию указанного направления. Результаты этого подхода представлены технологиями CORBA и RMI.

Глава 4 посвящена тематике web-технологий, которые с момента своего появления претендовали только на интеграцию задач информационного документооборота, но без которых сегодня невозможно представить развитие любых общественных или производственных систем. Во многом это связано и с практическими достижениями языка Java, обеспечившими реализацию технологий сервлетов и JSP-страниц, а также успешную разработку программного обеспечения уровня Enterprise Edition.

Глава 5 кратко описывает концепцию сервис-ориентированных архитектур, в основу которых положена концепция SOA, а также подходы частных решений, таких как облачные вычисления и GRID.

Данное учебное пособие не имеет целью дать описание всех существующих технологий создания распределенных вычислительных сетей, тем более, что указанное направление развивается очень быстро и является неустойчивым в приоритетах своего развития. В любом случае, изложенный учебный материал является достаточным для начального изучения предмета, ограниченного уровнем бакалавриата.

1 Тема 1. Введение в теорию вычислительных сетей

Терминологические разночтения в учебной литературе и научных работах, связанных с распределенной обработкой информации, затрудняют освоение соответствующих дисциплин. Назначение данной главы — уточнить значение используемых терминов, а также ограничить рамки изучаемого материала.

В фундаментальном курсе по архитектуре и структуре современных компьютерных средств, написанных С.А. Орловым и Б.Я. Цилькером [2], даются следующие определения понятиям «вычислительная машина» и «вычислительная система»: «... *Вычислительная машина (ВМ)* — совокупность технических средств, создающая возможность проведения обработки информации (данных) и получение результата в необходимой форме. Под техническими средствами понимают все оборудование, предназначенное для автоматизированной обработки данных. Как правило, в состав ВМ входит и системное программное обеспечение. ВМ, основные функциональные устройства которой выполнены на электронных компонентах, называют электронной вычислительной машиной (ЭВМ).

В свою очередь, *вычислительную систему (ВС)* стандарт ISO/IEC 2382/1-93 определяет как одну или несколько вычислительных машин, периферийное оборудование и программное обеспечение, которые выполняют обработку данных. ...».

Продолжая следовать рассуждениям учебника [2], мы приходим к понятию «Архитектура компьютера» (Computer architecture), а затем — к понятию уровни детализации структуры ВМ, например, так как показано на рисунке 1.1.

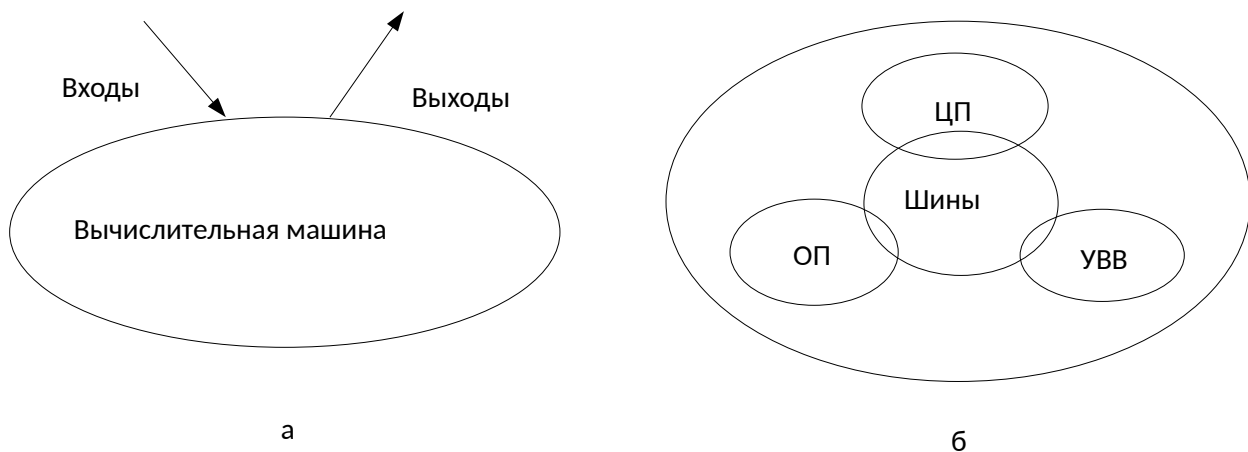


Рисунок 1.1 — Уровни детализации вычислительной машины [2]: а — уровень «чёрного ящика»; б — уровень общей архитектуры

Первоначально введенные термины и обозначения не вызывают особых вопросов. Например, в левой части рисунка «Входы» и «Выходы» обозначают связь отдельной вычислительной машины с другими ВМ посредством каналов связи или сетевых устройств, что позволяет создавать разнообразные вычислительные системы. Элементы правой части рисунка вводят обозначения связанной между собой совокупности технических средств ВМ. Все это соответствует общепринятой логике рассуждений и не вызывают гностических (познавательных) противо-

речий.

Терминологические сложности начинаются с момента содержательного анализа понятия «*Вычислительные системы*». Здесь все уважаемые источники [1-5] дают различные толкования, которые если и приемлемы для подготовленного специалиста, но — не допустимы для учебного пособия уровня бакалавриата. Особую проблему вызывает толкование ВС (как определение макроуровня), которое затем детализируется и раскрывается по мере описания решаемой задачи. В результате образуется комбинаторный набор понятий, которые разрушают иерархическую зависимость используемых определений и создают гносеологические трудности у обучающихся. Примером такого определения является базовое понятие «*Распределенные вычислительные системы*» используемое в названии учебного пособия [5].

Чтобы устранить указанные выше недостатки, в данном учебном пособии используется терминология ориентированная на последний выпуск государственного стандарта: «ГОСТ 33707-2016 (ISO/IEC 2382:2015). Информационные технологии (ИТ). Словарь» [9], где базовыми понятиями макроуровня являются: «... 4.1270 **система обработки данных**: СОД: Система, выполняющая автоматизированную обработку данных и включающая аппаратные средства, программное обеспечение и соответствующий персонал (data processing system, computer system, computing system). 4.1271 **система обработки информации**; СОИ: Совокупность технических средств и программного обеспечения, а также методов обработки информации и действий персонала, обеспечивающая выполнение автоматизированной обработки информации (information processing system). ...».

Существенная разница между СОД и СОИ, сложно воспринимаемая на слух, состоит в том, что:

- **СОД** — выполняет автоматизированную обработку данных;
 - **СОИ** — только обеспечивает автоматизированную обработку информации.
-

Далее, в пределах изучаемого материала, системы не конкретизируются до различий между СОД и СОИ. Тем не менее, за основу классификации изучаемых систем выбрано понятие **СОД**, которое подробно описано в учебнике [4]. Кроме того, для большего соответствия стандарту [9, раздел 2], используется следующий «Список сокращений и условных обозначений»:

ЭВМ — электронно-вычислительная машина;
СУБД — система управления базами данных;

ПК — персональный компьютер;

ПЗУ — постоянное запоминающее устройство;

ОС — операционная система;

ОЗУ — оперативное запоминающее устройство;

ЛВС — локальная вычислительная сеть;

ИС — 1. интегральная схема; 2. информационная система; 3. интеллектуальная собственность;

ИИ — искусственный интеллект;

ЗУ — запоминающее устройство;

ВС — вычислительная система;

БД — база данных.

Чтобы в должной степени раскрыть подробности изучаемой предметной области, выделим основные подразделы данной главы, где излагаются главные теоретические положения, касающиеся вычислительных сетей:

- **Общая классификация систем обработки данных** представлена мнением А.М. Ларионова, С.А. Майорова и Г.И. Новикова, изложенным в материалах первой главы ученика [4] и положенным в основу изучаемой дисциплины. Это мнение сравнивается с принципами и парадигмами фундаментального труда Э. Таненбаума [3].
- **Сетевые объектные системы** отражают начальную стадию теоретического построения распределенных систем, появившихся в эпоху бурного развития сетевых технологий. Различные технологические аспекты этого подраздела подробно излагаются в главах 3 и 4 данного учебного пособия.
- **Сервис-ориентированные технологии** отражают текущие теоретические построения, положенные в основу теории и практики современных распределённых систем в качестве концепции SOA. Теоретические положения этой части кратко раскрываются в последней главе 5.
- **Виртуальные системы** отражают общий методологический принцип построения современных сложных систем. В этом плане, виртуальные системы связывают общие вопросы проектирования различных технических систем с инструментальными средствами их разработки. Конечным пунктом такого видения являются языковые и инструментальные средства Java, базовые основы которого излагаются во второй главе данного учебного пособия и последовательно распространяются на весь учебный материал последующих глав.

1.1 Общая классификация систем обработки данных

Предметная область изучаемой дисциплины находится в пространстве понятий, которые определяются термином «Системы обработки данных» (СОД). Для содержательного раскрытия этого термина воспользуемся классификацией Ларионова (А.М. Ларионов, С.А. Майоров и Г.И. Новиков) [4], которая представлена в общем виде на рисунке 1.2.

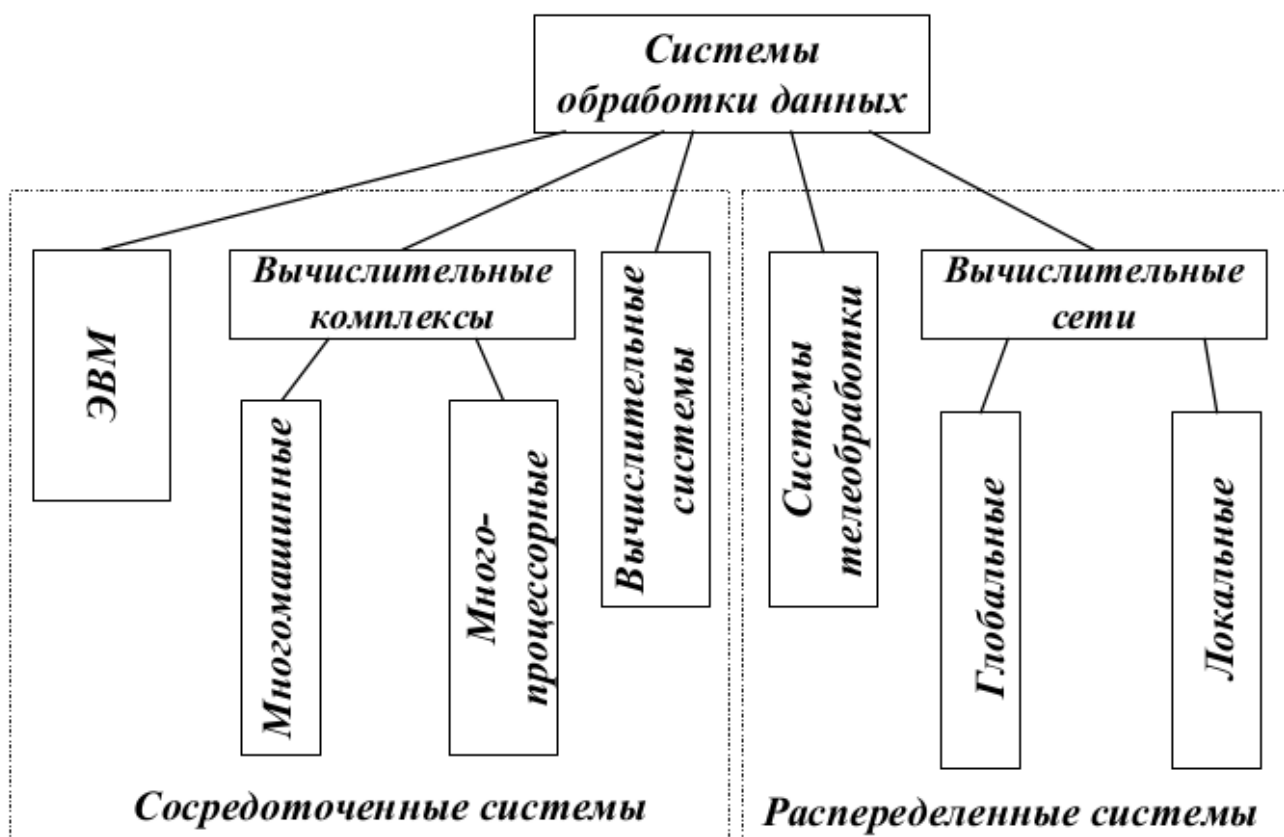


Рисунок 1.2 — Классификация СОД [4]

Представленная классификация снимает терминологические трудности определений, отмеченные во введении и начале данной главы. Хорошо видна иерархическая структура предложенной классификации, которая выделяет две группы систем: «Сосредоточенные системы» и «Распределенные системы». Достаточно чётко обозначена предметная область изучаемой дисциплины: «Вычислительные сети», а сами авторы так характеризуют предложенную классификацию [4, стр. 10]: «... СОД, построенные на основе отдельных ЭВМ, вычислительных комплексов и систем, образуют класс *сосредоточенных (централизованных) систем*, в которых вся обработка реализуется ЭВМ, вычислительным комплексом или специализированной системой. Системы телеобработки и вычислительные сети относятся к классу *распределенных систем*, в которых процессы обработки данных рассредоточены по многим компонентам. При этом системы телеобработки считаются распределенными в некоторой степени условно, поскольку

основные функции обработки данных здесь реализуются централизованно – в одной ЭВМ или вычислительном комплексе. ...».

Особое внимание уделяется *интерфейсам*, которые являются элементами сопряжения и подразделяются на: *параллельные, последовательные и связанные* интерфейсы. Для уточнения отметим [9]: «... 4.447 **интерфейс**: Совместно используемая граница между двумя функциональными единицами, определяемая различными функциональными характеристиками, параметрами физического соединения, параметрами взаимосвязи при обмене сигналами, а также другими характеристиками в зависимости от задаваемых требований (interface). **П р и м е ч а н и е** - Примерами интерфейсов являются RS232, RS422, RS485 и радиоинтерфейс. ...».

Следует отметить, что авторы учебника [4] являются специалистами очень высокого класса, участвовавшими в разработках большого количества отечественной цифровой вычислительной техники, включая многопроцессорный вычислительный комплекс (МПК) «Эльбрус».

Прежде чем перейти к рассмотрению компонент модели СОД отметим, что ЭВМ и другие выделенные системы включают не только аппаратные средства, показанные на рисунке 1.1б, но и программное обеспечение ОС, базовый состав которой адекватно соответствует изучаемой компоненте. Более того, современные ЭВМ фактически не выпускаются без программного обеспечения, которое включает ПО BIOS (UEFI) и firmware (программные «прошивки» аппаратной части ЭВМ). Этот факт хорошо известен студентам уже изучившим курс «*Операционные системы*». Например, дистрибутив ОС Arch Linux, установленный в апреле 2019 года, содержал ПО драйверов — порядка 86 МБ, а ПО firmware — 376 МБ.

1.1.1 Сосредоточенные системы

Следуя рассуждениям авторов учебника [4], выделяются: *ЭВМ*, «*Вычислительные комплексы*» и «*Вычислительные системы*».

ЭВМ — одномашинная СОД с традиционной однопроцессорной архитектурой и программным обеспечением ОС, допускающим развитие такой системы в плане установки и эксплуатации инструментального и прикладного ПО. Такой тип СОД хорошо известен студентам в плане изучения различных курсов, включая дисциплину «ЭВМ и периферийные устройства».

Вычислительные комплексы — СОД построенные по одному из двух принципов: «*Многомашинные вычислительные комплексы*» или «*Многопроцессорные вычислительные комплексы*». Основное внимание в таких архитектурах уделяется аппаратным средствам, которые обеспечивают распараллеливание вычислений, что в конечном итоге ориентировано на увеличение скорости работы приложений. Важной отличительной особенностью таких комплексов является наличие только базового программного обеспечения ОС, которое создаёт единую систему, скры-

вая детали аппаратной реализации.

Многомашинные вычислительные комплексы — разновидность систем, которые включают несколько ЭВМ и предназначены для повышения надёжности и производительности СОД. Они появились в начале 60-х годов и строились на основе высокоскоростных адаптеров, обеспечивая *прямую связь* между ЭВМ, или на основе внешних накопителей информации, обеспечивая *косвенную связь* между ЭВМ посредством магнитных дисков или магнитных лент.

Многопроцессорные вычислительные комплексы — разновидность систем, которые включают более одного процессора, но используют общую память СОД. Само взаимодействие процессоров и памяти осуществляется через специальную коммуникационную среду, определяющую различные качественные характеристики системы в целом.

Вычислительные системы — СОД настроенные на решение задач конкретной области применения. Другими словами, это — **Вычислительные комплексы**, дополненные прикладным программным обеспечением, специализированным для конкретной предметной области.

Таким образом, термин «*Вычислительные системы*» является эквивалентом термину «*Сосредоточенные системы*» и в таком виде используется не только в отдельных научных публикациях, но и в учебной литературе.

Чтобы обосновать заявленное утверждение, обратимся к другой классификации, известной как таксономия Флинна. Для этого процитируем, например, статью Википедии [11]: «... **Таксономия (Классификация) Флинна** (англ. *Flynn's taxonomy*) — общая классификация архитектур ЭВМ по признакам наличия параллелизма в потоках команд и данных. Была предложена Майклом Флинном в 1966 году и расширена в 1972 году. ... Все разнообразие архитектур ЭВМ в этой таксономии Флинна сводится к четырём классам:

- **ОКОД** — Вычислительная система с **о**диночным потоком **к**оманд и **о**диночным потоком **д**анных (SISD, single instruction stream over a single data stream).
- **ОКМД** — Вычислительная система с **о**диночным потоком **к**оманд и **м**ножественным потоком **д**анных (SIMD, single instruction, multiple data).
- **МКОД** — Вычислительная система со **м**ножественным потоком **к**оманд и **о**диночным потоком **д**анных (MISD, multiple instruction, single data).
- **МКМД** — Вычислительная система со **м**ножественным потоком **к**оманд и **м**ножественным потоком **д**анных (MIMD, multiple instruction, multiple data). ...».

Приведённый пример наглядно показывает неправомерное использование термина «*Вычислительные системы*» взамен гораздо более адекватного термина «*Вычислительные комплексы*». В частности, отметим [11]: «... Архитектура SISD — это традиционный компьютер фон-Неймановской архитектуры с одним процес-

сором, который выполняет последовательно одну инструкцию за другой, работая с одним потоком данных. В данном классе не используется параллелизм ни данных, ни инструкций, и, следовательно, SISD-машина не является параллельной. К этому классу также принято относить *конвейерные, суперскалярные и VLIW-процессоры*. ...».

Поскольку «*Сосредоточенные системы*» не являются предметом нашего изучения, то ограничим их рассмотрение приведённой выше детализацией.

1.1.2 Распределенные системы

На рисунке 1.2 «*Распределенные системы*» представлены как «*Системы телеобработки*» и «*Вычислительные сети*». Согласно [4, стр. 7]: «... **Системы телеобработки**. Уже первоначальное применение СОД для управления производством, транспортом и материально-техническим снабжением показало, что эффективность систем можно значительно повысить, если обеспечить ввод данных в систему непосредственно с мест их появления и выдачу результатов обработки к местам их использования. Для этого необходимо связать СОД и рабочие места пользователей с помощью каналов связи. Системы, предназначенные для обработки данных, передаваемых по каналам связи, называются *системами телеобработки данных*.

Состав технических средств системы телеобработки данных укрупненно представлено на рисунке 1.3. Пользователи (абоненты) взаимодействуют с системой посредством терминалов (абонентских пунктов), подключаемых через каналы связи к средствам обработки данных – ЭВМ или вычислительному комплексу. Данные передаются по каналам связи в форме сообщений – блоков данных, несущих в себе кроме собственно данных служебную информацию, необходимую для управления процессами передачи и защиты данных от искажений. Программное обеспечение систем телеобработки содержит специальные средства, необходимые для управления техническими средствами, установления связи между ЭВМ и абонентами, передачи данных между ними и организации взаимодействия пользователей с программами обработки данных. ...».

В современных представлениях, благодаря сетевым технологиям, «*Системы телеобработки данных*», представленные в виде рисунка 1.3, потеряли свою актуальность и используются только в специализированных областях, например, в системах дальней космической связи или в системах диспетчеризации (SCADA), обеспечивая удалённый (полевой) уровень сбора данных. На передний план вышли сетевые технологии, которые авторами учебника [4, стр. 8-9] характеризуются следующим образом: «... **Вычислительные сети**. С ростом масштабов применения электронной вычислительной техники в научных исследованиях, проектно-конструкторских работах, управлении производством и транспортом и прочих областях стала очевидна необходимость объединения СОД, обслуживающих отдельные предприятия и коллективы. Объединение разрозненных СОД обеспечивает доступ к данным и процедурам их обработки для всех пользователей, связанных общей сферой деятельности.

В конце 60-х годов был предложен способ построения вычислительных сетей, объединяющих ЭВМ (вычислительные комплексы) с помощью базовой сети передачи данных. Структура вычислительной сети в общих чертах представлена на рис. 1.4. Ядром является *базовая сеть передачи данных (СПД)*, которая состоит из каналов и *узлов связи (УС)*. Узлы связи принимают данные и передают их в направлении, обеспечивающем доставку данных абоненту. ЭВМ подключаются к узлам базовой сети передачи данных, чем обеспечивается возможность обмена данными между любыми парами ЭВМ. Совокупность ЭВМ, объединённых сетью передачи данных, образует *сеть ЭВМ*. ...».

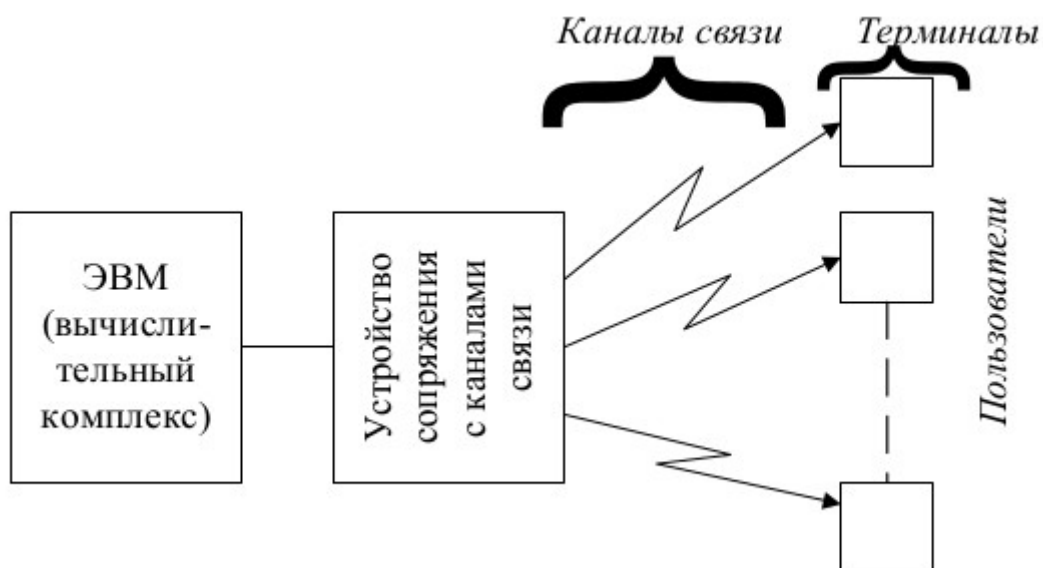


Рисунок 1.3 - Система телеобработки данных [4]

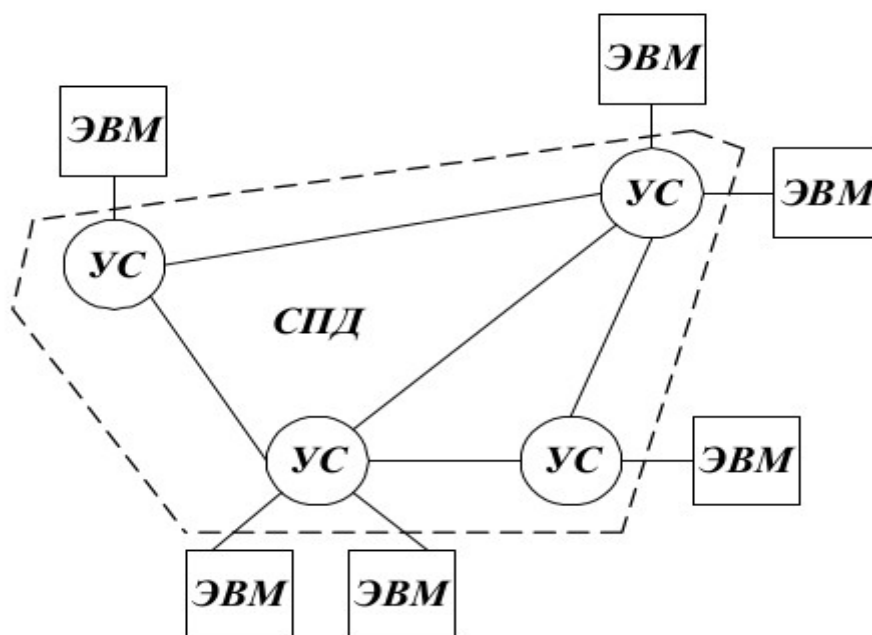


Рисунок 1.4 — Вычислительная сеть [4]

Такое представление термина «*Вычислительная сеть*» хорошо соответству-

ет предметному материалу дисциплины «Сети и телекоммуникации», а также ГОСТ 33707-2016 [9]: «...4.223 **вычислительная сеть**: Сеть, узлы которой состоят из компьютеров и аппаратуры передачи данных, а ветви которой являются линиями передачи данных (computer network)». В любом случае, приведённые определения не раскрывают содержание термина «*Распределенные системы*», а стандарт [9] содержит только близкое по смыслу содержание: «4.1166 **распределенная обработка данных**; DDP: Обработка данных, при которой выполнение операций распределено по узлам вычислительной сети. П р и м е ч а н и е — Для распределенной обработки данных требуется коллективное сотрудничество, которое достигается путём обмена данными между узлами (distributed data processing, DDP). 4.1167 **распределенная система базы данных**: Совокупность данных, распределенных между двумя или более базами данных (distributed database). ...».

Чтобы раскрыть содержательную часть термина «*Распределенные системы*», обратимся к фундаментальной работе Эндрю Таненбаума «Распределенные системы. Принципы и парадигмы» [3]. В ней указывается, что распределенные системы реализуются в виде *системы (службы) промежуточного уровня* (middleware), которая показана на рисунке 1.5.

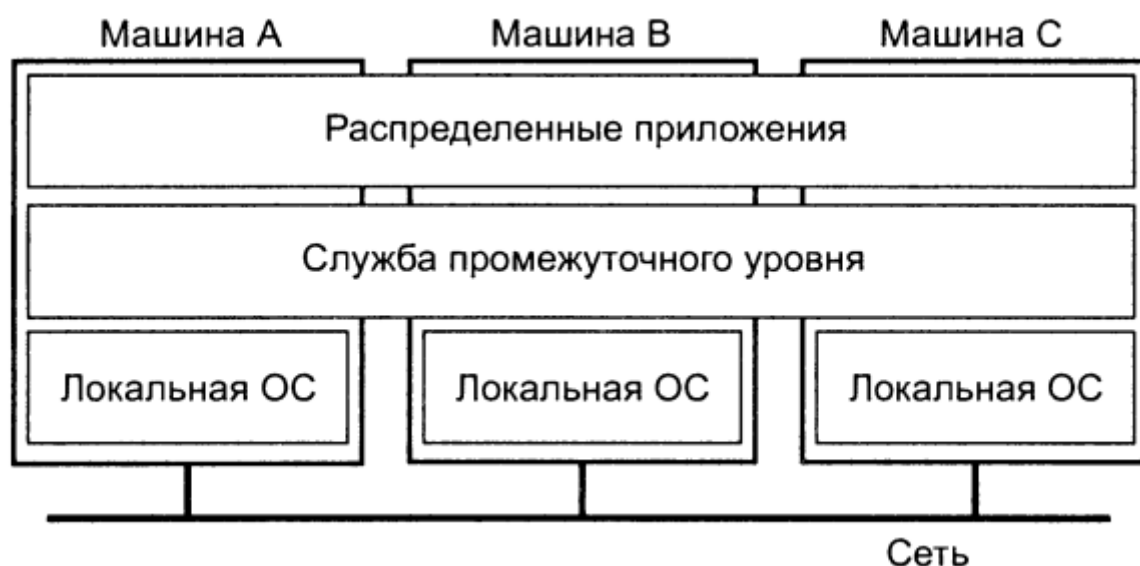


Рисунок 1.5 — Распределенная система [3]

Предполагается, что служба промежуточного уровня должна быть реализована в виде специального программного обеспечения, которое устанавливается на каждую ЭВМ, подключённую к сети. Перспективность такой модели Таненбаум обосновывает:

- значительным снижением цен на отдельные ЭВМ;
- широким распространением высокоскоростных *локальных сетей* (Local-Area Networks, LAN);
- значительным увеличением скорости обмена данными в *глобальных сетях* (Wide-Area Networks, WAN).

К сожалению работа Таненбаума [3] мало подходит в качестве базового

учебного пособия по тематике изучаемой дисциплины. Действительно, как отмечено в ее предисловии: «Эта книга первоначально задумывалась как пересмотренный вариант книги Distributed Operating Systems, но вскоре мы поняли, что с 1995 года произошли такие изменения, что простым пересмотром здесь не обойтись. Нужна совершенно новая книга. Эта новая книга получила и новое название Distributed Systems: Principles and Paradigm». Возможно в будущем автор больше уделит внимания согласованию классификационных принципов построения распределенных систем. Чтобы обосновать это мнение, обратимся к рисунку 1.6, на котором Таненбаум выделяет группы систем с *шинной* и *коммутируемой* архитектурами.

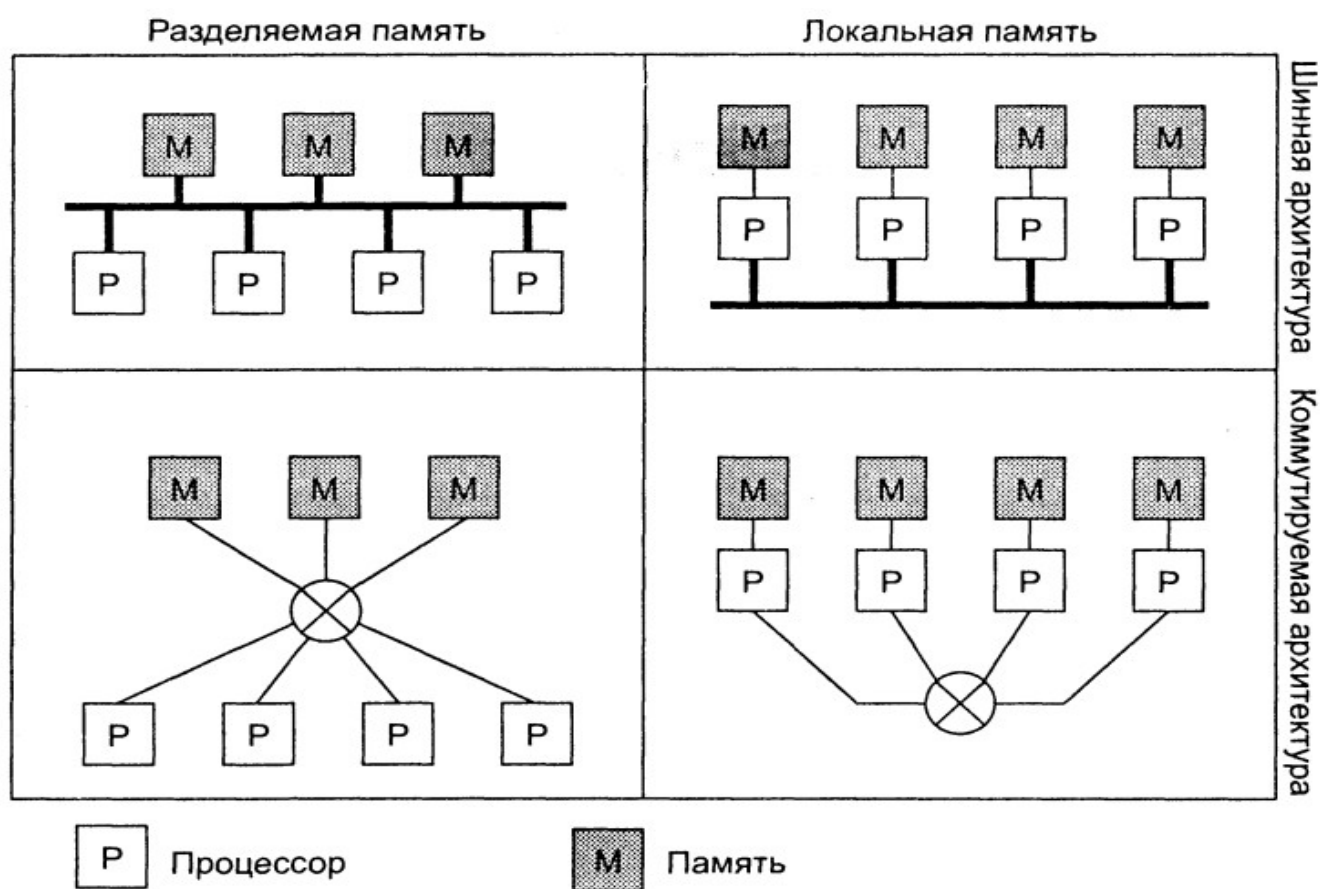


Рисунок 1.6 - Различные базовые архитектуры процессоров и памяти распределенных компьютерных систем [3]

Логически, показанные распределенные архитектуры можно рассматривать как в классе «Вычислительные комплексы», так и в классе «Вычислительные сети». Поэтому, чтобы избежать возможных последующих разночтений, введём более конкретный класс - «Распределенные вычислительные сети». Его и будем рассматривать как предметную область данного пособия.

1.1.3 Распределенные вычислительные сети

Определим предметную область «*Распределенных вычислительных сетей*» (РС-сетей) методом противопоставления понятий «*Распределенные системы*» и «*Сосредоточенные системы*».

Обратимся к публичному источнику Википедии, где прочитаем [12]: «... **Распределенная система** — система, для которой отношения местоположений элементов (или групп элементов) играют существенную роль с точки зрения функционирования системы, а, следовательно, и с точки зрения анализа и синтеза системы.

Для распределенных систем характерно распределение функций, ресурсов между множеством элементов (узлов) и отсутствие единого управляющего центра, поэтому выход из строя одного из узлов не приводит к полной остановке всей системы. Типичной распределенной системой является Интернет.

Примеры распределенных систем:

- Распределенная система компьютеров — компьютерная сеть.
- Распределенная система управления — система управления технологическим процессом.
- Распределенная энергетика.
- Распределенная экономика.
- Распределенная файловая система — сетевые файловые системы.
- Распределенные операционные системы.
- Системы распределенных вычислений.
- Распределенные системы контроля версий.
- Распределенные базы данных.
- Система доменных имен (DNS) — распределенная система для получения информации о доменах».

Можно обсуждать являются ли «*Распределенные системы*» необходимым условием отсутствия единого управляющего центра, но наличие элементов (или групп элементов), которые являются «*Сосредоточенными системами*», является обязательным. Например, DNS является распределенной системой, поскольку размещается на множестве DNS-серверов, которые обслуживают единую иерархическую структуру доменных имён. При этом, отдельный DNS-сервер представляет собой сосредоточенную систему и может рассматриваться как отдельная ЭВМ, комплекс или система. Другим примером являются «*Автоматизированные системы управления*» (АСУ, АСУП, АСУПП, АСУТП), практика создания которых хорошо согласуется с архитектурами рисунков 1.3 и 1.4, но для которых структуры рисунка 1.6 оказываются практически бессмысленными. В любом случае, обмен данными и управляющей информацией осуществляется между отдельными сосредоточенными системами.

Что касается современных «*Сосредоточенных систем*», то для них показанные на рисунке 1.6 архитектуры, действительно являются базовыми и исследуют-

ся в классе вычислительных комплексов. Тот факт, что Э. Таненбаум называет их распределенными, не обеспечивает полноценную реализацию распределенных приложений. В лучшем случае, такие приложения и на таких структурах могут только моделироваться или проектироваться, но не обеспечивать их нормальную целевую работу. Это подтверждается практикой создания различных АСУ, где обработка информации осуществляется параллельно и асинхронно на некотором наборе «Автоматизированных рабочих мест» (АРМ).

Чтобы более наглядно показать возможности «Сосредоточенных систем», вспомним их важнейшую характеристику — быстродействие вычислений. Для этого воспользуемся данными учебника [2] по иерархии запоминающих устройств компьютеров. Сами данные представлены в виде таблицы 1.1 и отражают общепринятое название устройства, ёмкость памяти и время выборки данных.

Таблица 1.1 — Иерархия запоминающих устройств (по данным источника [2])

Название устройства	Ёмкость памяти	Время выборки данных
Регистры процессора	32 бита — 64 бита	Несколько системных циклов
Кэш-память: уровень L1 уровень L2 уровень L3	32 КБ — 64 КБ 256 КБ — 2 МБ 4 МБ — 8 МБ	Порядка 10 нс Порядка 25 нс Порядка 50 нс
Основная память	1 ГБ — 4 ГБ	50 — 100 нс
Твердотельные диски	256 ГБ — 1 ТБ	50 — 200 нс
Магнитные диски	512 ГБ — 6 ТБ	2.5 — 16 мс
Оптические диски CD DVD BD	700 МБ 4.7 ГБ — 8.5 ГБ 25 ГБ — 128 ГБ	150 — 400 мс
Магнитные ленты	5 ТБ — 100 ТБ	500 — 1000 мс

Учитывая общие физические принципы распространения сигналов в электронных устройствах, можно утверждать, что основная тенденция развития сосредоточенных систем — концентрация всех их структур в минимальных физических объемах. Это позволяет создавать ВС различной мощности, включая суперкомпьютеры, но они остаются только элементами «Распределенных систем».

Из сказанного можно было бы сделать вывод, что термин «Распределенные вычислительные системы» (РВС) вполне подошёл бы для предметной области изучаемой дисциплины. Таким образом поступил и автор учебного пособия [5], тем самым игнорируя тот факт, что все приложения РВС явно взаимодействуют через систему, которая обозначена как СПД (система передачи данных). Чтобы более наглядно раскрыть эту ситуацию, кратко характеризуем классы сетевых объектных систем и сервис-ориентированные технологии.

1.2 Сетевые объектные системы

Любое взаимодействие приложений в сети описывается моделью OSI [13]: «... **Сетевая модель OSI** (англ. *open systems interconnection basic reference model* — **Базовая Эталонная Модель Взаимодействия Открытых Систем (ЭМВОС)**) — сетевая модель стека (магазина) сетевых протоколов OSI/ISO (ГОСТ Р ИСО/МЭК 7498-1-99). Посредством данной модели различные сетевые устройства могут взаимодействовать друг с другом. Модель определяет различные уровни взаимодействия систем. Каждый уровень выполняет определённые функции при таком взаимодействии. ...».

Сами уровни такой системы представлены на рисунке 1.7.

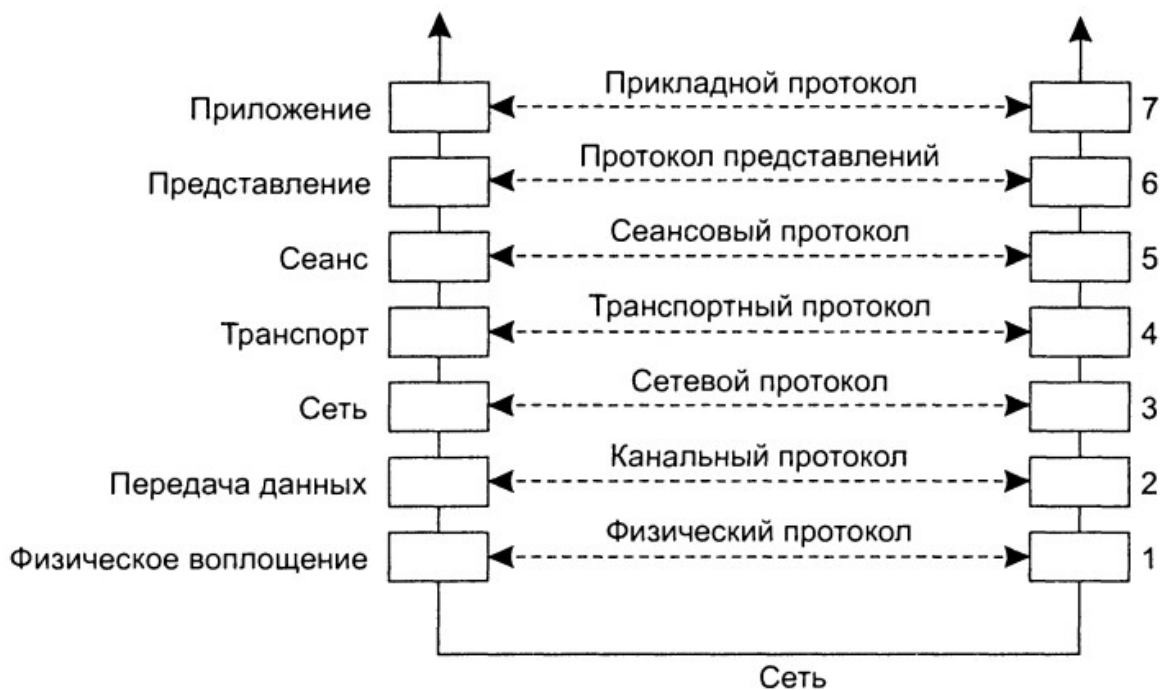


Рисунок 1.7 - Уровни, интерфейсы и протоколы модели OSI [3]

Студентам эта модель хорошо известна из курса «Сети и телекоммуникации», поэтому не будем описывать назначение каждого уровня. Для нас важно, что распределенные приложения обмениваются сообщениями, общая структура которых представлена на рисунке 1.8.

Технологии, основанные на передаче сообщений, породили различные подходы к реализации распределенных приложений. Эти подходы неуклонно смещались в сторону «Распределенных объектных систем», использующих:

- классические модели, основанные на модели **OSI**;
- инструменты распределенной вычислительной среды (**DCE**);
- специальные сетевые системы: **брокеры сообщений**;
- вызов методов удалённых объектов, например, **RMI**.



Рисунок 1.8 — Типовое сетевое сообщение [3]

1.2.1 Классические приложения модели OSI

В модели OSI, «Транспортный уровень» отвечает за надёжную передачу массивов данных, поэтому все современные ОС содержат набор библиотек, обеспечивающих разработку сетевых приложений, классическими примерами которых являются:

- **Telnet (teletype network)** — набор приложений и сетевой протокол для реализации текстового терминального интерфейса по сети; в современных реализациях, обеспечивающих защищённую работу в сети, заменён на приложения: **ssh (secure shell)** для ОС UNIX) и **PuTTY** (для ОС MS Windows);
- **FTP (File Transfer Protocol)** — набор приложений и сетевой протокол для передачи файлов по сети;
- **HTTP (HyperText Transfer Protocol)** - протокол передачи гипертекста, на основе которого создаются общеизвестные приложения: браузеры и web-сервера.

Все приведённые примеры имеют два главных общих признака:

- протоколы *Telnet*, *FTP* и *HTTP* являются оригинальными суммарными реализациями трёх протоколов верхнего уровня модели OSI: *сеансового*, *представления* и *прикладного*;
- все приложения реализованы по общей архитектуре [14]: «... **Клиент-сервер** (англ. *client-server*) — вычислительная или сетевая архитектура, в которой задания или сетевая нагрузка распределены между поставщиками услуг, называемыми серверами, и заказчиками услуг, называемыми клиентами. Фактически клиент и сервер — это программное обеспечение. Обычно эти программы расположены на разных вычислительных машинах и взаимодействуют между собой через вычислительную сеть посредством сетевых протоколов, но они могут быть расположены также и на одной машине. Про-

граммы-серверы ожидают от клиентских программ запросы и предоставляют им свои ресурсы в виде данных (например, загрузка файлов посредством HTTP, FTP, BitTorrent, потоковое мультимедиа или работа с базами данных) или в виде сервисных функций (например, работа с электронной почтой, общение посредством систем мгновенного обмена сообщениями или просмотр web-страниц во всемирной паутине). Поскольку одна программа-сервер может выполнять запросы от множества программ-клиентов, ее размещают на специально выделенной вычислительной машине, настроенной особым образом, как правило, совместно с другими программами-серверами, поэтому производительность этой машины должна быть высокой. Из-за особой роли такой машины в сети, специфики ее оборудования и программного обеспечения, ее также называют сервером, а машины, выполняющие клиентские программы, соответственно, - клиентами. ...».

Дополнительными признаками приведённых примеров являются:

- использование стека протоколов TCP/IP;
- первоначальная *реализация на языке программирования C* и последующий переход на объектно-ориентированные языки программирования (ООП), например, C++ и другие;
- использование *статической адресации ЭВМ (hosts, хостов)*, требующих обращения к протоколам третьего (*сетевого*) уровня, а также фиксации точек доступа (*портов*) на четвёртом (*транспортном*) уровне;
- использование *централизованного управления сетевым взаимодействием* посредством DNS-серверов.

В общем случае, предметная область распределенных приложений не требует привязки к конкретным протоколам транспортного и более нижних уровней. Тем не менее, мы будем опираться на стек протоколов TCP/IP, поскольку на нем построены многие широкоизвестные технологии и имеется соответствующее программное обеспечение.

1.2.2 Распределенная вычислительная среда (DCE)

Естественным образом, расширяя технологическую модель основанную на архитектуре клиент-сервер, мы приходим к идее распределённой вычислительной среды, которая могла бы формироваться программным окружением ОС, например, так [15]: «... **Распределительная вычислительная среда** (англ. *Distributed Computing Environment*, сокр. DCE) — система программного обеспечения, разработанная в начале 1990-х годов в Open Software Foundation, который представлял собой ассоциацию нескольких компаний: Apollo Computer, IBM, Digital Equipment Corporation и других. DCE предоставляет фреймворк и средства разработки клиент-серверных приложений. Фреймворк включает в себя:

- удалённый вызов процедур DCE/RPC;
- служба каталогов;
- служба связанная со временем;
- службу проверки подлинности;
- файловую систему DCE/DFS.

DCE стала важным шагом в направлении стандартизации архитектуры программного обеспечения, которые были зависимы от производителя. Как и сетевая модель OSI, DCE не получила большого успеха на практике, однако основные идеи (концепции) имели более существенное влияние, чем последующие исследования и разработки в этом направлении. ...».

Важнейшим технологическим инструментом DCE является удалённый вызов процедур [16]: «... **Удалённый вызов процедур**, реже **Вызов удалённых процедур** (от англ. *Remote Procedure Call, RPC*) — класс технологий, позволяющих компьютерным программам вызывать функции или процедуры в другом адресном пространстве (как правило, на удалённых компьютерах). Обычно реализация RPC-технологии включает в себя два компонента: сетевой протокол для обмена в режиме клиент-сервер и язык сериализации объектов (или структур, для необъектных RPC). ...».

Суть этой технологии отображена на рисунке 1.9.

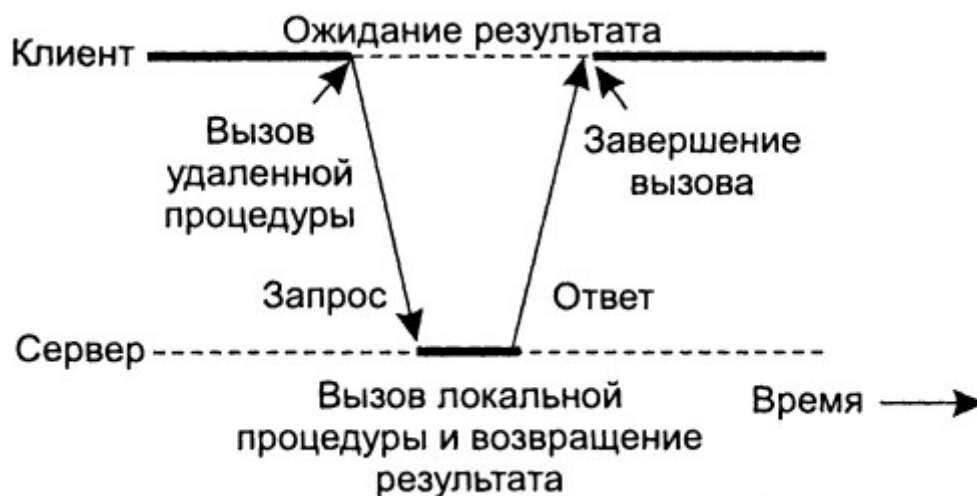


Рисунок 1.9 - Схема RPC между программами клиента и сервера [3]

Особенностью реализации технологии RPC является необходимость создания программных заглушек как на стороне клиента, так и на стороне сервера:

- **Client stub** — программная заглушка на стороне клиента;
- **Server stub** — программная заглушка на стороне сервера.

Поскольку вызов удалённых процедур осуществляется через заранее заданные интерфейсы, то для упрощения разработки интерфейсы часто описываются с использованием языка определения интерфейсов (**Interface Definition Language**,

IDL). В последующем, такие описания компилируются специальными утилитами для создания заглушек на конкретных языках программирования.

В современных условиях, DCE как технология может претендовать только на создание ПО распределенных ОС, поскольку, находясь в «плёну» низкоуровневых системных программных средств, не может обеспечивать необходимый уровень решений при создании сложных распределенных приложений.

1.2.3 Технология CORBA

Разработчикам распределенных систем требовались инструменты, которые бы манипулировали объектами на уровне сети ЭВМ.

Для этого, в 1989 году был создан консорциум [17]: «... **OMG** (сокр. от англ. *Object Management Group*, читается как [о-эм-джи]) — консорциум (рабочая группа), занимающийся разработкой и продвижением объектно-ориентированных технологий и стандартов. Это некоммерческое объединение, разрабатывающее стандарты для создания интероперабельных, то есть платформо-независимых, приложений на уровне предприятия. С консорциумом сотрудничает около 800 организаций — крупнейших производителей программного обеспечения. ...».

Основным достижением консорциума является брокерная архитектура взаимодействия ПО [18]: «... **CORBA** (обычно произносится [кóрба], иногда жарг. [кобра]; англ. *Common Object Request Broker Architecture* — общая архитектура брокера объектных запросов; типовая архитектура опосредованных запросов к объектам) — технологический стандарт написания распределённых приложений, продвигаемый консорциумом (рабочей группой) OMG и соответствующая ему информационная технология. ... Технология CORBA создана для поддержки разработки и развёртывания сложных объектно-ориентированных прикладных систем. CORBA является механизмом в программном обеспечении для осуществления интеграции изолированных систем, который даёт возможность программам, написанным на разных языках программирования, работающим в разных узлах сети, взаимодействовать друг с другом так же просто, как если бы они находились в адресном пространстве одного процесса. ... GIOP (General Inter-ORB Protocol) — абстрактный протокол в стандарте CORBA, обеспечивающий интероперабельность брокеров. ... Архитектура GIOP включает несколько конкретных протоколов:

1. Internet InterORB Protocol (IIOP) (Межброкерный протокол для Интернет) — протокол для организации взаимодействия между различными брокерами, опубликованный консорциумом OMG. IIOP используется GIOP в среде интернет, и обеспечивает отображение сообщений между GIOP и слоем TCP/IP.
2. SSL InterORB Protocol (SSLIOP) — IIOP поверх SSL, поддерживаются шиф-

- рование и аутентификация.
3. HyperText InterORB Protocol (HTIOP) — IIOP поверх HTTP. ...».

Главные теоретические и практические достижения технологии CORBA достаточно подробно рассматриваются в главе 3. Они наглядно показывают, что распределенные системы не могут полноценно строиться только на основе разнесения приложений по отдельным вычислительным системам подключённым к сети. «Службы промежуточного уровня» сами должны представлять собой систему, которая опирается на собственные протоколы, например, GIOP.

Этот вывод обосновывает заявленное название предметной области дисциплины как «Распределенные вычислительные сети».

1.2.4 Удалённый вызов методов

Как заявлено выше, удалённый вызов процедур — RPC является важнейшим технологическим инструментом DCE. Развитие этого инструмента применительно к взаимодействию объектов привело к созданию технологии **RMI** (*Remote Method Invocation*) — программному интерфейсу вызова удалённых методов в языке Java. Реализованный на основе протокола IIOP, RMI является простейшим инструментом реализации не очень сложных распределенных систем.

Технология RMI, как и технология CORBA, рассматриваются в третьей главе данного пособия. Она имеет прежде всего практический интерес, раскрывающий методики программирования, заложенные технологиями DCE. Общая схема взаимодействия объектов RMI показана на рисунке 1.10.

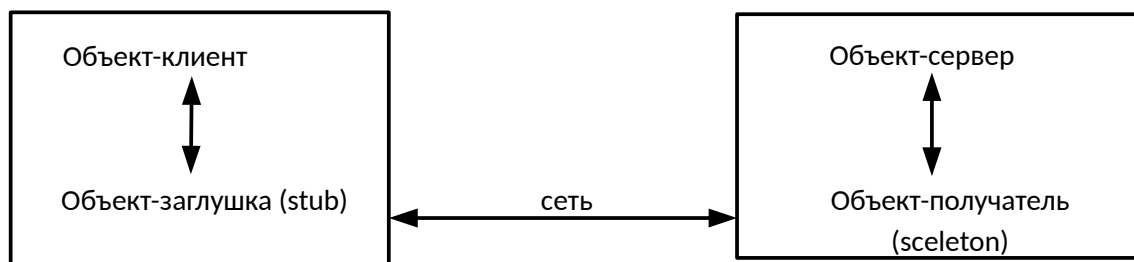


Рисунок 1.10 — Общая схема взаимодействия объектов RMI

Отметим, что развитие технологий RMI и CORBA, а также присущие им недостатки и ограничения, заложили основы более современных представлений, которые обобщенно называются - «**Сервис-ориентированные технологии**».

1.3 Сервис-ориентированные технологии

Рассмотренные ранее технологии, в той или иной степени, были ориентированы на *технические аспекты* применения ЭВМ и сетей. Кроме того, все элементы вычислительной техники и системное программное обеспечение стоят немалых денег и, в процессе развития компьютерной индустрии, возникает понимание того, что на результатах работы ЭВМ тоже можно строить самостоятельный *бизнес*. Соответственно, в конце 90-х годов появляется термин «сервис» и формируется парадигма: «*все есть сервис*». Окончательно, понимая, что этот термин охватывает слишком широкую сферу понятий, были сформированы более строгие трактовки:

- **Everything-as-a-Service** – все как услуга.
- **Infrastructure-as-a-Service (IaaS)** – инфраструктура как услуга.
- **Platform-as-a-Service (PaaS)** – платформа как услуга.
- **Software-as-a-Service (SaaS)** — программное обеспечение как услуга.
- **Hardware-as-a-Service (HaaS)** — аппаратное обеспечение как услуга.
- **Workplace-as-a-Service (WaaS)** – рабочее место как услуга.
- **Data-as-a-Service (DaaS)** – данные как услуга.
- **Secure-as-a-Service** — безопасность как сервис.

Хотя многие из перечисленных сервисов не получили должного развития или популярности, общая тенденция этого движения ведёт к *аутсорсингу*, трактовка которого даётся в следующих аспектах:

- **Аутсорсинг** - *outsourcing: outer-source-using* - использование внешнего источника/ресурса.
- **Аутсорсинг** - передача организацией на основании договора определённых *бизнес-процессов* или *производственных функций* на обслуживание другой компании, специализирующейся в соответствующей области.

Строго говоря, между терминами услуга сервиса и услуга аутсорсинга имеются следующие различия:

- **услуга сервиса** подразумевает разовый, эпизодический или случайный характер и ограниченна началом и концом;
- **услуга аутсорсинга** обычно подразумевает функции по профессиональной поддержке *бесперебойной работоспособности* отдельных систем и инфраструктуры на основе длительного контракта (не менее 1 года).

Краткое раскрытие тематики сервисно-ориентированных технологий приведено в пятой главе данного пособия. Здесь мы отметим только:

- теоретические аспекты различия между *функциями* и *сервисами*;
- особое значение *middleware* — как «Службы промежуточного уровня»;
- важную потребность формирования «Сервисной шины предприятия».

1.3.1 Функции и сервисы

Общее понятие сервиса интерпретируется в рамках информационных технологий (**ИТ**) с помощью модели **SOA**. Для этого, обратимся к публикации «SOA. Архитектурные особенности и практические аспекты» [19]: «... **Сервисно-ориентированная архитектура** (service-oriented architecture, SOA) — подход к разработке программного обеспечения, в основе которого лежат сервисы со стандартизированными интерфейсами. ... С помощью SOA реализуются три аспекта ИТ-сервисов, каждый из которых способствует получению максимальной отдачи от ИТ в бизнесе:

- **Сервисы бизнес-функций.** Суть этих сервисов заключается в автоматизации компонентов конкретных бизнес-функций, необходимых потребителю.
- **Сервисы инфраструктуры.** Данные сервисы выполняют проводящую функцию, посредством платформы, через которую поставляются сервисы бизнес-функций.
- **Сервисы жизненного цикла.** Эти сервисы являются своего рода «обёрткой», которая в большинстве случаев предоставляет ИТ-пользователям «настоящие сервисы». Сервисы жизненного цикла отвечают за дизайн, внедрение, управление, изменение сервисов инфраструктуры и бизнес-функций. ...».

Связь между парадигмами объектно-ориентированного и сервис-ориентированного подходами показана на рисунке 1.11.

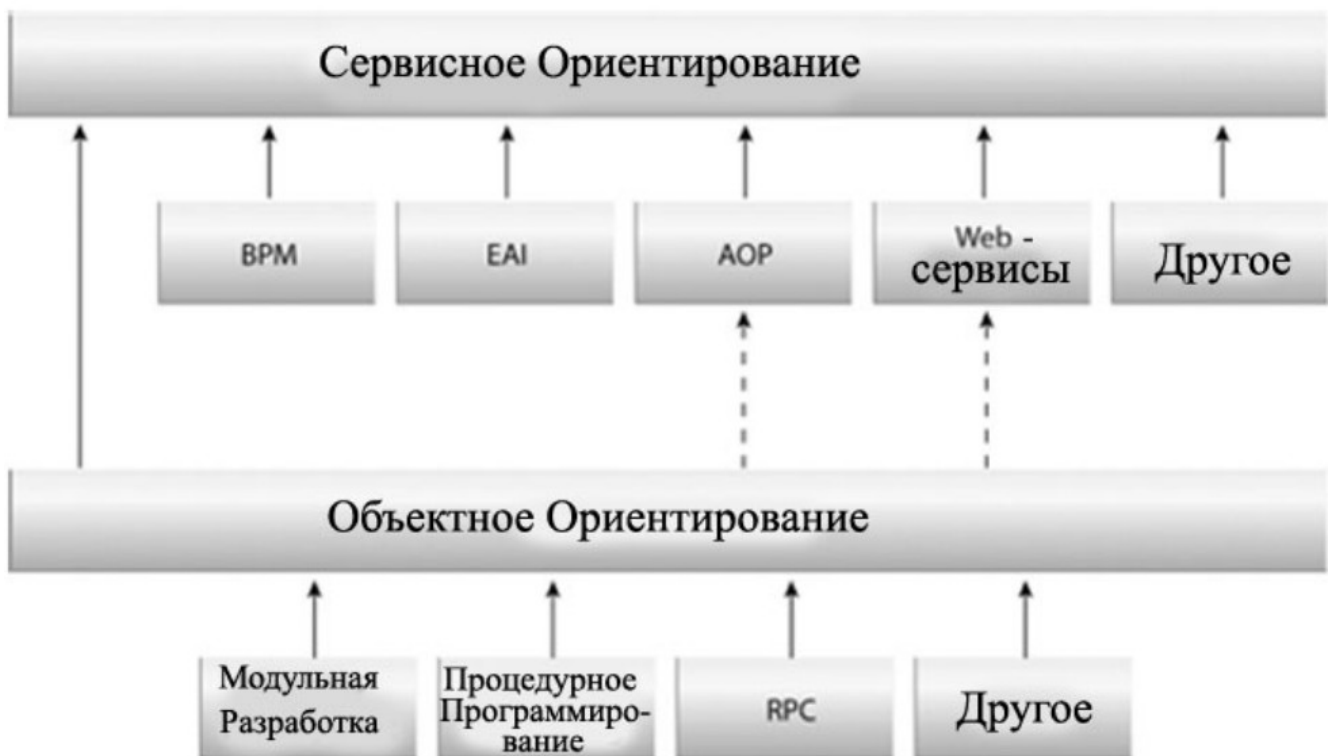


Рисунок 1.11 — Соотношение объектно-ориентированного и сервисного подходов в создании прикладных систем [19]

Из представленного рисунка хорошо видно, что сервис-ориентированный подход вводит следующие новые понятия:

- **BPM** (англ. *business process management*, **управление бизнес-процессами**) — концепция процессного управления организацией, рассматривающая бизнес-процессы как особые ресурсы, непрерывно адаптируемые к постоянным изменениям, и полагающаяся на такие принципы, как понятность и видимость бизнес-процессов в организации за счёт их моделирования с использованием формальных нотаций, использования программного обеспечения моделирования, симуляции, мониторинга и анализа бизнес-процессов [20].
- **EAI** (*Enterprise Application Integration*) — общее название сервиса интеграции прикладных систем предприятия.
- **AOP** (*Aspect-Oriented Programming*) — аспектно-ориентированное программирование; парадигма программирования, основанная на идее разделения функциональности программы на модули.

Википедия даёт этому подходу следующую характеристику [21]: «... **Сервис-ориентированная архитектура (SOA, англ. *service-oriented architecture*)** — модульный подход к разработке программного обеспечения, основанный на использовании распределённых, слабо связанных (англ. *close coupling*) заменяемых компонентов, оснащённых стандартизированными интерфейсами для взаимодействия по стандартизированным протоколам. Программные комплексы, разработанные в соответствии с сервис-ориентированной архитектурой, обычно реализуются как набор веб-служб, взаимодействующих по протоколу SOAP, но существуют и другие реализации (например, на базе *jini*, *CORBA*, на основе *REST*). ...».

1.3.2 Системы *middleware*

Концептуально, SOA реализуется на основе протокола SOAP [22]: «... **SOAP** (от англ. *Simple Object Access Protocol* — простой протокол доступа к объектам) — протокол обмена структурированными сообщениями в распределённой вычислительной среде. Первоначально SOAP предназначался в основном для реализации удалённого вызова процедур (RPC). Сейчас протокол используется для обмена произвольными сообщениями в формате XML, а не только для вызова процедур. Официальная спецификация последней версии 1.2 протокола никак не расширяет название SOAP. SOAP является расширением протокола XML-RPC. SOAP может использоваться с любым протоколом прикладного уровня: SMTP, FTP, HTTP, HTTPS и др. Однако его взаимодействие с каждым из этих протоколов имеет свои особенности, которые должны быть определены отдельно. Чаще всего SOAP используется поверх HTTP. SOAP является одним из стандартов, на которых базируются технологии веб-служб. ...».

В процессе создания и эксплуатации систем, созданных на основе подхода SOA, стало ясно, что необходима промежуточная сетевая система. Стандартизацией подхода SOA занялся некоммерческий консорциум [23]: «... **OASIS** (англ. *Organization for the Advancement of Structured Information Standards*) — глобальный кон-

сорциум, который управляет разработкой, конвергенцией и принятием промышленных стандартов электронной коммерции в рамках международного информационного сообщества. Данный консорциум является лидером по количеству выпущенных стандартов, относящихся к Веб-службам. Кроме этого он занимается стандартизацией в области безопасности, электронной коммерции; также затрагивается общественный сектор и рынки узкоспециальной продукции. В OASIS входит свыше 5000 участников, представляющих более 600 различных организаций из 100 стран мира. Консорциум спонсируется ведущими корпорациями ИТ индустрии такими, как IBM, Novell, Oracle, Microsoft. ...».

Казалось бы такой представительный форум должен был решить все проблемные вопросы, но этого не произошло. Поэтому стали создаваться различные (нестандартизованные) варианты *midl्लeware*-систем, реализующих потребности разработчиков SOA.

1.3.3 Сервисные шины предприятий

Поскольку термины *midl्लeware* и «Службы промежуточного уровня» достаточно абстрактны, а консорциум OASIS не предлагал им достойного эквивалента, то по аналогии с системной шиной ЭВМ был предложен термин [24]: «... **Сервисная шина предприятия** (англ. *Enterprise service bus, ESB*) — связующее программное обеспечение, обеспечивающее централизованный и унифицированный событийно-ориентированный обмен сообщениями между различными информационными системами на принципах сервис-ориентированной архитектуры. Понятие введено в начале 2000-х годов специалистами подразделения Progress Software — Sonic, разрабатывавшими MOM-продукт SonicMQ. ... Основной принцип сервисной шины — концентрация обмена сообщениями между различными системами через единую точку, в которой, при необходимости, обеспечивается транзакционный контроль, преобразование данных, сохранность сообщений. ... По состоянию на вторую половину 2011 года Forrester относит к «волне лидеров» следующие продукты со значительным присутствием на рынке: WebMethods ESB (Software AG, семейство продуктов WebMethods, поглощённой одноимённой компанией), ActiveMatrix Service Bus (Tibco), Oracle Service Bus (Oracle, семейство Fusion Midl्लeware), WebSphere Message Broker (IBM, семейство WebSphere). Среди продуктов с менее значительным присутствием на рынке упомянуты Sonic ESB (Progress Software), WebSphere ESB и ESBRE (IBM), FuseSource, с незначительным — MuleESB, WSO2, Jboss ESB (Red Hat). ...».

Таким образом, смысловое содержание понятия **ESB** является своеобразным «зонтичным брендом» специализированного программного обеспечения для создания *сетевых инфраструктур*, которые интегрируют *потребителей* и *поставщиков* различных сервисов. Такие инфраструктуры скрывают детали реализации сетей, определённых моделью OSI, обеспечивая доступность только протоколов прикладного уровня, таким как: SMTP, FTP, HTTP, HTTPS и другим.

1.4 Виртуальные системы

Понятие «*Виртуальные системы*», которое включает в себя *метасемантику*, присутствующую в изучаемой предметной области, но обычно употребляемую в виде прилагательного, например: «... **Виртуальная машина** (VM, от англ. *Virtual Machine*) — программная и/или аппаратная система, эмулирующая аппаратное обеспечение некоторой платформы (target — целевая, или гостевая платформа) и исполняющая программы для target-платформы на host-платформе (host — хост-платформа, платформа-хозяин) или виртуализирующая некоторую платформу и создающая на ней среды, изолирующие друг от друга программы и даже операционные системы (см.: песочница); также спецификация некоторой вычислительной среды (например: «виртуальная машина языка программирования Си»). Виртуальная машина исполняет некоторый машинно-независимый код (например, байт-код, шитый код, р-код) или машинный код реального процессора. Помимо процессора, VM может эмулировать работу как отдельных компонентов аппаратного обеспечения, так и целого реального компьютера (включая BIOS, оперативную память, жёсткий диск и другие периферийные устройства). В последнем случае в VM, как и на реальный компьютер, можно устанавливать операционные системы (например, Windows можно запускать в виртуальной машине под Linux или наоборот). На одном компьютере может функционировать несколько виртуальных машин (это может использоваться для имитации нескольких серверов на одном реальном сервере с целью оптимизации использования ресурсов сервера)» [25].

Следуя заявленной общей традиции, уточним применительно к тематике дисциплины метасемантику следующих концепций:

- виртуальные машины;
 - виртуализация вычислительных комплексов на уровне ОС;
 - виртуализация ПО на уровне языка (программирования);
 - виртуальная машина языка Java.
-

1.4.1 Виртуальные машины

Словосочетание «*Виртуальная машина*» является наиболее широко используемым понятием. Слово «*машина*» подразумевает некоторый механизм, который обычно имеет аппаратное исполнение и эффективно выполняет некоторый набор действий, имеющих некоторый результат. Например, процессор выполняет последовательность команд над заданным набором данных и выдаёт результат произведённых вычислений. Соответственно, прилагательное «*виртуальная*» задаёт метасемантику подмены первоначального объекта «*машина*» на другой объект «*машина-2*», которая конкретизируется в двух противоположных аспектах:

- *негативный аспект* конкретизации связан не только с фактом подмены, что ассоциируется с фактом обмана, но и с более низким качеством объекта под-

мены; например, замена аппаратной реализации машины программой обычно *снижает скорость вычислений*;

- *позитивный аспект* конкретизации варьируется в более широких пределах; простейший вариант обоснования — отсутствие необходимых аппаратных средств, поэтому виртуализация посредством программной реализации позволяет получить необходимый результат; более сложные варианты обоснования могут быть построены на очень сомнительных аргументах, например, идея реализации вычислительных машин на основе громоздких и ненадёжных электронно-ламповых схем взамен компактных и надёжных электромеханических переключателей (реле).

Важнейшим событием для нашей предметной области стал переход от гарвардской архитектуры ЭВМ к архитектуре фон-Неймана, поскольку в ней появилось понятие адреса как элемента конструктива не только аппаратных средств, но и программного обеспечения. Память становится той универсальной средой, где размещаются программы и данные, а местоположение программ и данных обеспечивается набором систем адресации, с помощью которых и организуется любой вычислительный процесс. Таким образом, согласно модели СОД мы приходим к модели ЭВМ как совокупности аппаратных средств и программного обеспечения ОС, реализующего конкретную виртуальную машину — ЭВМ. Теперь, рассматривая ЭВМ на уровне ОС (например, Linux), мы выделяем:

- *виртуальную машину ядра (kernel)*, создающую для всего пользовательского пространства: набор виртуальных устройств, например, виртуальные терминалы; виртуальную память на уровне страниц и сегментов; виртуальную файловую систему;
- *виртуальное адресное пространство* процессов размером 4 ГБ, первые три ГБ которого принадлежат пользовательскому пространству, а 4-й ГБ - пространству ядра;
- *разделяемые между процессами библиотеки и память*, которые каждый процесс использует в своём (виртуальном) адресном пространстве.

Главная особенность такой предметной интерпретации - «Виртуальная машина» является целостной системой в плане нормального функционирования всех ее компонент. Вся система ориентирована на организацию эффективного вычислительного процесса. Выход из строя любой компоненты существенно влияет на позитивное состояние системы, что банально устраняется ее ремонтом.

1.4.2 Виртуализация вычислительных комплексов на уровне ОС

По определению СОД, вычислительный комплекс является базовой основой вычислительной системы и содержит ОС, которая, реализуя виртуальную машину, позволяет рассматривать комплекс как отдельную ЭВМ. Поскольку комплексы со-

здаются с целью повышения эффективности работы ВС, то соответствующие проектные решения реализуются посредством:

- *виртуализации* многоядерных процессоров в виде отдельных вычислителей, объединённых общей памятью и построением систем симметричного мультипроцессирования (SMP-систем);
- *виртуализации* многомашинных систем средствами программного обеспечения Message Passing Interface (MPI, Интерфейс передачи сообщений), что также обеспечивает построение систем средствами ПО ОС.

Указанные средства могут иметь различную вычислительную мощность и архитектуру различной сложности, например, построенные на гомогенных архитектурах описанных Эндрю Таненбаумом [3], они, по сути, остаются в классе сосредоточенных систем.

1.4.2 Виртуализация ПО на уровне языка

Использование ЭВМ, построенных на архитектуре фон-Неймана, породило большое количество систем адресации, которые необходимо использовать как при проектировании, так при создании и эксплуатации различных систем. Большинство возникающих при этом проблем более или менее успешно решаются с помощью языков программирования. Инструментом решения указанных проблем является **именование** и **типизация имен** языковых конструкций, например:

- языки *ассемблеров*, посредством мнемоник команд, скрывают многие особенности методов адресации данных и размеров самих команд, перекладывая эти проблемы на компиляторы и линковщики программ;
- язык *Fortran*, ориентированный исключительно на вычисления, полностью освобождается от прямой адресации, но вынужден поддерживать типизацию и работу с массивами данных;
- в языке *C* — наоборот, вводится тип указателя, необходимый для решения системных задач и написания программ максимально заменяющих языки ассемблеров;
- языки ООП маскируют адресацию в конструкции объекта, инкапсулируя данные и методы в одном типе;
- языки, поддерживающие *динамическую типизацию*, например, Python, PHP, Perl, JavaScript и другие максимально скрывают саму типизацию, перекладывая решение вопросов адресации на интерпретаторы этих языков.

Здесь следует высказать ряд важных замечаний.

Языки обеспечивающие создание виртуальных машин ориентированы на потребности в собственной системе адресации. Они не охватывают адресацию

приложений, выходящих за рамки виртуальной машины ОС. Например, типичная реализация программы в архитектуре клиент-сервер на уровне стека протоколов TCP/IP требует от клиентской части программы знания адреса и порта сервера, а если адрес сервера задан его доменным именем, то - и адреса сервера DNS.

1.4.3 Виртуальная машина языка Java

Самым ярким событием середины 90-х годов можно считать появление виртуальной машины Java [26]: «... **Java Virtual Machine** (сокращённо **Java VM, JVM**) — виртуальная машина Java — основная часть исполняющей системы Java, так называемой *Java Runtime Environment* (JRE). Виртуальная машина Java исполняет байт-код Java, предварительно созданный из исходного текста Java-программы компилятором Java (javac). JVM может также использоваться для выполнения программ, написанных на других языках программирования. Например, исходный код на языке Ada может быть откомпилирован в байт-код Java, который затем может выполняться с помощью JVM. JVM является ключевым компонентом платформы Java. Так как виртуальные машины Java доступны для многих аппаратных и программных платформ, Java может рассматриваться и как связующее программное обеспечение, и как самостоятельная платформа. Использование одного байт-кода для многих платформ позволяет описать Java как «скомпилировано однажды, запускается везде» (compile once, run anywhere). Виртуальные машины Java обычно содержат Интерпретатор байт-кода, однако, для повышения производительности во многих машинах также применяется JIT-компиляция часто исполняемых фрагментов байт-кода в машинный код. ...».

Хотя сам язык Java является сильно типизированным объекто-ориентированным языком программирования [27], именно благодаря его виртуальной машине (JVM) удалось в кратчайшие сроки реализовать многие современные технологии объектных сетевых систем, web-технологии и сервис-ориентированные технологии. Дело в том, что в отличие от других скриптовых языков, исходный код Java-программы подвергается компиляции и может быть использован без последующего синтаксического анализа. Это значительно ускоряет выполнение Java-программ и позволяет выполнять кэширование серверных приложений.

Уникальными особенностями обладает и байт-код Java. Благодаря разработчикам, в JVM обеспечивается [27] «... полная независимость байт-кода от ОС и оборудования, что позволяет выполнять Java-приложения на любом устройстве, для которого существует соответствующая виртуальная машина. Другой важной особенностью является гибкая система безопасности, в рамках которой исполнение программы полностью контролируется виртуальной машиной. Любые операции, которые превышают установленные полномочия программы (например, попытка несанкционированного доступа к данным или соединение с другим компьютером), вызывают немедленное прерывание.

Часто к недостаткам концепции виртуальной машины относят снижение производительности. Ряд усовершенствований несколько увеличил скорость выполнения программ на языке Java:

- применение технологии трансляции байт-кода в машинный код непосредственно во время работы программы (JIT-технология) с возможностью сохранения версий класса в машинном коде,
- обширное использование платформо-ориентированного кода (native-код) в стандартных библиотеках,
- аппаратные средства, обеспечивающие ускоренную обработку байт-кода (например, технология Jazelle, поддерживаемая некоторыми процессорами архитектуры ARM).

По данным сайта shootout.alioth.debian.org, для семи разных задач время выполнения на Java составляет в среднем в полтора-два раза больше, чем для C/C++, в некоторых случаях Java быстрее, а в отдельных случаях в 7 раз медленнее. С другой стороны, для большинства из них потребление памяти Java-машиной было в 10—30 раз больше, чем программой на C/C++. Также примечательно исследование, проведенное компанией Google, согласно которому отмечается существенно более низкая производительность и большее потребление памяти в тестовых примерах на Java в сравнении с аналогичными программами на C++.

Идеи, заложенные в концепцию и различные реализации среды виртуальной машины Java, вдохновили множество энтузиастов на расширение перечня языков, которые могли бы быть использованы для создания программ, исполняемых на виртуальной машине. Эти идеи нашли также выражение в спецификации общезыковой инфраструктуры CLI, заложенной в основу платформы .NET компанией Microsoft. ...».

Наличие такого большого количества замечательных качеств виртуальной машины Java опирается на серьезную интеллектуальную работу сотрудников американской компании Sun Microsystems, которая совместно с нынешней правопреемницей ее собственности — корпорацией Oracle (*Oracle Corporation*), обеспечивших грамотную пакетную организацию программного обеспечения этой платформы. Благодаря проведенной работе, пакетная организация ПО Java может служить образцом стандартизации программного обеспечения для любых систем и входить во все учебники по программированию.

Отдельной статьи заслуживает описание достижений платформы JVM применительно к web-технологиям. Особенно следует выделить технологию апплетов, которая появилась в первой версии языка Java уже в 1995 году. Именно апплеты обеспечили запуск полноценных приложений в среде web-браузеров. В дальнейшем, подобные технологии распространились и на web-сервера (сервлеты, 1997 год), появилась технология JSP-страниц, реализованная в сервере Apache Tomcat, и так далее.

Таким образом, практическое применение технологических достижений виртуальной машины Java является важной (вспомогательной) частью предметной области изучаемой дисциплины.

1.5 Итоги теоретических построений

Общим итогом теоретических построений данной главы стало определение границ и понятийного содержания предметной области дисциплины, обозначенной как «*Распределенные вычислительные сети*» (РВ-сети). Необходимость более чёткого выделения таких границ вызвана разнообразием различных понятийных трактовок, присутствующих не только в широких публичных изданиях, но и в учебной литературе [1-5]. Важность более строгого выделения определения понятийного содержания рассматриваемой предметной области обусловлена учебным характером данной работы. Опираясь на классификационную схему Ларионова, нами проведён теоретический анализ модели СОД и выполнено уточнение таких понятий как «*Сосредоточенные системы*» и «*Распределенные системы*».

Сосредоточенные системы группируют «*классическое направление*» развития цифровых технологий, основанных на вычислительной мощности программно-аппаратных средств и локализованных во вполне обозримых границах. Для таких систем подключение и взаимодействие с другими подобными системами является допустимым явлением, но не главной характеристикой.

Непосредственное последующее толкование таких систем будет рассматриваться в трёх аспектах:

- **ЭВМ** — согласованный в функциональном назначении аппаратный конструктив, дополненный базовым ПО ОС до конкретной виртуальной машины; в последующем, такая машина рассматривается как элемент более сложных систем;
- **Вычислительные системы** — все прикладные аспекты программного обеспечения, реализованные поверх виртуальной машины ОС; в последующем, такие системы также могут рассматриваться как элементы более сложных систем;
- **Вычислительные комплексы** — системы аппаратных средств, ориентированные на повышение вычислительной мощности и надёжности всего конструктива, а также — дополненные соответствующей виртуальной машиной ОС; в последующем, такой комплекс также может рассматриваться как отдельная ЭВМ или составляющая часть вычислительной системы.

Распределенные системы группируют «*современное направление*» развития цифровых технологий, основной характеристикой которых является интеграция *сосредоточенных систем* (ЭВМ, ВС) на основе сетевых технологий. Важной особенностью таких систем является их *свойство несводимости* к моделям сосредоточенных систем. Обычно, это определяется самой природой интегрируемых

приложений, например, программное обеспечение клиента банка должно быть отделено от ПО самого банка в силу нормативных требований к проведению финансовых операций.

Что же касается декомпозиции распределенных систем, представленной на рисунке 1.2, то являясь по сути правильной, она морально устарела с точки зрения современных достижений. Действительно:

- **Системы телеобработки**, детализированные на рисунке 1.3, с успехом и без ущерба для производительности и стоимости их эксплуатации, могут быть реализованы на основе современных сетевых технологий;
- **Вычислительные сети**, детализированные на рисунке 1.4, также отражают устаревшие технологические представления, поскольку базовая сеть передачи данных (СПД), построенная на узлах связи (УС), охватывает только нижние три уровня модели OSI (см. рисунок 1.7); современные же сетевые системы, как это показано в пунктах 1.2 и 1.3, создаются на основе протоколов прикладного уровня, что охватывает все уровни модели OSI.

Таким образом, суммируя все перечисленные выше доводы, *распределенные системы* будем детализировать одной моделью *РВ-сетей*, показанной на рисунке 1.12.

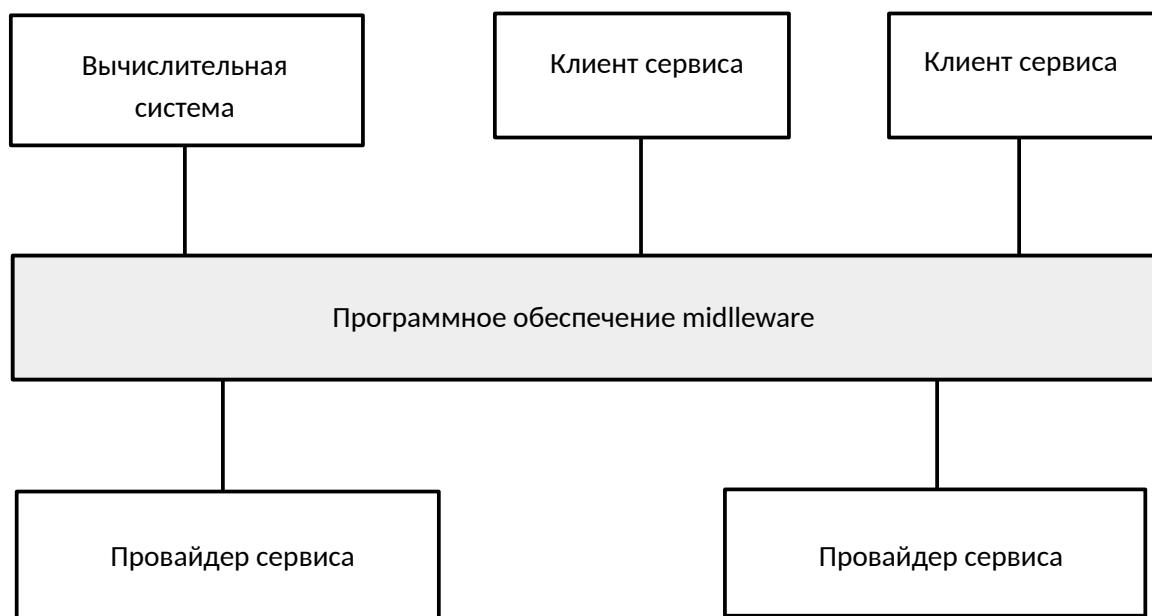


Рисунок 1.12 — Модель РВ-сети

Представленная модель РВ-сети будет конкретизироваться в зависимости от тематики рассматриваемых вопросов. Соответственно, «Программное обеспечение middleware» будет обозначать модель OSI (обычно стек протоколов TCP/IP), а окружающие прикладные объекты будут отображать *сосредоточенные системы*.

Вопросы для самопроверки

1. Какие уровни детализации допускает понятие «*Вычислительная машина*»?
2. На какие две группы делит системы классификация СОД?
3. Что подразумевает понятие «*Сосредоточенные системы*»?
4. Чем отличаются понятия «*Вычислительные системы*» и «*Вычислительные комплексы*»?
5. На какие четыре части делит архитектуры ЭВМ таксономия Флинна?
6. Что входит в состав классического понятия «*Распределенные системы*»?
7. Какие элементы включает в себя понятие «*Вычислительная сеть*»?
8. Что означает понятие «*Служба промежуточного уровня*»?
9. Что означает понятие «*Распределенная вычислительная среда*»?
10. Что означает понятие «*Remote Procedure Call*»?
11. Что такое - «*Client stub*» и «*Server stub*», а также в чем их различие?
12. Каково назначение и область применения аббревиатуры — IDL?
13. В чем состоит суть технологии CORBA?
14. В чем отличие технологии RMI от технологии CORBA?
15. На какие общие услуги подразделяется парадигма «*Сервис-ориентированные технологии*»?
16. В чем состоит отличие понятий «*Функции*» и «*Сервисы*»?
17. Какая практическая интерпретация применима к понятию «*Middleware*»?
18. Что означает понятие «*Виртуальная машина*» применительно к программным системам?
19. Что означает понятие «*Java Virtual Machine*»?
20. Что такое - «*Промышленная шина предприятия*»?

2 Тема 2. Инструментальные средства языка Java

Настоящая тема посвящена краткому изучению базовых средств языка Java, который был разработан американской компанией Sun Microsystems [28]. Считается, что он начал проектироваться в начале 90-х годов, а датой официального выпуска считается 23 мая 1995 года [27]. В настоящее время, правопреемницей технологических достижений языка является корпорация Oracle [29].

Поскольку технологическая основа Java предполагает реализацию виртуальной машины JVM [26], то, первоначально, широкое распространение языка было под большим сомнением. Действительно, в начале 90-х годов большинство персональных компьютеров использовало 20-битную адресацию команд, что ограничивало объем основной памяти (ОП) ЭВМ размером 1 МБ, из которого только 640 КБ предназначалось для запуска программ. Тем не менее, при вероятной поддержке корпорации Oracle, выпустившей в начале 80-х годов свою знаменитую СУБД с одноимённым названием, Sun Microsystems продолжила свою работу и получила широкое признание своих достижений. Позже корпорация Oracle даже включила JVM в свою СУБД для реализации триггеров и функций ядра ориентированных на численные расчёты.

Сам язык Java, как и язык C++, унаследовал синтаксис языка C, поэтому студенты, изучавшие курс ООП, могут легко начинать программировать даже на основе статьи Википедии [27]. Мы, при изучении материала, будем опираться на руководство [8], но предварительно сделаем ряд общих замечаний.

В декабре 1998 года вышла версия языка под номером 1.2, в которую была добавлена структура коллекций (collections framework) и ряд других незначительных изменений. Этот факт стал основой различных публикаций, использующих обозначение языка как Java 2. Такое обозначение можно найти и в современных книгах, учебных пособиях и используемых сокращениях, хотя они относятся к одному и тому же языку Java. Кроме того, часто мажорный номер версии опускается, а указывается только минорный. Например, словосочетание «Java 7» точно подразумевает версию языка: «Java 1.7».

Большая популярность и широкое распространение технологии Java на различные платформы потребовало разделения первоначально единого программного обеспечения. В настоящее время выделяются следующие платформы:

- **J2EE** или Java EE (начиная с v.1.5) — Java Enterprise Edition, для создания программного обеспечения уровня предприятия;
- **J2SE** или Java SE (начиная с v.1.5) — Java Standard Edition, для создания пользовательских приложений, например, — для настольных систем;
- **J2ME** или Java ME или Java Micro Edition, - для использования в устройствах, ограниченных по вычислительной мощности, например, мобильных, КПК или встроенных системах;
- **Java Card** — для использования в устройствах без собственного человеко-машинного интерфейса, например, - в смарт-картах.

Дополнительно, стандартные дистрибутивы Java подразделяются на два основных вида поставок, за которыми сейчас нужно обращаться на сайты корпорации Oracle:

- **JRE** (*Java Runtime Environment*) — среда исполнения Java-приложений, обязательным компонентом которой является *JVM*;
- **JDK** (*Java Development Kit*) — дистрибутив инструментальных средств для разработки приложений на языке Java, обязательно включающий в себя базовый дистрибутив *JRE*;
- другие, альтернативные виды дистрибутивов, за которыми следует обращаться к другим поставщикам; например, **OpenJDK** — дистрибутив, обычно формируемый и поставляемый провайдерами ОС Linux.

Особо следует отметить, что дистрибутивы поставляются с фиксированным номером версии и аппаратной платформы. Например, используемая в данном учебном пособии дистрибутив: *OpenJDK_1.8* для 64-битной платформы ОС Arch Linux. Чтобы не перегружать деталями вводную часть этой главы, в подразделе 2.1 предоставляется общая вводная часть, которая касается работы с языком в командной строке интерпретатора shell и в инструментальной среде системы Eclipse EE. Более конкретная и детальная информация предоставляется в методических руководствах по лабораторным работам, например, [6, 7] и соответствующих указаниях по практическим занятиям, если таковые предусмотрены, например, [8].

Остальные подразделы данной главы посвящены краткому описанию синтаксиса и семантики базовых конструкций языка Java и завершаются рассмотрением базовых сетевых приложений, а именно:

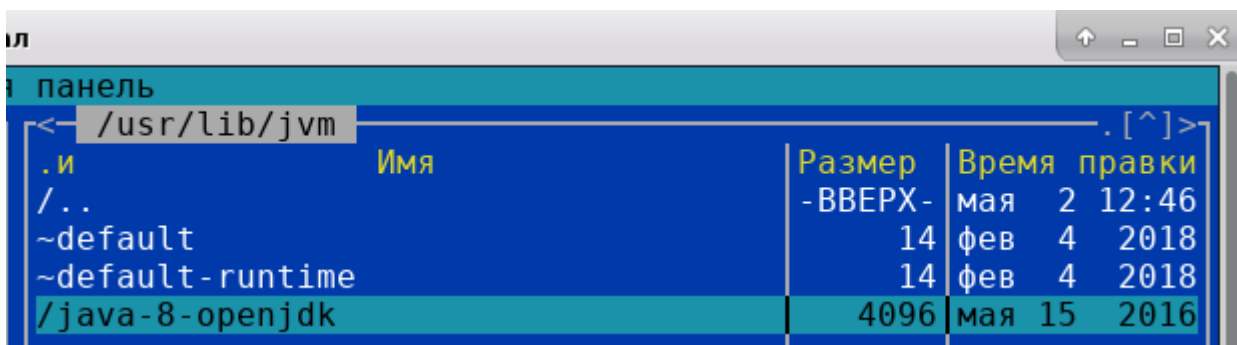
- подразделы 2.2-2.4 — содержат описания простейших типов, классов, методов и организацию ввода-вывода; это обеспечивает студентов знаниями и навыками написания обычных расчётных программ;
- подраздел 2.5 - готовит студентов к написанию простейших клиент-серверных программ на базе транспортного уровня стека протоколов TCP/IP;
- подраздел 2.6 — обеспечивает студентов навыками работы с базами данных на языке Java.

В целом, учебный материал данной главы, хоть и имеет выраженную прикладную направленность, но ориентирован на подготовку студента к освоению теоретического и практического материала последующих глав дисциплины.

2.1 Общее описание инструментальных средств языка

Поскольку технологии Java обладают высокой межплатформенной переносимостью, поэтому среды исполнения (JRE) и инструментальные средства (JDK) работают в них практически одинаково. Тем не менее, в различных ОС обычно устанавливаются разные дистрибутивы. Различным является и их размещение, хотя программное обеспечение располагается компактно и не вызывает затруднений в его использовании.

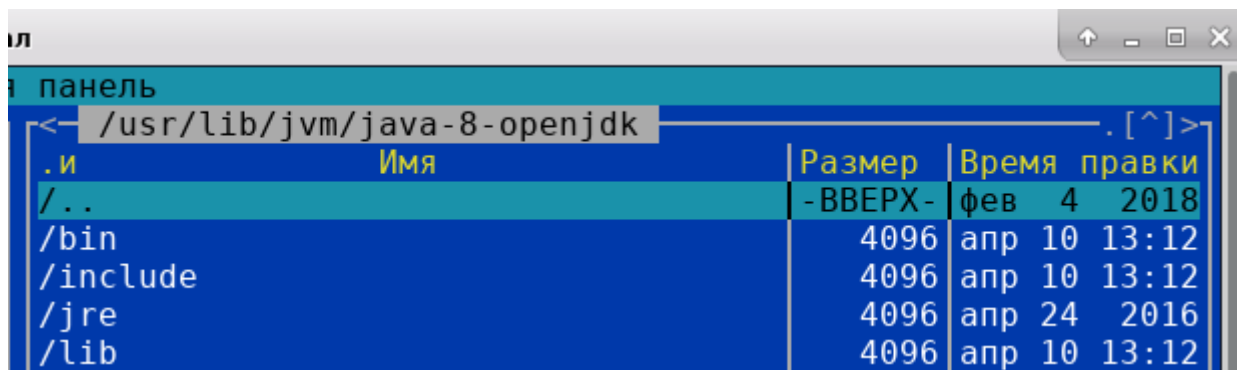
В учебном пособии используется дистрибутив OpenJDK версии 1.8, который является частью учебного программного комплекса [6] и с которым студенты хорошо знакомы в процессе изучения дисциплины «Операционные системы». Стандартное размещение OpenJDK находится в корне директории `/usr/lib/jvm`, как это показано на рисунке 2.1.



Имя	Размер	Время	правки
..	-ВВЕРХ-	мая 2	12:46
~default	14	фев 4	2018
~default-runtime	14	фев 4	2018
java-8-openjdk	4096	мая 15	2016

Рисунок 2.1 — Базовая директория установки дистрибутивов Java

Теоретически, здесь может быть установлено несколько разных дистрибутивов, поэтому для удобства настроек используются две ссылки `default` и `default-runtime`, конкретизирующие дистрибутив по умолчанию. На рисунке 2.1 они указывают на одну и ту же директорию `java-8-openjdk`, содержимое которой представлено на рисунке 2.2.



Имя	Размер	Время	правки
..	-ВВЕРХ-	фев 4	2018
bin	4096	апр 10	13:12
include	4096	апр 10	13:12
jre	4096	апр 24	2016
lib	4096	апр 10	13:12

Рисунок 2.2 — Типовое содержимое директории OpenJDK, где каталог `jre` является корнем дистрибутива среды исполнения JRE

Что касается ОС MS Windows, то установка Java производится специальным

инсталлятором, который скачивается с сайта корпорации Oracle. По умолчанию, инсталляция производится в базовый каталог **C:\Program Files (x86)\Java**, а структура каталогов дистрибутивов JDK оказывается такой же, как и на рисунке 2.2.

С целью избежания возможных скрытых недоразумений, студент должен учитывать, что все примеры программ, приведённые в данном пособии, реализованы в среде ОС Linux и в рабочей области пользователя **vgr**. Это сделано для того, чтобы студент приложил некоторые минимальные усилия для самостоятельной реализации всех учебных проектов.

2.1.1 Инструментальные средства командной строки

Инструментальные средства языка Java содержат некоторое множество различных программ (утилит), которыми необходимо владеть для успешного использования всех технологических возможностей языка. Две такие утилиты являются наиболее важными и часто используемыми:

- **java** — главная программа, запускающая виртуальную машину Java;
- **javac** — компилятор исходных текстов языка, входящий в дистрибутив JDK.

Хотя современные инсталляторы ОС обычно обеспечивают все необходимые настройки для конечного пользователя, разработчику приходится самому исправлять имеющиеся ошибки, проектировать и отлаживать новые дистрибутивы. К счастью ПО Java ориентировано на достаточно автономную базовую установку, что упрощает ее последующую эксплуатацию. Действительно, как показано на рисунке 2.2, все дистрибутивы JDK имеют общую архитектуру, сосредоточенную в следующих директориях:

- **bin** — каталог для хранения исполняемых файлов JDK (для ОС MS Windows исполняемые файлы имеют стандартное расширение **.exe**);
- **lib** — для хранения библиотек (пакетов) Java, имеющих нативное (родное) расширение **jar**, а также платформозависимые расширения **.so** или **.dll**;
- **jre** — каталог размещения дистрибутива JRE, также имеющий подкаталоги **bin** и **lib** с аналогичным назначением.

Дополнительные настройки инструментальной среды Java определяются с помощью четырёх переменных среды ОС:

- **PATH** — список директорий, где ОС будет искать исполняемые файлы для запуска;
- **CLASSPATH** — список директорий, где Java будет искать нужные библиотеки (пакеты библиотек);
- **JRE_HOME** — указывает на директорию инсталляции JRE;
- **JAVA_HOME** — указывает на директорию инсталляции JDK.

Чтобы проверить правильную настройку инструментальных средств Java,

следует выполнить команды:

```
$ java -version
$ javac -version
```

где знак «\$» означает, что команда запущена от имени пользователя **vgr**, а знак «#» используется, если команды запускаются от имени пользователя **root**.

Для демонстрации работы **java** и **javac** воспользуемся исходным текстом простейшей программы, представленной на листинге 2.1.

Листинг 2.1 — Исходный текст класса Example1

```
public class Example1 {

    public static void main(String[] args) {
        System.out.println("Здравствуй, Мир!");
    }

}
```

Здесь приведено описание публичного класса *Example1*, содержащего единственный публичный статический метод *main(...)*, который печатает на стандартный вывод (объект *System.out*) строку «Здравствуй, Мир!» с помощью метода *println(...)*. Полное определение классов и методов будет дано в следующем подразделе, а сейчас отметим, что в языке Java принято называть классы именами, начинающимися с большой буквы, а методы — с маленькой. Кроме того, исходный текст каждого класса хранится в отдельном файле с именем класса и стандартным расширением **.java**.

Учитывая эти требования, создадим рабочую директорию **/home/vgr/src/rvs**, перейдём в неё и сохраним содержимое листинга 2.1 в файле **Example1.java**. Затем, проведём компиляцию данного файла командой:

```
$ javac Example1.java
```

В результате, получим файл **Example1.class**, содержащий байт-код нашей программы. Запустим этот класс на выполнение командой:

```
$ java Example1
```

Программа выведет на терминал строку: «Здравствуй, Мир» и закончит свою работу. Чтобы продемонстрировать более сложные случаи запуска программ Java, необходимо рассмотреть его пакетную организацию.

2.1.2 Пакетная организация языка Java

Все языки программирования характеризуются некоторой логической структурой, которая проецируется на его базовые возможности и последующие расширения на прикладные области применения. Обычно это представляется в виде набора взаимосвязанных библиотек, которые необходимы не только для реализации любого прикладного проекта, но и становятся необходимой понятийной базой любого программиста, характеризующей его профессиональную подготовку.

Важной особенностью языка Java является хорошая структуризация его системных программных средств. Эта структуризация основана на базе множества пакетов, группирующих классы языка в соответствии с их основной прикладной направленностью и строгим именованием, что демонстрируется ниже:

<i>java.lang</i>	Базовый пакет, обеспечивающий основные возможности языка: объекты, классы, исключения, математические функции, интерфейс с JVM и другие.
<i>java.util</i>	Инструментальный пакет классов: коллекции, дата, время, ...
<i>java.io</i>	Операции с файлами, потоковый ввод\вывод, ...
<i>java.math</i>	Набор функций: sin, cos и другие.
<i>java.net</i>	Операции с сетью, сокет, URL, ...
<i>java.security</i>	Генерация ключей, шифрование и дешифрование, ...
<i>java.sql</i>	Java Database Connectivity (JDBC) доступ к базам данных
<i>java.awt</i>	Базовый пакет для работы с графикой ...
<i>javax.swing</i>	Графические компоненты для разработки приложений: кнопки, текстовые поля, ...

Практически все из перечисленных пакетов, кроме двух последних, будут использованы в учебном материале данной главы. В частности, базовый пакет ***java.lang*** уже неявно был использован в примере листинга 2.1. Это является особенностью современных реализаций языка Java. В первых версиях такое не допускалось и требовалось явное подключение всех классов, использованных в программе.

Явное подключение используемых классов осуществляется с помощью оператора ***import***, после которого необходимо правильно указать полный путь к ним. Поскольку это в большинстве случаев является затруднительным, то возможно подключение всех классов пакета, в формате:

```
import имя_пакета.*;
```

Покажем явное подключение пакетов на примере использования объекта класса ***Date***, содержащегося в пакете ***java.util*** и задающего текущие дату и время.

Для этого, модифицирует класс **Example1**, как показано на листинге 2.2.

Листинг 2.2 — Первая модификация текста класса *Example1*

```
import java.util.*; // Подключение всех классов пакета

public class Example1 {

    public static void main(String[] args) {
        System.out.println("Здравствуй, Мир! - " + new Date());
    }

}
```

Проведя компиляцию и запуск программы листинга 2.2, получим результат показанный на рисунке 2.3.

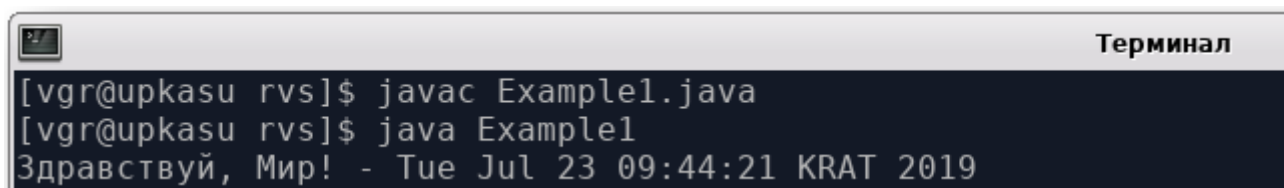


Рисунок 2.3 — Результат компиляции и запуска программы листинга 2.2

Приведённый пример демонстрирует важные особенности программирования на языке Java, связанные с его пакетной организацией классов:

- объект **out**, обеспечивающий стандартный вывод языка, находится в базовом пакете **java.lang** и определён статическим типом класса, поэтому разрешается прямое обращение к методу **java.lang.System.out.println(...)**, а используемое сокращение допускается наличием однозначности его вызова;
- класс **Date()** является динамическим типом, поэтому объект даты генерируется с помощью оператора **new**, а однозначность генерации обеспечивается отсутствием других пакетов, содержащих класс **Date()**;
- строка любого текста, заключённого в двойные кавычки, является объектом, имеющим свои методы и оператор конкатенации «+»;
- метод **println(...)** демонстрирует также приведение объекта типа **Date()** к объекту строкового типа.

Пакетная организация языка Java имеет простую семантику, привязанную к каталогам файловой системы ОС, что напрямую отображается синтаксисом следующего формата:

```
package имя_каталога1.имя_каталога2. ... .имя_каталогаN;
```

Строка с ключевым словом ***package*** помещается в начале каждого файла с исходным текстом описания класса, задавая относительную адресацию байт-кода класса в пределах реализуемого проекта. Обычно, используемые имена каталогов имеют дополнительную семантику, организующую некоторую доменную структуру и характеризующие страну, организацию, язык, автора и другие свойства целевого пакета. Например, приведённые выше имена базовых пакетов Java, начинаются именем **java**, а второе имя характеризует его целевую направленность. Для наглядности демонстрации влияния оператора ***package***, проведём вторую модификацию класса `Example1`, которая представлена на листинге 2.3.

Листинг 2.3 — Вторая модификация текста класса `Example1`

```
package ru.tusur.asu; // Описание пути для класса Example1.class

import java.util.*;    // Подключение всех классов пакета

public class Example1 {

    public static void main(String[] args) {
        System.out.println("Здравствуй, Мир! - " + new Date());
    }
}
```

Компиляция этой программы проходит успешно и в текущем каталоге появляется файл `Example1.class`, но как показано на рисунке 2.4, запустить на выполнение его не удаётся. Появляется сообщение, что класс не найден и не может быть загружен.

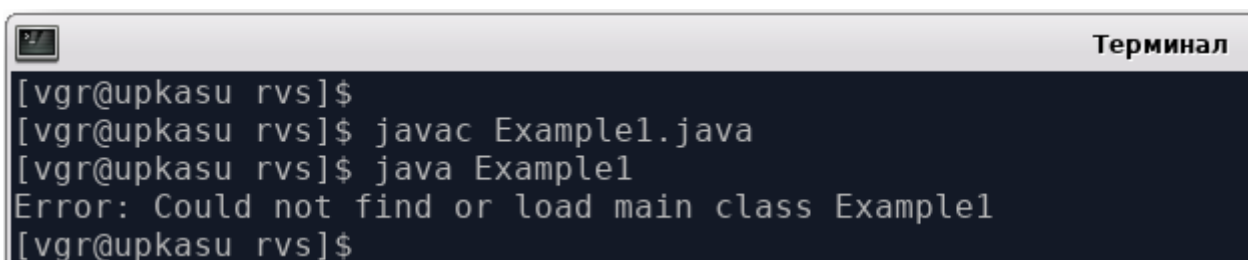


Рисунок 2.4 — Результат компиляции и запуска программы листинга 2.3

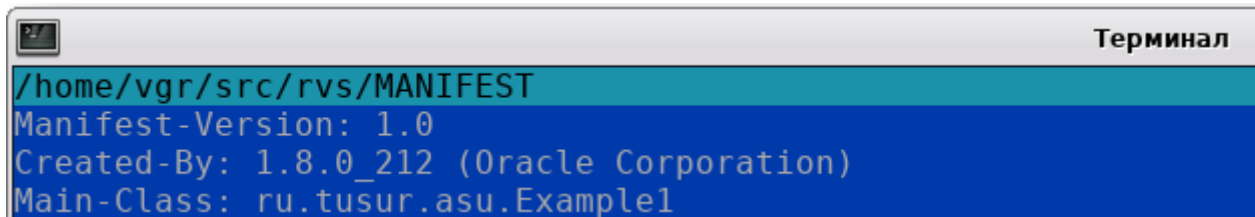
Чтобы исправить ситуацию, нужно:

- в текущем каталоге создать дерево директорий ***ru/tusur/asu*** и перенести туда файл `Example1.class`;
- вернувшись в исходную директорию, запустить программу командой:

```
$ java ru.tusur.asu.Example1
```

Таким образом, каждый класс, разрабатываемый в рамках какого-либо проекта, помещается в свой каталог, определяемый оператором **package**. Совокупность всех классов проекта размещается в некотором дереве каталогов, имеющем общую вершину. В последующем, такой проект оформляется в виде отдельного **jar**-файла — библиотеки языка Java, которая создается с помощью утилиты **jar**.

Сама утилита **jar** имеет множество параметров и показывает их запуском без параметров. Мы используем ее только для нашего примера, предварительно создав файл манифеста, представленного на рисунке 2.5.



Терминал

```
/home/vgr/src/rvs/MANIFEST
Manifest-Version: 1.0
Created-By: 1.8.0_212 (Oracle Corporation)
Main-Class: ru.tusur.asu.Example1
```

Рисунок 2.5 — Минимальное содержимое файла манифеста для jar-архива

Такой файл манифеста указывает статический класс архива, содержащий статический метод **main(...)**, используемый для запуска программы.

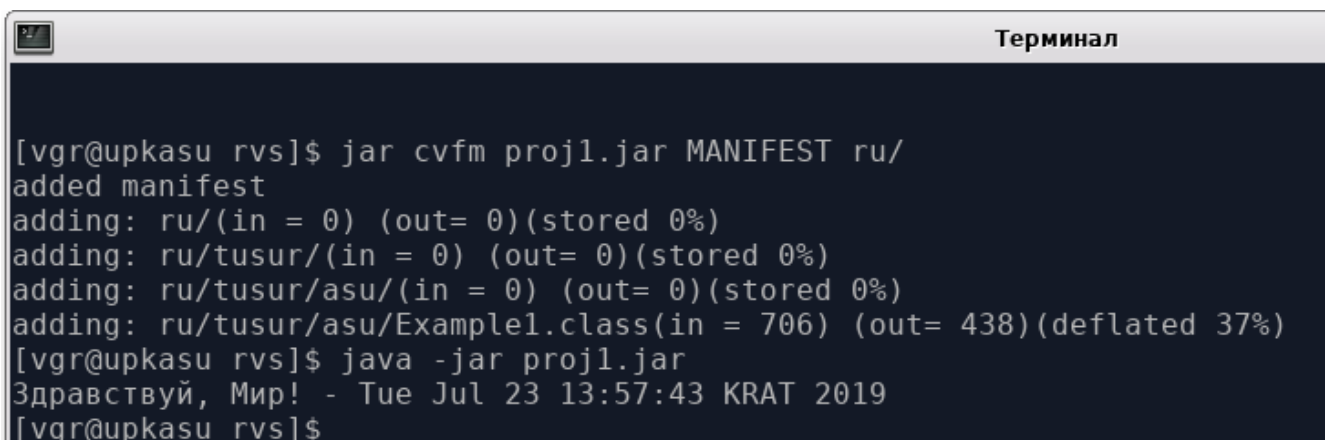
Теперь создадим архив с именем **proj1.jar** командой:

```
jar cvfm proj1.jar MANIFEST ru/
```

В текущей директории появится архив с именем **proj1.jar** и наш пример может запускаться командой:

```
java -jar proj1.jar
```

Общий результат создания архива и запуска приложения показан на рисунке 2.6.



Терминал

```
[vgr@upkasu rvs]$ jar cvfm proj1.jar MANIFEST ru/
added manifest
adding: ru/(in = 0) (out= 0)(stored 0%)
adding: ru/tusur/(in = 0) (out= 0)(stored 0%)
adding: ru/tusur/asu/(in = 0) (out= 0)(stored 0%)
adding: ru/tusur/asu/Example1.class(in = 706) (out= 438)(deflated 37%)
[vgr@upkasu rvs]$ java -jar proj1.jar
Здравствуй, Мир! - Tue Jul 23 13:57:43 KRAT 2019
[vgr@upkasu rvs]$
```

Рисунок 2.6 — Результат создания архива proj1.jar и запуска приложения

2.1.3 Инструментальные средства Eclipse

Возможности командной строки - достаточно широки, но в плане разработки программного обеспечения значительно уступают интегрированным инструментальным средствам IDE (*Integrated Development Environment*). В данном пособии будет использоваться IDE Eclipse EE, которое ориентировано на разработку приложений уровня предприятия средствами технологий языка Java. Мы будем придерживаться варианта ПО установленного в учебном программном комплексе [6], поэтому отметим, что упомянутый инструмент установлен опционально в среду ОС Linux и монтируется в директорию **/opt/eclipseEE**. Запуск инструмента осуществляется специальным значком, размещённым на рабочем столе пользователя, или эквивалентной командой:

```
/opt/eclipseEE/eclipse -data rvs
```

Поскольку студенты хорошо знакомы с инструментальной средой Eclipse в процессе изучения курса «*Операционные системы*», программируя на языках C/C++, поэтому уделим внимание только тем аспектам технологии IDE, которые связаны с реализацией проектов на языке Java. Для конкретизации примера, создадим проект с именем **proj1**, реализующим программу с исходным текстом листинга 2.3.

В главном меню запущенной IDE, выберем: **File** → **New** → **Java Project**. Появится окно рисунка 2.7, в котором укажем имя проекта **proj1** и создадим проект, нажав кнопку «**Finish**».

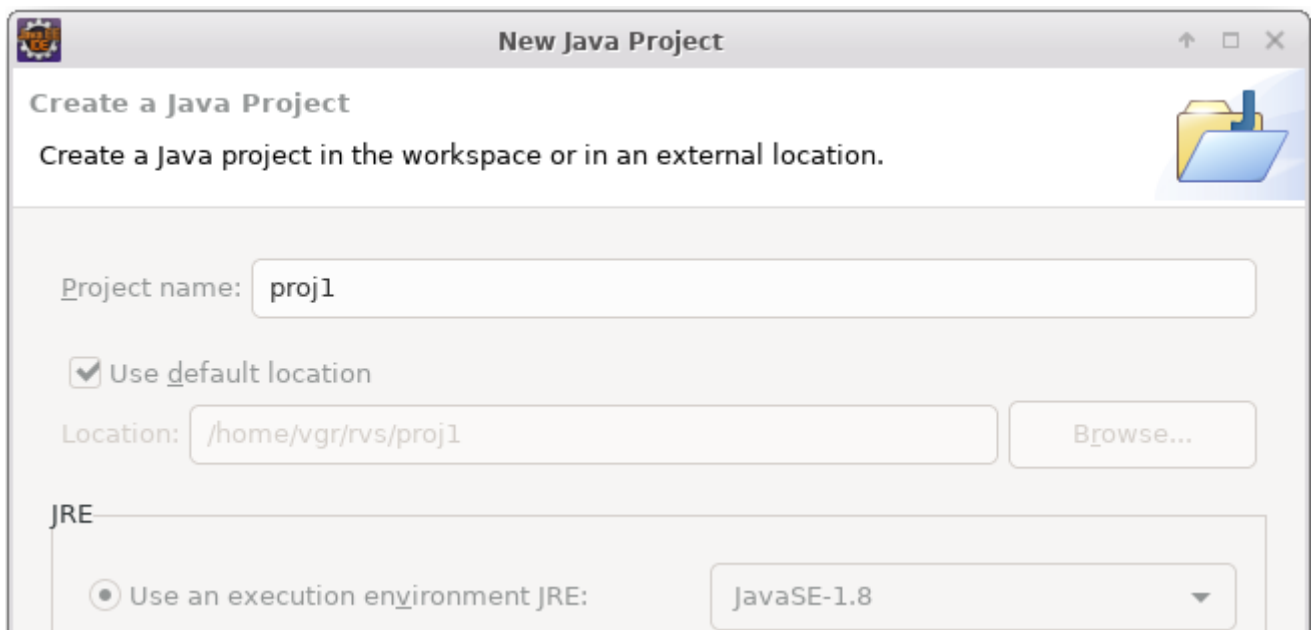


Рисунок 2.7 — Выбор названия проекта

В окне «**Package Explorer**» выделим мышкой **proj1/src** и правой кнопкой ак-

тивируем меню **New** → **Class**, а появившееся окно заполним, как показано на рисунке 2.8.

New Java Class

Java Class
Create a new Java class.

Source folder:

Package:

☐ Enclosing type:

Name:

Modifiers: ☒ public ☐ package ☐ private ☐ protected
☐ abstract ☐ final ☐ static

Superclass:

Interfaces:

Which method stubs would you like to create?

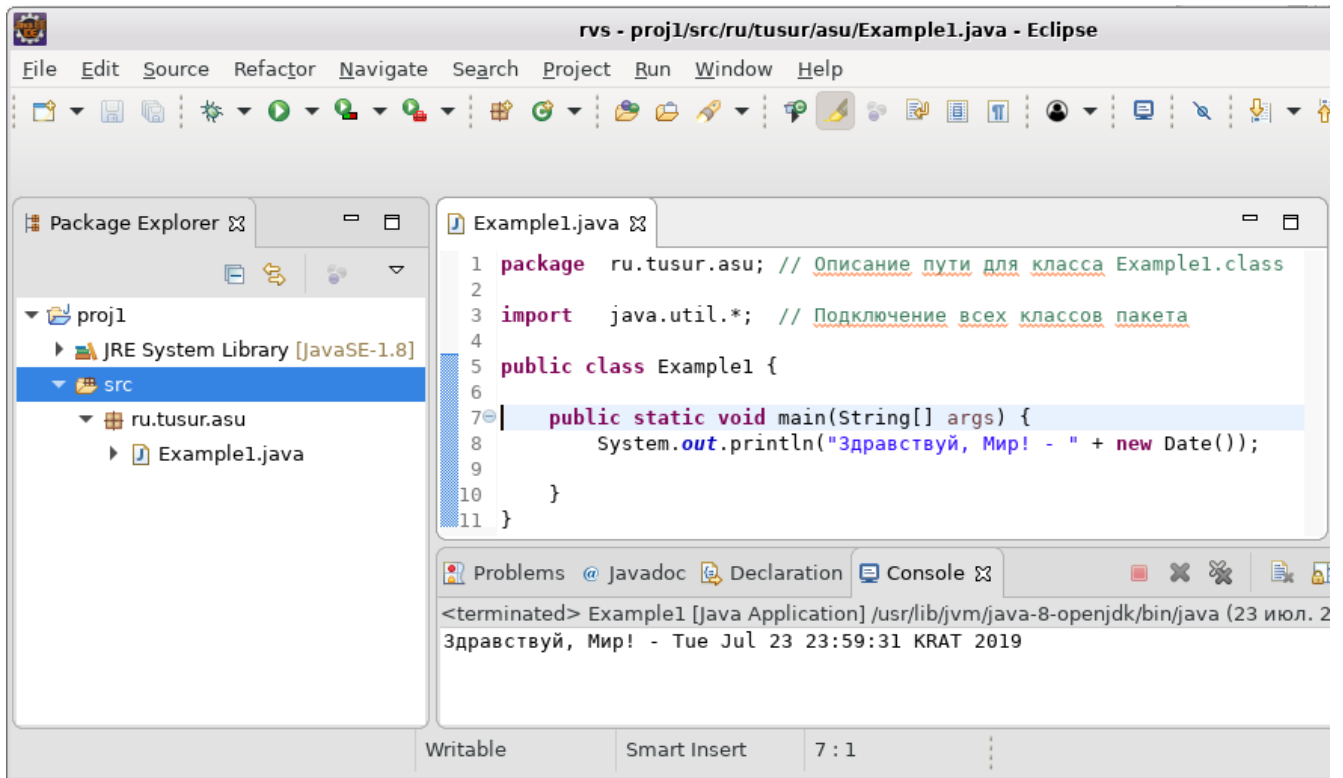
☒ public static void main(String[] args)
☐ Constructors from superclass
☒ Inherited abstract methods

Do you want to add comments? (Configure templates and default value [here](#))
☐ Generate comments

Рисунок 2.8 — Окно создания класса Example1 в проекте proj1

Завершив создание класса кнопкой «**Finish**», мы получаем в среде IDE вкладку **Example1.java** с исходным текстом шаблона класса **Example1**. Внесём в этот шаблон изменения согласно листингу 2.3 и сохраним результат. Проект готов к запуску.

Запустив программу на выполнение, мы по содержимому рисунка 2.9 убеждаемся, что результат ее работы полностью совпадает с результатом рисунка 2.3.

Рисунок 2.9 — Результат запуска проекта `proj1`

Теперь проведём анализ каталога `/home/vgr/rvs/proj1`, где расположены каталоги нашего проекта:

- директория **`bin`** соответствует дереву каталогов для классов проекта;
- директория **`src`** соответствует дереву каталогов для исходных текстов проекта;
- рисунок 2.10 показывает места хранения **`Example1.class`** и **`Example1.java`**.

Терминал						
Левая панель		Файл		Команда		Правая панель
~ / rvs / proj1 / bin / ru / tusur / asu		Имя		Размер		~ / rvs / proj1 / src / ru / tusur / asu
..		-ВВЕРХ-		Время правки		..
Example1.class		769		июл 23 22:59		Example1.java
				июл 23 23:59		

Рисунок 2.10 — Места хранения файлов `Example1.class` и `Example.java`

Убедившись, что Eclipse также придерживается правил хранения классов, задаваемых оператором **`package`**, и приступим к созданию **`jar`**-архива нашего проекта. Для этого, в окне «**`Package Explorer`**» выделим мышкой проект **`proj1`** и правой кнопкой активируем меню «**`Export...`**».

Далее, в появившемся окне «**`Export`**» выбираем «**`Runnable JAR file`**» и кнопкой «**`Next`**» переходим к окну, показанному на рисунке 2.11 и заполняем его с учётом того, что архив имеет имя **`proj1.1.jar`**, а директория - `/home/vgr/src/rvs`.

Теперь, нажав кнопку «**`Finish`**», можно сравнить полученный архив с архивом **`proj1.jar`**.

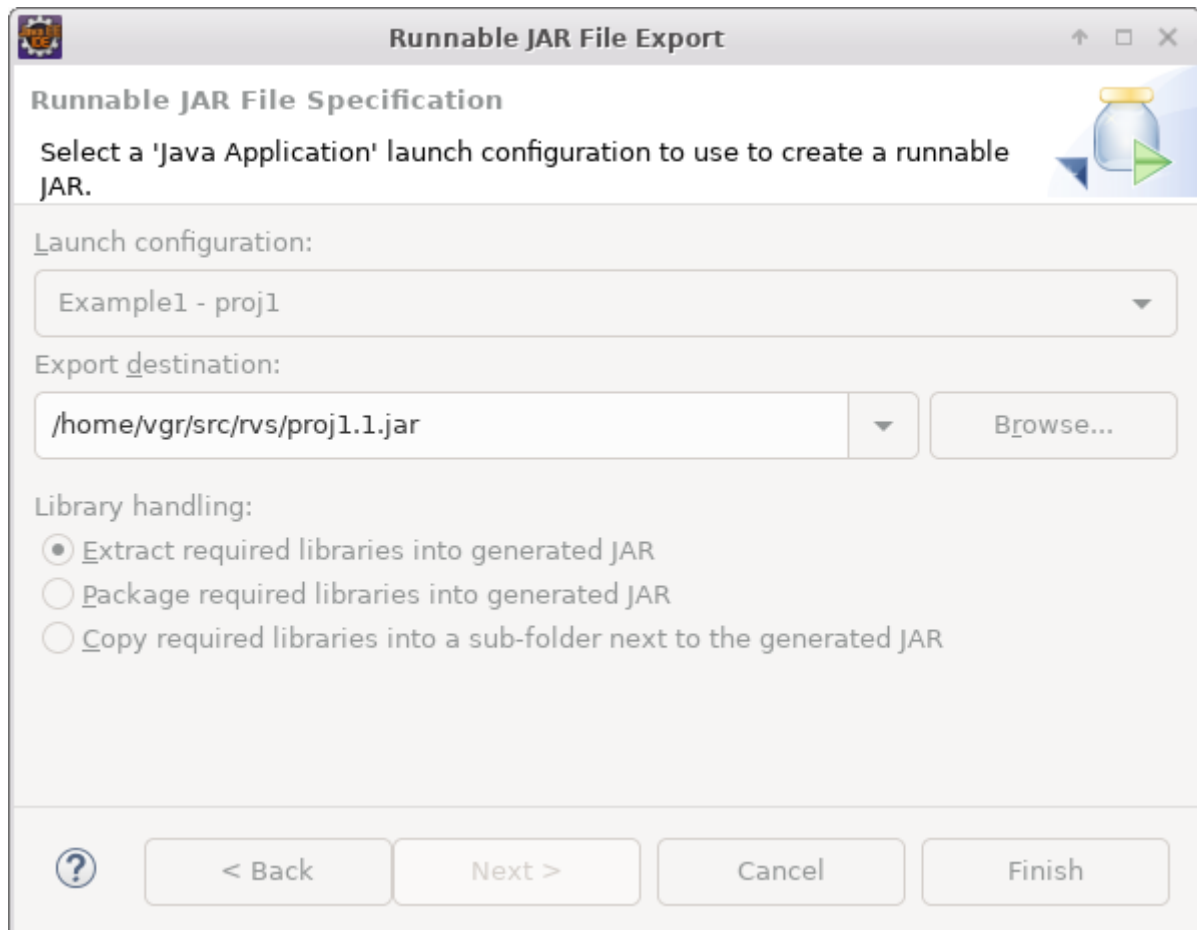


Рисунок 2.11 — Выбор местоположения архива проекта proj1

Запустив программу командой `java -jar proj1.1.jar`, убеждаемся, что она работает также, как и программа архива **proj1.jar**. Можно ещё распаковать архивы командами:

```
jar xf proj1.jar
jar xf proj1.1.jar
```

и сравнить их содержимое.

Теперь, завершая данный подраздел, можно находиться в уверенности, что студенту представлены все инструменты языка Java, необходимые для изучения последующего материала этой главы.

2.2 Классы и простые типы данных

Как было отмечено ранее, Java является сильно типизированным языком объектно-ориентированного программирования, программное обеспечение которого группируется в виде специализированных пакетов, а дистрибутивы группируются по четырём платформам: *J2EE*, *J2SE*, *J2ME* и *Java Card*. Общая взаимосвязь ПО языка является сложной, но хорошо продуманной системой. Например, официальная документация на платформу Java SE версии 6 группирует все технологии в виде концептуальной диаграммы [30], представленной на рисунке 2.12. Следует также заметить, что на самом сайте эта диаграмма является интерактивной и позволяет легко переходить на соответствующие разделы документации.

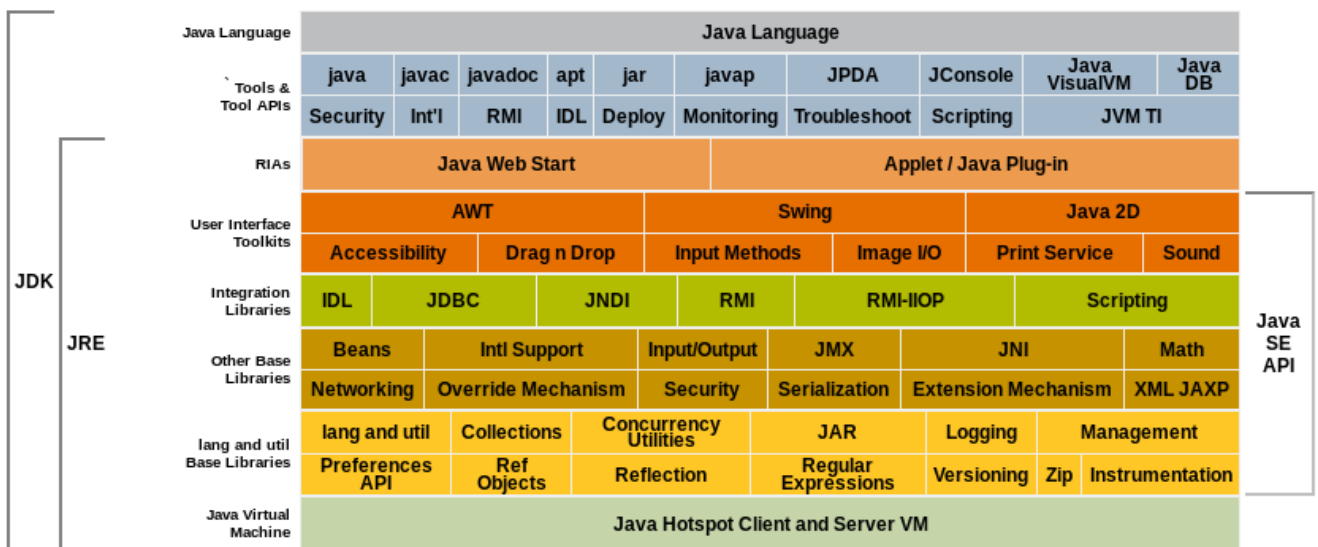


Рисунок 2.12 — Концептуальная диаграмма технологий Java SE версии 6 [30]

Тема данного подраздела посвящена синтаксическим конструкциям языка Java, соответствующим общему понятию ООП «класс». Простейшие примеры использования этого понятия были рассмотрены в предыдущем подразделе, где также отмечалось, что инструментальная поддержка соответствующих конструкций сосредоточена в пакете ***java.lang***. Если говорить более точно, то базовой конструкцией языка Java является:

```
public class Object,
```

где *Object* адресуется как ***java.lang.Object*** и является суперклассом для всех других классов. Все объекты, созданные классами языка Java, включают методы класса *Object*.

В целом учебный материал излагается в справочном стиле. Расчёт идёт на знание студентом теории и синтаксических конструкций языка C++. Подраздел также не содержит примеров, предполагая дополнительные занятия по лабораторным работам. Изложение учебного материала начинается с представления опера-

торов и семантики простых типов данных, имеющих соответствующий эквивалент во всех языках программирования.

2.2.1 Операторы и простые типы данных

Язык Java имеет восемь простейших типов данных, имеющих аналоги в других языках и перечисленных в таблице 2.1.

Таблица 2.1 - Простые типы языка Java

Тип	Длина (байт)	Диапазон или набор значений
boolean	не определено	true, false
byte	1	-128..127
char	2	0..2 ¹⁶ -1, или 0..65535
short	2	-2 ¹⁵ ..2 ¹⁵ -1, или -32768..32767
int	4	-2 ³¹ ..2 ³¹ -1, или -2147483648..2147483647
long	8	-2 ⁶³ ..2 ⁶³ -1, или примерно -9.2·10 ¹⁸ ..9.2·10 ¹⁸
float	4	-(2·2 ⁻²³)·2 ¹²⁷ ..(2·2 ⁻²³)·2 ¹²⁷ , или примерно -3.4·10 ³⁸ ..3.4·10 ³⁸
double	8	-(2·2 ⁻⁵²)·2 ¹⁰²³ ..(2·2 ⁻⁵²)·2 ¹⁰²³ , или примерно -1.8·10 ³⁰⁸ ..1.8·10 ³⁰⁸

Типы **float** и **double** имеют также специальные значения бесконечности и **NaN**, которые мы не будем здесь рассматривать. Кроме того, имеется набор зарезервированных слов, которые перечислены в таблице 2.2.

Таблица 2.2 - Зарезервированные слова языка Java

abstract	boolean	break	byte	byvalue
case	cast	catch	char	class
const	continue	default	do	double
else	extends	false	final	finally
float	for	future	generic	goto
if	implements	import	inner	instanceof
int	interface	long	native	new
null	operator	outer	package	private
protected	public	rest	return	short
static	super	switch	synchronized	this
throw	throws	transient	true	try
var	void	volatile	while	

Перечень операторов языка Java представлен в таблице 2.3.

Таблица 2.3 - Операторы языка Java

+	+=	-	-=
*	*=	/	/=
	=	^	^=
&	&=	%	%=
>	>=	<	<=
!	!=	++	--
>>	>>=	<<	<<=
>>>	>>>=	&&	
==	=	~	?:
	instanceof	[]	

Назначение перечисленных обозначений можно уточнить по любому руководству языка Java, мы отметим, что неявные преобразования встроенных типов полностью совпадает с преобразованием типов в языках C/C++, согласно правилам:

1. Если один операнд имеет тип *double*, другой тоже преобразуется к типу *double*.
2. Иначе, если один операнд имеет тип *float*, другой тоже преобразуется к типу *float*.
3. Иначе, если один операнд имеет тип *long*, другой тоже преобразуется к типу *long*.
4. Иначе оба операнда преобразуются к типу *int*.

2.2.2 Синтаксис определения классов

Кроме перечисленных восьми простых типов данных, все остальные типы языка Java относятся к классам, а чтобы иметь возможность преобразования между ними, используются классы-обертки представленные в таблице 2.4.

Таблица 2.4 - Простые типы данных и классы-обертки языка Java

boolean	byte	char	short	int	long	float	double
Boolean	Byte	Character	Short	Integer	Long	Float	Double

Общее определение класса задаётся следующим синтаксическим шаблоном

при негласном соглашении, что имя класса должно начинаться с *заглавной* буквы:

```
[модификаторы] class имя-класса
    [extends суперкласс]
    [implements список_интерфейсов]
{
    ...
    // переменные и методы класса
    ...
}
```

В данном и последующих определениях, ключевые слова выделяются полужирным шрифтом, а необязательные элементы конструкции выделены скобками «[...]». Соответственно, в шаблоне класса - выделены три ключевых слова:

- **class** – слово, являющееся обязательным при объявлении класса;
- **extends** – слово, которое используется при объявлении нового класса, на основе уже созданного класса (суперкласса); объявляемый класс называется классом-потомком уже существующего класса;
- **implements** – слово, которое указывает, что класс поддерживает методы, определённые в последующем списке интерфейсов; отдельные элементы списка интерфейсов разделяются запятыми.

Объявляемый класс может не иметь модификатора, тогда «*областью его видимости*» является файл, в котором он определён. Если модификатор присутствует, то он может быть одним из трёх видов:

- **abstract** — класс, имеющий хотя бы один абстрактный метод (метод объявлен, но не имеет реализации); может иметь классы-потомки;
- **public** — открытые (публичные) классы, которые можно использовать: внутри программы, внутри пакета программ или за пределами пакета программ;
- **final** — класс, который не может иметь потомков; примерами таких классов являются: **String** и **StringBuffer**.

2.2.3 Синтаксис и семантика методов

Общее определение метода класса задаётся следующим синтаксическим шаблоном при негласном соглашении, что имя его имя должно начинаться со *строчной* буквы:

```
[спецификатор_доступа]
    [static] [abstract] [final] [native] [synchronized]
    тип_данных имя_метода
    ([параметр1], [параметр2], ...)
    [throws список_исключений]
```

Спецификатор_доступа — может отсутствовать, но в любом случае он определяет область видимости не только методов, но и переменных:

- **public** - имеется видимость переменной константы или метода из любого класса;
- **private** — доступ разрешается только внутри класса;
- **protected** — доступ разрешается как внутри класса, в котором даётся определение, так и внутри классов-потомков;
- **<пробел>** - доступ разрешается для любых классов пакета, в котором класс определён.

Модификаторы — определяют особенности использования методов, если к ним имеется доступ:

- **static** — метод, принадлежащий классу и используемый всеми объектами класса; обращение к методу производится указанием имени класса и, после точки, указывается метод (оператор **new** указывать не нужно);
- **abstract** — объявляет метод, но не даёт его реализации;
- **final** — метод нельзя будет переопределять (перегружать) в классах-потомках;
- **native** — определяет метод, реализованный на других языках, отличных от языка Java, например, - на языке C;
- **synchronized** — при обращении к методу, доступ к остальным методам класса прекращается до завершения работы данного метода.

Тип_данных — возвращаемый методом тип результата, к которым относятся: типы объявленных классов, простейшие типы данных или **void**, если метод ничего не возвращает.

throws список_исключений — если в методе используются классы или ситуации, приводящие к исключениям, то указывает список необрабатываемых в методе исключений; имена исключений в списке разделяются запятыми.

В качестве замечания отметим, что в языке Java, как и в языках C/C++, имеется особый метод **main(...)**, синтаксис которого имеет стандартный вид:

```
public static void main(String[] args) {
    // Команды языка Java
}
```

Если такой метод присутствует в каком-либо классе, то такой класс может быть запущен как самостоятельная программа. Обратите внимание, что этот метод должен быть публичным (*public*) и статическим (*static*), чтобы загрузчик программ мог его увидеть и запустить на выполнение.

Теперь, рассмотрим особые конструкции Java — *интерфейсы*, которые можно рассматривать как специальные типы классов.

2.2.4 Синтаксис определения интерфейсов

Интерфейсы были предложены как альтернатива *абстрактным* типам классов, чтобы расширить возможности последних. Общий синтаксис объявления интерфейсов имеет вид:

```
[public] interface имя_интерфейса
    [extends интерфейс1, интерфейс2, ...]
{
    [тип_переменной1 имя_переменной1 = значение1;]
    [тип_переменной2 имя_переменной2 = значение2;]
    ...
    [метод_доступа] тип_метода название_метода1([тип_аргумента1
аргумент1], [тип_аргумента2 аргумент2], ...);
    [метод_доступа] тип_метода название_метода2([тип_аргумента1
аргумент1], [тип_аргумента2 аргумент2], ...);
    ...
}
```

В определении интерфейса может присутствовать необязательный модификатор **public** и два ключевых слова:

- **interface** – слово, являющееся обязательным при объявлении интерфейса;
- **extends** – слово, которое используется при объявлении нового интерфейса, включающего один или более уже объявленных интерфейсов.

Тело интерфейса может содержать:

- объявленные и проинициализированные типы данных;
- описание не реализованных методов.

Интерфейсы можно интерпретировать как незавершённые классы, подобные абстрактным классам, но в отличие от последних они допускают объявление объединений других интерфейсов. Как показано выше, список интерфейсов может присутствовать при объявлении класса, подключая к классу методы, которые должны быть в нем реализованы.

2.2.5 Объекты и переменные

В отличие от языка C++, в языке Java имеются только динамически создаваемые объекты и все переменные языка являются объектными ссылками. С другой стороны, в языке Java отсутствует понятие ссылки и соответствующие операции адресации, доступные в языках C/C++. Чтобы наглядно показать сказанное, обратимся к исходному тексту класса Example2, представленному на листинге 2.4.

Листинг 2.4 — Исходный текст класса *Example2* из среды Eclipse EE

```

package ru.tusur.asu;

public class Example2 {

    // Объявление переменной типа String
    String text1;

    // Объявление статического массива из двух целых чисел
    static int[] im = new int[2];

    // Конструктор класса
    Example2(String text2, int n) {
        // Присваиваем значение части переменных
        text1 = text2;
        im[0] = n;
    }

    // Первый (обычный) метод
    public void print1() {
        System.out.println(text1 + ":"
            + " im[0]=" + im[0]
            + " im[1]=" + im[1]);
    }

    // Второй (статический) метод
    public static void print2() {
        System.out.println("Статический метод:"
            + " im[0]=" + im[0]
            + " im[1]=" + im[1]);
    }
}

```

Этот пример демонстрирует объявление класса *Example2* со следующими особенностями:

- объявляется объектная переменная *text1*, которая неявно инициализируется «пустой» строкой;
- объявляется *статический* массив из двух целых чисел *im*, который неявно инициализируется нулевыми значениями и становится объектом класса;
- объявляется специальный метод *Example2(...)*, который является конструктором класса и, при создании объекта, будет инициализировать значение объекта *text1*, а также — первый элемент массива *im*; таких конструкторов может быть много, но они должны отличаться списком аргументов;
- объявляется обычный *публичный* метод *print1*, который с помощью *статического* метода *println(...)* объекта *java.lang.System.out* распечатывает содержимое строки *text1* и массива *im*;
- объявляется *публичный статический* метод *print2*, который с помощью того же метода *println(...)* распечатывает содержимое массива *im*.

Теперь, чтобы показать детали использования перечисленных особенностей класса *Example2*, определим класс *Example3*, представленный на листинге 2.5.

Листинг 2.5 — Исходный текст класса *Example3* из среды Eclipse EE

```

package ru.tusur.asu;

public class Example3 {

    public static void main(String[] args) {

        // Объявляем и создаём объект obj1 класса Example2
        Example2 obj1 =
            new Example2("Первый вызов обычного метода", 1);

        // Вызываем обычный метод объекта obj1
        obj1.print1();

        // Вызываем статический метод класса Example2
        Example2.print2();

        // Объявляем и инициализируем объект obj2 класса Example2
        Example2 obj2 = obj1;

        // Вызываем обычный метод объекта obj2
        obj2.print1();

        // Пересоздаем объект obj1 класса Example2
        obj1 =
            new Example2("Второй вызов обычного метода", 2);

        // Вызываем обычный метод объекта obj1
        obj1.print1();

        // Вызываем статический метод класса Example2
        Example2.print2();

        // Удаляем объект для obj1 и obj2: "сборка мусора"
        obj1 = null;
        obj2 = null;

    }
}

```

Реализуем оба класса *Example2* и *Example3* в одном проекте **proj2** среды IDE Eclipse EE. Запуск класса **Example3** на исполнение приведёт к созданию объектов класса *Example2* и выполнению его различных методов. Результат работы проекта **proj2** представлен на рисунке 2.13.

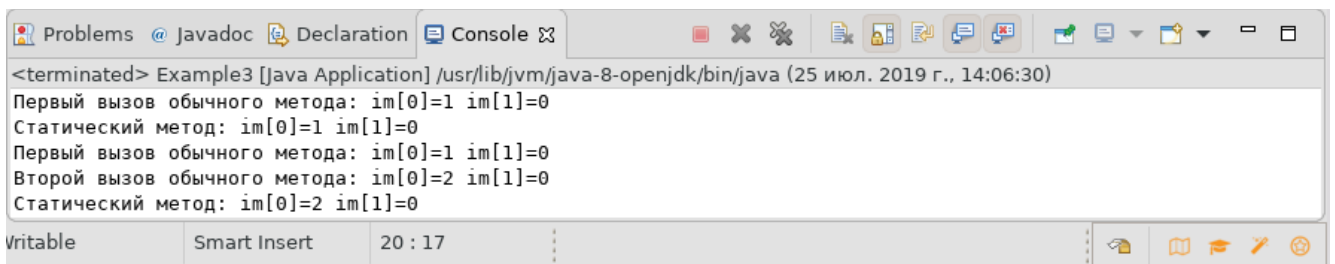


Рисунок 2.13 — Результат запуска проекта proj2 в среде Eclipse EE

2.3 Управляющие операторы языка

Управляющие операторы языка Java практически полностью совпадают с аналогичными конструкциями языков C/C++, поэтому их описание приведено в шаблонном виде и без комментариев.

Оператор if

```
if (логическое_условие)
{
    //Блок операторов, выполняемых при истинности
    //логического условия
}
else if (логическое_условие)
{
    //Блок операторов, выполняемых при истинности
    //логического условия
}
...
else
{
    //Блок операторов, выполняемых при не истинности
    //логического условия
}
```

Оператор switch

```
switch (Проверяемое_выражение)
{
    case значение1:
        ...
        //Блок выполняемых операторов
        break;
    case значение2:
        ...
        //Блок выполняемых операторов
        break;
    default:
        ...
        //Блок выполняемых операторов
}
```

Проверяемое_выражение — это один из перечисляемых типов данных (переменных): *int*, *boolean*, *char* или *byte*.

Оператор цикла while

```
while (логическое_условие)
{
    ...
    //Блок выполняемых операторов
}
```

Оператор цикла do ... while

```
do
{
    ...
    //Блок выполняемых операторов
}
while (логическое_условие)
```

Оператор цикла for

```
for (инициализация; условие; итератор)
{
    ...
    //Блок выполняемых операторов
}
```

Операторы перехода

Операторы перехода используются для более гибкого управления работой циклических операторов. Всего имеется три таких операторов: **break**, **continue** и **return**:

- **break** - обеспечивает безусловный выход из цикла на один уровень;
- **continue** — позволяет игнорировать текущую итерацию цикла и перейти к следующей итерации цикла;
- **return** — используется для безусловного возврата из метода; для методов возвращающих значение, указывает возвращаемый объект или простой тип данных.

2.4 Потоки ввода-вывода

Излишне утверждать, что в разработке сетевых приложений ввод-вывод играет очень важную роль. В языке Java для организации ввода-вывода используются специальные классы, которые должны агрегироваться во вновь определяемые классы и создаваемые объекты. Технология программирования на языке Java использует два общих подхода:

- методы *стандартного* ввода-вывода;
- *обобщённые* методы, основанные на классах *InputStream* и *OutputStream*.

Рассмотрим сначала первый подход.

2.4.1 Стандартный ввод-вывод

Все языки программирования содержат средства *стандартного* ввода/вывода. Язык Java не является исключением. Чтобы раскрыть семантику этого факта, рассмотрим рисунок 2.14, хорошо известный студентам по дисциплине «Операционные системы».

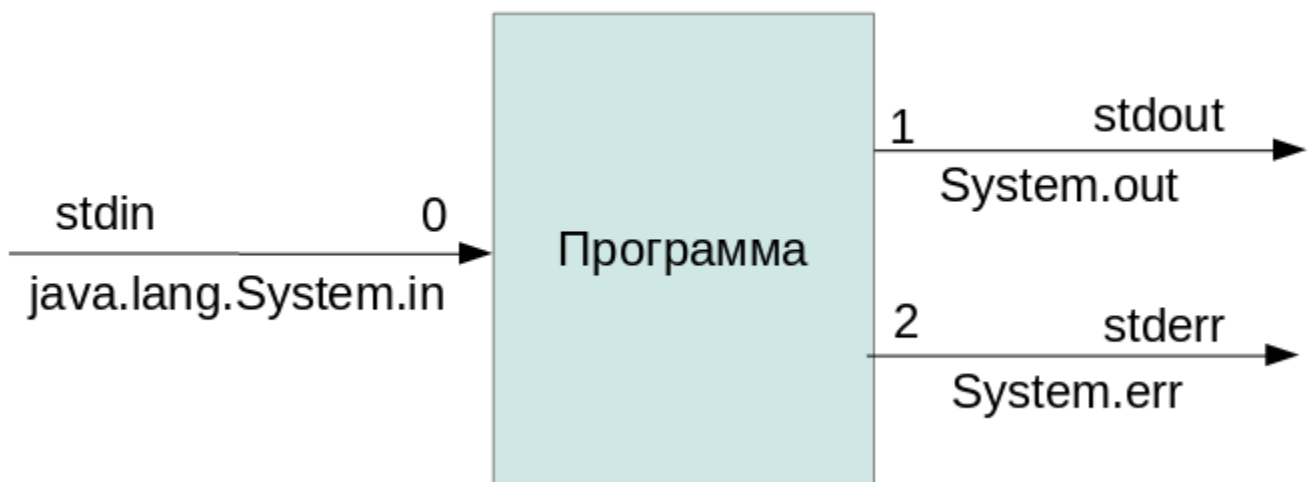


Рисунок 2.14 — Демонстрация семантики стандартного ввода-вывода

Здесь показано, что любая нормально запущенная программа ОС, сразу имеет:

- в командном интерпретаторе - доступные дескрипторы (устройства) 0, 1 и 2;
- в языке C — доступные потоки *stdin*, *stdout* и *stderr*;
- в языке Java — доступные объекты *System.in*, *System.out* и *System.err*.

Документация по языку Java формально определяет объекты стандартного ввода-вывода в пакете *java.lang*:

```
public static final InputStream in;
```

```
public static final PrintStream out;  
public static final PrintStream err;
```

Таким образом, объекты стандартного ввода-вывода являются *статическими*, что уже отмечалось в рассмотренных примерах, и *финальными*, т.е. — не имеющими возможность иметь дочерние классы.

Объект ввода ***in*** порожден из класса ***InputStream*** и имеет три основных метода:

<code>public abstract int read() throws IOException</code>	Читает по одному байту из потока ввода и возвращает целое число в пределах значений от 0 до 255.
<code>public int read(byte[] b) throws IOException</code>	Читает содержимое потока ввода и помещает его в массив байт <i>b</i> . Возвращает число прочитанных байт.
<code>public int read(byte[] b, int off, int len) throws IOException</code>	Читает <i>len</i> байт из потока ввода и помещает его в массив байт <i>b</i> со смещением <i>off</i> . Возвращает число прочитанных байт.

Объекты ***out*** и ***err*** порождены классом ***PrintStream***, который сам порождён от класса ***OutputStream***. Они имеют много методов, но мы представим только четыре:

<code>public void print(String s)</code>	Обеспечивает вывод строки, производя конкатенацию и преобразование типов. В конце строки <i>не добавляет</i> символ перевода строки.
<code>public void println(String s)</code>	Обеспечивает вывод строки, производя конкатенацию и преобразование типов. В конце строки <i>добавляет</i> символ перевода строки.
<code>public void write(byte[] b) throws IOException</code>	Выводит массив байт <i>b</i> .
<code>public void write(byte[] buf, int off, int len) throws IOException</code>	Выводит <i>len</i> байт из массива <i>b</i> , начиная со смещения <i>off</i> .

Программируя ввод-вывод, необходимо всегда помнить, что многие методы порождают исключения ***IOException***, поэтому, определяя собственные методы, необходимо проектировать их обработку или игнорирование. Рассмотрим это примером программы, представленной на листинге 2.6, которая в цикле читает символы с буфера клавиатуры и выводит их на стандартный вывод в виде пары символ-значение.

Листинг 2.6 — Исходный текст класса Example4 из среды Eclipse EE

```
package ru.tusur.asu;  
  
import java.io.IOException; // Для обработки исключения  
  
public class Example4 {
```

```

public static void main(String[] args) {
    // Определение целочисленной рабочей переменной
    int myKey;

    System.out.println("Для выхода из программы, нажмите:\n" +
        "Ctrl-Z в DOS/Windows\n" +
        "Ctrl-D в UNIX/Linux");

    try
    {
        while ((myKey = System.in.read()) != -1)
        {
            System.out.println((char)myKey + ": " + myKey);
        }
    }
    catch(IOException e)
    {
        System.out.println(e.getMessage());
    }
}

```

Другой пример, использующий динамически определяемый байтовый буфер, представлен на листинге 2.7.

Листинг 2.7 — Исходный текст класса Example5 из среды Eclipse EE

```

package ru.tusur.asu;

import java.io.IOException; // Для обработки исключения

public class Example5 {

    public static void main(String[] args) {

        byte[] b; // Объявление байтового массива
        int n;     // Переменная хранения размера массива

        System.out.println("Программа Example5 делает 5 циклов:\n");
        try
        {
            for (int i = 0; i < 5; i++)
            {
                // Ожидаем ввод и определяем число символов
                // в буфере терминала
                while ((n = System.in.available()) == 0) ;
                b = new byte[n]; // Создаём массив буфера b
                System.in.read(b); // Читаем символы в буфер b
                System.out.println(new String(b)); // Печатаем весь буфер
            }
        }
        catch(IOException e)
        {
            System.out.println(e.getMessage());
        }
    }
}

```

2.4.2 Классы потоков ввода

Для решения общих задач ввода-вывода язык Java использует специальный пакет **java.io**, обладающий набором классов и интерфейсов, позволяющих управлять процессами внутри различных потоков. Базовым классом для методов ввода является **InputStream**. Его диаграмма наследования представлена на рисунке 2.15, а общий синтаксис объявления имеет вид:

```
InputStream имя_объекта =  
    new Конструктор_одного_из_классов_ввода
```

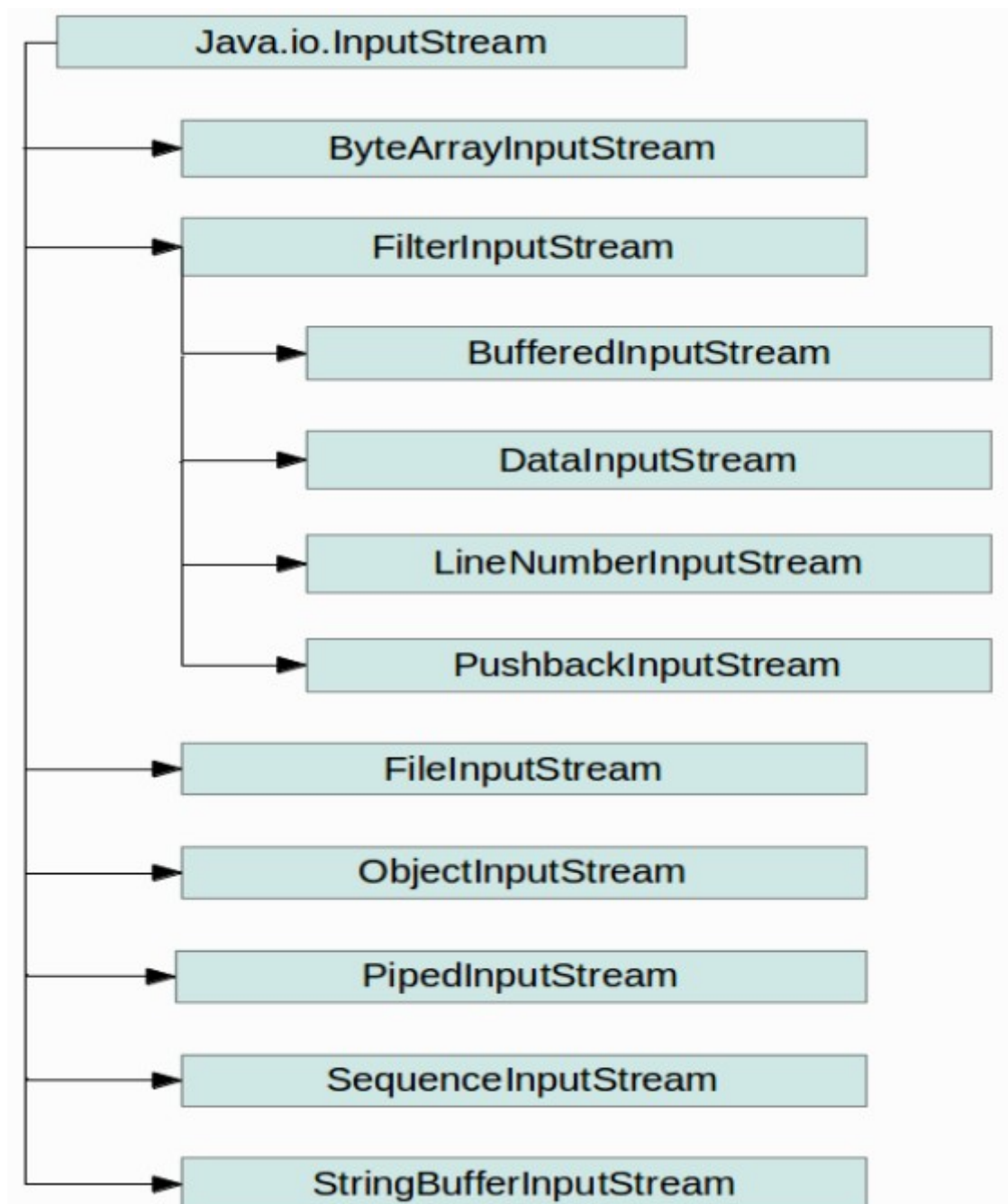


Рисунок 2.15 — Диаграмма наследования класса `InputStream`

Объектам класса ***InputStream*** доступны все методы, которые также доступны стандартному вводу. Обычно, на практике используются ещё методы:

<i>available()</i>	Возвращает число байт, доступных для чтения в потоке ввода.
<i>skip(long N)</i>	Пропускает во входном потоке N байт.
<i>close()</i>	Закрывает входной поток ввода.

Рассмотрим программу листинга 2.8, демонстрирующую использование входного потока ***FileInputStream*** на примере чтения текстового файла и отображения результата чтения на стандартный вывод.

Листинг 2.8 — Исходный текст класса Example6 из среды Eclipse EE

```
package ru.tusur.asu;

import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;

public class Example6 {

    public static void main(String[] args) {

        // Объявление буфера и счётчика читаемых байт
        byte[] b;
        int n;

        System.out.println("Демонстрация потока FileInputStream.\n"
            + "Чтение файла: /home/vgr/src/rvs/Example1.java\n"
            + "-----\n");

        try{
            InputStream in =
                new FileInputStream("/home/vgr/src/rvs/Example1.java");

            while ((n = in.available()) > 0) // Читаем в цикле
            {
                b = new byte[n];
                in.read(b);
                System.out.print(new String(b));
            }

            in.close();          // Закрываем поток ввода

        }catch(IOException e)    // Обрабатываем исключение
        {
            System.out.println(e.getMessage());
        }

    }
}
```

Следует обратить внимание, что чтение файла осуществляется в цикле, потому что входной буфер всегда ограничен в размере и реальное чтение, в общем случае, происходит блоками переменной длины.

2.4.3 Классы потоков вывода

Для реализации общих задач вывода информации в языке Java используется базовый класс ***OutputStream***, также принадлежащий пакету ***java.io***. Диаграмма наследования его классов/потоков приведена на рисунке 2.16, а общий синтаксис объявления имеет вид:

OutputStream имя_объекта =
new Конструктор_одного_из_классов_вывода

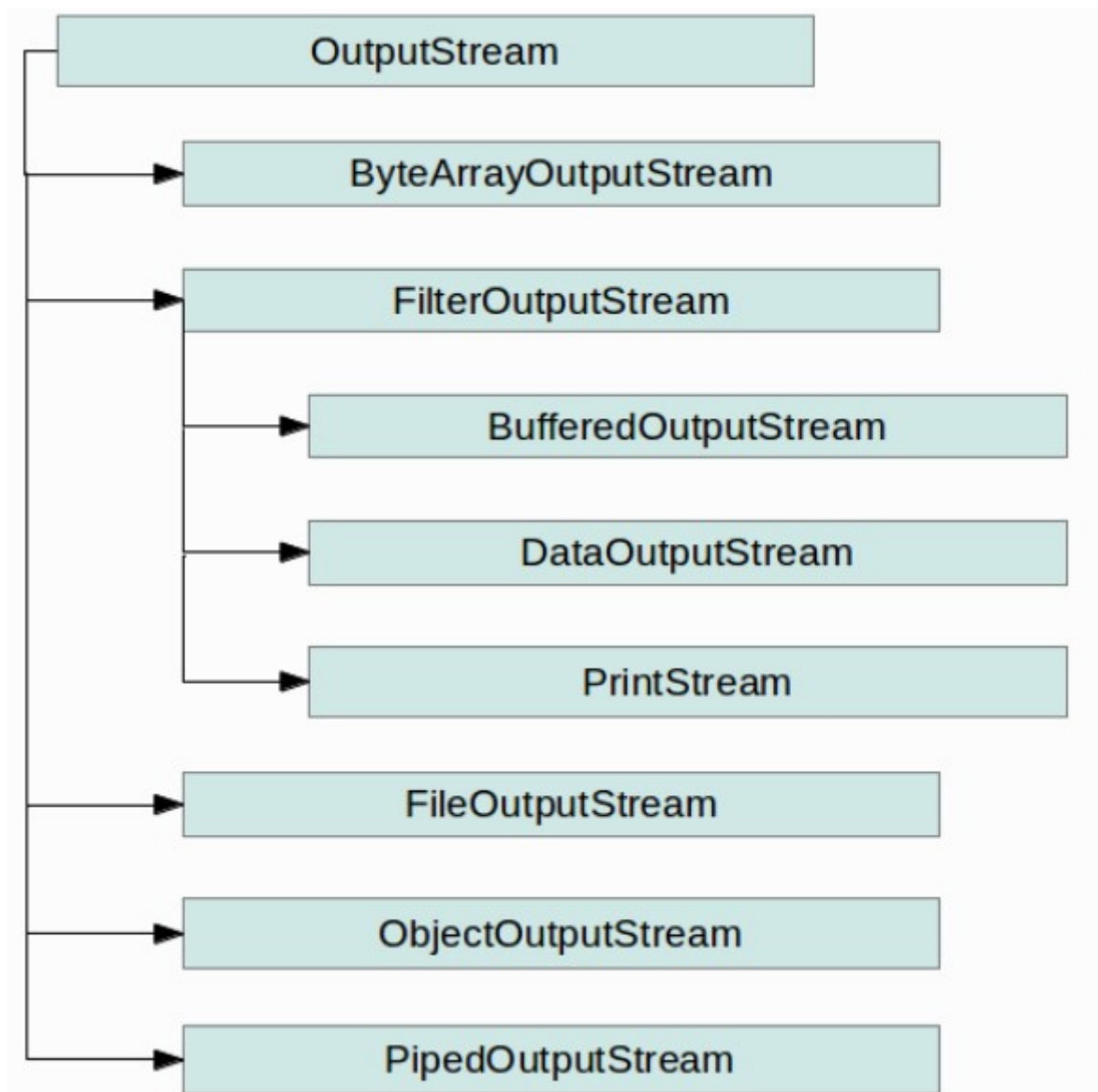


Рисунок 2.16 — Диаграмма наследования класса ***OutputStream***

Обратим внимание, что на диаграмме присутствует класс стандартного вывода ***PrintStream***, который является дочерним от класса ***FilterOutputStream***. Сам же класс ***OutputStream*** имеет много методов, подобных стандартному выводу, а наиболее значимые из них:

<code>write(int b)</code>	Запись одного байта <i>b</i> в поток вывода.
<code>write(byte[] b)</code>	Запись массива байт <i>b</i> в поток вывода.
<code>write(byte[] b, int off, int len)</code>	Запись в поток вывода части массива <i>b</i> длиной <i>len</i> , начиная с позиции <i>off</i> .
<code>flush()</code>	Принудительный сброс данных из промежуточного буфера в поток вывода.
<code>close()</code>	Закрывает выходной поток вывода.

Рассмотрим программу листинга 2.9, демонстрирующую использование выходного потока ***FileOutputStream*** на примере создания текстового файла с помощью объекта класса ***OutputStream***, последующего чтения этого файла и отображения результата чтения на стандартный вывод.

Листинг 2.9 — Исходный текст класса Example7 из среды Eclipse EE

```
package ru.tusur.asu;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

public class Example7 {

    public static void main(String[] args) {

        // Объявление буфера и счётчика читаемых байт
        byte[] b;
        int n;

        System.out.println("Демонстрация потока FileOutputStream.\n"
            + "Создание и чтение файла: /home/vgr/demo7.txt  \n"
            + "-----\n");

        try{

            OutputStream out =
                new FileOutputStream("/home/vgr/demo7.txt");

            // Пишем строки в файл
            out.write("Привет всем друзьям!\n".getBytes());
            out.write("Мы написали программу записи в файл.\n".getBytes());
            out.write("Посмотрим, что из этого получится...\n".getBytes());
            out.flush();          // Сбрасываем поток вывода
            out.close();           // Закрываем поток вывода

            InputStream in =
                new FileInputStream("/home/vgr/demo7.txt");

            while ((n = in.available()) > 0) // Читаем в цикле
            {
                b = new byte[n];
                in.read(b);
            }
        }
    }
}
```

```

        System.out.print(new String(b));
    }

    in.close(); // Закрываем поток ввода
} catch (IOException e)
{
    System.out.println(e.getMessage());
}
}
}

```

Следует особо обратить внимание, что при записи в файл:

- используется метод **write(byte[] b)**, оперирующий с заранее созданным массивом байт;
- строки символов, заключённые в двойные кавычки, являются объектами и к ним применяется *статический* метод **getBytes()** из класса **java.lang.String**, динамически преобразующий объект строки в массив байт;
- перед закрытием любого выходного потока методом **close()** всегда нужно сбрасывать промежуточный буфер вывода методом **flush()**.

Завершая изучение общих вопросов программирования на языке Java, рассмотрим пример использования класса **File**, представленный на листинге 2.10 и демонстрирующий ряд характеристик уже созданного файла **/home/vgr/demo7.txt**.

Листинг 2.10 — Исходный текст класса *Example8* из среды *Eclipse EE*

```

package ru.tusur.asu;

import java.io.File;
import java.util.Date;

public class Example8 {

    public static void main(String[] args) {
        // Определяем и создам объект класса File
        File myf = new File("/home/vgr/demo7.txt");

        System.out.println("Это - файл: " + myf.isFile());
        System.out.println("Это - директория: " + myf.isDirectory());

        System.out.println("Можно писать в файл: " + myf.canWrite());
        System.out.println("Можно читать файл: " + myf.canRead());
        System.out.println("Можно запускать файл: " + myf.canExecute());

        System.out.println("Имеет родителя: " + myf.getParent());
        System.out.println("Имеет путь: " + myf.getPath());
        System.out.println("Имеет имя: " + myf.getName());

        System.out.println("Длина файла: " + myf.length());
        System.out.println("Последняя модификация: "
            + new Date(myf.lastModified()));
    }
}

```

2.5 Управление сетевыми соединениями

Управление сетевыми соединениями является важной темой для нашей дисциплины. В языке Java имеется специальный пакет **java.net**, содержащий базовые классы и методы для работы со стеком протоколов TCP/IP и предполагающий, что компьютер студента подключён к сети и может пользоваться службой доменных имён (DNS). Мы исходим из того, что студентом уже изучен курс «Сети и телекоммуникации», поэтому изложение материала не содержит терминов и определений, излагаемых в этой дисциплине. Следуя содержанию ПО пакета **java.net**, в данном подразделе выделяются следующие части:

- адресация в Internet, основанная на классе **InetAddress**;
- адресация в Internet на основе классов **URL** и **URLConnection**;
- сокет протокола **TCP**;
- сокет протокола **UDP**;
- пример технологии клиент-сервер.

2.5.1 Адресация на базе класса **InetAddress**

Работая с Internet в стеке протоколов **TCP/IP версии 4**, мы привыкли к цифровым адресам в виде четырёх чисел, разделённых точками, и доменным адресам, представляющим слова, разделённые точками. Например:

- **192.168.0.20** — цифровой адрес компьютера в локальной сети класса C;
- **asu.tusur.ru** — доменное имя web-сервера кафедры АСУ.

В языке Java в качестве адресов используются объекты класса **InetAddress**, а привычные нам адреса Internet — в качестве строковых аргументов в трёх *статических* методах этого класса:

getLocalHost()	Создаёт объект адреса для локального компьютера.
getByName(String host)	Создаёт объект адреса для host : строкового значения цифрового или доменного адреса Internet. Если вместо host указать null , то объект соответствует цифровому имени: 127.0.0.1
getAllByName(String host)	Создаёт массив объектов адресов для hosts : строкового значения цифрового или доменного адреса Internet. Если вместо host указать null , то объект соответствует цифровому имени: 127.0.0.1

Созданные объекты класса **InetAddress** используются в методах других классов, например в классах сокетов, но также имеют два собственных метода:

getHostAddress()	Возвращает адрес для объекта InetAddress .
getHostName()	Возвращает имя для объекта InetAddress .

Для примера рассмотрим программу листинга 2.11, в которой создаются объекты класса **InetAddress** для локального компьютера, доменного имени кафедр-

ры АСУ и доменного имени **www.yandex.ru** (см. также рисунок 2.17).

Листинг 2.11 — Исходный текст класса *Example9* из среды *Eclipse EE*

```
package ru.tusur.asu;

import java.net.InetAddress;
import java.net.UnknownHostException;

public class Example9 {

    public static void main(String[] args) {

        System.out.println("Использование класса InetAddress.\n"
            + "-----");

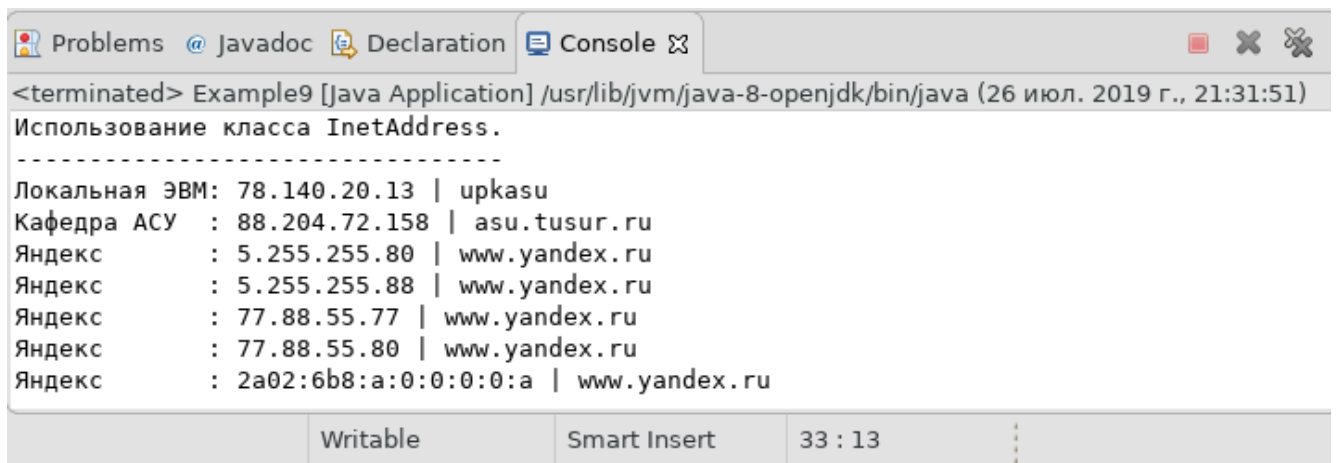
        try {
            // Создаём объекты адресов
            InetAddress addr1 =
                InetAddress.getLocalHost();
            InetAddress addr2 =
                InetAddress.getByName("asu.tusur.ru");
            InetAddress[] addr3 =
                InetAddress.getAllByName("www.yandex.ru");

            // Выводим на печать содержимое объектов
            System.out.println("Локальная ЭВМ: "
                + addr1.getHostAddress() + " | " + addr1.getHostName());

            System.out.println("Кафедра АСУ : "
                + addr2.getHostAddress() + " | " + addr2.getHostName());

            for(int i=0; i < addr3.length; i++)
                System.out.println("Яндекс      : "
                    + addr3[i].getHostAddress() + " | "
                    + addr3[i].getHostName());

        } catch (UnknownHostException e) {
            System.out.println(e.getMessage());
        }
    }
}
```



The screenshot shows the Eclipse IDE's console window. The title bar includes tabs for Problems, Javadoc, Declaration, and Console. The console output is as follows:

```
<terminated> Example9 [Java Application] /usr/lib/jvm/java-8-openjdk/bin/java (26 июл. 2019 г., 21:31:51)
Использование класса InetAddress.
-----
Локальная ЭВМ: 78.140.20.13 | upkasu
Кафедра АСУ : 88.204.72.158 | asu.tusur.ru
Яндекс      : 5.255.255.80 | www.yandex.ru
Яндекс      : 5.255.255.88 | www.yandex.ru
Яндекс      : 77.88.55.77 | www.yandex.ru
Яндекс      : 77.88.55.80 | www.yandex.ru
Яндекс      : 2a02:6b8:a:0:0:0:0:a | www.yandex.ru
```

At the bottom of the console window, there is a status bar with the text "Writable", "Smart Insert", and "33 : 13".

Рисунок 2.17 — Результат работы программы листинга 2.11

2.5.2 Адресация на базе URL и URLConnection

Для работы с web-протоколами такими как, **http**, **ftp** и другими, в пакете **java.net** имеются классы:

- **URL** — класс для непосредственной работы с URL-адресами;
- **URLConnection** — класс, методы которого обеспечивают процессы загрузки ресурсов Internet и их информационную поддержку с использованием объектов класса **URL**.

Поддерживается следующая схема URL-адресов **protocol://host:port/file**, которая обеспечивается тремя конструкторами класса **URL**:

```
URL(String saddr);
URL(String protocol, String host, String file);
URL(String protocol, String host, int port, String file);
```

Объекты класса **URL** имеют ряд важных методов:

- **getProtocol()**, **getHost()**, **getPort()**, **getFile()** - возвращают значения компонент соответствующего URL-адреса;
- **openConnection()** - возвращает объект класса **URLConnection**;
- **openStream()** - возвращает объект входного потока класса **InputStream**.

На листинге 2.12 представлен исходный текст программы, использующей URL-адрес «<http://asu.tusur.ru/>» для чтения доступного по нему ресурса и записи его в файл **/home/vgr/demo10.html**.

Листинг 2.12 — Исходный текст класса *Example10* из среды Eclipse EE

```
package ru.tusur.asu;

import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.MalformedURLException;
import java.net.URL;

public class Example10 {

    public static void main(String[] args) {

        // Объявление буфера и счётчика читаемых байт
        byte[] b;
        int n;

        System.out.println("Использование класса URL\n"
            + "Сохраняем в файле: /home/vgr/demo10.html\n"
            + "-----");

        try
        {
```

```

URL url = // Объявление и создание объекта
          new URL("http://asu.tusur.ru/");

// Печатаем компоненты URL-адреса
System.out.println("Протокол: " + url.getProtocol() + "\n"
    + "Хост      : " + url.getHost() + "\n"
    + "Порт      : " + url.getPort() + "\n"
    + "Файл      : " + url.getFile() + "\n");

InputStream in = // Открываем входной поток
                url.openStream();
OutputStream out = // Открываем выходной поток
                  new FileOutputStream("/home/vgr/demo10.html");

while ((n = in.available()) > 0) // Читаем и пишем в цикле
{
    b = new byte[n];
    in.read(b);
    //System.out.print(new String(b));
    out.write(b);
}

in.close(); // Закрываем потоки
out.close();

}
catch(MalformedURLException e1)
{
    System.out.println(e1.getMessage());
}
catch(IOException e2)
{
    System.out.println(e2.getMessage());
}
}
}

```

Полученный программой файл можно посмотреть в любом браузере, но следует помнить, что объекты класса **URL** предназначены для работы с адресами и не содержат средств анализа адресуемого ресурса.

Для манипулирования адресуемыми ресурсами используется объект класса **URLConnection**, который можно получить командой:

```
URLConnection ucon = url.openConnection();
```

где *url* — объект класса **URL**, имеющий множество методов. Например:

- **connect()** - устанавливает соединение с ресурсом, если оно ещё не было установлено, или вызывает исключение, если соединение — невозможно;
- **getContentLength()** - возвращает размер ресурса;
- **getContentType()** - возвращает тип содержимого адресуемого ресурса;
- **getDate()** - возвращает дату загрузки;
- **getExpiration()** - возвращает срок хранения;
- **getPermission()** - определяет параметры доступа к ресурсу;
- **getInputStream()** - возвращает объект потока ввода класса **InputStream**.

2.5.3 Сокеты протокола TCP

Для организации взаимодействия между двумя программами по протоколу TCP пакет **java.net** предоставляет два класса:

- **Socket** — класс для создания объектов клиентского сокета;
- **ServerSocket** — класс создающий объекты, используемые серверными программами для организации соединения с программы клиентов.

Чтобы программа-клиент могла соединиться с сервером, она должна знать адрес и порт сервера, а затем создать объект клиентского сокета командой:

```
Socket client =  
    new Socket(InetAddress address, int port);
```

Чтобы программа-сервер могла принимать соединения от клиентов, она должна создать объект типа **ServerSocket** командой:

```
ServerSocket server =  
    new ServerSocket(int port);
```

Получив запрос на соединение, программа-сервер должна согласиться на него и создать объект клиентского сокета командой:

```
Socket client =  
    server.accept();
```

Для возможности непосредственного обмена данными обе программы, и клиент и сервер, должны создать входные и выходные потоки, используя классы пакета **java.io**: и методы объектов класса **Socket**:

```
InputStream in =  
    client.getInputStream();  
  
OutputStream out =  
    client.getOutputStream();
```

Далее, обмен данными между программами выполняется через объекты потоков ввода-вывода с помощью соответствующих методов **read(...)** и **write(...)**.

Описанная выше технология обмена данными между двумя программами называется — **технология клиент-сервер**, а метод взаимодействия — **синхронным**. Алгоритм, по которому осуществляется обмен данными, - **протоколом**. Соответствующий пример программы приводится в пункте 2.5.5.

2.5.4 Сокеты протокола UDP

Для организации асинхронного взаимодействия между программами пакет *java.net* предоставляет классы *DatagramSocket* и *DatagramPacket*, реализованные на транспортном уровне протокола UDP.

Сокет протокола UDP создаётся объектом класса *DatagramSocket*, который имеет три конструктора:

```
DatagramSocket();
DatagramSocket(int port);
DatagramSocket(int port, InetAddress addr);
```

Каждый объект класса *DatagramSocket* имеет следующие основные методы:

- ***getInetAddress()*** - возвращает адрес, к которому осуществляется подключение;
- ***getPort()*** - возвращает порт, к которому осуществляется подключение;
- ***getLocalAddress()*** - возвращает локальный адрес компьютера, с которого осуществляется подключение;
- ***getLocalPort()*** - возвращает локальный порт, через который осуществляется подключение;
- ***send(DatagramPacket pack)*** — передача пакета;
- ***receive(DatagramPacket pack)*** - приём пакета.

Пакеты протокола UDP создаются объектами класса *DatagramPacket*, который имеет два конструктора:

- ***DatagramPacket(byte[] buf, int length)*** — создать пакет данных размера *length* на основе массива байт - *buf*;
- ***DatagramPacket(byte[] buf, int length, InetAddress addr, int port)*** - создать полный пакет данных размера *length* на основе массива байт — *buf*, предназначенный для передачи по адресу *addr* и порту *port*.

Каждый объект класса *DatagramPacket* имеет следующие основные методы:

- ***getAddress()*** и ***setAddress()*** - возвращает или устанавливает адрес подключения;
- ***getPort()*** и ***setPort()*** - возвращает или устанавливает порт подключения;
- ***getLength()*** и ***setLength()*** - возвращает или устанавливает размер дейтаграммы;
- ***getData()*** - возвращает массив байт содержимого дейтаграммы;
- ***getData(byte[] buf)*** — позволяет записать данные в пакет.

Естественно, что двусторонний обмен между программами с помощью дейтаграмм требует организовать с каждой стороны как коды клиента, так и коды сервера. Обычно, это требует мультинитевой организации взаимодействующих программ. Примеры таких программ — это тема лабораторных работ.

2.5.5 Простейшая задача технологии клиент-сервер

Решение задач технологии *клиент-сервер* требует от программиста трёх навыков:

- определение общего состава прикладных функций решаемой задачи;
- распределение прикладных функций между программой клиента и программой сервера;
- описание протокола обмена данными между программой клиента и программой сервера;

Поскольку познавательная цель данного пункта - демонстрация использования классов и методов пакета **java.net** языка Java, то минимизируем прикладную часть задачи до простейшего уровня:

- программа-**клиент** передаёт на **сервер** конечную последовательность текстовых сообщений, ожидая последовательное подтверждение на приём каждого из них;
- программа-**сервер** принимает отдельные сообщения и подтверждает каждое из них передачей **клиенту** собственного текстового сообщения.

Решение задачи проведём на базе протокола TCP, который предполагает организацию соединения между программами до передачи и приема сообщениями, а также разрыв соединения после обмена данными. Для конкретизации технологии *клиент-сервер* предполагаем, что:

- программа-**сервер** запускается и ждёт запрос на соединение от программы-**клиента**; после установки соединения, она производит обмен сообщениями, а после разрыва соединения по инициативе **клиента** завершает свою работу;
- программа-**клиент** иницирует соединение с сервером, производит обмен сообщениями, разрывает соединение и завершает свою работу.

Реализация протокола поставленной задачи требует определения *средств синхронизации* процессов обмена данными между взаимодействующими программами. Такие средства являются обязательными для задач, реализуемых на базе протоколов транспортного уровня. Они включают:

- определение аварийных ситуаций, фиксируемых программными средствами языка Java;
- определение состояний блокировки обмена данными между взаимодействующими программами;
- организацию перехода программ в фиксированные исходные состояния для продолжения процессов взаимодействия (сам процесс синхронизации).

В рамках нашей задачи, средства синхронизации фиксируются следующим набором правил:

- любая аварийная ситуация, фиксируемая программными средствами языка Java, приводит к завершению программ;
- монитором и инициатором синхронизации является программа-**клиент**;
- событием конца отдельного сообщения является приём любой стороной диалога символа перевода строки «\n»;
- состояние блокировки обмена данными определяет программа-**клиент** посредством нарушения границ фиксированного тайм-аута при ожидании подтверждения на приём сообщения от программы-**сервера**;
- программа-**клиент** иницирует синхронизацию посылкой серверу символа возврата каретки «\r»;
- программа-**сервер** должна вернуть символ «\r» в пределах границы тайм-аута, иначе программа-**клиент** разрывает соединение и завершает работу.

В пределах изложенной постановки задачи, программа-**сервер** реализована в виде класса **TCPServer** и представлена на листинге 2.13.

Листинг 2.13 — Исходный текст класса TCPServer из среды Eclipse EE

```
package asu.server;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.InetAddress;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.UnknownHostException;
import java.util.Date;

/*
 * Сервер стартует и прослушивает порт.
 * Сервер акцептирует запрос на соединение, создаёт потоки ввода/вывода
 * и в цикле выполняет:
 * 1) читает входной поток посимвольно и выводит их на экран;
 * 2) получив символ "\n" == 10, посылает клиенту строку "OK\n";
 * 3) получив символ "\r" == 13, посылает его назад клиенту;
 * 4) завершает работу по любому исключению или закрытию входного потока.
 */

public class TCPServer {

    public static void main(String[] args)
    {
        int port; //Номер порта сервера
        String smes= "OK\n"; //Сообщение посылаемое клиенту

        ServerSocket svrsocket; //Серверный сокет
        Socket clientsocket; //Клиентский сокет

        //Время получения первого пакета
        long statime = new Date().getTime();

        //Текущее время относительно акцепта соединения
        long curtime = 0;

        InputStream in; //Входной поток сервера
```

```

OutputStream out;           //Выходной поток сервера

int ch = (int)'\n';          //Символ конца строки
System.out.println("\n\n = " + ch);
int cr = (int)'\r';          //Символ синхронизации
System.out.println("\r = " + cr + "\n");

try{
    port = // Читаем номер порта как аргумент программы
           new Integer(args[0]).intValue();

    InetAddress localhost = //Адресс сервера
        InetAddress.getLocalHost();
    System.out.println("Server Address: "
        + localhost.getHostAddress());
    System.out.println("Port Address: "
        + port);           //args[0]);

    //Объявляем порт прослушивания
    svrsocket = new ServerSocket(port);
    System.out.println("TCPserver start: " + new Date());

    //Прослушиваем порт
    clientsocket = svrsocket.accept(); //Ожидаем соединения
    statime = new Date().getTime();
    System.out.print(curtime +
        ": TCP-server - accept conection...\n\n" + "0: ");

    //Открываем потоки ввода/вывода
    in = clientsocket.getInputStream();
    out = clientsocket.getOutputStream();

    //Цикл диалога с клиентом
    // Читаем по байту, пока не закроется входной поток сокета
    while((ch=in.read()) != -1){

        System.out.print((char)ch);
        //Перевод строки - значит строка закончилась
        if(ch == 10){
            curtime = new Date().getTime();
            System.out.print((curtime - statime) + ": ");

            //Отвечаем клиенту "OK\n"
            out.write(smes.getBytes());
        }
        //Возврат каретки - значит синхронизация с клиентом
        if(ch == 13) out.write("\r".getBytes());
    }

    out.flush();           // Освобождаем поток вывода
    out.close();           // Закрываем потоки
    in.close();            //
    clientsocket.close();  // Закрываем сокеты
    svrsocket.close();     //

}catch(UnknownHostException ue){
    System.out.println("UnknownHostException: " + ue.getMessage());
}catch(IOException e){
    System.out.println("IOException: " + e.getMessage());
}catch(Exception ee){
    System.out.println("Exception: " + ee.getMessage());
}

```

```

        // Подсказка для запуска программы
        System.out.println("\nrun: java asu.server.TCPServer port\n");
    }

    curtime = new Date().getTime(); // Подводим итог
    System.out.println("\n" + (curtime - statime)
        + ": TCP-server - stop");

    System.exit(0); // Системное завершение программы
}
}

```

Для нормального запуска программы-**сервера**, необходимо указать в качестве ее аргумента номер используемого порта, например, - число 8888. А поскольку мы реализуем программы в среде Eclipse EE, то необходимо зайти в каталог **bin** ее проекта и выполнить команду:

```
$ java asu.server.TCPServer 8888
```

Результат такого запуска сервера показан на рисунке 2.18.

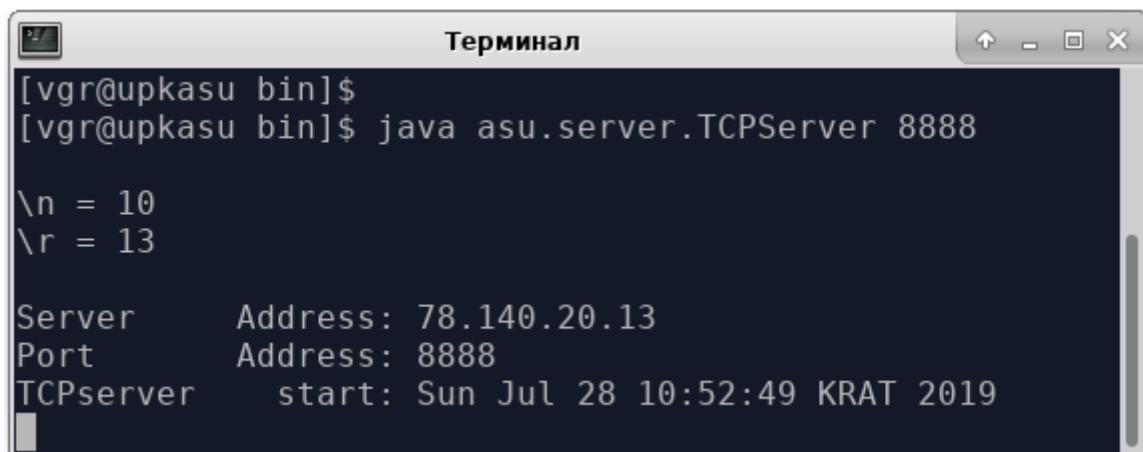


Рисунок 2.18 — Результат работы программы-сервера

Соответствующая программа-**клиент** реализована в виде класса **TCPClient** и представлена на листинге 2.14.

Листинг 2.14 — Исходный текст класса TCPClient из среды Eclipse EE

```

package asu.client;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.InetAddress;
import java.net.Socket;
import java.net.UnknownHostException;
import java.util.Date;

/*

```



```

//Устанавливаем соединение с сервером
clientsocket =
    new Socket(remotehost, port);
System.out.println("TCPclient connect: " + new Date());
long statime = new Date().getTime();

//Создание потоков ввода/вывода
in = clientsocket.getInputStream();
out = clientsocket.getOutputStream();

//Цикл диалога с сервером
for(int i=1; i<=nn; i++){
    //Посылаем серверу сообщение
    mes = String.valueOf(i) + " " + args[4] + "\n";
    out.write(mes.getBytes());

    //Фиксируем границу тайм-аута
    rTime = new Date().getTime() + dTime;
    curtime = new Date().getTime();
    System.out.println((curtime - statime) + ": " + mes);

    //ожидаем ответ сервера
    boolean flag = true;
    while(flag){
        if(in.available() > 0){

            //Блокирующая операция чтения одного байта
            ch = in.read();
            System.out.print((char)ch);
            if(ch == 10){ //Если пришло подтверждение
                curtime = new Date().getTime();
                System.out.print((curtime - statime) + ": ");
                flag = false;
                flagRepeat = false;
            }
            if(ch == 13){ //Если пришла синхронизация
                //Заново отправляем пакет
                out.write(mes.getBytes());
                rTime = new Date().getTime() + dTime;
            }
        }else{ //Проверка тайм-аута
            System.out.print(".");
            if(new Date().getTime() > rTime){
                if(flagRepeat) flag = false;
                else{ //Инициация синхронизации
                    out.write("\r".getBytes());
                    rTime = new Date().getTime() + dTime;
                }
            }
        }
    }
    if(flagRepeat) break;
}
out.flush();
out.close();
in.close();
clientsocket.close();
}catch(UnknownHostException ue)
{
    System.out.println("\nUnknownHostException: "

```



```

        + ue.getMessage());
    }catch(IOException e)
    {
        System.out.println("\nIOException: " + e.getMessage());
    }catch(Exception ee)
    {
        System.out.println("\nException: " + ee.getMessage());

        //Посказка для запуска программы
        System.out.println("\nrun: java asu.client.TCPClients address "
            + "port time_out count_message message\n");
    }

    System.out.println("\nTCPclient stop: " + new Date());
    System.exit(0);
}
}

```

Общий формат команды запуска программы-**клиента** из командной строки имеет вид:

```
$ java asu.client.TCPClient address port time_out count_message message
```

При исполнении команды запуска клиента следует учитывать адрес и номер порта запущенного сервера. При учете данных, отображённых на рисунке 2.18, аргументы могут быть следующие:

- **address** = 78.140.20.13
- **port** = 8888
- **time_out** = 1000
- **count_message** = 100
- **message** = "Text message from client program"

При таких аргументах, команда запуска программы-**клиента** будет иметь вид:

```
$ java asu.client.TCPClient 78.140.20.13 8888 1000 100 \
    "Text message from client program"
```

Обе программы должны быть запущены в разных терминалах, а результат должен быть следующий:

- программа-**клиента** — последовательно отправит серверу 100 одинаковых сообщений и распечатает их на своём терминале;
- программа-**сервера** примет эти сообщения и распечатает их на своём терминале;
- по завершении работы, программа-**клиента** выведет на терминал общее время передачи всех сообщений.

2.6 Организация доступа к базам данных

Классическим примером распределенных систем является совокупность приложений, обрабатывающих данные в некоторой вычислительной сети. Принципиальным здесь является тот факт, что система разделена как минимум на две части, которые как программное обеспечение функционируют автономно, но связаны некоторым набором общих данных, которые они вместе обрабатывают. Сама обработка данных осуществляется по технологии «*клиент-сервер*», где:

- программа-**клиент** — приложение, выполняющее запросы к серверу;
- программа-**сервер** — СУБД, обслуживающее запросы клиента.

Познавательная цель данного подраздела — описание классов и методов языка Java, сосредоточенных в пакете ***java.sql*** и организующих общий доступ приложений к СУБД. Технология организации такого доступа основана на выполнении следующих четырёх этапов:

1. Загрузка драйвера необходимой СУБД.
2. Установка соединения между Java-программой и СУБД.
3. Передача в базу данных команд языка SQL.
4. Получение и обработка результатов таких команд (SQL-запросов).

Организация учебного материала проведена с учётом того, что студент уже освоил теорию и практику работы с реляционными СУБД в дисциплине «*Базы данных*» и способен формулировать правильные запросы на языке SQL. Это даёт основание сосредоточить основное внимание на инструментальных средствах пакета ***java.sql*** и детализировать изложение материала на примере конкретной СУБД — ***Apache Derby*** [31]. Такой подход закладывает познавательный фундамент для изучения материала последующих глав. Само изложение текущего учебного материала разделено на три пункта:

- описание инструментальных средств СУБД Apache Derby;
- описание классов и методов для формирования SQL-запросов к СУБД и подключения необходимых драйверов, обеспечивающих подключение Java-программ к конкретным СУБД;
- демонстрация Java-программы, реализующей простейший типовой пример работы с СУБД.

2.6.1 Инструментальные средства СУБД Apache Derby

Проект ***Apache Derby***, появившийся в 1997 году, является примером СУБД полностью написанным на языке Java. Как отмечает Википедия [32]: «***Apache Derby*** — реляционная СУБД, написанная на Java, предназначенная для встраивания в Java-приложения или обработки транзакций в реальном времени. Занимает 2

МБ на диске. Распространяется на условиях лицензии Apache 2.0. Ранее известна как **IBM Cloudscape**. Oracle распространяет те же бинарные файлы под именем **Java DB**», но в действительности, эта СУБД может работать как нормальный сетевой сервер, а размер ее зависит как от версии разработки, так и варианта установленного дистрибутива. Например, минимальный вариант, используемый в данной дисциплине, имеет размер 5 МБ. Официальный сайт проекта [33] предоставляет множество версий СУБД, которые обновляются регулярно раз в год и позволяют выбрать нужную конфигурацию, рабочая часть которой пригодна для установки как в среде Linux, так и в среде MS Windows. Сама процедура установки — достаточно проста и предполагает:

- выбор корневого каталога для установки СУБД Derby и копирования в него содержимого архива дистрибутива;
- настройку переменных среды ОС, учитывающих местоположение как самой СУБД, так и среды исполнения Java (JRE).

Для студентов, изучающих данную дисциплину, дистрибутив СУБД Derby установлен в архиве его личной рабочей области, которая подключается к среде учебного программного комплекса (УПК) [6] во время проведения им лабораторных работ. В проекции такого варианта и продолжим дальнейшее изложение учебного материала.

СУБД Derby инсталлировано в среду УПК следующим образом:

- ***\$HOME/derby*** — корневой каталог установленной СУБД;
- ***\$HOME/derby/bin*** — каталог размещения баз данных, утилит и сценариев запуска различных компонент СУБД;
- ***\$HOME/derby/lib*** — каталог размещения библиотек СУБД.

Для правильной настройки среды ОС необходимо правильно настроить четыре переменных среды в файле ***\$HOME/.bashrc***, как показано на рисунке 2.19.

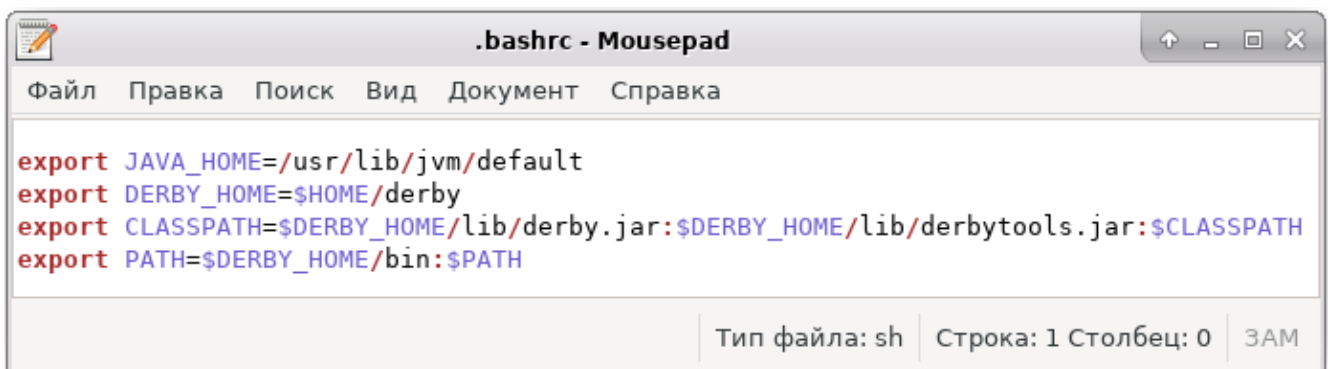
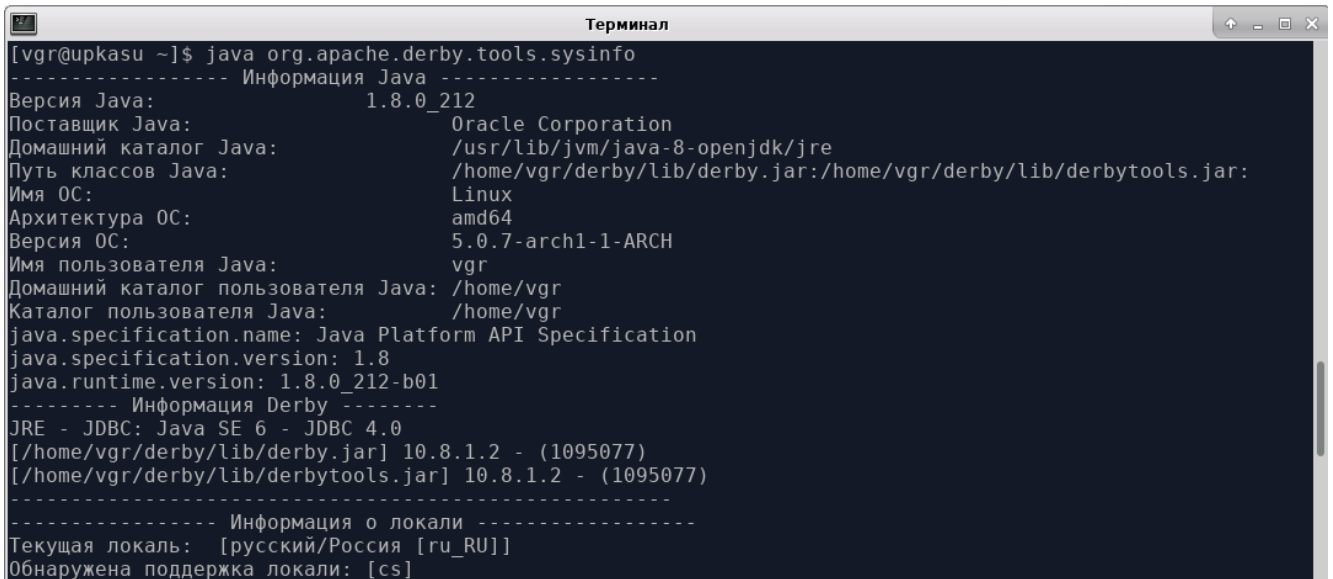


Рисунок 2.19 — Правильная настройка переменных среды ОС

Для проверки произведённых настроек, необходимо запустить новый терминал и выполнить команду:

```
$ java org.apache.derby.tools.sysinfo
```

Если настройки выполнены правильно, то на терминал будут выведены сообщения об используемых версиях дистрибутивов Java и Derby, показанные на рисунке 2.20.



```

Терминал
[vgr@upkasu ~]$ java org.apache.derby.tools.sysinfo
----- Информация Java -----
Версия Java: 1.8.0_212
Поставщик Java: Oracle Corporation
Домашний каталог Java: /usr/lib/jvm/java-8-openjdk/jre
Путь классов Java: /home/vgr/derby/lib/derby.jar:/home/vgr/derby/lib/derbytools.jar:
Имя ОС: Linux
Архитектура ОС: amd64
Версия ОС: 5.0.7-arch1-1-ARCH
Имя пользователя Java: vgr
Домашний каталог пользователя Java: /home/vgr
Каталог пользователя Java: /home/vgr
java.specification.name: Java Platform API Specification
java.specification.version: 1.8
java.runtime.version: 1.8.0_212-b01
----- Информация Derby -----
JRE - JDBC: Java SE 6 - JDBC 4.0
[/home/vgr/derby/lib/derby.jar] 10.8.1.2 - (1095077)
[/home/vgr/derby/lib/derbytools.jar] 10.8.1.2 - (1095077)
-----
----- Информация о локали -----
Текущая локаль: [русский/Россия [ru_RU]]
Обнаружена поддержка локали: [cs]

```

Рисунок 2.20 — Сообщение ПО Derby при правильной настройке среды ОС

Если СУБД Derby используется в серверном варианте, необходимо определиться с адресом и портом, по которым СУБД будет принимать соединения. Общий формат использования сценария, запускающего сервер, имеет вид:

```
$ $DERBY_HOME/bin/startNetworkServer -h адрес -p порт &
```

По умолчанию (без аргументов), указанный сценарий будет запускать сервер по адресу **localhost** и порту **1527**. Чтобы служба безопасности разрешила такой запуск для нашего дистрибутива Java, необходимо добавить следующий оператор в файл **/etc/java-8-openjdk/security/java.policy**:

```
grant {
    permission java.net.SocketPermission "localhost:1527", "listen";
};
```

После внесенных изменений, команда:

```
$ $DERBY_HOME/bin/startNetworkServer &
```

запустит сервер, как это показано на рисунке 2.21, а запуск сценария:

```
$ $DERBY_HOME/bin/stopNetworkServer
```

остановит сервер, показано на рисунке 2.22.

```

Терминал
[vgr@upkasu bin]$ ./startNetworkServer &
[1] 4076
[vgr@upkasu bin]$ Tue Jul 30 10:41:41 KRAT 2019 : DRDA_SecurityInsta
lled.I
Tue Jul 30 10:41:41 KRAT 2019 : Сетевой сервер Apache Derby Network
Server - 10.8.1.2 - (1095077) запущен и готов принимать соединения н
а порту 1527 на {3}

```

Рисунок 2.21 — Локальный старт сервера Apache Derby

```

Терминал
at org.apache.derby.drda.NetworkServerControl.main(Unknown S
ource)
Tue Jul 30 10:43:55 KRAT 2019 : Сетевой сервер Apache Derby - 10.8.1
.2 - (1095077) завершение работы в {2}
Tue Jul 30 10:43:55 KRAT 2019 : Сетевой сервер Apache Derby - 10.8.1
.2 - (1095077) завершение работы в {2}
[1]+  Завершён      ./startNetworkServer
[vgr@upkasu bin]$

```

Рисунок 2.22 — Завершение работы сервера Apache Derby

Проведенных настроек уже вполне достаточно для полноценной работы с СУБД Derby, которая обеспечивается интерактивной утилитой **ij**. Она позволяет создавать и удалять базы данных, делать к ним различные SQL-запросы и выводить результаты на стандартный ввод-вывод. На рисунке 2.23 показан пример запуска **ij** в интерактивном режиме и выполнение в ней команды **help**, которая выводит список доступных инструментальных средств этой утилиты.

```

Терминал
[vgr@upkasu ~]$
[vgr@upkasu ~]$ ij
версия ij 10.8
ij> help;

Поддерживаемые команды:

PROTOCOL 'протокол JDBC' [ AS идент ];
                                -- задать протокол по умолчанию или указанный протокол
DRIVER 'класс драйвера';      -- загрузить указанный класс
CONNECT 'url базы данных' [ PROTOCOL указанный_протокол ] [ AS имя_соединения ];
                                -- соединиться с указанным URL базы данных
                                -- и также позволяет назначить идентификатор
SET CONNECTION имя_соединения; -- переключиться на указанное соединение
SHOW CONNECTIONS;              -- вызвать список всех соединений
AUTOCOMMIT [ ON | OFF ];       -- задать режим автоматического принятия для соединения

```

Рисунок 2.23 — Выполнение команды help в интерактивном режиме утилиты ij

Самой утилитой мы воспользуемся в пункте 2.6.3, а сейчас перейдём к изучению классов и методов пакета *java.sql*.

2.6.2 SQL-запросы и драйверы баз данных

Для взаимодействия любой СУБД в языке Java предусмотрены специальные средства, называемые **JDBC**. Википедия так характеризует эти средства [34]: «**JDBC** (англ. Java DataBase Connectivity — *соединение с базами данных на Java*) — платформенно независимый промышленный стандарт взаимодействия Java-приложений с различными СУБД, реализованный в виде пакета *java.sql*, входящего в состав Java SE. JDBC основан на концепции так называемых драйверов, позволяющих получать соединение с базой данных по специально описанному URL. Драйверы могут загружаться динамически (во время работы программы). Загрузившись, драйвер сам регистрирует себя и вызывается автоматически, когда программа требует URL, содержащий протокол, за который драйвер отвечает».

Непосредственно в пакете *java.sql* реализована только группа интерфейсов под общим названием **Driver**, а само ПО драйверов поставляется в дистрибутивах соответствующих СУБД или ищутся на сайтах вендоров, например, перечисленных в таблице 2.5.

Таблица 2.5 — Примеры официальных сайтов вендоров СУБД

Название СУБД	Адрес сайта
Derby DB (clients) Derby DB (embedded)	http://db.apache.org/derby/derby_downloads.html
MySQL	http://www.mysql.com/downloads/connector/j/
PostgreSQL	http://jdbc.postgresql.org/download.html
Microsoft SQLServer	http://go.microsoft.com/fwlink/?LinkId=245496
Oracle Database	http://www.oracle.com/technetwork/database/enterprise-edition/jdbc-112010-090769.html
HSQLDB	http://sourceforge.net/projects/hsqldb/files/hsqldb/
SQLite	http://www.xerial.org/trac/Xerial/wiki/SQLiteJDBC
Sybase Adaptive Server Enterprise	http://sourceforge.net/projects/jtds/files/
Sybase Adaptive Server IQ	http://www.sybase.com/products/allproductsa-z/softwaredeveloperkit/jconnect
JDBC/ODBC	Интегрирован в JDK

Что касается СУБД Derby, то нужные драйвера находятся в библиотеках:

- **derby.jar** — содержит драйвер для встроенного подключения к СУБД;
- **derbyclient.jar** - содержит драйвер для удалённого (сетевого) подключения к

серверу СУБД.

Список имён классов, реализующих интерфейс **Driver**, для разных СУБД представлен в таблице 2.6.

Таблица 2.6 - Имена классов драйверов для некоторых СУБД

Название СУБД	Класс драйвера
Derby DB (for remote clients)	<i>org.apache.derby.jdbc.ClientDriver</i>
Derby DB (embedded)	<i>org.apache.derby.jdbc.EmbeddedDriver</i>
MySQL	<i>com.mysql.jdbc.Driver</i>
PostgreSQL	<i>org.postgresql.Driver</i>
Microsoft SQL Server	<i>com.microsoft.jdbc.sqlserver.SQLServerDriver</i>
Oracle Database	<i>oracle.jdbc.driver.OracleDriver</i>
HSQLDB	<i>org.hsqldb.jdbc.JDBCDriver</i>
SQLite	<i>org.sqlite.JDBC</i>
Sybase Adaptive Server Enterprise	<i>net.sourceforge.jtds.jdbc.Driver</i>
Sybase Adaptive Server IQ	<i>com.sybase.jdbc3.jdbc.SybDriver</i>
JDBC/ODBC	<i>sun.jdbc.odbc.JdbcOdbcDriver</i>

Непосредственное подключение драйвера к коду Java-программы выполняется в формате шаблона:

```
try {
    Class.forName("your_driver_class_name");
} catch (ClassNotFoundException ex) {
    System.out.println(ex.getMessage());
}
```

Для манипулирования драйверами и подключения к СУБД, пакет **java.sql** содержит класс **DriverManager** со статическими методами:

registerDriver(Driver driver)	Регистрирует драйвер.
deregisterDriver(Driver driver)	Удаляет драйвер.
getDrivers(void)	Возвращает перечисление Enumeration<Driver>.
getConnection(String jurl)	Возвращает объект класса Connection.
getConnection(String jurl, String user, String password)	Возвращает объект класса Connection.

где **jurl** — строка, определяющая абсолютный адрес для JDBC, построенный по следующему общему принципу:

jdbc:название_набора_драйверов:имя_и_путь_к_базе_данных

Для каждой СУБД, строка **jurl** имеет свой уникальный формат, определённый вендором драйвера. В частности, она может включать аргументы **user** и **password**, которые соответствуют имени пользователя и его пароля, объявленные при создании базы данных. Формат этой строки для СУБД Derby будет рассмотрен в следующем пункте.

Остальные классы и методы пакета **java.sql** не зависят от используемого драйвера или адреса JDBC. Так, объекты класса **Connection** имеют два основных метода связанных с формированием SQL-запросов:

- **createStatement()** - возвращает объект класса **Statement**, методы которого отправляют полностью сформированный SQL-запрос;
- **prepareStatement(String sql)** — создаёт объект класса **PreparedStatement**, содержащий не полностью сформированный SQL-запрос.

Объекты класса **Statement** предлагают два метода:

- **executeQuery(String sql)** — отправляет СУБД запрос, полностью сформированной, на основе оператора **SELECT**, строкой **sql**; возвращает объект класса **ResultSet**;
- **executeUpdate(String sql)** — отправляет СУБД запрос, полностью сформированной, на основе оператора **DELETE** или **INSERT** или **UPDATE**, строкой **sql**; возвращает целое число изменённых строк.

Объекты класса **PreparedStatement** сформированы на не полностью определённой строке SQL-запроса, где неизвестные параметры помечаются символом «?». Их методы разделены на две группы:

- первая группа — методы **setInt(int n, int i)**, **setString(int n, String s)** и другие, где первый аргумент задаёт номер изменяемого параметра, а второй — само значение параметра, согласно его типу;
- вторая группа — методы **executeQuery()** и **executeUpdate()**, аналогичные классу **Statement**.

Объекты класса **ResultSet** содержит упакованный по строкам результат возврата SQL-запроса **SELECT**. Имеют три группы методов:

- первая группа — метод **getMetaData()** возвращает только объект класса **ResultSetMetaData**;
- вторая группа — **first()**, **last()**, **previous()** и **next()** обеспечивают перемещение по строкам результата запроса;
- третья группа — **getBytes(...)**, **getInt(...)**, **getString(...)** и другие, обеспечивают извлечение из объекта запроса типизированных данных; в качестве аргумента этих методов указывается номер столбца или его имя.

Объект класса **ResultSetMetaData** содержит метаданные результата запроса, которые извлекаются методами не требующими пояснений: **getColumnCount()**, **getColumnName()**, **getColumnType()** и **isReadOnly(int column)**.

2.6.3 Типовой пример выборки данных

Завершив в предыдущем пункте общее описание интерфейсов, классов и методов пакета **java.sql**, перейдём к рассмотрению конкретного примера, который наглядно покажет как использовать эти инструментальные средства пакета. Для этого возьмём задачу ведения личных записей в таблице с именем **notepad**, хранящейся в базе данных с именем **exampleDB**. Чтобы не путаться в альтернативных вариантах, конкретизируем условия задачи следующими ограничениями:

- будем использовать *встроенный вариант* использования базы данных, который нам потребуется и в других примерах, предполагающий размещение ее в домашней директории пользователя в каталоге: **\$HOME/databases**;
- таблица **notepad** имеет два поля, первое из которых является целочисленным уникальным ключём с именем **notekey**, а второе — поле переменной длины с именем **text**, содержащее отдельную запись пользователя;
- доступ прикладных программ к базе данных **exampleDB** осуществляется от имени пользователя **upk** с паролем **upkasu**.

Приступая к реализации приложений работы с базой данных **notepad**, конкретизируем содержание строки **jurl**, определяющей абсолютный адрес JDBC для доступа к ней. В условиях поставленной задачи, полная строка для *встроенного варианта* имеет вид:

```
jdbc:derby:/home/vgr/databases/exampleDB;create=true;user=upk;password=upkasu;
```

где опции **create**, **user** и **password** могут отсутствовать в зависимости от вариантов их использования. Например, опция **create=true** означает, что база данных будет создана, если она — отсутствует.

Соответственно, *альтернативный вариант* для серверного использования базы данных, когда она будет создана в каталоге **\$DERBY_HOME/derby/bin**, имеет вид:

```
jdbc:derby://localhost:1527/exampleDB;create=true;user=upk;password=upkasu;
```

Разобравшись с **jurl** JDBC, создадим базу данных **exampleDB**, которая обычно должна существовать перед запуском и отладкой прикладных программ, работающих с ней. А для этого, воспользуемся служебным инструментом СУБД представленным в виде утилиты **ij**.

Общая технология разработки баз данных предполагает создание порождающего сценария, реализующего некоторый первоначальный ее вариант. Для нашего примера задачи такой сценарий представлен на рисунке 2.24, отображающего содержимое файла **createDB.sql**, размещённого в каталоге **\$HOME/src/rvs/** и запускаемого из него командой:

```
$ ij createDB.sql
```



```

-----
-- Сценарий создания базы данных: exampleDB.
-- Создается таблица notepad – личные записи студента.
-----
-- Вариант сервера:
-- CONNECT 'jdbc:derby://localhost:1527/exampleDB;create=true;'
--   USER 'upk' PASSWORD 'upkasu';
-----
-- Вариант встроенной БД:
CONNECT 'jdbc:derby:/home/vgr/databases/exampleDB;create=true;'
  USER 'upk' PASSWORD 'upkasu';
-----
-- Сначала удаляем все таблицы, если они были созданы.
drop table notepad;
-----
-- Создаем таблицу notepad

create table notepad (
    notekey int not null,      -- ключ записи - ключ таблицы
    text varchar(50),         -- текст записи
    primary key (notekey)
);
-----
-- Вносим в таблицу некоторое количество записей:
insert into notepad values(321, 'Запись 1');
insert into notepad values(743, 'Запись 2');
-----
select * from notepad;
-----
-- Завершение работы сценария:
commit;
disconnect;
exit;
-----

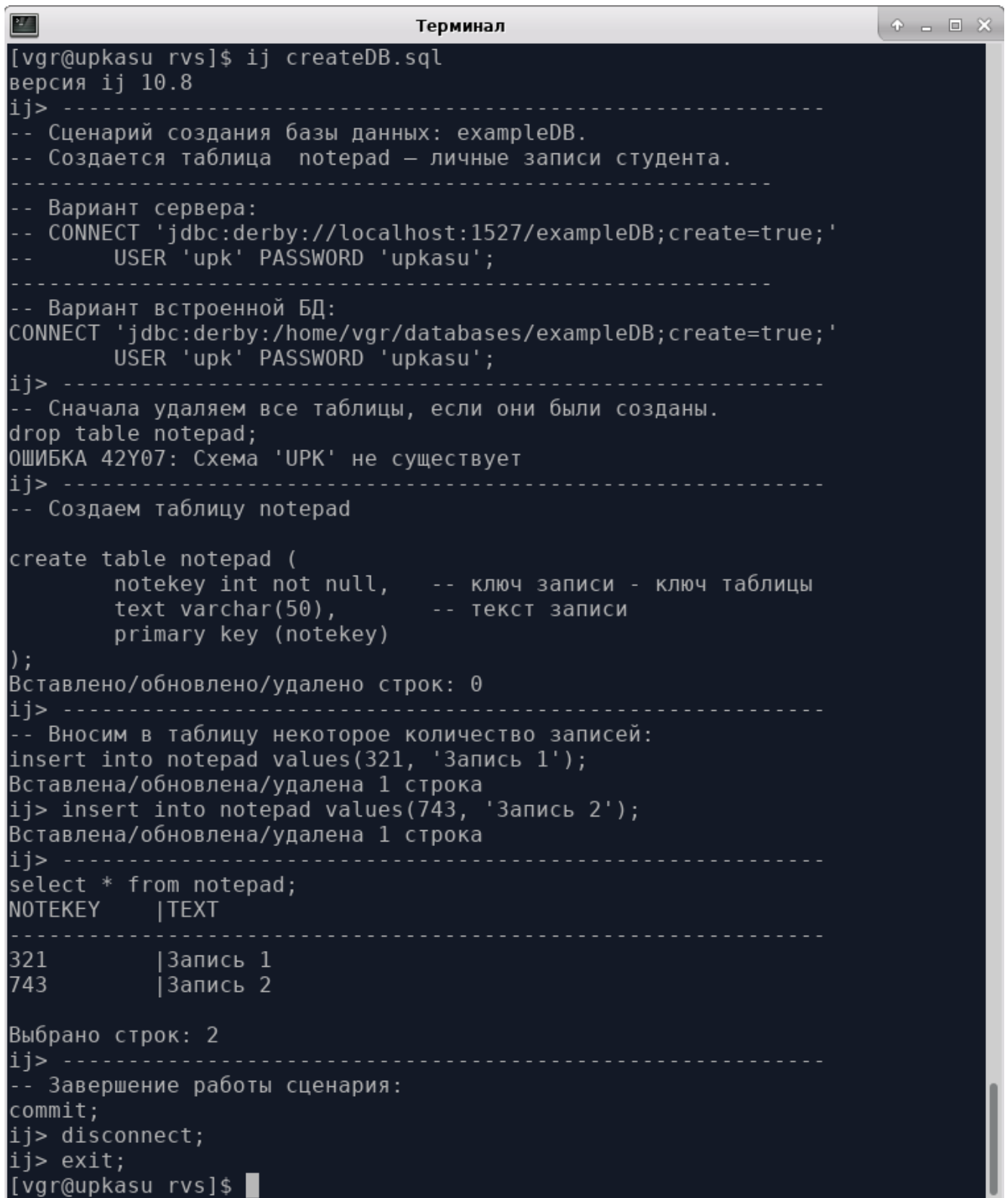
```

Тип файла: SQL | Строка: 34 Столбец: 58 | ЗАМ

Рисунок 2.24 — Сценарий создания первоначального варианта базы данных ExampleDB

Результат запуска этого сценария, создающего *встроенный вариант* базы данных, представлен на рисунке 2.25. Если запустить сетевой вариант СУБД и изменить в сценарии строку соединения, то такая же база данных будет создана в директории *\$HOME/derby/bin/* и обсуживать сетевые приложения.

Что касается приложений, работающих с базой данных *exampleDB*, то они с успехом могут быть реализованы в виде набора **SQL**-сценариев и функций любого языка *shell*, использующих нужные запуски утилиты *ij*.



```

[vg@upkasu rvs]$ ij createDB.sql
версия ij 10.8
ij> -----
-- Сценарий создания базы данных: exampleDB.
-- Создается таблица notepad — личные записи студента.
-----
-- Вариант сервера:
-- CONNECT 'jdbc:derby://localhost:1527/exampleDB;create=true;'
-- USER 'upk' PASSWORD 'upkasu';
-----
-- Вариант встроенной БД:
CONNECT 'jdbc:derby:/home/vgr/databases/exampleDB;create=true;'
USER 'upk' PASSWORD 'upkasu';
ij> -----
-- Сначала удаляем все таблицы, если они были созданы.
drop table notepad;
ОШИБКА 42Y07: Схема 'UPK' не существует
ij> -----
-- Создаем таблицу notepad

create table notepad (
    notekey int not null,    -- ключ записи - ключ таблицы
    text varchar(50),        -- текст записи
    primary key (notekey)
);
Вставлено/обновлено/удалено строк: 0
ij> -----
-- Вносим в таблицу некоторое количество записей:
insert into notepad values(321, 'Запись 1');
Вставлена/обновлена/удалена 1 строка
ij> insert into notepad values(743, 'Запись 2');
Вставлена/обновлена/удалена 1 строка
ij> -----
select * from notepad;
NOTEKEY    |TEXT
-----
321         |Запись 1
743         |Запись 2

Выбрано строк: 2
ij> -----
-- Завершение работы сценария:
commit;
ij> disconnect;
ij> exit;
[vg@upkasu rvs]$

```

Рисунок 2.25 — Результат запуска сценария createDB.sql

Для начального изучения практики применения классов и методов пакета *java.sql*, рассмотрим пример интерактивной программы, которая устанавливает соединение со встроенным вариантом БД *exampleDB* и в цикле выполняет следующие действия;

- читает все записи таблицы *notepad* и печатает их на стандартный вывод;

- последовательно запрашивает у пользователя содержимое полей **notekey** и **text**, формируя к базе данных SQL-запрос на вставку новой записи;
- в случае любого исключения или, если пользователь ввёл в поле **notekey**, строку нулевой длины, то программа завершает свою работу.

Описанная программа реализована в виде класса **Example11**, а ее исходный текст представлен на листинге 2.15. Следует обратить внимание на стиль ее написания, предполагающий выделение отдельных специализированных методов и обработку в каждом из них возникающих исключений:

- метод конструктора **Example11(...)** обеспечивает подключение драйвера и создание соединения с БД;
- метод **getResultSet()** читает содержимое таблицы **notepad**;
- метод **setInsert()** записывает в таблицу **notepad** новую строку;
- специализированные методы **getKey()** и **getString()** ориентированы на чтение со стандартного ввода;
- метод **setClose()** закрывает объекты созданные конструктором;
- статический метод **main(...)** создаёт объект класса **Example11** и, с помощью других методов этого объекта, организует интерактивный интерфейс с пользователем.

Листинг 2.15 — Исходный текст класса **Example11** из среды **Eclipse EE**

```
package ru.tusur.asu;

import java.io.IOException;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class Example11 {

    // Объекты класса
    boolean flag = true;
    Connection conn;
    ResultSet rs;

    // Конструктор
    Example11(String dburl, String dbuser, String dbpassword)
    {

        try {
            //Подключаем необходимый драйвер
            Class.forName("org.apache.derby.jdbc.EmbeddedDriver");

            //Устанавливаем соединение с БД
            conn = DriverManager.getConnection(dburl,
                                                dbuser, dbpassword);

        } catch (ClassNotFoundException e1){
```

```

        System.out.println(e1.getMessage());
        flag = false;
    } catch (SQLException e2){
        System.out.println(e2.getMessage());
        flag = false;
    }
}

// Метод, реализующий SELECT
public int getResultSet()
{
    String sql1 = "SELECT * FROM notepad ORDER BY notekey";

    try
    {
        Statement st =
            conn.createStatement();
        rs = st.executeQuery(sql1);
        return 1;

    } catch (SQLException e2){
        System.out.println(e2.getMessage());
        return 0;
    }
}

// Метод, реализующий INSERT
public int setInsert(int key, String str)
{
    String sql2 = "INSERT INTO notepad values( ? , ? )";

    try
    {
        PreparedStatement pst =
            conn.prepareStatement(sql2);

        // Установка первого параметра
        pst.setInt(1, key);
        // Установка второго параметра
        pst.setString(2, str);

        return pst.executeUpdate();

    } catch (SQLException e2){
        System.out.println(e2.getMessage());
        return -1;
    }
}

// Метод чтения целого числа со стандартного ввода
public int getKey()
{
    int ch1 = '0';
    int ch2 = '9';
    int ch;
    String s = "";

    try
    {
        while(System.in.available() == 0) ;

        while(System.in.available() > 0)

```

```

    {
        ch = System.in.read();
        if (ch == 13 || ch < ch1 || ch > ch2)
            continue;
        if (ch == 10)
            break;

        s += (char)ch;
    };
    if (s.length() <= 0)
        return -1;
    ch = new Integer(s).intValue();
    return ch;

} catch (IOException e1){
    System.out.println(e1.getMessage());
    return -1;
}
}

// Метод чтения строки текста со стандартного ввода
public String getString()
{
    String s = "\r\n";
    String text = "";
    int n;
    char ch;
    byte b[];

    try
    {
        //Ожидаем поток ввода
        while(System.in.available() == 0) ;
        s = "";
        while((n = System.in.available()) > 0)
        {
            b = new byte[n];
            System.in.read(b);
            s += new String(b);
        };
        // Удаляем последние символы '\n' и '\r'
        n = s.length();
        while (n > 0)
        {
            ch = s.charAt(n-1);
            if (ch == '\n' || ch == '\r')
                n--;
            else
                break;
        }
        // Выделяем подстроку
        if (n > 0)
            text = s.substring(0, n);
        else
            text = "";
        return text;

    } catch (IOException e1){
        System.out.println(e1.getMessage());
        return "Ошибка...";
    }
}
}

```

```

// Закрытие соединения
public void setClose()
{
    try
    {
        rs.close();
        conn.commit();
        conn.close();

    } catch (SQLException e2){
        System.out.println(e2.getMessage());
    }
}

public static void main(String[] args)
{
    System.out.println("Программа ведения записей в БД exampleDB.\n"
        + "Используются методы класса Example11\n"
        + "-----");

    // Исходные данные
    String url = "jdbc:derby:/home/vgr/databases/exampleDB";
    String user = "upk";
    String password = "upkasu";

    // Создаем объект класса
    Example11 obj =
        new Example11(url, user, password);
    if (!obj.flag)
    {
        System.out.println("Не могу создать объект класса Example11...");
        System.exit(1);
    }

    int ns;           // Число прочитанных строк
    int nb;           // Число прочитанных байт
    String text, s;   // Строка введенного текста

    // Цикл обработки запросов
    while(obj.flag)
    {
        //Печатаем заголовок ответа
        System.out.println("      Ключ   Текст\n"
            + "-----");
        obj.getResultSet();

        if (obj.rs == null)
        {
            System.out.println("Отсутствует результат SELECT...");
            break;
        }
        ns = 0;
        try
        {
            //Выводим (построчно) результат запроса к БД
            while(obj.rs.next()){
                System.out.format("%10d", obj.rs.getInt(1));
                System.out.println("  " + obj.rs.getString(2));
                ns++;
            }
            // Выводим итог запроса

```

```

System.out.println(
    "-----\n"
    + "Прочитано " + ns + " строк\n"
    + "-----\n"
    + "Формируем новый запрос!");

System.out.print("\nВведи ключ или Enter: ");
nb = obj.getKey();

if (nb == -1)
    break; // Завершаем работу программы

System.out.print("Строка текста или Enter: ");
s = obj.getString();
text = s;

while (s.length() > 0)
{
    System.out.print("Строка текста или Enter: ");
    text += ("\n" + s);
    s = obj.getString();
}

if (text.length() <= 0)
    text = "Нет данных...";

ns = obj.setInsert(nb, text);
if (ns == -1)
    System.out.println("\nОшибка добавления строки !!!");
else
    System.out.println("\nДобалено " + ns + " строк...");

} catch (SQLException e1){
    System.out.println(e1.getMessage());
    break;
}

}

//Закрываем все объекты и разрываем соединение
//obj.setClose();
System.out.println("Программа завершила работу...");
}
}

```

Демонстрация приведённой программы — это занятие для лабораторной работы, но отметим, что ее идейная основа ложится в учебный материал последующих глав.

На этом завершается учебный материал главы 2, посвящённый инструментальным средствам языка Java.

Вопросы для самопроверки

1. Назовите две стандартные поставки инструментальных средств Java.
2. В чем состоит основное отличие языка Java от языков C/C++?
3. На какие четыре платформы подразделяются дистрибутивы языка Java?
4. Какие переменные среды ОС необходимо настроить для правильной работы инструментальных средств Java?
5. Что такое - пакетная организация ПО Java?
6. Сколько простых типов данных имеет язык Java и чем они отличаются от объектов?
7. Чем интерфейсы отличаются от классов?
8. Перечислите объекты стандартного ввода-вывода языка Java.
9. Перечислите базовые классы языка Java, обеспечивающие его ввод-вывод.
10. Какие классы сетевой адресации использует Java и какому пакету они принадлежат?
11. Для чего предназначены классы *InputStream* и *OutputStream* языка Java?
12. Что такое — сокет языка Java и какими классами они поддерживаются?
13. Что такое — синхронное и асинхронное взаимодействие и какие классы пакета *java.net* их поддерживают?
14. В чем состоит отличие между классами *Socket* и *ServerSocket* пакета *java.net*?
15. В каком пакете языка Java сосредоточены базовые средства доступа к базам данных?
16. Какая СУБД полностью написана на языке Java?
17. Какие переменные среды ОС необходимо настроить для правильной работы с СУБД Apache Derby?
18. Что такое адресация JDBC и для чего она используется?
19. Что такое драйверы баз данных и где их взять?
20. В чем состоят особенности встроенного использования СУБД Derby?

3 Тема 3. Объектные распределенные системы

В подразделе 1.2 главы 1 уже была дана общая характеристика сетевых объектных систем. Она касалась классических приложений модели OSI, распределенных вычислительных сред (*DCE*), технологии CORBA и удалённых вызовов методов (*RMI*). Все эти приложения демонстрируют различные подходы к реализации общей модели «Клиент-сервер». В данной главе эти темы рассматриваются более подробно, хотя и с разной степенью детализации. Они разделены на три части:

- подраздел 3.1 описывает брокерные архитектуры, которые составляют идейную основу объектных распределенных систем;
- подраздел 3.2 посвящён технологии CORBA, которая стандартизирует общую архитектуру брокерных систем, обеспечивая сетевые приложения абстрактным протоколом взаимодействия GIOP и теоретическим набором стандартных служб;
- подраздел 3.3 содержит информацию о технологии RMI, достаточную не только для получения теоретических представлений о «Межброкерном протоколе для Интернет» (*IIOP, Internet InterORB Protocol*), но и практической реализации приложений с использованием языка Java.

При изучении предметной области данной главы необходимо хорошо представлять проблематику классических технологий, потребовавших новых подходов к реализации модели «Клиент-сервер».

Главной причиной, потребовавшей новых идей, является интенсивное развитие сетевых технологий, которые объективно увеличили количество ЭВМ и различных вычислительных систем, участвующих в создании распределенных систем. Поскольку количество потенциальных связей квадратично зависит от числа взаимодействующих элементов, то соответственно увеличивается и нагрузка на программирование сетевых взаимодействий между ними. В таких условиях, критически важным является создание инструментальных средств, которые бы полностью или частично освободили прикладных программистов от рутинной работы, связанной с учётом сетевых адресов распределенных объектов, деталей реализации сетевых протоколов модели OSI и служб сетевой безопасности.

Другой, но не менее важной причиной является необходимость создания *отдельной сетевой объектной адресации*, которая позволяла бы именовать элементы распределенных приложений независимо от особенностей реализации самой сети. Это обеспечило бы не только развитие традиционных инструментальных средств ООП, но и позволило бы создать новые распределенные информационные среды для реализации вычислительных сетей.

Таким образом, мы естественно приходим к архитектурам, использующим некоторых посредников — **брокеров**, выполняющих вспомогательные функции для приложений более высокого уровня.

3.1 Брокерные архитектуры

Брокер — ПО-посредник, которое в сетевой модели «Клиент-сервер» принимает запросы от программы-**клиента**, предаёт их программе-**серверу**, получает от сервера ответ и передаёт его клиенту.

Хотя такое определение вызывает массу нареканий, следует учесть, что сам термин является зонтичным и заимствован из сфер финансовой, торговой или страховой деятельности. Что же касается нашей тематики, то он может пониматься как служба промежуточного уровня (***Middleware***), например, как это представлено в первой главе на рисунке 1.5, или как посредник в передаче сообщений и объектных запросов, что рассматривается в данной главе. В любом случае, контекст его применения раскрывается дополнительно и, в первую очередь, рассматривается как развитие классической модели «Клиент-сервер».

Рассмотрим модель взаимодействия программ клиента и сервера, представленную на рисунке 3.1 и реализуемую средствами сетевой архитектуры OSI.

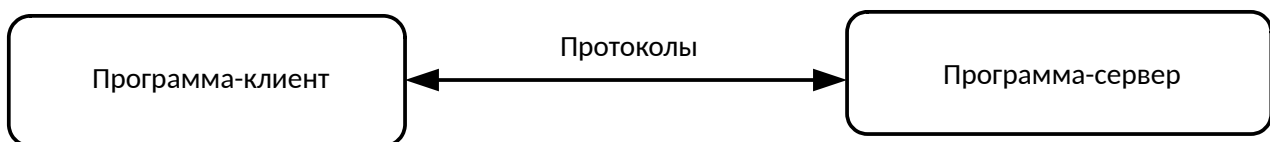


Рисунок 3.1 — Классическая модель взаимодействия «Клиент-сервер»

Конкретизируя этот рисунок на реализованные Java-программы клиента и сервера (см. пункт 2.5.5, листинги 2.13 и 2.14), мы констатируем, что:

- обе программы соответствуют классической модели «Клиент-сервер», реализованной на базе протокола TCP пакета ***java.net***;
- сами программы реализуют *собственный протокол* взаимодействия, предполагающий: передачу на сервер последовательности текстовых сообщений, передачу клиенту текстовых подтверждений на каждое принятое сообщение и проведение синхронизации процесса взаимодействия, если такие ситуации возникают;
- учебная цель обеих программ — демонстрация технологических возможностей языка Java в пределах сетевой модели OSI, поэтому она содержит минимальный прикладной контекст — синхронное взаимодействие программ посредством строк символов;
- прикладное наполнение этих программ требует разработки дополнительных протоколов; например, проведение структуризации передаваемых сообщений в виде команд и последующей реализации программного обеспечения, поддерживающего эти команды.

Следует обратить особое внимание, что, с ростом функционального наполнения классической модели, растёт не только объем прикладного ПО клиента и сервера, но и объем ПО, поддерживающего все новые протоколы взаимодействия, поскольку каждая задача реализуется самостоятельно.

С этой точки зрения — интересен типовой пример выборки данных, реализованный в пункте 2.6.3, в виде программы-**клиента**, и представленный на листинге 2.15 как Java-программа, использующая инструментальные средства пакета **java.sql**. Здесь также выделяются *клиент*, *сервер* и *протоколы*, но две последних составляющих реализуются не самим программистом, а берутся им как законченные продукты или инструментальные средства. Более того, в зависимости от выбранной СУБД, программист должен найти и использовать совместимый с пакетом Java драйвер JDBC. Таким образом, хотя в прикладном плане мы и можем рассматривать этот пример как классическую модель «Клиент-сервер», но в плане используемой технологии должны констатировать наличие промежуточного ПО — **Middleware**.

В любом случае, технологии рассмотренных примеров не зависят от того, использовался ли язык ООП Java или, например, язык С. Программист вынужден или сам реализовывать прикладные протоколы или, в частных случаях, использовать специализированные инструменты, например, СУБД. Поэтому брокерные модели становятся актуальными, а ее схема взаимодействия демонстрируется рисунком 3.2.

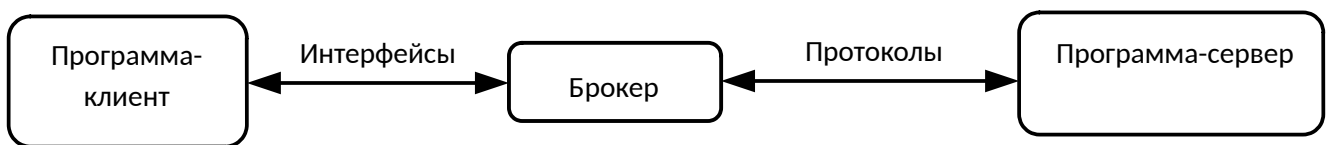


Рисунок 3.2 — Брокерная модель взаимодействия «Клиент-сервер»

Брокерная модель нацелена на упрощение технологии создания программ-**клиентов**. Она предоставляет клиентским программам наборы интерфейсов в виде описаний функций или методов соответствующего языка программирования. Таким образом, логика реализации программ-**клиентов** сводится к технологиям вызова удалённых процедур (*RPC*) или запросам к удалённым объектам, посредством вызова удалённых методов, а ПО брокера выступает в качестве **proxy**-сервера, скрывающего детали реальных запросов к программе-**серверу**.

Технология использования **proxy**-серверов получила очень широкое распространение [35]: «**Прокси-сервер** (от англ. *proxy* — «представитель», «уполномоченный»), **сервер-посредник** — промежуточный сервер (комплекс программ) в компьютерных сетях, выполняющий роль посредника между пользователем и целевым сервером (при этом о посредничестве могут как знать, так и не знать обе стороны), позволяющий клиентам как выполнять косвенные запросы (принимая и передавая их через прокси-сервер) к другим сетевым службам, так и получать ответы. Сначала клиент подключается к прокси-серверу и запрашивает какой-либо ресурс (например e-mail), расположенный на другом сервере. Затем прокси-сервер либо подключается к указанному серверу и получает ресурс у него, либо возвращает ресурс из собственного кэша (в случаях, если прокси имеет свой кэш). В некоторых случаях запрос клиента или ответ сервера может быть изменён прокси-сервером в определённых целях...».

Обычно, технология прокси-серверов дополняется технологией языков описания интерфейсов [36]: «**IDL**, или **язык описания интерфейсов** (англ. *Interface Description Language* или *Interface Definition Language*) — язык спецификаций для описания интерфейсов, синтаксически похожий на описание классов в языке C++. Реализации:

- AIDL: Реализация IDL на Java для Android, поддерживающая локальные и удалённые вызовы процедур. Может быть доступна из нативных приложений посредством JNI.
- CORBA IDL — *язык описания интерфейсов* распределённых объектов, разработанный рабочей группой OMG. Создан в рамках обобщённой архитектуры CORBA.
- IDL DCE, *язык описания интерфейсов* спецификации межплатформенного взаимодействия служб, которую разработал консорциум Open Software Foundation (теперь The Open Group).
- MIDL (Microsoft Interface Definition Language) — *язык описания интерфейсов* для платформы Win32 определяет интерфейс между клиентом и сервером. Предложенная Microsoft технология использует реестр Windows и используется для создания файлов и файлов конфигурации приложений (ASF), необходимых для дистанционного вызова процедуры интерфейсов (RPC) и COM/DCOM-интерфейсов.
- COM IDL — *язык описания интерфейсов* между модулями COM. Является преемником языка IDL в технологии DCE (с англ. — «среда распределённых вычислений») - спецификации межплатформенного взаимодействия служб, которую разработал консорциум Open Software Foundation (теперь The Open Group)...».

Архитектурная модель рисунка 3.2 обычно детализируется в более конкретных представлениях, поэтому рассмотрим отдельно архитектуры вызовов удалённых процедур и запросы к удалённым объектам.

3.1.1 Вызов удалённых процедур

Как было отмечено в пункте 1.2.2 предыдущей главы, технология удалённого вызова процедур (RPC) стала формироваться в начале 90-х годов в рамках проектов распределённых вычислительных сред (DCE). Основу RPC составляла синхронная схема взаимодействия программ клиента и сервера, показанная на рисунке 1.9. Соответственно как показано на рисунке 3.3, ПО брокера распределялось между программами клиента и сервера в виде программ-**заглушек** (*Client stub* и *Server stub*), которые и реализовывали все протоколы взаимодействия.

Технологическая новинка такого подхода — использование C-подобного IDL и специальных компиляторов, генерирующих исходный код программ-**заглушек**. Э. Таненбаум [3] демонстрирует такую технологию схемой, представленной на рисунке 3.4.

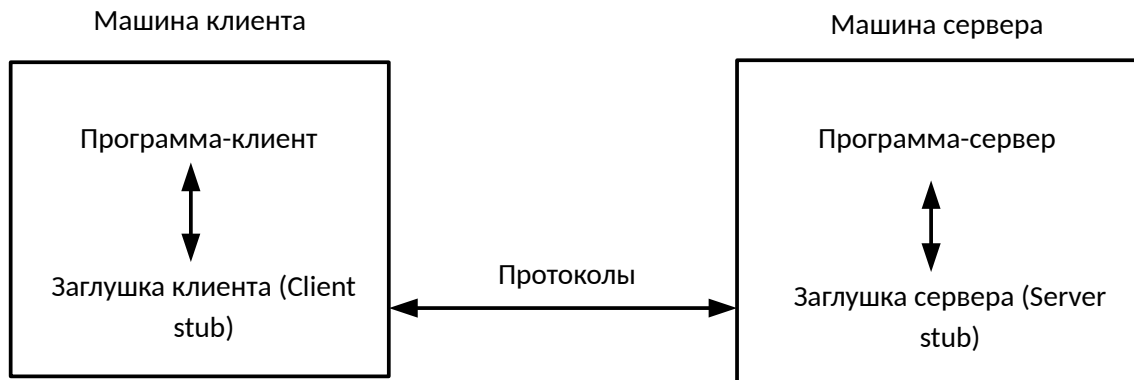


Рисунок 3.3 — Брокерная модель RPC

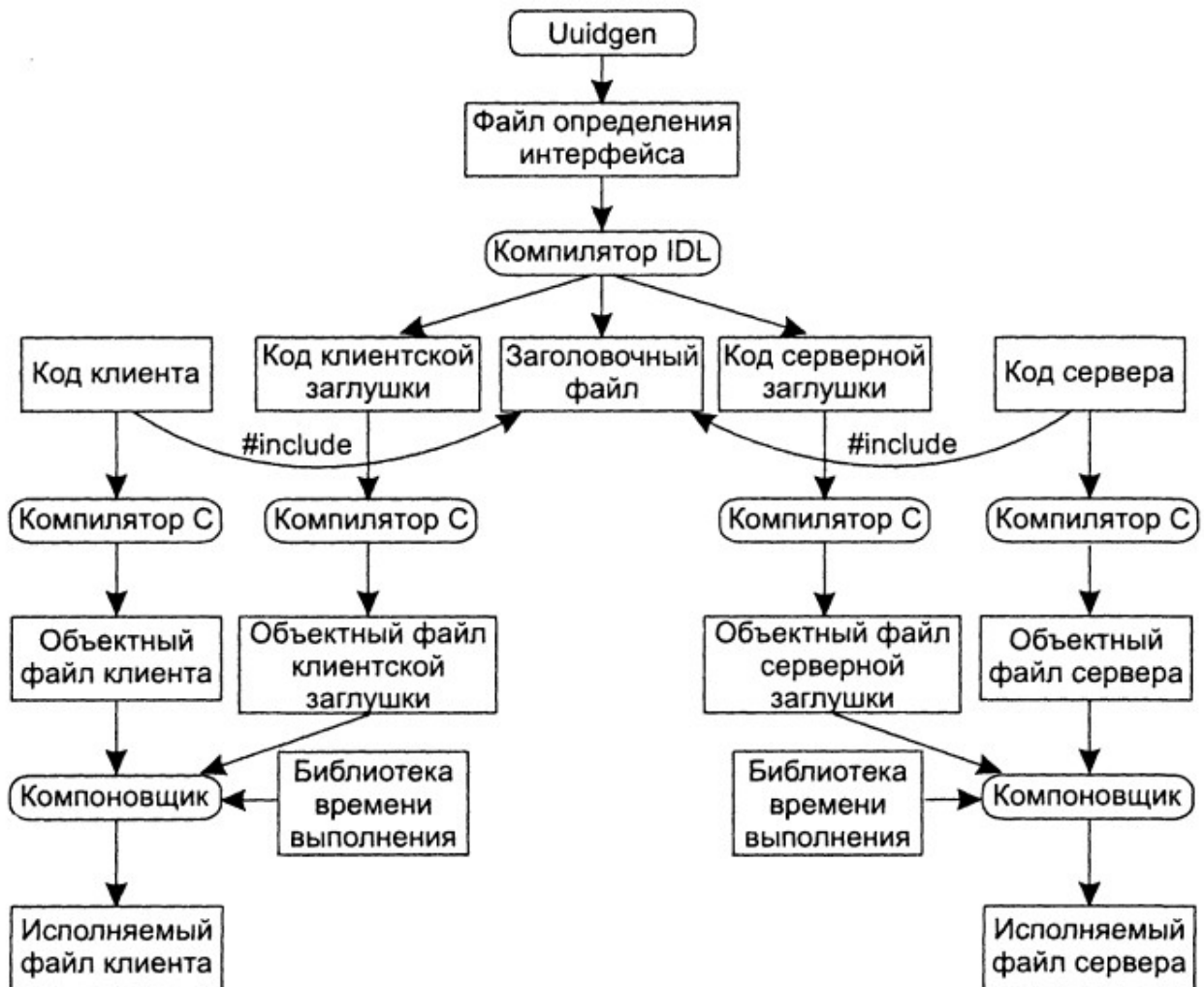


Рисунок 3.4 — Этапы реализации DCE RPC [3]

Пояснения требуют только начальные этапы представленной технологии:

- **Uuidgen** (точнее **uuidgen**) — утилита, генерирующая уникальный универсальный идентификатор, который вставляется в исходный код описания интерфейса; соответственно, этот код используется в ПО заглушек, тем самым идентифицируя реализацию проекта.

- **Файл определения интерфейса** — исходный текст описания интерфейса на языке IDL DCE.
- **Компилятор IDL** — программа, преобразующая файл определения интерфейса в три файла с исходным текстом на языке C: *общий заголовочный файл, код клиентской заглушки и код серверной заглушки*.
- Далее, технология DCE RPC разделяется на две части типичной технологии языка C, обеспечивающей создание программ клиента и сервера.

Мы не будем рассматривать конкретные примеры реализации брокерной модели RPC, поскольку, как отмечено выше, существует множество вариантов реализации языков IDL. Имеются также расширения этой модели ориентированные на асинхронное взаимодействие клиента и сервера, но в целом, в современных представлениях, технология RPC считается устаревшей. Ее основной недостаток — ограниченное время жизни вызываемой процедуры на стороне сервера. В условиях принципиальной ненадёжности взаимодействия в сети, возникают проблемы подтверждения результата или ошибок исполнения вызываемых процедур. Желаям подробно изучить эту проблематику следует обратиться к главе 2 источника [3].

3.1.2 Использование удалённых объектов

Значительным развитием брокерной модели взаимодействия «Клиент-сервер» стало использование объектов, реализуемых средствами языков ООП. Это позволило устранить основной недостаток технологии RPC, сохраняя объекты результатов запросов клиентов на стороне сервера нужное количество времени.

В целом, брокерная модель взаимодействия объектов очень похожа на брокерную модель RPC. Ранее, на примере рисунка 1.10 технологии RMI, было показано, что:

- на стороне программы-**клиента** создаётся объект-**заглушка** (*stub, proxy*), реализующий методы *маршалинга*: преобразования имени объекта, вызываемого метода и его аргументов в поток данных, передаваемых по протоколам серверу;
- на стороне программы-**сервера** создается объект-**заглушка** (*skeleton*), реализующий методы *демаршалинга*: преобразования входного потока данных в запрос к объекту сервера.

В общем случае считается, что удалённый объект (**Remote object**), расположенный на сервере, имеет:

- **Состояние** (*State*) — данные инкапсулируемые (включаемые) объектом.
- **Методы** (*Methods*) — операции над данными состояния объекта.
- **Интерфейс** (*Interface*) — программный код, обеспечивающий доступ к методам объекта.

На основании конструктивных понятий *состояния*, *метода* и *интерфейса* вводится ряд более общих определений.

Распределенный объект (*Distributed object*) — удалённый объект, интерфейсы которого расположены на машинах клиентов, что демонстрируется рисунком 3.5.

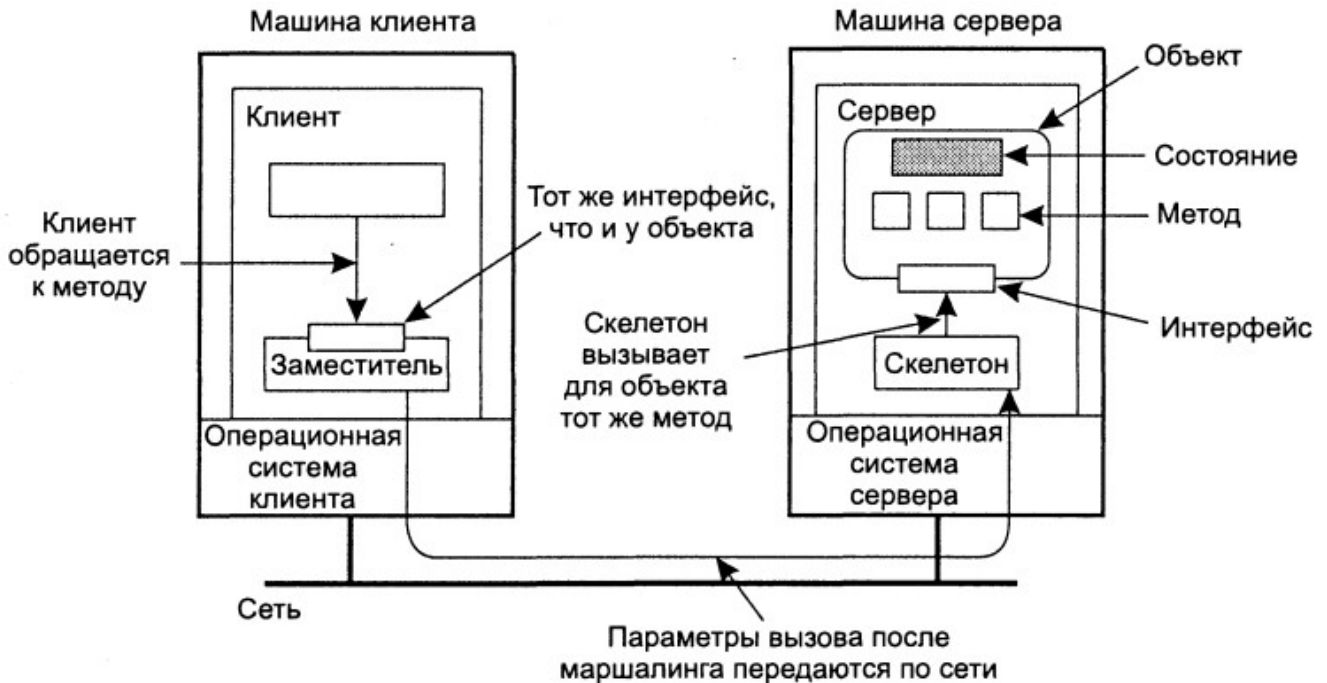


Рисунок 3.5 — Обобщённая организация удалённых объектов [3]

Объект времени компиляции — объект, который может быть полностью описан в рамках своего класса и интерфейсов. В качестве таких описаний могут использоваться объекты и интерфейсы языка Java.

Адаптер объектов (*Object adapter*) — программная оболочка (*wrapper*), единственная задача которой — придать реализации видимость объекта.

Сохраненный объект (*Persistent object*) — объект способный существовать за пределами адресного пространства серверного процесса.

Нерезидентный объект (*Transient object*) — объект способный существовать только под управлением запущенного сервера.

Удалённое обращение к методам (*Remote Method Invocation*) — объектная технология, например - *RMI*, реализованная на языке Java.

Статическое обращение (*Static invocation*) — способ использования предопределённых интерфейсов, предполагающий их предварительную компиляцию в виде заглушек для программ клиентов и сервера.

Динамическое обращение (*Dynamic invocation*) — способ, предполагающий выбор метода удалённого объекта во время выполнения приложения.

Наличие множества возможных подходов к реализации удалённого доступа к объектам потребовало разработки стандартов такого взаимодействия. Наиболее общим подходом является стандартизация проекта CORBA.

3.2 Технология CORBA

Как уже было отмечено в первой главе (см. пункт 1.2.3), CORBA является общей архитектурой брокера объектных запросов, имеет собственный абстрактный протокол GIOP, на основе которого разработаны три Internet-протокола: IIOP, SSLIOP и HTIOP. Дополнительно можно отметить, что CORBA является конкурентом более частной архитектуры **DCOM** (*Distributed Component Object Model, Distributed COM*), которая развивается корпорацией Microsoft.

В данном подразделе рассмотрены только общие концепции архитектуры CORBA. Мы не будем рассматривать альтернативные варианты. Это связано с тем, что брокерных моделей достаточно много, они сложны и поддерживаются множеством несовместимых инструментальных средств. Желающих более подробно изучить эту тему, отправляем к фундаментальному труду Э. Таненбаума [3, глава 9], где он подробно описывает три варианта таких архитектур: CORBA, DCOM и экспериментальную распределенную систему Globe.

Технология CORBA даёт описание распределённых приложений независимо от языков их реализации. Мы будем интерпретировать ее архитектуру, предполагая реализацию на языке Java. Исходя из этого, общая последовательность учебного материала разделена на шесть этапов, которых студент должен придерживаться при проектировании распределенных приложений:

1. **Этап 1** (пункт 3.2.1) — общее описание брокерной архитектуры CORBA. Дополнительно рассматриваются инструментальные средства Java для работы с этой архитектурой.
2. **Этап 2** (пункт 3.2.2) — пример приложения (**NotePad**), которое рассматривается как серверная часть будущего распределенного объекта.
3. **Этап 3** (пункт 3.2.3) — пример приложения (**Example12**), которое рассматривается как клиентская часть будущего распределенного объекта.
4. **Этап 4** (пункт 3.2.4) — генерация базового описания распределенного класса (**OrbPad**), который демонстрирует использование инструментального средства языка IDL, применительно к языку Java.
5. **Этап 5** (пункт 3.2.5) — пример реализации удалённого объекта по описаниям распределенного класса **OrbPad**.
6. **Этап 6** (пункт 3.2.6) — пример реализации клиентской части по описаниям распределенного класса **OrbPad** и общей функциональной части приложения, соответствующего возможностям класса **Example12**.

3.2.1 Брокерная архитектура CORBA

Главной особенностью CORBA является использование компонента **ORB** (*Object Resource Broker – брокер ресурсов объектов*) ранее показанного на рисунке 3.2 и формирующего «мост» между приложениями: программной-**клиентом** и программной-**сервером**. Источник [3] демонстрирует глобальную архитектуру ORB в виде рисунка 3.6.

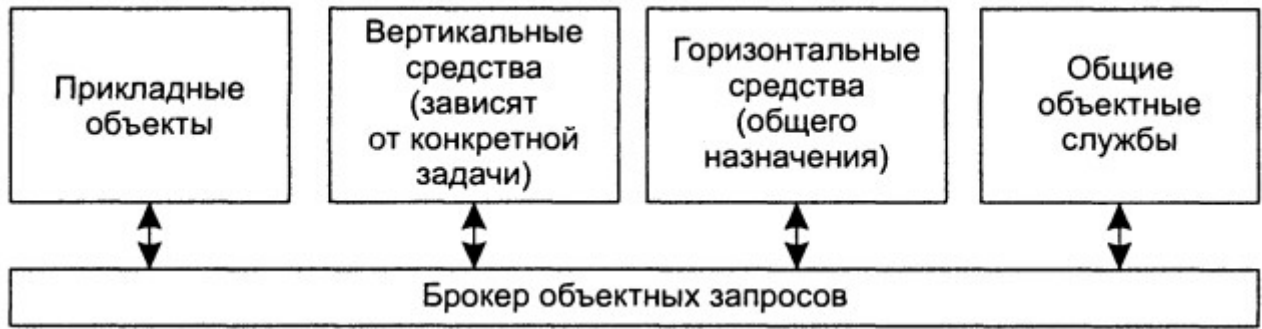


Рисунок 3.6 — Глобальная архитектура CORBA [3]

Здесь, «**Брокер объектных запросов**» - ORB, являющийся реальным сервером, объединяет другие компоненты:

- **Прикладные объекты** — части конкретных распределенных приложений.
- **Общие объектные службы** — базовые сервисы, доступные всем объектам, подключённым к ORB.
- **Горизонтальные средства** (*CORBA horizontal facilities*) — высокоуровневые службы общего назначения, независимые от прикладной области использующих их программ: *средства мобильных агентов, средства печати и средства локализации.*
- **Вертикальные средства** (*CORBA vertical facilities*) — это домены или прикладные области CORBA. Первоначально было выделено одиннадцать рабочих групп, которые работали в следующих сферах применения: *корпоративные системы, финансы и страхование, электронная коммерция, промышленность и другие.*

Для разработчиков приложений наиболее важными являются шестнадцать общих объектных служб, являющиеся ядром CORBA-систем или **сервисами**:

1. Сервис имён (*Naming Service*).
2. Сервис управления событиями (*Event Managment Service*).
3. Сервис жизненных циклов (*Life Cycle Service*).
4. Сервис устойчивых состояний (*Persistent Service*).
5. Сервис транзакций (*Transaction Service*).
6. Сервис параллельного исполнения (*Concurency Service*).
7. Сервис взаимоотношений (*Relationship Service*).
8. Сервис экспорта (*Externalization Service*).
9. Сервис запросов (*Query Service*).
10. Сервис лицензирования (*Licensing Service*).
11. Сервис управления ресурсами (*Property Service*).
12. Сервис времени (*Time Service*).
13. Сервис безопасности (*Security Service*).
14. Сервис уведомлений (*Notification Service*).
15. Сервис трейдинга (*Trader Service*) - анализ текущей ситуации на рынке и заключение торговых сделок.
16. Сервис коллекций (*Collections Service*).

Хотя не все из перечисленных сервисов необходимы для разработки отдельных приложений, хорошо видно, что технология CORBA предоставляет инструменты для самых сложных реализаций распределённых систем, которые в парадигме «Клиент-сервер» разделяют ПО ORB на три части, показанные на рисунке 3.7:

- ORB-сервер;
- ORB клиента;
- ORB сервера.

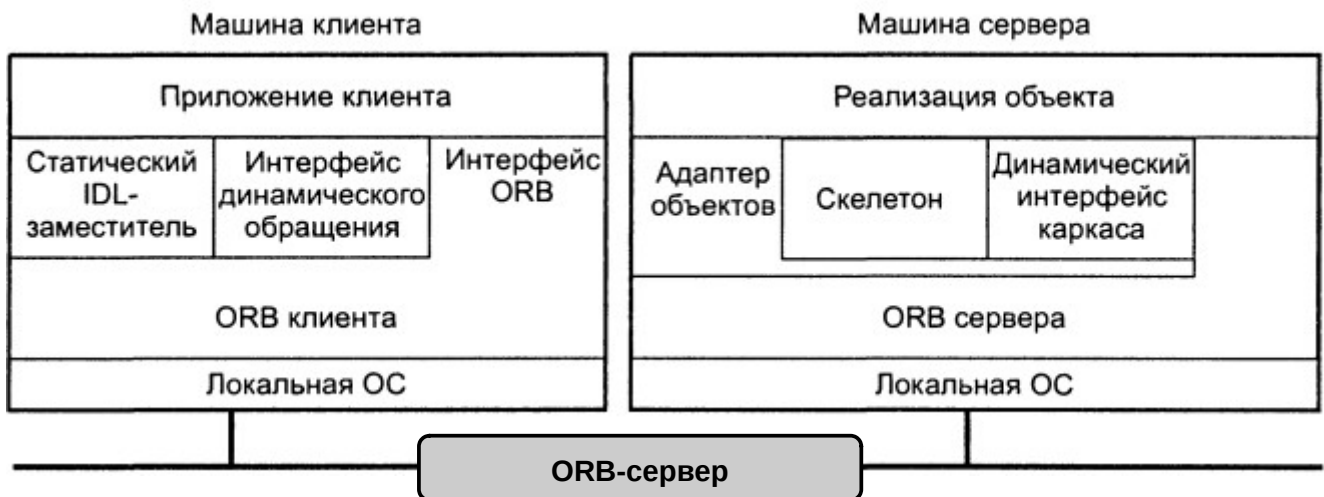


Рисунок 3.7 — Общая организация системы CORBA [3]

В процессе конкретизации клиент-серверной технологии CORBA, применительно к инструментальным средствам языка Java, будем опираться на официальный глоссарий, представленный источником [37], и документацию учебной среды ОС УПК АСУ [6], но основное внимание сосредоточим на реализации конкретного примера.

В современных дистрибутивах Java, все необходимые компоненты ORB поставляются вместе со средой времени исполнения (*JRE*) и доступны для программирования в пределах отдельного пакета **org.omg**. Сам ORB-сервер представлен программой **orbd**, которой для запуска необходимо указать два параметра: **IP-адрес** и **номер порта**. Более подробную информацию о запуске ORB-сервера можно получить из руководства ОС (запусти: **man orbd**), а из командной строки терминала полный запуск по локальному адресу выглядит так:

```
$ export JAVA_HOME=/usr/lib/jvm/default
$ export JAVA_JRE=$JAVA_HOME/jre
$ $JAVA_JRE/bin/orbd -ORBInitialPort 1050 -ORBInitialHost localhost
```

В общем случае запуск сервера можно проводить по любому доступному адресу и номеру порта, но порт **1050/TCP,UDP** специально выделен для CORBA Management Agent (**CMA**). Другие две компоненты ORB подробно опишем в пунктах 3.2.4-3.2.6, когда подготовим конкретное приложение для демонстрации.

3.2.2 Проект серверной части приложения *NotePad*

Технология CORBA содержит множество служебных классов, обеспечивающих многочисленные сервисы. Хотя большинство из них сосредоточено в одном основном пакете *org.omg*, их подробное изучение вызывает большую проблему и требует непомерного количества времени. С другой стороны, любая начальная разработка распределенного приложения выполняется по некоторому шаблону, который задействует небольшое число основных классов и обеспечивает освоение технологии в рамках учебного процесса.

Первым шагом любого шаблона проектирования является выделение той функциональной части приложения, которая будет выполняться на машине сервера. Необходимо, чтобы эта часть была оформлена в виде класса, имеющего чётко описанные интерфейсы методов, а чтобы достичь необходимого качества такого описания — нужно реализовать и отладить такой класс в среде локального приложения.

В качестве прототипа такого распределенного приложения выберем пример работы с СУБД Derby, рассмотренный в пункте 2.6.3 и описанный в виде класса *Example11* на листинге 2.15. Функциональную часть будущего серверного приложения определим в виде класса *NotePad*, который имеет конструктор и интерфейсное описание следующих методов:

- *NotePad()* — конструктор, устанавливающий соединение с уже существующей базой данных *examoleDB*;
- *boolean isConnected()* - метод, проверяющий наличие соединения с БД;
- *Object[] getList()* - метод, получающий содержимое таблицы *notepad* БД *exampleDB* в виде списка текстовых строк;
- *int setInsert(int key, String str)* - метод, добавляющий текст к содержимому таблицы *notepad* БД; также учитывается уникальность ключа *key*;
- *int setDelete(int key)* - метод, удаляющий по заданному ключу *key* запись из таблицы *notepad* БД;
- *void setClose()* - метод, закрывающий соединение с базой данных.

В таком виде класс *NotePad* максимально скрывает все детали работы с СУБД, предоставляя использующим его программам только методы, входящие в базовые средства языка Java. Исходный текст реализации данного класса приведён на листинге 3.1.

Листинг 3.1 — Исходный текст класса NotePad из среды Eclipse EE

```
package asu.rvs;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
```

```

import java.util.ArrayList;

public class NotePad {

    // Объекты класса
    static boolean flag;
    static Connection conn;

    // Конструктор
    public NotePad()
    {
        // Исходные данные для соединения
        String url =
            "jdbc:derby:/home/vgr/databases/exampleDB";
        String user =
            "upk";
        String password =
            "upkasu";

        if(conn != null)
            return;
        else
            flag = false;
        try {
            //Подключаем необходимый драйвер
            Class.forName("org.apache.derby.jdbc.EmbeddedDriver");

            //Устанавливаем соединение с БД
            conn = DriverManager.getConnection(url,
                user, password);

            if(conn.isValid(0))
                flag = true;
            else
                flag = false;
        }
        catch (ClassNotFoundException e1)
        {
            System.out.println("ClassNotFoundException: "
                + e1.getMessage());
            flag = false;
        }
        catch (SQLException e2)
        {
            System.out.println("SQLException: "
                + e2.getMessage());
            flag = false;
        }
    }
    /**
     * Метод, проверяющий наличие соединения с БД
     * @return true, если соединение присутствует.
     */
    public boolean isConnect()
    {
        return flag;
    }
    /**
     * Метод, реализующий SELECT
     * @return Список текстовых строк типа Object[]

```

```

*/
public Object[] getList()
{
    // Защита от неправомерного соединения
    if(!flag) return null;

    String sql = "SELECT * FROM notepad ORDER BY notekey";
    ArrayList<String> al = new ArrayList<String>();

    try
    {
        Statement st =
            conn.createStatement();
        ResultSet rs =
            st.executeQuery(sql);
        if(rs == null)
            return null;

        while(rs.next())
        {
            al.add(rs.getString(1) + "\t" + rs.getString(2));
        }
        return al.toArray();
    }
    catch (SQLException e2)
    {
        System.out.println(e2.getMessage());
    }
    return null;
}

/**
 * Метод, реализующий INSERT
 * @param key - ключ записи;
 * @param str - строка добавляемого текста.
 * @return число измененных строк или -1, если - ошибка.
 */
public int setInsert(int key, String str)
{
    // Защита от неправомерного соединения
    if(!flag)
        return -1;

    String sql = "INSERT INTO notepad values( ? , ? )";

    try
    {
        PreparedStatement pst =
            conn.prepareStatement(sql);

        // Установка первого параметра
        pst.setInt(1, key);
        // Установка второго параметра
        pst.setString(2, str);

        return pst.executeUpdate();
    }
    catch (SQLException e)
    {
        System.out.println(e.getMessage());
        return -1;
    }
}

```

```

    }
}

/**
 * Метод, реализующий DELETE
 * @param key - ключ записи.
 * @return число измененных строк или -1, если - ошибка.
 */
public int setDelete(int key)
{
    // Защита от неправомерного соединения
    if(!flag)
        return -1;

    String sql = "DELETE FROM notepad WHERE notekey = ?";

    try
    {
        PreparedStatement pst =
            conn.prepareStatement(sql);

        // Установка первого параметра
        pst.setInt(1, key);

        return pst.executeUpdate();

    }
    catch (SQLException e)
    {
        System.out.println(e.getMessage());
        return -1;
    }
}

/**
 * Закрытие соединения с БД.
 */
public void setClose()
{
    try
    {
        if(!conn.isClosed())
        {
            conn.commit();
            conn.close();
        }

    }
    catch (SQLException e)
    {
        System.out.println(e.getMessage());
    }
}

/**
 * Тестовый метод
 * @param args
 */
public static void main(String[] args)
{
    System.out.println(
        "Проверка соединения Notepad с БД exampleDB.\n"
        + "-----");
}

```

```

// Создаём объект класса
NotePad obj =
    new NotePad();

if(obj.isConnected())
    System.out.println("БД - подключилась...");
else
    System.out.println("БД - не подключилась!!!");

obj.setClose();
System.out.println("Программа завершила работу...");
}
}

```

Изучая исходный текст листинга 3.1, следует обратить внимание, что конструктор **NotePad()** подключает драйвер СУБД Derby и открывает соединение с БД, которое сохраняется на все время работы с серверной частью приложения. Обычно указанные действия являются критичными для всего проекта и требуют особой проверки, поэтому в текст листинга специально добавлен метод **main(...)**, включающий проверку правильной реализации приложения. Кроме того, дополнительно, необходимо проверить работу приложения, предварительно оформив проект в виде исполняемого **jar**-архива.

В пределах данного пособия, все создаваемые архивы будут размещаться в каталоге пользователя: **\$HOME/lib**. Для заявленного тестирования класса **NotePad** будет использоваться архив **notepad.jar**, поэтому в среде Eclipse EE выполним следующие действия:

- выделим мышкой реализованный проект (в нашем случае — **proj8**);
- правой кнопкой мыши активируем контекстное меню и выберем «**Export...**»;
- в появившемся окне «**Export**» выделим «**Runnable JAR file**», как это показано на рисунке 3.8.

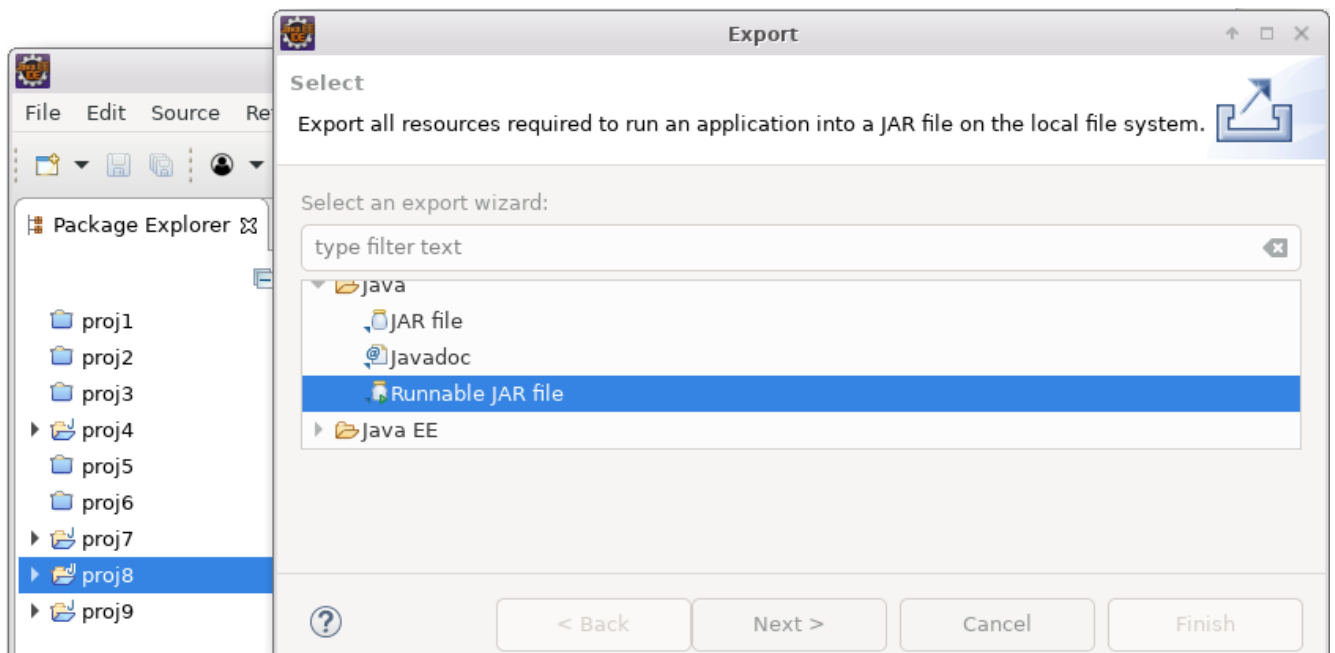


Рисунок 3.8 — Выбор типа экспортируемого проекта

Далее, активировав кнопку «**Next** >», переходим к следующему окну и заполняем его, как показано на рисунке 3.9.

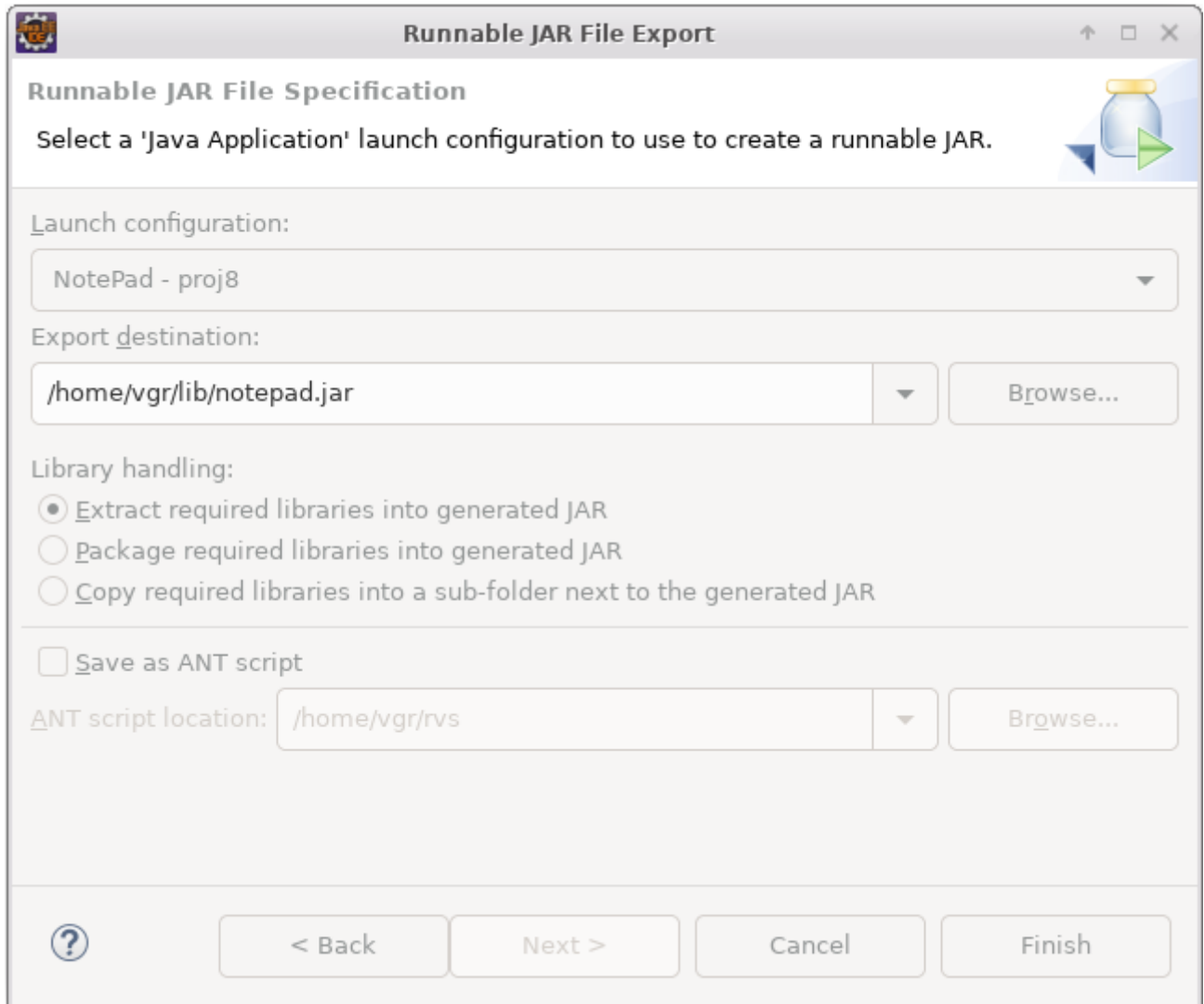


Рисунок 3.9 — Указание имени и места размещения архива

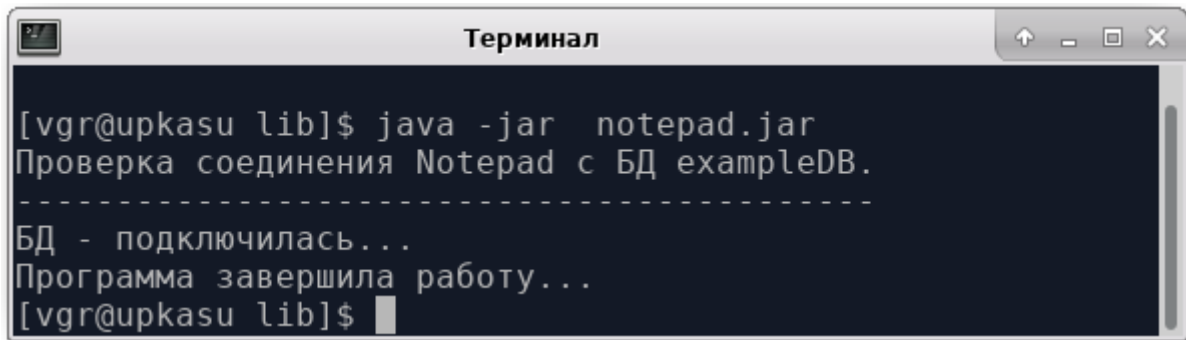
Теперь, активировав кнопку «**Finish**», мы завершаем создание архива класса **NotePad**.

В будущем, все архивы приложений следует создавать по указанной выше схеме.

Для окончательного тестирования класса **NotePad**, следует перейти в каталог **\$HOME/lib**, где выполнить команду:

```
$ java -jar notepad.jar
```

В случае правильно выполненной работы, результат запуска приложения будет выглядеть так, как показано на рисунке 3.10.



```

Терминал
[vgr@upkasu lib]$ java -jar notepad.jar
Проверка соединения Notepad с БД exampleDB.
-----
БД - подключилась...
Программа завершила работу...
[vgr@upkasu lib]$

```

Рисунок 3.10 — Результат тестирования класса NotePad

Естественно, проведённое тестирование не гарантирует полной работоспособности серверной части приложения и может быть завершено только после реализации клиентской части.

3.2.3 Проект клиентской части приложения *Example12*

На втором шаге нашего шаблона проектирования проведём реализацию клиентской части приложения Java-проектом *proj9*, в виде класса *Example12*, куда из класса *Example11* перенесём методы чтения со стандартного ввода:

- `int getKey()` - чтение целого положительного числа;
- `String getString()` - чтение строки произвольной длины.

Сам алгоритм клиентского приложения реализуется в методе *main(...)*, который, по сравнению с алгоритмом *Example11*, дополнен функцией удаления записи из таблицы *notepad* БД *exampleDB*. Внешний результат алгоритма следующий:

- если на предложение программы *ввести ключ* пользователь нажимает клавишу «*Enter*», то программа — завершается;
- если на предложение программы *ввести строку текста* пользователь нажимает клавишу «*Enter*», то введенный ранее *ключ* используется в команде удаления строки из таблицы *notepad*;
- если *введённая строка текста* не является пустой, то введенный ранее *ключ* используется в команде добавления строки в таблицу *notepad*.

Исходный текст реализации класса *Example12* приведён на листинге 3.2.

Листинг 3.2 — Исходный текст класса *Example12* из среды Eclipse EE

```

package ru.tusur.asu;

import java.io.IOException;
import asu.rvs.NotePad;

public class Example12 {

    /**
     * Метод чтения целого числа со стандартного ввода

```

```

    * @return целое число или -1, если - ошибка.
    */
    public int getKey()
    {
        int ch1 = '0';
        int ch2 = '9';
        int ch;
        String s = "";

        try
        {
            while(System.in.available() == 0) ;

            while(System.in.available() > 0)
            {
                ch = System.in.read();
                if (ch == 13 || ch < ch1 || ch > ch2)
                    continue;
                if (ch == 10)
                    break;

                s += (char)ch;
            };
            if (s.length() <= 0)
                return -1;
            ch = new Integer(s).intValue();
            return ch;

        }
        catch (IOException e1)
        {
            System.out.println(e1.getMessage());
            return -1;
        }
    }
    /**
     * Метод чтения строки текста со стандартного ввода
     * @return строка текста.
     */
    public String getString()
    {
        String s = "\r\n";
        String text = "";
        int n;
        char ch;
        byte b[];

        try
        {
            //Ожидаем поток ввода
            while(System.in.available() == 0) ;
            s = "";
            while((n = System.in.available()) > 0)
            {
                b = new byte[n];
                System.in.read(b);
                s += new String(b);
            };
            // Удаляем последние символы '\n' и '\r'
            n = s.length();
            while (n > 0)
            {

```

```

        ch = s.charAt(n-1);
        if (ch == '\n' || ch == '\r')
            n--;
        else
            break;
    }
    // Выделяем подстроку
    if (n > 0)
        text = s.substring(0, n);
    else
        text = "";
    return text;
}
catch (IOException e1)
{
    System.out.println(e1.getMessage());
    return "Ошибка...";
}
}
/**
 * Реализация алгоритма работы с серверной частью
 * приложения.
 * @param args
 */
public static void main(String[] args)
{
    System.out.println(
        "Example12 для ведения записей в БД exampleDB.\n"
        + "\t1) если ключ - пустой, то завершаем программу;\n"
        + "\t2) если текст - пустой, то удаляем по ключу;\n"
        + "\t3) если текст - не пустой, то добавляем его.\n"
        + "Нажми Enter - для продолжения ...\n"
        + "-----");

    // Создаем объекты классов
    Example12 ex =
        new Example12();
    ex.getKey();

    NotePad obj =
        new NotePad();
    if (!obj.isConnected())
    {
        System.out.println("Не могу создать объект класса NotePad...");
        System.exit(1);
    }

    int ns;           // Число прочитанных строк
    int nb;           // Число прочитанных байт
    String text, s;    // Строка введенного текста
    Object[] ls;

    // Цикл обработки запросов
    while(true)
    {
        //Печатаем заголовок ответа
        System.out.println(
            "-----\n"
            + "Ключ\tТекст\n"
            + "-----");
        ls = obj.getList();
    }
}

```

```

if (ls == null )
{
    System.out.println("Нет соединения с базой данных...");
    break;
}

//Выводим (построчно) результат запроса к БД
ns = 0;
nb = ls.length;

while(ns < nb){
    System.out.println(ls[ns] + "\n");
    ns++;
}
// Выводим итог запроса
System.out.println(
    "-----\n"
    + "Прочитано " + ls.length + " строк\n"
    + "-----\n"
    + "Формируем новый запрос!");

System.out.print("\nВведи ключ или Enter: ");
nb = ex.getKey();

if (nb == -1)
    break; // Завершаем работу программы

System.out.print("Строка текста или Enter: ");
s = ex.getString();
text = s;

while (s.length() > 0)
{
    System.out.print("Строка текста или Enter: ");
    s = ex.getString();
    if(s.length() <= 0)
        break;
    text += ("\n" + s);
}

if (text.length() <= 0)
{
    ns = obj.setDelete(nb);
    if (ns == -1)
        System.out.println("\nОшибка удаления строки !!!");
    else
        System.out.println("\nУдалено " + ns + " строк...");
}
else
{
    ns = obj.setInsert(nb, text);
    if (ns == -1)
        System.out.println("\nОшибка добавления строки !!!");
    else
        System.out.println("\nДобалено " + ns + " строк...");
}

System.out.println("Нажми Enter ...");
ex.getKey();
}

```

```

        //Закрываем все объекты и разрываем соединение
        obj.setClose();
        System.out.println("Программа завершила работу...");
    }
}

```

После тестирования реализованного приложения в среде Eclipse EE, необходимо:

- создать архив приложения с именем **example12.jar**, по технологии описанной в предыдущем пункте;
- перейти в директорию **\$HOME/lib** и запустить приложение, как показано на рисунке 3.11.

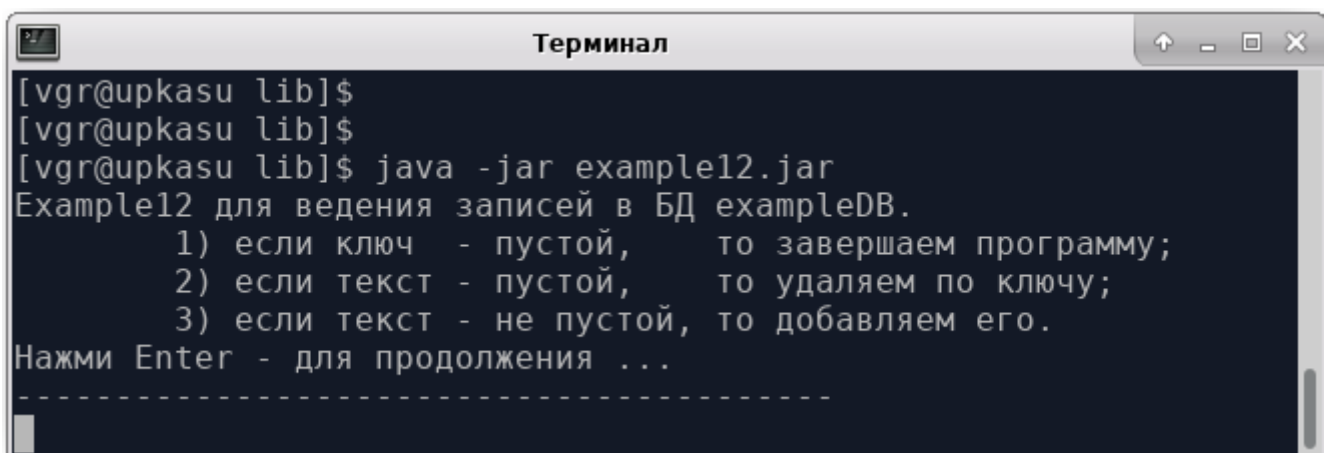


Рисунок 3.11 — Запуск на тестирования приложения Example12

Если тестирование приложения **Example12** прошло успешно, то тогда тестирование класса **NotePad** - полностью завершено и можно переходить к созданию распределённой системы.

3.2.4 Генерация распределенного объекта OrbPad

На третьем шаге проектирования проведём генерацию компонент распределенной системы **OrbPad**, которая:

- для программ-**клиентов** — будет предоставляться реализованный интерфейс с именем **OrbPad**, имеющий те же методы, что и класс **NotePad**;
- для программы-**сервера** — на каждый запрос клиента будет генерироваться объект с именем «**OrbPad**», обеспечивающий функциональность методов класса **NotePad**.

Работа по созданию распределенных объектов начинается с описания интерфейсов, которые удалённый объект, созданный на сервере, будет предоставлять программам-**клиентам**. Технология CORBA требует, чтобы используемые интерфейсы были описаны на языке IDL, независимым от языка реализации. Затем, это

описание компилируется в конкретный язык реализации.

Язык Java имеет собственный компилятор *idlj*, обеспечивающий преобразование формального описания IDL в набор файлов с шаблонами исходных текстов на языке Java. Поскольку IDL использует собственные типы языковых конструкций, то для правильного описания интерфейсов следует пользоваться таблицей 3.1, показывающей соответствие типов IDL и Java. Этих данных вполне достаточно для построения большинства мыслимых описаний интерфейсов.

Таблица 3.1 — Соответствие типов языков IDL и Java

IDL Type	Java Type
module	package
boolean	boolean
char, wchar	char
octet	byte
string, wstring	java.lang.String
short, unsigned short	short
long, unsigned long	int
long long, unsigned long long	long
float	float
double	double
fixed	java.math.BigDecimal
void	void
enum, struct, union	class
sequence, array	array
interface (non-abstract)	signature interface и operations interface, helper class, holder class
interface (abstract)	signature interface, helper class, holder class
exception	class
Any	org.omg.CORBA.Any
typedef	helper classes
readonly attribute	accessor method
readwrite attribute	accessor and modifier methods
operation	method

Дальнейшую разработку будем вести в среде Eclipse EE, где откроем проект

с именем **proj10**. Выделим в **Package Explorer/proj10/src** и правой кнопкой мыши активируем меню, в котором выберем **New → File**. В появившемся окне укажем имя файла **orbpad.idl**, как показано на рисунке 3.12.

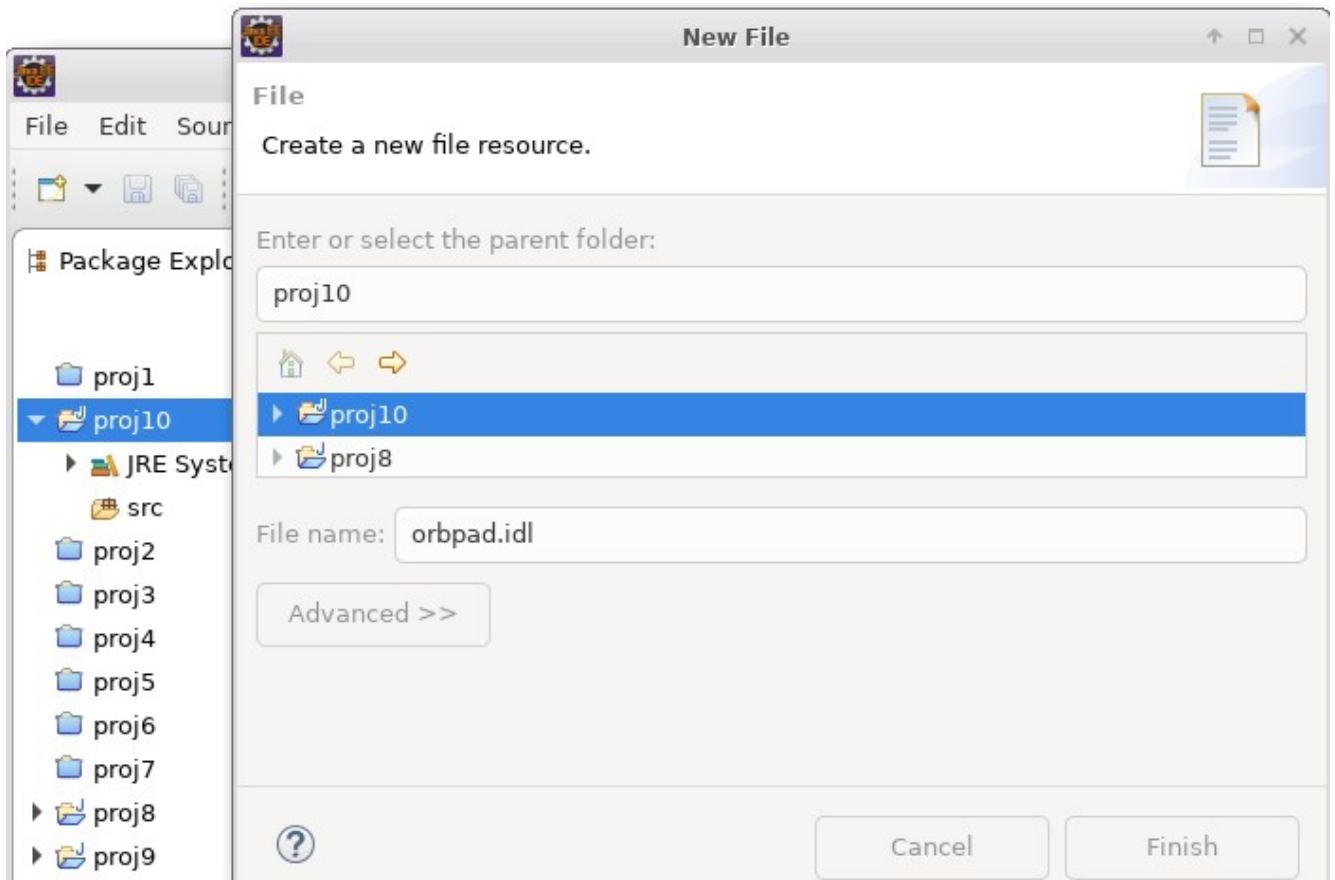


Рисунок 3.12 — Открытие IDL-файла в проекте proj10

Далее, нажимаем кнопку «**Finish**» и запустится редактор **Mousepad**, куда нужно ввести описание интерфейса распределенного объекта **OrbPad**.

Само описание интерфейса проведём на основе методов класса **NotePad**, реализованного в пункте 3.2.2. С учётом преобразования типов таблицы 3.1, оно будет выглядеть как показано на рисунке 3.13.

По синтаксису, язык IDL очень похож на язык C++. Его формальное описание можно найти, например, в источнике [38]. Мы лишь отметим, что полное описание любого интерфейса должно быть помещено в синтаксическую конструкцию:

```
module имя_модуля { ... };
```

причём, **имя_модуля** будет частью операторов **package** в генерируемых шаблонах языка Java, а описания методов будут помещаться в конструкции типа:

```
interface имя_интерфейса { ... };
```

где **имя_интерфейса** — будет присутствовать в именах генерируемых файлов

компонент (не менее шести файлов на один интерфейс).

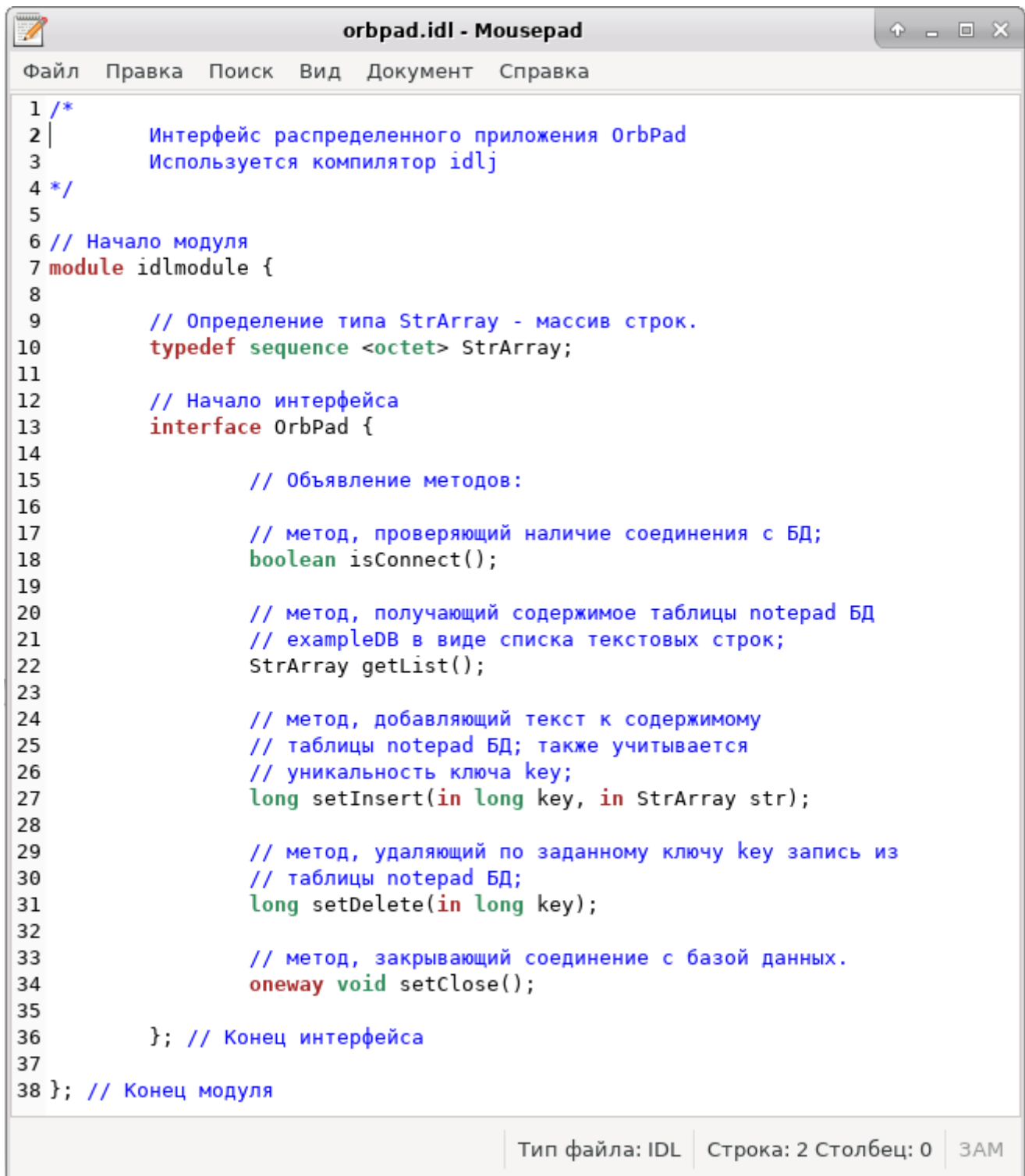


Рисунок 3.13 — Описание интерфейсов OrbPad на языке IDL

Что касается описания методов, то здесь необходимо учитывать следующие особенности:

- комментарии переносятся в исходные тексты генерируемых компонент, что облегчает их последующее использование, но писать их нужно на английском языке, поскольку компилятор *idlj* не поддерживает национальные язы-

ки;

- не следует использовать строки (**string**) и массивы строк (**sequence string**), поскольку возникают ошибки кодирования/декодирования; лучше их заменять массивами байт (**sequence octet**);
- все описываемые методы будут иметь модификатор **public**;
- если метод не возвращает данные (имеет тип **void**) и клиент не будет ожидать завершения этого метода, то следует использовать специальный модификатор **oneway**;
- при описании аргументов методов используются ключевые слова **in**, **inout** и **out**; модификатор **in** применяется тогда, когда методу передаётся копия значения аргумента; модификатор **out** указывает, что методу передаётся ссылка на Java-объект, содержащий в себе другой Java-объект; в этом случае IDL-компилятор генерирует специальные классы типа **HOLDER**, которые доступны в пакете **org.omg.CORBA**; модификатор **inout** использует и ту, и другую семантику.

В целом, работа по описанию интерфейсов может быть достаточно сложной и объёмной. Она также требует хорошего знания тонкостей технологии CORBA, поэтому мы ограничимся только рассмотренным примером и генерацией компонент интерфейса в среде системы Eclipse EE. Для этого запустим компилятор **idlj**, размещённый в каталоге **\$JAVA_HOME/bin**, в виде:

```
idlj <Опции> имя_IDL_файла
```

где стандартный набор <Опции> - включает:

- **-f**<client | server | all>
- **-pkgPrefix** <имя модуля> <добавляемый префикс>
- **-td** <выходной каталог генерации компонент>

Чтобы провести генерацию компонент объекта **OrbPad** в среде Eclipse EE и на основе файла **orbpad.idl** проекта **proj10**, нужно запустить IDL-компилятор **idlj** с правильными параметрами. Для этого, в главном меню среды разработки выбираем «**Run->External Tools->External Tools Configuration...**», а в левой части появившегося окна активируем «**New_configuration**» и заполняем правую часть так, как показано на рисунке 3.14.

Нажав последовательно кнопки «**Apply**» и «**Run**», мы запустим компилятор **idlj**, который создаст *восемь файлов* с расширением **.java**, показанных на рисунке 3.15 и достаточных как для создания серверной части распределенного объекта, так и для клиентских программ.

Обязательно, после генерации компонент проекта необходимо выделить его имя (**proj10**) и правой кнопкой мыши активировать меню, выбрав «**Refresh**». Это нужно для того, чтобы среда разработки Eclipse EE обнаружила внесённые изменения.

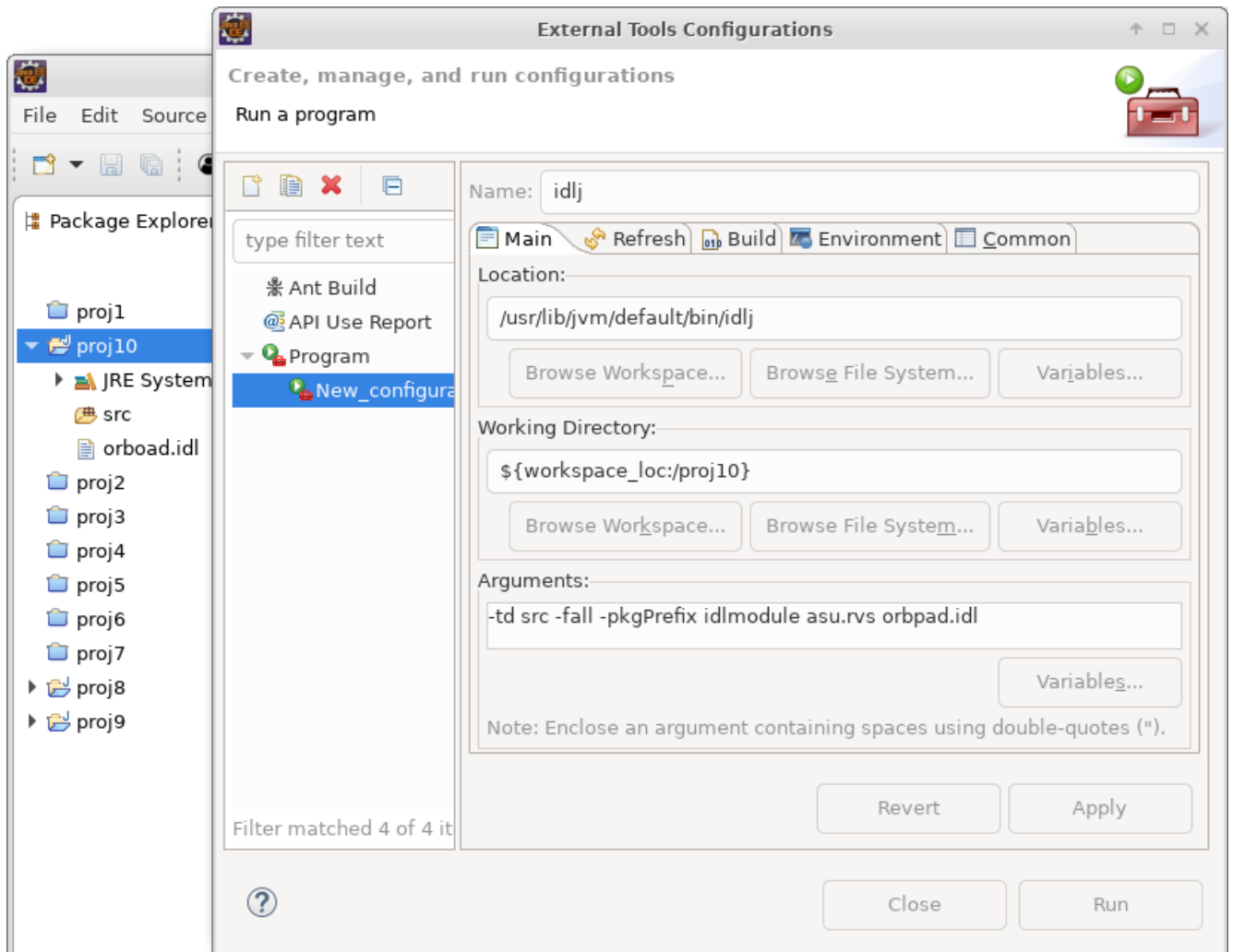


Рисунок 3.14 — Задание параметров вызова компилятора idlj

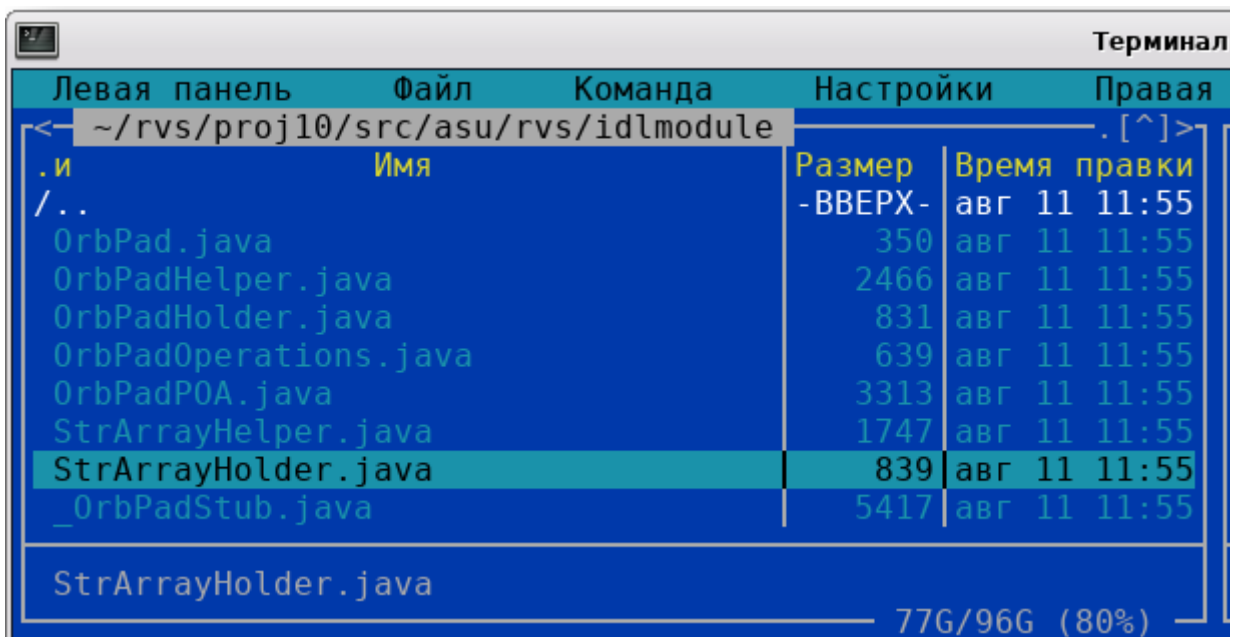


Рисунок 3.15 — Полный список сгенерированных файлов проекта proj10

Обычно, в простейших случаях генерируются *по шесть файлов* на каждый описанный интерфейс. В нашем случае, добавлены два файла ***StrArrayHelper.java*** и ***StrArrayHolder.java***, необходимые для описания типа ***StrArray***, объявленного в файле ***orbpad.idl*** (см. рисунок 3.13).

Главная компонента нашего проекта — файл ***OrbPad.java***, показанный на листинге 3.3 и являющийся интерфейсом для сервера и клиентских приложений. Он описывает простое объединение трёх интерфейсов:

- ***org.omg.CORBA.Object*** и ***org.omg.CORBA.portable.IDLEntity*** — два интерфейса пакета ***org.omg.CORBA***;
- ***OrbPadOperations*** — определённый в файле ***OrbPadOperations.java*** интерфейс, объявляющий методы распределенного объекта ***OrbPad*** (см. листинг 3.4).

Листинг 3.3 — Исходный текст файла OrbPad.java из среды Eclipse EE

```
package asu.rvs.idlmodule;

/**
 * asu/rvs/idlmodule/OrbPad.java .
 * Generated by the IDL-to-Java compiler (portable), version "3.2"
 * from orbpad.idl
 * 11 августа 2019 г. 11:55:56 KRAT
 */

// Начало интерфейса
public interface OrbPad extends OrbPadOperations,
    org.omg.CORBA.Object,
    org.omg.CORBA.portable.IDLEntity
{
} // interface OrbPad
```

Листинг 3.3 — Исходный текст файла OrbPadOperations.java из среды Eclipse EE

```
package asu.rvs.idlmodule;

/**
 * asu/rvs/idlmodule/OrbPadOperations.java .
 * Generated by the IDL-to-Java compiler (portable), version "3.2"
 * from orbpad.idl
 * 11 августа 2019 г. 11:55:56 KRAT
 */

//Начало интерфейса
public interface OrbPadOperations
{
    // 'метод, проверяющий наличие соединения с БД;'
    boolean isConnect ();

    // "метод, получающий содержимое таблицы notepad БД"
    // exampleDB в виде списка текстовых строк;
    String[] getList ();

    // метод, добавляющий текст к содержимому
    // таблицы notepad БД; также учитывается
```

```
// уникальность ключа key;
int setInsert (int key, String str);

// метод, удаляющий по заданному ключу key запись из
// таблицы notepad БД;
int setDelete (int key);

// метод, закрывающий соединение с базой данных.
void setClose ();
} // interface OrbPadOperations
```

Обязательно проверьте листинг 3.4 на правильность описания методов.

Более подробно содержимое сгенерированных компонент рассмотрим в следующих пунктах данного подраздела, а сейчас выделим наиболее важные методы абстрактного класса **org.omg.CORBA.ORB**, полное описание которых можно найти в источнике [39].

Обычно, объекты абстрактного класса **ORB** создаются статическим методом **init(String[] args, Properties props)**, например:

```
ORB orb = ORB.init(args, null);
```

Далее, такой объект **orb** может быть использован программой клиента или сервера для регистрации себя на ORB-сервере. Кроме того:

- **org.omg.CORBA.Object resolve_initial_references(String object_name)** - разрешает ссылку конкретной цели от набора доступных начальных имён службы; такими объектами являются: «**RootPOA**» — менеджер объектного адаптера сервера, а также «**NameService**» — общая служба имен ORB-сервера;
- **void run()** - используется сервером для ожидания запросов на выполнение методов, одновременно блокируя завершение его работы;
- **void shutdown(boolean wait_for_completion)** — используется сервером для завершения метода **run()**; обычно используется в виде: **orb.shutdown(false)**;
- **void destroy()** - уничтожает объект **orb**, созданный ранее методом **init()**;

Теперь перейдём к реализации программы сервера.

3.2.5 Реализация серверной части ORB-приложения

На четвертом шаге проектирования создадим серверную часть ORB-приложения, которая обеспечивает интерфейс удалённого объекта **OrbPad**. Простейший вариант архитектуры такого приложения состоит из двух классов:

- собственно класса сервера (назовём его **OrbPadServer**), имеющего статический метод **main()**, организующего приём запросов от клиентов и отправку им результатов выполненных методов;
- класса серванта (*servant* или *слуга*, - назовём его **OrbPadServant**), реализующего только методы объявленного интерфейса.

Первым реализуется класс серванта, поскольку он обеспечивает функциональность основного сервера.

Технология создания серванта **OrbPadServant** основана на расширении абстрактного класса **OrbPadPOA**, который уже сгенерирован на основе IDL-описания интерфейса и находится в файле **OrbPadPOA.java**. Сам абстрактный класс определён с точностью до интерфейса **OrbPadOperations**, показанного выше на листинге 3.3. Таким образом, класс **OrbPadServant** должен состоять из собственного конструктора, вспомогательного метода, передающего в сервант объекты классов **ORB** и **NotePad**, и пяти методов, реализующих объявленный интерфейс удалённого объекта. Исходный текст серванта представлен на листинге 3.4.

Листинг 3.4 — Исходный текст серванта OrbPadServant из среды Eclipse EE

```
package asu.rvs.server;

import org.omg.CORBA.ORB;
import asu.rvs.NotePad;
import asu.rvs.idlmodule.*;

/**
 * Сервант, реализующий методы интерфейса
 * OrbPadOperations.
 * Конструктор серванта объявлен неявно.
 * @author vgr
 */
public class OrbPadServant extends OrbPadPOA
{
    private NotePad obj;
    private ORB orb;

    /**
     * Специальный метод, используемый сервером для
     * передачи в сервант объектов двух классов:
     * @param obj - Объект NotePad
     * @param orb - Объект ORB
     */
    public void setObjects(NotePad obj, ORB orb)
    {
        this.obj = obj;
        this.orb = orb;
    }

    /**
     * Методы, реализующие объявленный интерфейс:
     *
     * метод, проверяющий наличие соединения с БД;
     * @return
     */
    public boolean isConnect ()
    {
        return obj.isConnect();
    }

    /**
     * метод, получающий содержимое таблицы notepad БД
     * exampleDB в виде списка текстовых строк и передающий
     * его клиенту в виде массива байт;
     * @return
     */
}
```

```

*/
public byte[] getList ()
{
    java.lang.Object[] os =
        obj.getList();
    if(os == null)
        return "Нет данных...".getBytes();

    int ns =
        os.length;
    // Конструирование общей строки
    String ss = "" + os[0].toString();

    for(int i=1; i < ns; i++)
    {
        // Вставка разделителя строк
        ss = ss + "###" + os[i];
    }

    return ss.getBytes();
}

/**
 * метод, добавляющий текст к содержимому
 * таблицы potepad БД, где также учитывается
 * уникальность ключа key;
 */
public int setInsert (int key, byte[] bt)
{
    return obj.setInsert(key,
        new String(bt));
}

/**
 * метод, удаляющий по заданному ключу key запись
 * из таблицы potepad БД;
 */
public int setDelete (int key) {
    return obj.setDelete(key);
}

/**
 * метод, закрывающий соединение с базой данных.
 * мы только останавливаем orb.run()
 * остальное должен сделать сервер.
 */
public void setClose ()
{
    // аргумент обязательно должен быть: false
    orb.shutdown(false);
}
}

```

Обратите внимание, что реализованные методы принимают от клиента и передают ему строковые данные в виде потока байт. Это является вынужденной мерой, поскольку имеющаяся реализация технологии CORBA не способна нормально передавать текст национальных языков.

После создания класса серванта, реализуется класс сервера *OrbPadServer*. Его задача:

- создать объект класса *NotePad* и проверить его готовность к работе;
- создать и инициализировать объект класса *ORB*, являющийся локальным представителем брокера в программе сервера и показанный ранее на рисунке 3.7 как компонент *ORB сервер*;
- создать объект класса *POA*, являющийся локальным представителем (адаптером) переносимых объектов сервера, и активировать его;
- создать объект класса *OrbPadServer*, являющийся сервантом, и передать ему ссылки на созданные объекты классов *NotePad* и *ORB*;
- с помощью менеджера *POA* создать объект класса *org.omg.CORBA.Object*, являющийся специальной ссылкой на объект серванта;
- с помощью сгенерированного класса *OrbPadHelper* создать объект класса *OrbPad*, являющийся ссылкой на объект серванта и, тем самым, удаленным объектом, с которым будут работать программы клиентов;
- создать объекты-ссылки на службу имён «*NameService*» отдельного сервера *ORB*, который уже должен быть запущен;
- провести регистрацию удалённого объекта на сервере ORB под именем «*OrbPad*»;
- запустить цикл приема запросов от программ клиентов, который обеспечивается методом *run()* локального объекта класса *ORB*.

Исходный текст сервера, обеспечивающего решение перечисленных выше задач, представлен на листинге 3.5.

Листинг 3.5 — Исходный текст сервера *OrbPadServer* из среды *Eclipse EE*

```
package asu.rvs.server;

import java.util.Properties;
import org.omg.CORBA.ORB;
import org.omg.CosNaming.NameComponent;
import org.omg.CosNaming.NamingContextExt;
import org.omg.CosNaming.NamingContextExtHelper;
import org.omg.PortableServer.POA;
import org.omg.PortableServer.POAHelper;
import asu.rvs.NotePad;
import asu.rvs.idlmodule.OrbPad;
import asu.rvs.idlmodule.OrbPadHelper;

/**
 * Реализация сервера распределенного объекта OrbPad
 * @author vgr
 */
public class OrbPadServer {

    public static void main(String[] args) {
        /**
         * Создаем объект класса NotePad
         */
        NotePad obj =
            new NotePad();
```



```

if(!obj.isConnect())
{
    System.out.println("Не могу создать объект класса NotePad ...");
    return;
}
/**
 * Задаем параметры сервера по умолчанию
 */
Properties props = System.getProperties();
props.put( "org.omg.CORBA.ORBInitialHost",
"localhost" );
props.put( "org.omg.CORBA.ORBInitialPort",
"1050" );

try{
    // Открываем и инициализируем ORB
    ORB orb = ORB.init(args, props);

    /**
     * Получаем ссылку на объект адаптера и
     * активируем его менеджер POAManager
     */
    POA rootpoa =
        POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
    rootpoa.the_POAManager().activate();

    // Создаем servant и регистрируем в нем NotePad и ORB
    OrbPadServant servant =
        new OrbPadServant();
    servant.setObjects(obj, orb);

    /**
     * Получаем ссылку на объект серванта и создаем
     * удаленный объект класса OrbPad, как ссылку на сервант
     */
    org.omg.CORBA.Object ref =
        rootpoa.servant_to_reference(servant);
    OrbPad orbpad =
        OrbPadHelper.narrow(ref);

    /**
     * Получаем объектную ссылку на службу NameService,
     * которую обеспечивает ORB-сервер
     */
    org.omg.CORBA.Object objRef =
        orb.resolve_initial_references("NameService");

    /**
     * Используем NamingContextExt, который является частью
     * спецификации Interoperable Naming Service (INS)
     */
    NamingContextExt ncRef =
        NamingContextExtHelper.narrow(objRef);

    /**
     * Регистрируем удаленный объект
     * в службе имен ORB-сервера
     */
    NameComponent path[] =
        ncRef.to_name( "OrbPad" );
    ncRef.rebind(path, orbpad);
}

```

```

        System.out.println("OrbPadServer готов и ждет ...");

        // Цикл ожидания входящих запросов от клиентов
        orb.run();

        // Нормальное завершение работы сервера
        obj.setClose();
        orb.destroy();
    }
    catch (Exception e)
    {
        System.err.println("ERROR: " + e);
        obj.setClose();
    }
    System.out.println("OrbPadServer завершил работу ...");
}
}

```

При нормальном запуске, сервер должен выдать сообщение «*OrbPadServer готов и ждёт ...*» и ожидать приема запросов от программ клиентов. Следует создать запускаемый jar-архив проекта **proj10**, поместить его в каталог **\$HOME/lib** под именем, например **orbpadserver.jar**, перейти в указанный каталог и проверить нормальный запуск сервера командой:

```
$ java -jar orbpadserver.jar
```

Теперь, можно перейти к реализации клиентского приложения.

3.2.6 Реализация клиентской части ORB-приложения

Приложение клиента можно реализовать и в проекте **proj10**. По крайней мере, среда Eclipse EE позволяет это сделать, но мы реализуем его в отдельном проекте **proj11**, демонстрируя реальную ситуацию, когда разработчику доступно только описание интерфейса на языке IDL. Поэтому:

- создадим новый проект;
- скопируем в него файл описания интерфейса **orbpad.idl**;
- проведём генерацию компонент приложения, как это было описано в предыдущем пункте.

Приложение клиента реализуем в виде класса **OrbPadClient**. Для чего выполним следующие действия:

- создадим новый Java-класс с указанным именем, где в качестве операнда оператора **package** укажем **asu.rvs.client**;
- перенесём в созданный класс методы класса **Example12** (см. листинг 3.2): метод **getKey()**, метод **getString()** и ту часть метода **main(...)**, которая касается основного цикла обработки запросов.

Дополнительно, в классе **OrbPadClient** должен быть реализован доступ к удалённому объекту, который зарегистрирован на отдельном сервере **ORB** под именем «**OrbPad**». Для этого, в методе **main(...)** необходимо:

- создать объект класса **OrbPadClient** — для доступа к собственным методам класса;
- создать и инициализировать локальный объект класса **ORB** — для доступа к серверу **ORB**;
- создать объекты-ссылки — для доступа к серверу **ORB**;
- создать объект класса **OrbPad**, являющийся ссылкой на удалённый объект и зарегистрированный на сервере **ORB** под именем «**OrbPad**», — для доступа к методам интерфейса;
- включить созданные объекты в алгоритм работы приложения.

Полная реализация клиентской части приложения, адекватная реализованному ранее приложению **Example12**, приведена на листинге 3.6.

Листинг 3.6 — Исходный текст клиента **OrbPadClient** из среды **Eclipse EE**

```
package asu.rvs.client;

import java.io.IOException;
import java.util.Properties;
import org.omg.CORBA.ORB;
import org.omg.CosNaming.NamingContextExt;
import org.omg.CosNaming.NamingContextExtHelper;

import asu.rvs.idlmodule.OrbPad;
import asu.rvs.idlmodule.OrbPadHelper;

/**
 * Реализация клиента распределенного объекта OrbPad
 * @author vgr
 */
public class OrbPadClient {
    /**
     * Метод чтения целого числа со стандартного ввода
     * @return целое число или -1, если - ошибка.
     */
    public int getKey()
    {
        int ch1 = '0';
        int ch2 = '9';
        int ch;
        String s = "";

        try
        {
            while(System.in.available() == 0) ;

            while(System.in.available() > 0)
            {
                ch = System.in.read();
                if (ch == 13 || ch < ch1 || ch > ch2)
                    continue;
                if (ch == 10)
                    break;
            }
        }
    }
}
```

```

        s += (char)ch;
    };
    if (s.length() <= 0)
        return -1;
    ch = new Integer(s).intValue();
    return ch;

}
catch (IOException e1)
{
    System.out.println(e1.getMessage());
    return -1;
}
}
/**
 * Метод чтения строки текста со стандартного ввода
 * @return строка текста.
 */
public String getString()
{
    String s = "\r\n";
    String text = "";
    int n;
    char ch;
    byte b[];

    try
    {
        //Ожидаем поток ввода
        while(System.in.available() == 0) ;
        s = "";
        while((n = System.in.available()) > 0)
        {
            b = new byte[n];
            System.in.read(b);
            s += new String(b);
        };
        // Удаляем последние символы '\n' и '\r'
        n = s.length();
        while (n > 0)
        {
            ch = s.charAt(n-1);
            if (ch == '\n' || ch == '\r')
                n--;
            else
                break;
        }
        // Выделяем подстроку
        if (n > 0)
            text = s.substring(0, n);
        else
            text = "";
        return text;
    }
    catch (IOException e1)
    {
        System.out.println(e1.getMessage());
        return "Ошибка...";
    }
}

```

```

public static void main(String[] args) {
    System.out.println(
        "OrbPadClient для работы с удаленным объектом OrbPad.\n"
        + "\t1) если ключ - пустой, то завершаем программу;\n"
        + "\t2) если текст - пустой, то удаляем по ключу;\n"
        + "\t3) если текст - не пустой, то добавляем его.\n"
        + "Нажми Enter - для продолжения ...\n"
        + "-----");

    // Создаем объект локального класса
    OrbPadClient opc =
        new OrbPadClient();
    opc.getKey();

    /**
     * Задаем параметры клиента по умолчанию
     */
    Properties props = System.getProperties();
    props.put( "org.omg.CORBA.ORBInitialHost",
        "localhost" );
    props.put( "org.omg.CORBA.ORBInitialPort",
        "1050" );

    try{
        // Открываем и инициализируем ORB
        ORB orb = ORB.init(args, props);

        /**
         * Получаем объектную ссылку на службу NameService,
         * которую обеспечивает ORB-сервер
         */
        org.omg.CORBA.Object objRef =
            orb.resolve_initial_references("NameService");

        /**
         * Используем NamingContextExt, который является частью
         * спецификации Interoperable Naming Service (INS)
         */
        NamingContextExt ncRef =
            NamingContextExtHelper.narrow(objRef);

        // Получаю объектную ссылку на удаленный объект
        OrbPad orbpad =
            OrbPadHelper.narrow(ncRef.resolve_str("OrbPad"));

        /**
         * Основной цикл приложения
         */
        int ns;           // Число прочитанных строк
        int nb;           // Число прочитанных байт
        String text, s;    // Строка введенного текста
        String[] ls;

        // Цикл обработки запросов
        while(true)
        {
            //Печатаем заголовок ответа
            System.out.println(
                "-----\n"
                + "Ключ\tТекст\n"
                + "-----");

```

```

s = new String(orbpad.getList());

//Выводим (построчно) результат запроса к БД
ns = 0;
ls = s.split("###");
nb = ls.length;

while(ns < nb){
    System.out.println(ls[ns] + "\n");
    ns++;
}
// Выводим итог запроса
System.out.println(
    "-----\n"
    + "Прочитано " + ls.length + " строк\n"
    + "-----\n"
    + "Формируем новый запрос!");

System.out.print("\nВведи ключ или Enter: ");
nb = opc.getKey();

if (nb == -1)
    break;    // Завершаем работу программы

System.out.print("Строка текста или Enter: ");
s = opc.getString();
text = s;

while (s.length() > 0)
{
    System.out.print("Строка текста или Enter: ");
    s = opc.getString();
    if(s.length() <= 0)
        break;
    text += ("\n" + s);
}

if (text.length() <= 0)
{
    ns = orbpad.setDelete(nb);
    if (ns == -1)
        System.out.println("\nОшибка удаления строки !!!");
    else
        System.out.println("\nУдалено " + ns
            + " строк...");
}
else
{
    ns = orbpad.setInsert(nb, text.getBytes());
    if (ns == -1)
        System.out.println("\nОшибка добавления строки !!!");
    else
        System.out.println("\nДобавлено " + ns + " строк...");
}

System.out.println("Нажми Enter ...");
opc.getKey();
}

//Закрываем все объекты и разрываем соединение
System.out.print("\nЗавершить работу сервера? (Enter - нет): ");

```

```

        s = opc.getString();

        if (s.length() > 0)
            orbpad.setClose(); // Завершаем работу сервера

        orb.destroy();
        System.out.println("Программа завершила работу...");
    }
    catch (Exception e)
    {
        System.out.println("ERROR : " + e) ;
    }
}
}

```

После написания и отладки программы в среде Eclipse EE, ее необходимо представить в виде jar-архива, например с именем **orbpadclient.jar**, и поместить в каталог **\$HOME/lib**. Запуск программы клиента осуществляется также, как и серверной программы, что показано на рисунке 3.16.

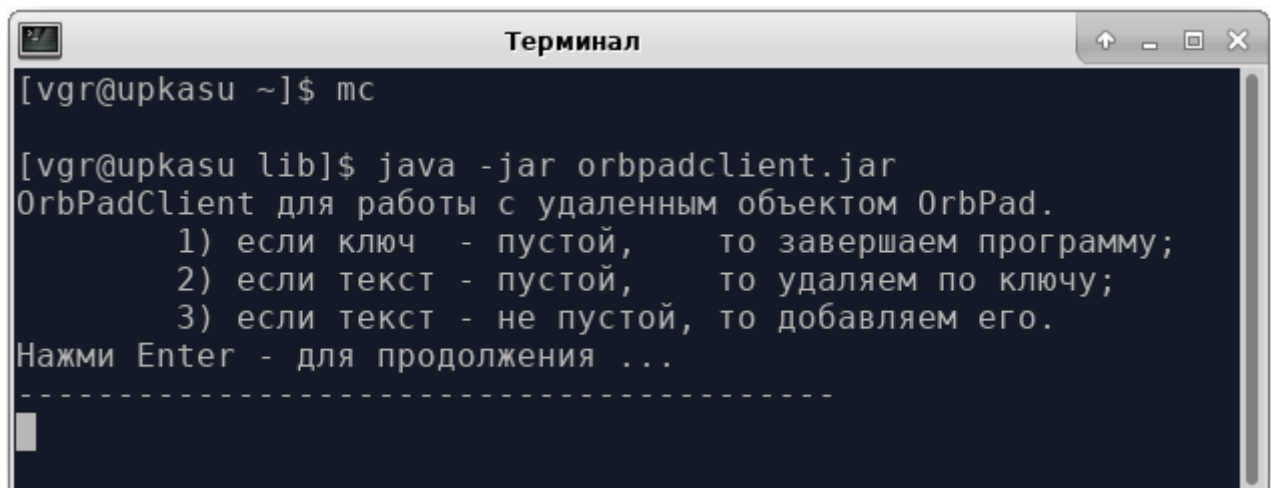


Рисунок 3.16 — Запуск программы ORB-клиента

Таким образом, в данном подразделе рассмотрен базовый набор методов технологии CORBA, позволяющий реализовывать распределенные системы, основанные на использовании брокеров объектных запросов. Хотя сама технология может показаться сложной для начинающего программиста, она обеспечивает разработчиков сложных приложений инструментами, облегчающими создание приложений клиентов на разных языках программирования. Для этого предоставляется универсальный язык описания интерфейсов (IDL) и специализированные компиляторы, которые освобождают программистов от деталей реализации сетевого взаимодействия посредством генерации необходимого набора компонент как для клиентской, так и серверной частей распределенных приложений.

3.3 Технология RMI

Можно считать, что технология **RMI** (*Remote Method Invocation*), является упрощённым (специализированным для языка Java) вариантом уже изученной технологии CORBA. Действительно, общую организацию технологии RMI можно представить рисунком 3.7, если **ORB-сервер** заменить на сервер **rmiregistry**, а для взаимодействия клиента и сервера использовать протокол **IIOP**.

Современная реализация технологии RMI, основанная на собственном протоколе **JRMP** (*Java Remote Method Protocol*), не требует генерации «стабов» (*client stub*) или «скелетонов» (*server stub, sceleton*), хотя использует своего брокера **rmiregistry** как службу «Сервиса имён». В такой интерпретации, общая организация технологии RMI может быть представлена рисунком 3.17.

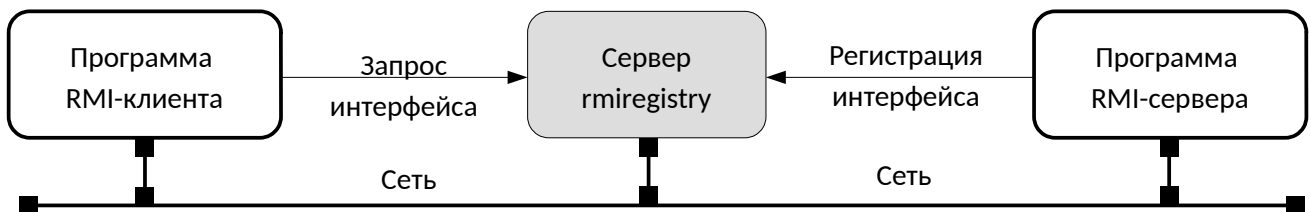


Рисунок 3.17 — Общая организация технологии RMI

Как и технология CORBA, RMI основана на описании интерфейса удалённого объекта, но отличается тем, что:

- является стандартным описанием интерфейса на языке Java;
- является расширением интерфейса **java.rmi.Remote**;
- каждый метод, включенный в интерфейс, обязан генерировать исключение **java.rmi.RemoteException**.

Центральным местом организации технологии RMI является сервер службы имён — **rmiregistry**, который находится в каталоге **\$JRE_HOME/bin/** и запускается командой:

```
$ rmiregistry [ port ]
```

где параметр **port** — не обязателен, поскольку по умолчанию используется **1099**.

Соответственно, технология проектирования распределенного приложения сводится к трём основным этапам:

1. Написание интерфейса распределенного приложения.
2. Написание *программы сервера*, реализующей функциональность удалённого объекта и обеспечивающей способность регистрации интерфейса объекта на сервере **rmiregistry**.
3. Написание *программы клиента*, реализующей функциональность локального приложения и обеспечивающей способность получения интерфейса удалённого объекта с сервера **rmiregistry**.

Поскольку теория создания объектных распределенных систем уже изучена студентами в предыдущем подразделе, на примере технологии CORBA, то особенности использования RMI рассмотрим на уже освоенном примере ведения записей в таблице *notepad* встроенного варианта БД — *exampleDB*.

3.3.1 Интерфейсы удалённых объектов

Как уже было отмечено выше, в технологии RMI интерфейсы удалённых объектов описываются классическими интерфейсами языка Java, которые должны расширять интерфейс *java.rmi.Remote*, входящий в отдельный пакет — *java.rmi*. Для конкретизации изложения вопроса рассмотрим интерфейс удалённого объекта на языке IDL, реализованного в проекте *proj10* и представленного на рисунке 3.13.

Эквивалентным по функциональному назначению будет интерфейс, реализованный в файле *RmiPad.java* проекта *proj12*. Он показан на рисунке 3.18. Хорошо видно, что описания интерфейсов очень похожи. Я специально сохранил те же комментарии, присутствующие в его прототипе.

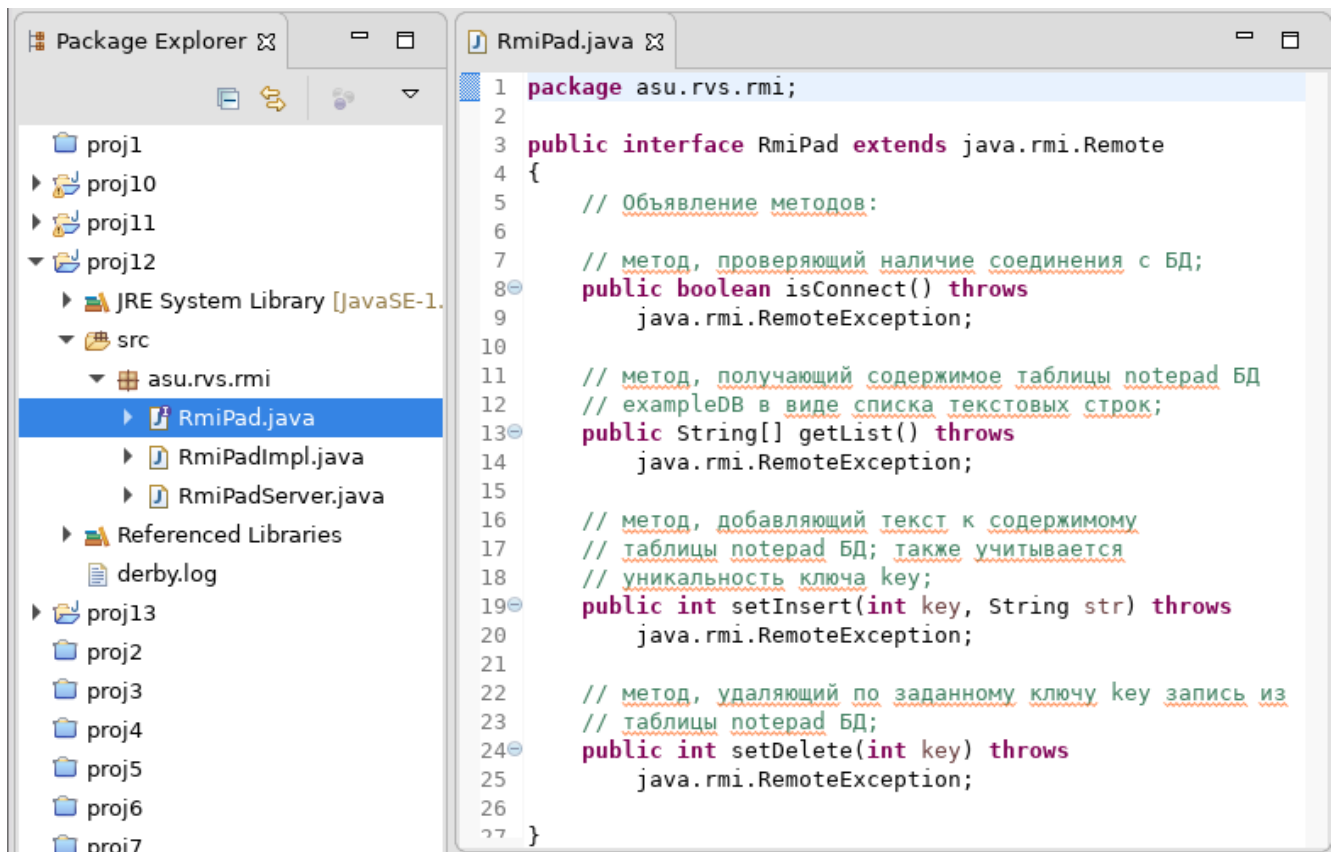


Рисунок 3.18 — Пример интерфейса удалённого объекта в технологии RMI

Что же касается практического плана, то предъявленный интерфейс имеет следующие особенности:

- удалён метод **void setClose()**, поскольку его отсутствие не уменьшает функциональности приложения, а назначение является достаточно одиозным для

удалённых объектов, разрешающих клиентским приложениям останавливать сервера;

- аргументы методов, передающих строковые значения, заменены на «родной» для языка Java тип — ***String***;
- метод ***getList()*** теперь возвращает массив строк (тип ***String[]***), что не требует при его реализации и использовании дополнительных преобразований, связанных с *упаковкой/распаковкой* содержимого.

Таким образом, интерфейс для технологии RMI — описан и готов к использованию как на стороне сервера, так и на стороне клиента.

3.3.2 Реализация RMI-сервера

Реализация отдельного RMI-сервера обычно состоит из трёх компонент на языке Java, содержащих: описание интерфейса, реализацию интерфейса и исходного текста самого сервера, обеспечивающего регистрацию интерфейса на сервере ***rmiregistry*** и принимающего запросы программ клиентов. Непосредственно для нашего примера, - это будут файлы:

- ***RmiPad.java*** — описание интерфейса, уже реализованного в проекте ***proj12***;
- ***RmiPadImpl.java*** — файл реализации интерфейса, класс которого обычно и регистрируется на сервере ***rmiregistry*** и рассматривается как удалённый объект; суффикс ***Impl*** — сокращение от английского слова ***Implementation*** (*реализация*), часто используемый в программировании как признак хорошего стиля;
- ***RmiPadServer.java*** — файл реализации самого сервера.

Поскольку интерфейс уже описан, то в проекте ***proj12*** создадим класс с именем ***RmiPadImpl***, содержимое которого представлено на листинге 3.7.

Листинг 3.7 — Исходный текст класса ***RmiPadImpl*** из среды Eclipse EE

```
package asu.rvs.rmi;

import java.rmi.RemoteException;
import asu.rvs.NotePad;

public class RmiPadImpl
    extends java.rmi.server.UnicastRemoteObject
    implements asu.rvs.rmi.RmiPad
{
    /**
     * Версия реализации
     */
    private static final long serialVersionUID = -8284829856213988639L;

    /**
     * Ссылка на класс NotePad, передаваемая экземпляру
     * класса RmiPadImpl через его конструктор.
     */
}
```

```

private NotePad obj = null;

// Конструктор
protected RmiPadImpl(NotePad obj)
    throws RemoteException
{
    super();
    this.obj = obj;
}

/**
 * Методы, реализующие объявленный интерфейс:
 *
 * метод, проверяющий наличие соединения с БД;
 * @return
 */
public boolean isConnect()
    throws java.rmi.RemoteException
{
    return obj.isConnect();
}

/**
 * метод, получающий содержимое таблицы notepad БД
 * exampleDB в виде списка текстовых строк и передающий
 * его клиенту в виде массива байт;
 * @return
 */
public String[] getList()
    throws java.rmi.RemoteException
{
    java.lang.Object[] os =
        obj.getList();
    if(os == null)
        return null;

    int ns =
        os.length;

    String[] ss = new String[ns];

    for(int i=0; i < ns; i++)
        ss[i] = os[i].toString();

    return ss;
}

/**
 * метод, добавляющий текст к содержимому
 * таблицы notepad БД, где также учитывается
 * уникальность ключа key;
 */
public int setInsert(int key, String str)
    throws java.rmi.RemoteException
{
    return obj.setInsert(key, str);
}

/**
 * метод, удаляющий по заданному ключу key запись
 * из таблицы notepad БД;
 */

```

```

    public int setDelete(int key)
        throws java.rmi.RemoteException
    {
        return obj.setDelete(key);
    }
}

```

Студенту следует обратить внимание на следующие особенности реализации класса **RmiPadImpl**, экземпляр которого собственно и является удаленным объектом:

- должен расширять класс **java.rmi.server.UnicastRemoteObject**, обеспечивающий экспорт удалённого объекта с помощью протокола **JRMP**;
- реализовать описанный интерфейс (в нашем случае — **asu.rvs.rmi.RmiPad**);
- являться публичным классом (**public**) и имеет статический номер версии (**serialVersionUID**), который генерируется и вставляется средой Eclipse EE;
- имеет один или несколько защищённых (**protected**) конструкторов (в нашем случае имеется один конструктор, передающий в класс ссылку на экземпляр класса **NotePad**, созданный компонентой сервера);
- все реализованные методы должны быть публичными (**public**) и генерировать исключение **RemoteException**.

Наконец, реализуется сам сервер, исходный текст которого представлен на листинге 3.8.

Листинг 3.8 — Исходный текст сервера RmiPadServer из среды Eclipse EE

```

package asu.rvs.rmi;

import java.rmi.Naming;
import asu.rvs.NotePad;

/**
 * Программа RmiPadServer.
 * @param args
 */
public class RmiPadServer
{
    public static void main(String[] args) {
        /**
         * Создается экземпляр класса NotePad.
         */
        NotePad obj =
            new NotePad();

        if(!obj.isConnected())
        {
            System.out.print("RmiPadServer: не могу стартовать NotePad... ");
            System.exit(1);
        }
        System.out.println("RmiPadServer: NotePad - стартовал... ");

        try {
            /**
             * Создается экземпляр удалённого объекта класса RmiPadImpl,
             * которому передаётся ссылка на экземпляр класса NotePad.

```

```

    */
    RmiPadImpl impl =
        new RmiPadImpl(obj);

    /**
     * Регистрация экземпляра удаленного объекта
     * на сервере rmiregistry.
     */
    Naming.rebind("//localhost:1099/RmiPad", (RmiPad)impl);

    System.out.println("RmiPadServer: ... ");
}
catch (Exception e)
{
    System.out.println("Исключение: " + e);
}
}
}

```

Собственно, сам сервер, во время своего запуска, выполняет три базовых действия:

1. Создает объект класса **NotePad**, который обеспечивает взаимодействие с таблицей **notepad** базы данных **exampleDB**.
2. Создает удаленный объект класса **RmiPadImpl**, передавая ему через конструктор ссылку на объект класса **NotePad**.
3. Регистрирует удаленный объект класса **RmiPadImpl** на сервере **rmiregister**, переходя в состояние прослушивания некоторого *TCP-порта* локальной машины для приема запросов от клиентских программ.

Если первые два действия — достаточно очевидны с прикладной точки зрения, то третье действие требует пояснения, поскольку оно выполняется статическим методом **rebind(...)** еще не изученного нами класса **java.rmi.Naming** пакета **java.rmi**.

Официальная документация [39] так характеризует этот класс: «Класс Naming предоставляет методы для хранения и получения ссылок на удаленные объекты в реестре удаленных объектов. Каждый метод класса Naming принимает в качестве одного из аргументов имя, которое представляет собой **java.lang.String** в формате URL (без компонента схемы) в форме:

//host:port/name

где **host** - это хост (удаленный или локальный), где находится реестр, **port** - номер порта, на который реестр принимает вызовы, а **name** - простая строка, не интерпретируемая реестром.

И **host** и **port** не являются обязательными. Если хост не указан, то по умолчанию используется локальный хост. Если порт не указан, то по умолчанию используется порт **1099**, «хорошо известный» порт, который использует реестр RMI, **rmiregistry**.

Привязка имени к удаленному объекту - это привязка или регистрация имени для удаленного объекта, который можно использовать позднее для поиска этого

удалённого объекта. Удалённый объект может быть связан с именем с помощью методов привязки или перепривязки класса *Naming*.

Как только удалённый объект зарегистрирован (привязан) в реестре RMI на локальном хосте, вызывающие абоненты на удалённом (или локальном) хосте могут искать удалённый объект по имени, получать его ссылку, а затем вызывать удалённые методы объекта. Реестр может совместно использоваться всеми серверами, работающими на хосте, или отдельный серверный процесс может создавать и использовать, если это необходимо, свой собственный реестр, (смотри метод *java.rmi.registry.LocateRegistry.createRegistry*)...».

Приведенная выдержка хорошо показывает назначение и возможности класса *Naming*. В частности, отмечается, что сервер может создавать свой реестр с помощью класса *java.rmi.registry.LocateRegistry*. При этом, используемый метод *createRegistry(...)* может проводить регистрацию отдельных серверов без участия сервера службы имен — *rmiregistry*. Мы не будем изучать методы класса *LocateRegistry*, поскольку этот подход выходит за парадигму использования единого брокера запросов. Студенты, желающие изучить этот класс, могут воспользоваться официальной документацией [40].

В целом, использование класса *Naming* не вызывает семантических трудностей, поскольку он имеет всего пять статических методов:

- *void bind(String name, Remote obj)* — регистрирует удалённый объект *obj* под именем *name*;
- *String[] list(String name)* — возвращает массив имён, связанных в реестре и представляющих собой строки в формате URL (без компонента схемы); массив содержит снимок имен, присутствующих в реестре на момент вызова;
- *Remote lookup(String name)* — возвращает ссылку на удалённый объект, связанный с указанным именем.
- *void rebind(String name, Remote obj)* — проводит перерегистрацию удалённого объекта *obj* под именем *name*;
- *void unbind(String name)* — уничтожает привязку для указанного имени, связанного с удалённым объектом.

Трудности с классом *Naming* возникают при использовании методов регистрации *bind(...)* и *rebind(...)*, когда программа запускаемого сервера выдаёт исключение *RemoteException* с сообщением, что не найдена ссылка на объект интерфейса (для нашего примера — это сообщение, показанное на рисунке 3.19).

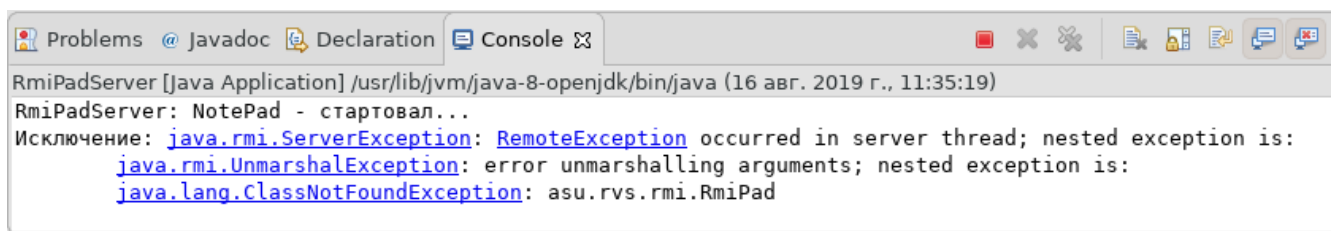


Рисунок 3.19 — Ошибочное сообщение при запуске сервера RmiPadServer

Причина состоит в том, что, во время регистрации, сервер **rmiregistry** не может найти файл: **RmiPad.class**, местоположение которого должно присутствовать в переменной среды ОС - **CLASSPATH**.

Чтобы устранить эту ошибку, следует учесть условия запуска нашего сервера **RmiPadServer**:

- программа разработана пользователем **vgr**, поэтому **\$HOME=/home/vgr**;
- программа разработана в среде Eclipse EE с базовым адресом всех проектов **\$HOME/rvs**;
- программа разработана в проекте **proj12**, поэтому адрес искомого файла, с учётом префикса **asu.rvs.rmi**, будет: **/home/vgr/rvs/proj12/bin**.

Исходя из проведённых вычислений, необходимо выполнить следующие действия:

1. Остановить сервер **rmiregistry**.
2. Добавить в файл **\$HOME/.bashrc** строку, как показано на рисунке 3.20.
3. Запустить новый виртуальный терминал, котором выполнить команду:

```
$ rmiregistry
```

4. Запустить в Eclipse EE сервер **RmiPadServer**.

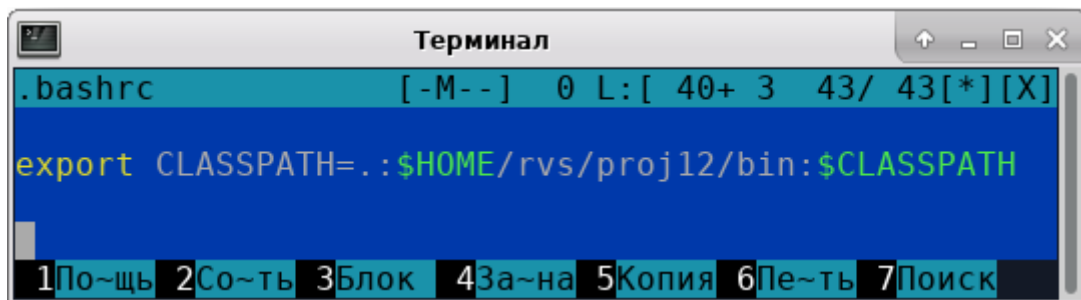


Рисунок 3.20 — Добавление пути в системную переменную CLASSPATH

В результате, запущенный сервер сообщит о нормальной регистрации на сервере имён **rmiregistry**, как показано на рисунке 3.21.

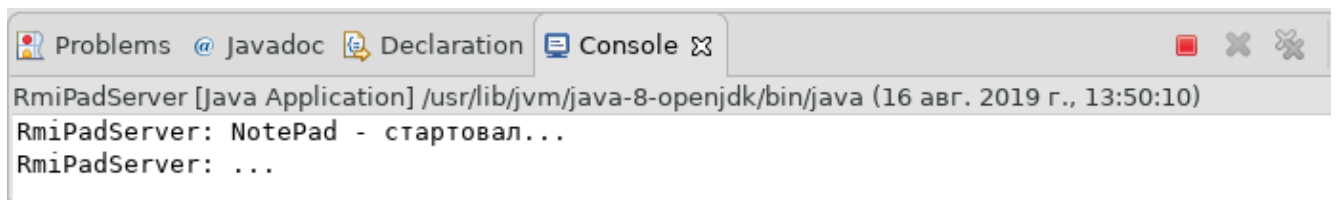


Рисунок 3.21 — Нормальный старт сервера RmiPadServer

Запущенный **RmiPadServer** будет ожидать запросы от клиентских программ по некоторому произвольному порту и внешнему адресу компьютера, которые хранятся на сервере **rmiregistry**. Клиентские RMI-программы будут обращаться к серверу имён и получать от него координаты доступа к **RmiPadServer**. Все это

скрыто внутри технологии RMI, требует знания основ администрирования сетей ЭВМ и обсуждается на лабораторных и практических занятиях.

3.3.3 Реализация RMI-клиента

Принципиально, клиентскую программу для нашего примера можно было бы реализовать в проекте сервера, но, для убедительности, мы создадим ее в новом проекте **proj13**.

Сначала, в проекте создадим интерфейс с именем **RmiPad** и перенесём в него содержимое файла **RmiPad.java** из проекта **proj12**. Это необходимо для того, чтобы клиентская программа могла работать с описанием интерфейса удалённого объекта.

Затем, в проекте **proj13** создадим класс с именем **RmiPadClient** и пакетным префиксом **asu.rmiclient**, а первоначальное содержимое его скопируем из класса **OrbPadClient** проекта **proj11**, в котором уже реализовано приложение клиента, но для технологии CORBA.

После необходимых изменений, мы получим приложение RMI-клиента, показанное на листинге 3.9.

Листинг 3.9 — Исходный текст клиента **RmiPadClient** из среды **Eclipse EE**

```
package asu.rmiclient;

import java.io.IOException;
import java.rmi.Naming;
import asu.rvs.rmi.RmiPad;

/**
 * Реализация клиента распределенного объекта RmiPad
 * @author vgr
 */
public class RmiPadClient {

    /**
     * Метод чтения целого числа со стандартного ввода
     * @return целое число или -1, если - ошибка.
     */
    public int getKey()
    {
        int ch1 = '0';
        int ch2 = '9';
        int ch;
        String s = "";

        try
        {
            while(System.in.available() == 0) ;

            while(System.in.available() > 0)
            {
                ch = System.in.read();
                if (ch == 13 || ch < ch1 || ch > ch2)
                    continue;
            }
        }
    }
}
```



```

        if (ch == 10)
            break;

        s += (char)ch;
    };
    if (s.length() <= 0)
        return -1;

    ch = new Integer(s).intValue();
    return ch;

}
catch (IOException e1)
{
    System.out.println(e1.getMessage());
    return -1;
}
}

/**
 * Метод чтения строки текста со стандартного ввода
 * @return строка текста.
 */
public String getString()
{
    String s = "\r\n";
    String text = "";
    int n;
    char ch;
    byte b[];

    try
    {
        //Ожидаем поток ввода
        while(System.in.available() == 0) ;
        s = "";
        while((n = System.in.available()) > 0)
        {
            b = new byte[n];
            System.in.read(b);
            s += new String(b);
        };
        // Удаляем последние символы '\n' и '\r'
        n = s.length();
        while (n > 0)
        {
            ch = s.charAt(n-1);
            if (ch == '\n' || ch == '\r')
                n--;
            else
                break;
        }
        // Выделяем подстроку
        if (n > 0)
            text = s.substring(0, n);
        else
            text = "";
        return text;
    }
    catch (IOException e1)
    {

```

```

System.out.println(e1.getMessage());
return "Ошибка...";
}
}

public static void main(String[] args)
{
    System.out.println(
        "RmiPadClient для работы с удаленным объектом RmiPad.\n"
        + "\t1) если ключ - пустой, то завершаем программу;\n"
        + "\t2) если текст - пустой, то удаляем по ключу;\n"
        + "\t3) если текст - не пустой, то добавляем его.\n"
        + "Нажми Enter - для продолжения ...\n"
        + "-----");

    // Создаем объект локального класса
    RmiPadClient rpc =
        new RmiPadClient();
    rpc.getKey();

    try{
        // Получаю и печатаю список всех регистраций.
        String[] sss =
            Naming.list("//localhost:1099/RmiPad");
        for(int i=0; i<sss.length; i++)
            System.out.println(sss[i]);

        // Получаю объектную ссылку на удаленный объект
        RmiPad rmipad =
            (RmiPad)Naming.lookup("//localhost:1099/RmiPad");

        /**
         * Основной цикл приложения
         */
        int ns;          // Число прочитанных строк
        int nb;          // Число прочитанных байт
        String text, s;  // Строка введенного текста
        String[] ls;

        // Цикл обработки запросов
        while(true)
        {
            //Печатаем заголовок ответа
            System.out.println(
                "-----\n"
                + "Ключ\tТекст\n"
                + "-----");
            ls = rmipad.getList();

            //Выводим (построчно) результат запроса к БД
            ns = 0;
            nb = ls.length;

            while(ns < nb){
                System.out.println(ls[ns] + "\n");
                ns++;
            }
            // Выводим итог запроса
            System.out.println(
                "-----\n"
                + "Прочитано " + ls.length + " строк\n");
        }
    }
}

```

```

+ "-----\n"
+ "Формируем новый запрос!");

System.out.print("\nВведи  ключ  или Enter: ");
nb = rpc.getKey();

if (nb == -1)
    break;    // Завершаем работу программы

System.out.print("Строка текста или Enter: ");
s = rpc.getString();
text = s;

while (s.length() > 0)
{
    System.out.print("Строка текста или Enter: ");
    s = rpc.getString();
    if(s.length() <= 0)
        break;
    text += ("\n" + s);
}

if (text.length() <= 0)
{
    ns = rmipad.setDelete(nb);
    if (ns == -1)
        System.out.println("\nОшибка удаления строки !!!");
    else
        System.out.println("\nУдалено " + ns
            + " строк...");
}
else
{
    ns = rmipad.setInsert(nb, text);
    if (ns == -1)
        System.out.println("\nОшибка добавления строки !!!");
    else
        System.out.println("\nДобавлено " + ns + " строк...");
}

    System.out.println("Нажми Enter ...");
    rpc.getKey();
}

System.out.println("Программа завершила работу...");
}
catch (Exception e)
{
    System.out.println("ERROR : " + e) ;
}
}
}

```

Основное отличие приведённого листинга от текста ORB-клиента (см. листинг 3.6) заключается в том, что RMI-клиент подключает интерфейс удалённого объекта всего лишь одним методом **lookup()** класса **Naming**:

```
// Получаю объектную ссылку на удалённый объект
```

```
RmiPad rmipad =
    (RmiPad)Naming.lookup("//localhost:1099/RmiPad");
```

все остальные изменения связаны только с различными типами разных технологий и приложений.

Запустив на выполнение программу RMI-клиента, можно убедиться в ее работоспособности, как показано на рисунке 3.22.

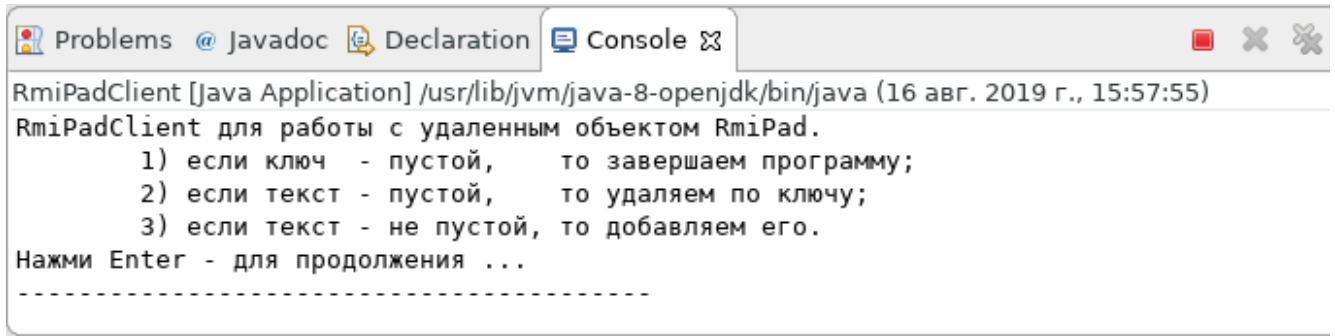


Рисунок 3.22 — Начало диалога приложения RmiPadClient

3.3.4 Завершение реализации RMI-проекта

В завершении данного подраздела необходимо реализовать и протестировать наше распределенное приложение в виде двух Jar-архивов: **rmipadserver.jar** и **rmipadclient.jar**. Для этого, предварительно нужно:

- в среде разработки Eclipse EE: остановить работу RMI-клиента, а затем — RMI-сервера;
- в виртуальном терминале остановить работу сервера **rmiregistry**.

Сначала, проведём архивацию и тестирование серверной части RMI-проекта. Для этого, выделим в Eclipse EE проект с именем **proj12** и проведём создание архива, как это уже было описано в пункте 3.2.2 данной главы. Сам архив сохраним в файле с абсолютным путём доступа: **\$HOME/lib/rmipadserver.jar**.

Теперь отредактируем переменную среды CLASSPATH, как это показано на рисунке 3.23, чтобы **rmiregistry** мог найти архив сервера.

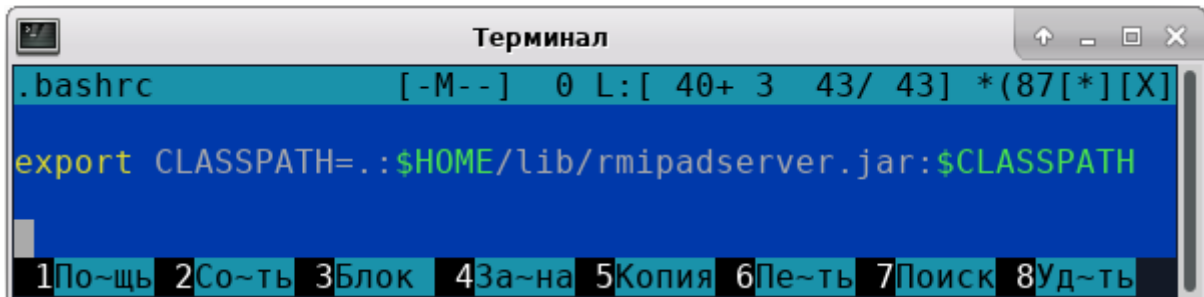


Рисунок 3.23 — Редактирование .bashrc для архива RMI-сервера

Сохранив изменения в файле **.bashrc**, запустим новый виртуальный терминал, в котором стартуем сервер **rmiregistry**. Затем, перейдя в каталог **\$HOME/lib/**, запустим сервер как показано на рисунке 3.24.

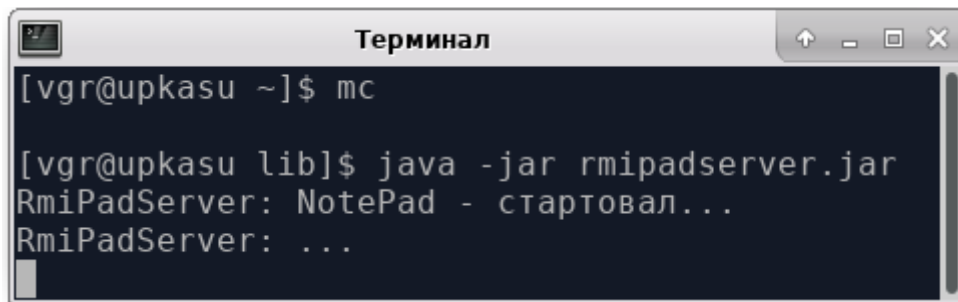


Рисунок 3.24 — Запуск RMI-сервера из архива **rmipadserver.jar**

Хорошо видно, что сервер стартовал — нормально, но здесь надо отметить, что если бы мы не внесли изменения в файл **.bashrc**, то **rmiregistry** регистрировал бы приложение, размещённое в проекте **proj12** среды Eclipse EE.

Запустив нормально RMI-сервер, следует провести:

- тестирование RMI-клиента в среде Eclipse EE;
- создание архива проекта **proj13** и размещение его в каталоге **\$HOME/lib/**;
- перейти в каталог **\$HOME/lib/** и запустить RMI-клиента, как это показано на рисунке 3.25.

Тестирование распределенного приложения можно продолжить, запуская приложение RMI-клиента на множестве виртуальных терминалов командами:

```
java -jar rmipadclient.jar
```

каждое отдельно запущенное приложение клиента будет работать самостоятельно, но с единой базой данных.

На этом, мы завершаем изучение технологии RMI и всей темы, посвящённой объектным распределённым системам. На рисунке 3.26 представлен список и размеры всех созданных в данной главе Jar-архивов приложений. В целом, они посвящены решению одной задачи: ведению записей в таблице **notepad** базы данных **exampleDB**, поддерживаемой встроенным вариантом СУБД Apache Derby.

Общая методика их реализации отражает общие правила проектирования:

- первоначально создаётся локальный вариант решения задачи, предполагающий будущее разделение на клиентскую и серверную части; такие части реализованы в архивах **notepad.jar** и **example12.jar**;
- решение этой задачи средствами технологии CORBA представлена архивами **orbpadserver.jar** и **orbpadclient.jar**;
- решение этой задачи средствами технологии RMI представлена архивами **rmipadserver.jar** и **rmipadclient.jar**.

```

[vg@upkasu lib]$ java -jar rmipadclient.jar
RmiPadClient для работы с удаленным объектом RmiPad.
    1) если ключ - пустой,      то завершаем программу;
    2) если текст - пустой,     то удаляем по ключу;
    3) если текст - не пустой,  то добавляем его.
Нажми Enter - для продолжения ...
-----
//localhost:1099/RmiPad
-----
Ключ      Текст
-----
124      Разработал программу для
технологии RMI.

125      Исправил исходные тексты.

126      Стартовал RMI-сервер
из архива ~/lib/rmipadserver.jar.

-----
Прочитано 3 строк
-----
Формируем новый запрос!

Введи  ключ   или Enter: 

```

Рисунок 3.25 — Запуск RMI-клиента из архива rmipadclient.jar

Имя	Размер	Время правки
./..	-ВВЕРХ-	авг 17 08:47
example12.jar	3944982	авг 10 11:46
notepad.jar	3941998	авг 10 11:32
orbpadclient.jar	13222	авг 13 15:01
orbpadserver.jar	3954577	авг 12 22:41
rmipadclient.jar	3747	авг 16 19:00
rmipadserver.jar	3944420	авг 16 18:22

Рисунок 3.26 — Сравнительные размеры созданных Jar-архивов приложений

Вопросы для самопроверки

1. Что такое — DCE?
2. Что такое — CORBA?
3. Что такое — RMI?
4. Что такое — RPC?
5. Что такое — GIOP?
6. Что такое — IIOP?
7. Что такое — брокер и в чем суть брокерной архитектуры?
8. Чем брокерная архитектура отличается от модели «Клиент-сервер»?
9. Что такое — middleware?
10. Что такое — прокси-сервер?
11. Что такое — IDL?
12. Что такое — объект времени компиляции?
13. Для чего в технологии CORBA используется адаптер объектов?
14. Для чего в JRE языка Java используется утилита idlj?
15. Для чего в JRE языка Java используется программа orbd?
16. Для чего в языке Java используется соответствие типов языку IDL?
17. Каким образом в языке Java создаётся ORB-объект и зачем он нужен?
18. Чем отличается технология RMI от технологии CORBA?
19. Какой специальный сервер используется в технологии RMI?
20. Какой класс технологии RMI используется как для регистрации серверной программы, так и для доступа к удалённому объекту в клиентской программе?

4 Тема 4. Web-технологии распределенных систем

Данная глава посвящена изучению технологий **web**, зародившихся как частная задача интеграции научного информационного документооборота сотрудников «Европейского совета по ядерным исследованиям» (CERN), но со временем превратившихся в глобальный гипертекстовый проект, известный как «Всемирная паутина». Согласно Википедии [41]: «**Всемирная паутина** (англ. *World Wide Web*) — распределенная система, предоставляющая доступ к связанным между собой документам, расположенным на различных компьютерах, подключённых к сети Интернет. Для обозначения Всемирной паутины также используют слово **веб** (англ. *web* «паутина») и аббревиатуру **WWW**. Всемирную паутину образуют сотни миллионов веб-серверов. Большинство ресурсов Всемирной паутины основано на технологии гипертекста. Гипертекстовые документы, размещаемые во Всемирной паутине, называются веб-страницами. Несколько веб-страниц, объединённых общей темой или дизайном, а также связанных между собой ссылками и обычно находящихся на одном и том же веб-сервере, называются веб-сайтом. Для загрузки и просмотра веб-страниц используются специальные программы — браузеры (англ. *browser*). Всемирная паутина вызвала настоящую революцию в информационных технологиях и дала мощный толчок развитию Интернета. ...».

Первоначально, **web** не создавался для реализации широкого класса распределённых систем, но он предоставил три технологических новинки:

1. **URI** (*Uniform Resource Identifier*) — унифицированный идентификатор ресурса.
2. **HTML** (*HyperText Markup Language*) — гипертекстовый язык разметки.
3. **Браузер** (*web browser*) — прикладная программа для просмотра содержимого HTML-файлов, включающих файлы рисунков.

Существуют различные мнения, но на мой взгляд — именно браузеры, которые стали распространяться с каждой ОС, сделали общедоступным язык HTML, что привлекло к участию в развитии web-технологий широкий круг специалистов различных областей, сделало **web** популярным и экспериментальной площадкой для новых технологических решений в Интернет. Появился даже термин «Тонкий клиент», обозначающий браузер — как «Автоматизированное рабочее место» (АРМ). Что касается нашей предметной области, то наиболее важной является идея URI, которая на практике показала свою состоятельность и обратила внимание теоретиков распределенных систем на проблему адресации большого количества ресурсов.

Сам учебный материал данной главы разделен на три части:

- первая часть — общее описание технологии web;
- вторая часть — развитие модели «Клиент-сервер»;
- третья часть — реализация web-технологий с использованием языка Java.

4.1 Общее описание технологии web

Считается, что Тимоти Джон Бернерс-Ли придумал web в 1990 году. Им были изобретены идентификаторы URI, язык HTML и протокол HTTP. В период с 1991 по 1993 годы он усовершенствовал эти стандарты и опубликовал их. Кроме того, им впервые в мире были написаны:

- web-сервер, названный «*httpd*»;
- web-браузер, названный «*WorldWideWeb*».

В концептуальном плане Бернерс-Ли ввёл понятие *ресурса*, которым может быть что угодно: произвольные рисунки, записи, чертежи и все остальное, что может быть идентифицировано и передано с одного компьютера на другой. В такой проекции, распределенная система рассматривается как некий «механизм» по перемещению ресурсов с одной машины на другую. Дополнительно, Бернерс-Ли показал, что такой «механизм» может быть реализован и для этого нужны три простые технологии:

1. *Адресация ресурсов*. Необходимы гибкие и расширяемые способы именования произвольных ресурсов, например, URI.
2. *Представление ресурсов*. Необходимо такое представление ресурсов, чтобы они могли быть представлены в виде потока бит и переданы по сети. Кроме того, такое представление должно быть понятно всем и принято всеми, например, HTML.
3. *Передача ресурсов*. Необходим протокол передачи по типу «Клиент-сервер», который поддерживает минимальный необходимый набор операций передачи данных, например, HTTP.

В такой последовательности мы и рассмотрим технологии web, понимая, что в конце 80-х годов единственным претендентом на глобальное применение был лишь стек протоколов TCP/IP.

4.1.1 Унифицированный идентификатор ресурсов (URI)

Официально, адресация ресурсов web осуществляется с помощью **URI** [42]: «URI — символьная строка, позволяющая идентифицировать какой-либо ресурс: документ, изображение, файл, службу, ящик электронной почты и т. д. Прежде всего, речь идёт о ресурсах сети Интернет и Всемирной паутины. URI предоставляет простой и расширяемый способ идентификации ресурсов. Расширяемость URI означает, что уже существуют несколько схем идентификации внутри URI, и ещё больше будет создано в будущем. ...».

URI имеет две формы представления:

1. **URL** — определяет, где и как найти ресурс. В 2002 году (см. RFC 3305) анонсировано устаревание термина URL.
2. **URN** — определяет, как ресурс идентифицировать.

Согласно [43]: «**Единый указатель ресурса** (от англ. *Uniform Resource Locator* — унифицированный указатель ресурса, **URL** — система унифицированных адресов электронных ресурсов, или единообразный определитель местонахождения ресурса (файла). Используется как стандарт записи ссылок на объекты в Интернет (Гипертекстовые ссылки во «всемирной паутине» *www*). ...».

Именно такие ссылки мы обычно используем в работе с Интернет, а также будем использовать в учебной работе, поэтому рассмотрим формат URL(URI) более подробно:

<схема>:[//[<логин>[:<пароль>]@]<хост>[:<порт>]]/[<URL-путь>][?<параметры>][#<якорь>]

где, в этой записи:

- **квадратные скобки** - необязательные конструкции;
- <схема> - схема обращения к ресурсу, обычно обозначает протокол: *http*, *https*, *ftp* и другие;
- <логин> - имя пользователя, используемое для доступа к ресурсу;
- <пароль> - пароль пользователя;
- <хост> - доменное имя компьютера или его цифровой IP-адрес;
- <порт> - номер порта транспортного уровня;
- <URL-путь> - путь к ресурсу относительно корневого каталога, определённого программным обеспечением сервера;
- <параметры> - строка запроса к серверу, допустимая в методе **GET** протокола HTTP (HTTPS); параметры передаются парами **имя=значение** и разделяются символами «&»;
- <якорь> - элемент языка HTML, дающий команду браузеру переместиться к нужной части принятой страницы.

Недостатки адресации URL — хорошо известны:

- использование ограниченного набора ASCII-символов, делающего нечитаемыми слова национальных языков;
- сильная привязка к стеку протоколов TCP/IP, требующая указания адреса сети и порта транспортного соединения;
- наличие привязок к тексту HTML и методам протокола HTTP.

В целом, адресация URL имеет достаточно простую семантику, что обеспечило ей большую популярность, хотя сам Бернерс-Ли считает ее излишней, поскольку раскрывает структуру распределенных систем.

Другой подход, связанный с идентификацией ресурсов, имеет следующие характеристики [44]: «**URN** (англ. *Uniform Resource Name*) — единообразное название (имя) ресурса. ... URN — это *постоянная* последовательность символов, идентифицирующая абстрактный или физический ресурс. URN является частью концепции **URI** (англ. *Uniform Resource Identifier*) — единообразных идентификаторов ресурса. Имена URN призваны в будущем заменить локаторы **URL** (англ. *Uniform Resource Locator*) — единообразные определители местонахождения ре-

сурсов. Но имена URN, в отличие от URL, не включают в себя указания на местонахождение и способ обращения к ресурсу. Стандарт URN специально разработан так, чтобы он мог включать в себя другие пространства имён. ...».

Общий формат URN:

urn:<NID>:<NSS>

В этой записи:

- <NID> - нечувствительный к регистру идентификатор пространства имён (*Namespace Identifier*), представляющий собой статическую интерпретацию NSS;
- <NSS> - строка некоторого определённого пространства имён (*Namespace Specific String*), состоящая, как и URL, из ограниченного набора ASCII-символов.

Мы не будем подробно рассматривать адресацию URN, отправляя желающих узнать подробности к источникам RFC 3401 - RFC 3406.

4.1.2 Общее представление ресурсов (HTML)

Язык HTML безусловно является символом Интернет [45]: «HTML (от англ. *HyperText Markup Language* — «язык гипертекстовой разметки») — стандартизированный язык разметки документов во Всемирной паутине. Большинство веб-страниц содержат описание разметки на языке HTML (или XHTML). Язык HTML интерпретируется браузерами; полученный в результате интерпретации форматированный текст отображается на экране монитора компьютера или мобильного устройства. Язык HTML до 5-й версии определялся как приложение SGML (стандартного обобщённого языка разметки по стандарту ISO 8879). Спецификации HTML5 формулируются в терминах DOM (объектной модели документа). Язык XHTML является более строгим вариантом HTML, он следует синтаксису XML и является приложением языка XML в области разметки гипертекста. ...».

Язык HTML — достаточно прост в изучении. Имеет множество учебников, описывающих его. Базовые принципы этого языка можно изучить даже в Википедии [46]. Нас он интересует прежде всего как средство представления ресурсов распределённых систем. В этом плане, он первоначально обеспечивал:

- предоставление ссылок (адресацию) на различные ресурсы Интернет;
- форматирование текста, включая представление списков и таблиц;
- отображение рисунков в формате *.gif*;
- интерактивные средства взаимодействия клиента с удалённым приложением сервера с помощью конструкции <FORM>, включающей поля ввода текста, радиокнопок, кнопок отправки формы по методам GET и POST протокола HTTP, а также кнопки очистки формы.

В целом, язык HTML содержит мало средств, ориентированных на описание самих ресурсов распределенных приложений. К тому же он только даёт описание разметки, а само представление обеспечивает приложение — браузер. Поэтому разработчики браузеров стали включать в HTML различные расширения:

- вставку различных объектов: не-HTML документов и *media*-файлов;
 - вставку различных языков сценариев, например, *JavaScript* [47].
-

Существенным событием web-технологий стала возможность включения в текст языка HTML объектов языка Java, известных как *апплеты* [48]: «**Java-апплет** — прикладная программа, чаще всего написанная на языке программирования Java в форме байт-кода. Java-апплеты выполняются в веб-обозревателе с использованием виртуальной Java машины (JVM), или в Sun's AppletViewer, автономном средстве для испытания апплетов. Java-апплеты были внедрены в первой версии языка Java в 1995 году. Java-апплеты обычно пишутся на языке программирования Java, но могут быть написаны и на других языках, которые компилируются в байт-код Java, таких, как Jython. Поддержка апплетов исключена из Java, начиная с версии 11. Апплеты используются для предоставления интерактивных возможностей веб-приложений, которые не могут быть предоставлены HTML. Так как байт-код Java платформо-независим, то Java-апплеты могут выполняться с помощью плагинов браузерами многих платформ, включая Microsoft Windows, UNIX, Apple Mac OS и GNU/Linux. ...».

В настоящее время, наиболее популярной является технология AJAX [49]: «**AJAX**, Ajax (от англ. *Asynchronous Javascript and XML* — «асинхронный JavaScript и XML») — подход к построению интерактивных пользовательских интерфейсов веб-приложений, заключающийся в «фоновом» обмене данными браузера с веб-сервером. В результате, при обновлении данных веб-страница не перезагружается полностью, и веб-приложения становятся быстрее и удобнее. ...».

Серьёзные претензии к HTML возникли в процессе создания проекта электронного документа. Были предложения полностью перейти на язык XML [50], но пока этого не случилось и язык HTML остаётся полноправным членом всех существующих web-технологий.

4.1.3 Протокол передачи гипертекста (HTTP)

Не менее значимым чем HTML в web-технологиях является протокол HTTP [51]: «**HTTP** (англ. *HyperText Transfer Protocol* — «протокол передачи гипертекста») — протокол прикладного уровня передачи данных изначально — в виде гипертекстовых документов в формате «HTML», в настоящий момент используется для передачи произвольных данных. Основой HTTP является технология «клиент-сервер», то есть предполагается существование:

- Потребителей (клиентов), которые иницируют соединение и посылают запрос;

- Поставщиков (серверов), которые ожидают соединения для получения запроса, производят необходимые действия и возвращают обратно сообщение с результатом. ...

HTTP используется также в качестве «транспорта» для других протоколов прикладного уровня, таких как SOAP, XML-RPC, WebDAV. Основным объектом манипуляции в HTTP является ресурс, на который указывает URI (Uniform Resource Identifier) в запросе клиента. Обычно такими ресурсами являются хранящиеся на сервере файлы, но ими могут быть логические объекты или что-то абстрактное. Особенностью протокола HTTP является возможность указать в запросе и ответе способ представления одного и того же ресурса по различным параметрам: формату, кодировке, языку и т. д. (в частности, для этого используется HTTP-заголовок). Именно благодаря возможности указания способа кодирования сообщения, клиент и сервер могут обмениваться двоичными данными, хотя данный протокол является текстовым. ...».

Широкую популярность в приложениях протокол HTTP получил благодаря опоре на стек протоколов транспортного уровня TCP/IP. Это делает его универсальным способом прикладного взаимодействия в Интернет. Кроме того, благодаря текстовому характеру передаваемых сообщений, он легко адаптируется, подвергается анализу и отладке во время проектирования распределенного взаимодействия, а также включает в себя достаточно большой набор стандартизированных методов:

- OPTIONS — метод, используемый для получения информации о поддерживаемых расширениях сервера;
- GET — основной метод запроса ресурсов в браузерах;
- POST — дополнительный метод запроса ресурса, используемый в браузерах с помощью специальных конструкций <FORM>;
- HEAD — метод, по форме запроса аналогичный методу GET, но ответ сервера не содержит тела ресурса;
- PUT — аналогичен методу POST, но содержит более детальный диалог;
- PATCH - аналогичен методу PUT, но применяется к фрагменту ресурса;
- DELETE — предназначен для удаления ресурсов;
- TRACE — обеспечивает трассировку запросов;
- CONNECT — использует соединение с сервером как TCP/IP-туннель.

Таким образом, идея Бернерса-Ли о трёх базовых ресурсах Всемирной паутины получила не только простейшую реализацию, применимую для научного информационного документооборота, но и стала развиваться в других приложениях, которые, в свою очередь, стимулировали ее развитие и все большую популярность. Такая ситуация очень быстро привела к необходимости дополнительных исследований, связанных со все более возрастающим объёмом обрабатываемой информации и источников ее размещения. Это потребовало более подробного изучения базовой модели сетевого взаимодействия модели «Клиент-сервер». Рассмотрим этот вопрос более подробно.

4.2 Модели «Клиент-сервер»

Во всем предшествующем учебном материале, начиная с первой главы (см. пункт 1.2.1), присутствует базовая сетевая парадигма «Клиент-сервер» [14]. Она присутствует как в концепции самого сетевого взаимодействия модели OSI [13], так и в базовом программном обеспечении (см. подраздел 2.5, «Управление сетевыми соединениями»). Далее ее классическая интерпретация рассматривается в подразделе 2.6 как «Организация доступа к базам данных», и, наконец, вся тема 3 полностью посвящена брокерным архитектурам, относящихся к группе «Объектных распределенных систем».

Фактически, с момента своего появления словосочетание «Клиент-сервер» превратилось в некий зонтичный термин, в котором, по определению, присутствуют две части:

1. **Клиент** — активная часть, иницилирующая сетевое взаимодействие и являющаяся конечным потребителем результатов вычислений.
2. **Сервер** — пассивная часть, обслуживающая одного или множество клиентов.

Простое масштабирование данной парадигмы, как модели построения распределённых вычислительных сетей (РВ-сетей), потребовало введение промежуточного программного обеспечения (*Middleware*), которое для объектного подхода было реализовано в виде ПО брокеров, регистрирующих наличие и местоположение серверов. Сами брокеры не участвовали в конечном прикладном взаимодействии клиента и сервера. Они обеспечивали только начало такого взаимодействия, реализуя «Службу имён». Широкое распространение web-технологий актуализировало другой аспект сетевого взаимодействия, предложив использовать парадигму «Тонкий клиент» взамен парадигмы «Автоматизированное рабочее место» (АРМ).

Сразу заметим, что идея *тонкого клиента*, под которым понимается программное обеспечение браузеров, является отражением более общей проблемы, с которой сталкиваются все проектировщики РВ-сетей. Она выражается в виде простой дилеммы о распределении функциональной нагрузки между ПО клиента и ПО сервера. В таком виде, использование тонкого клиента напоминает концепцию виртуальной машины Java (JVM), реализуемой для каждой платформы ЭВМ, только на более высоком (прикладном) уровне. Но прямое применение этой идеи позитивно лишь до тех пор, пока в силу не вступают вопросы надёжности, безопасности и другие технические или юридические ограничения.

В общем случае, к приложениям можно применить два общих типа распределения: *вертикальные* и *горизонтальные*. В следующих пунктах мы рассмотрим:

- распределение приложений по уровням, отдельно выделяющим и отражающим концепцию тонкого клиента;
- выделение ряда общих типов клиент-серверной архитектуры, включающих вертикальные и горизонтальные распределения.

4.2.1 Распределение приложений по уровням

Парадигма «Тонкий клиент», ставшая основой публичного использования web-технологий, породила много дебатов и споров. В результате, сторонники бурно развивающихся в 90-е годы информационных технологий предложили трёх-уровневую архитектуру построения РВ-сетей. Она включала:

- *верхний уровень представления* (пользовательского интерфейса);
- *средний уровень бизнес-логики* (функциональная поддержка приложений);
- *нижний уровень данных* (накопление, хранение и извлечение данных).

В чем-то такая архитектура напоминала идеи сетевых моделей OSI и DoD (*Department of Defense*), перенесённых в понятийное пространство прикладных систем. Рассмотрим эту архитектуру более подробно.

Уровень представления — ПО поддержки пользовательского интерфейса, расположенное на машине клиента. Оно может варьироваться от простейших терминальных систем, обеспечивающих только символьный доступ к ПО сервера, до сложных мультимедиа систем, обеспечивающих все возможности графики, звука и других интерактивных технологий. Появился даже новый термин: «*Богатый клиент*».

Уровень бизнес-логики – это совокупность правил, принципов и зависимостей поведения объектов предметной области системы. Фактически, - это и есть функциональная часть приложения, реализованная на сервере, но не включающая хранилище используемых данных, которое реализовано как отдельная система.

Уровень данных — это сервер или набор серверов, содержащих программы, которые хранят и предоставляют данные программам уровня бизнес-логики. Фактически, — это сервера, содержащие базы данных управляемые некоторыми СУБД.

Хотя такая архитектура может показаться устаревшей и напоминающей эпоху использования мейнфреймов, она имеет серьёзные позитивные обоснования:

- тенденция создания *сложных масштабных приложений*, работоспособность которых может быть обеспечена только организациями, содержащими высококлассных специалистов;
- тенденция создания *важных приложений*, которые юридически могут обслуживаться только специализированными организациями;
- тенденция *мобильности и низкого уровня подготовки клиентов*, которые требуют централизованного и профессионального обслуживания.

Сложно прогнозировать, будет ли указанная архитектура преобладающей в будущих РВ-сетях, поскольку все современные тенденции негативно относятся к централизованным решениям, но ее простота и наглядность вполне применимы к небольшим проектам уровня предприятия.

4.2.2 Типы клиент-серверной архитектуры

В предыдущем пункте был рассмотрен *вертикальный тип* распределения приложений, который называется «*Трёхзвенной архитектурой*». В целом, можно выделить три варианта таких архитектур.

Однозвенная архитектура — модель, по функциональности эквивалентная терминальному доступу к мейнфреймам. В простейшем виде она соответствует системам телеобработки данных, ранее показанным на рисунке 1.3. В более сложных случаях, это могут быть, например, графические X-станции, которые в своё время выпускала компания Sun Microsystems. Такие станции представляли собой компьютер со встроенным сетевым программным обеспечением. После включения, такая станция соединялась с сервером, загружала программу **login** и, после успешной авторизации, загружала программное обеспечение графического X-сервера, который обеспечивал доступ к размещённой на самом сервере рабочей области пользователя.

Двухзвенная архитектура — модель, демонстрирующая функциональность АРМ или «*толстого клиента*». Она показана на рисунке 4.1.

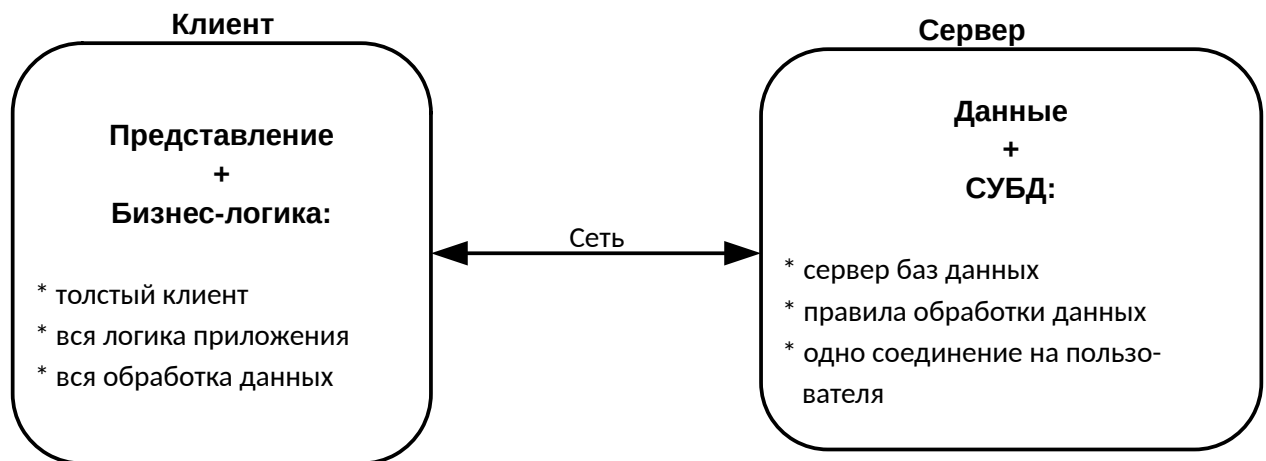


Рисунок 4.1 — Двухзвенная архитектура «Клиент-сервер»

Примером такой архитектуры является приложение **Example11**, реализованное в пункте 2.6.3 главы 2, а также подобные реализации в вариантах технологий CORBA и RMI. Тот факт, что в этих примерах использовался вариант встроенной СУБД Apache Derby не играет никакой роли, поскольку сама технология доступа к СУБД предполагает соединение с сервером баз данных.

Трёхзвенная архитектура - модель, демонстрирующая функциональность «*тонкого клиента*». Ее архитектура представлена на рисунке 4.2, а особенности функционирования — следующие:

- **Клиент** обеспечивает только представление данных и диалог с сервером приложений;
- **Сервер приложений** несет всю нагрузку обработки данных, определённую бизнес-логикой приложения.



Рисунок 4.2 — Трёхзвенная архитектура «Клиент-сервер»

Общей характерной особенностью вертикального типа распределения приложений является *размещение на разных машинах логически разных компонент приложения*.

Альтернативой вертикальному распределению является *горизонтальное распределение*, когда логически одинаковые компоненты клиентов и серверов размещаются на разных машинах. Схематически, такой вариант показан на рисунке 4.3, где центральный элемент — «Сервер распределения» - распределяет нагрузку запросов от множества клиентов по множеству серверов.

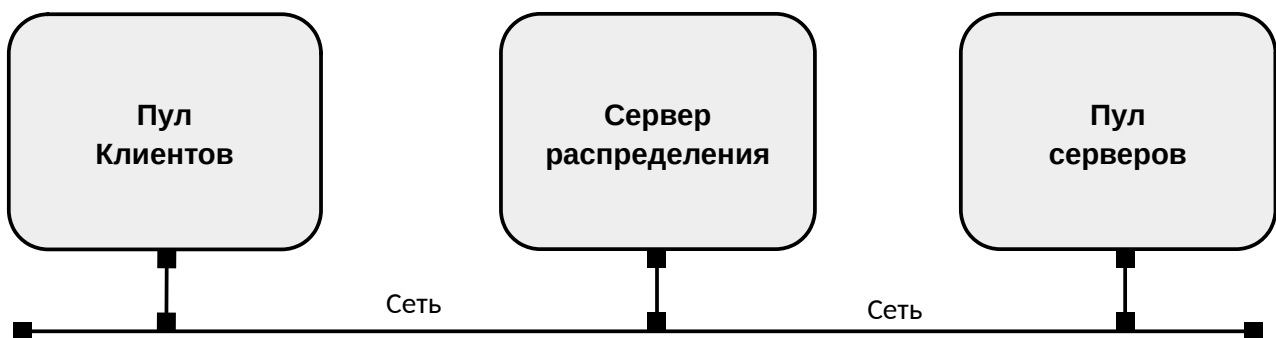


Рисунок 4.3 — Горизонтальное распределение архитектуры «Клиент-сервер»

В учебных условиях достаточно сложно реализовать горизонтальную архитектуру распределения компонент приложения, поэтому мы ее больше не будем рассматривать. Главное, что необходимо отметить, — горизонтальное распределение обычно создаётся *методами репликации*, когда логически одинаковые компоненты приложений постоянно синхронизируются между собой. Другими словами, если какой-то логический компонент изменил своё значение, то аналогичные компоненты изменяют своё значение на всех машинах сети.

4.3 Технология Java-сервлетов

Первоначально, *www-технологии* зародились из желания иметь публикации, которые по качеству представления были бы сравнимы с книгами, газетами и другими традиционными документами, но могли бы находиться на компьютерах и свободно распространяться по сети Интернет. Для этой цели были разработаны:

- язык *HTML*, для упрощённой разметки электронного документа;
- *browser*, способный обеспечить разметку и просмотр электронного документа, написанного на языке HTML;
- технология *CGI* (*Common Gateway Interface*), описывающая общие требования к web-серверу, способному хранить или формировать страницы HTML;
- протокол *HTTP*, обеспечивающий взаимодействие браузера и web-сервера по передаче запросов на получение документов.

При таком подходе, простейшие публикации, предоставляющие статический текст на языке HTML, могли формироваться любой программой или в любом текстовом редакторе. Пример такого простейшего текста показан на листинге 4.1.

Листинг 4.1 — Исходный текст простейшей HTML-страницы

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Проект proj14</title>
</head>
<body>
  <h2 align="center">
    РАСПРЕДЕЛЕННЫЕ ВЫЧИСЛИТЕЛЬНЫЕ СЕТИ
  </h2>
  <h4 align="center">
    бакалавриат кафедры АСУ
  </h4>
  <hr> Адрес этой страницы:
  <a href="http://localhost:8080/proj14/Title.html">
    http://localhost:8080/proj14/Title.html
  </a>
  <hr>
  <a href="http://localhost:8080/proj14/Example14">
    Запуск сервлета Example14
  </a>
  <hr>
</body>
</html>
```

Текст листинга содержит: титульную часть, заголовок в виде надписей разным размером шрифта и две адресные ссылки. Если этот текст поместить, например в файл *\$HOME/src/rvs/Title.html*, то его можно запустить из файлового менеджера и отразить в браузере, что и показано на рисунке 4.4.

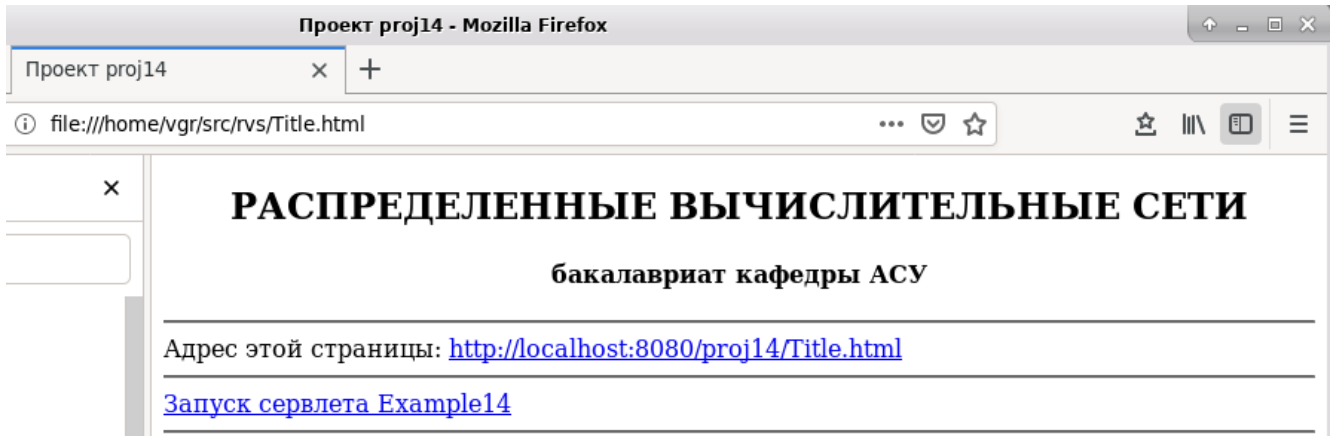


Рисунок 4.4 — Демонстрация в браузере статического HTML-текста

В 1995 году, на основе общих требований CGI, появляются новые инструменты:

- свободный web-сервер Apache [52]: «**Apache HTTP-сервер** (... назван именем группы племён североамериканских индейцев апачей; ...) — свободный веб-сервер. Apache является кроссплатформенным ПО, поддерживает операционные системы Linux, BSD ... Основными достоинствами Apache считаются надёжность и гибкость конфигурации. Он позволяет подключать внешние модули для предоставления данных, использовать СУБД для аутентификации пользователей, модифицировать сообщения об ошибках и т. д. Поддерживает IPv6. ...»;
- серверный препроцессорный язык PHP [53]: «**PHP** (англ. *PHP: Hypertext Preprocessor* — «PHP: препроцессор гипертекста»; первоначально *Personal Home Page Tools* — «Инструменты для создания персональных веб-страниц») — скриптовый язык общего назначения, интенсивно применяемый для разработки веб-приложений. В настоящее время поддерживается подавляющим большинством хостинг-провайдеров и является одним из лидеров среди языков, применяющихся для создания динамических веб-сайтов. ...»;
- встраиваемый язык JavaScript [47]: «**JavaScript** (... аббр. **JS** ...) — мультипарадигменный язык программирования. ... Является реализацией языка ECMAScript (стандарт ECMA-262). JavaScript обычно используется как встраиваемый язык для программного доступа к объектам приложений. Наиболее широкое применение находит в браузерах как язык сценариев для придания интерактивности веб-страницам. ...».

Благодаря новым технологиям возможности Всемирной паутины значительно расширились, появилась возможность стандартными средствами языка PHP создавать динамические web-страницы. Тем не менее для разработки приложений уровня предприятия они еще не годились. Причина была в том, что сценарии PHP, которые на стороне сервера вставлялись в текст HTML-страниц, постоянно подвергались синтаксическому контролю. Это не позволяло кэшировать страницы, что снижало быстродействие и требовало больших ресурсов web-серверов.

В 1997 году, Sun Microsystems предложила технологию сервлетов [54]: «**Сервлет** является интерфейсом Java, реализация которого расширяет функциональные возможности сервера. Сервлет взаимодействует с клиентами посредством принципа запрос-ответ. Хотя сервлеты могут обслуживать любые запросы, они обычно используются для расширения веб-серверов. Для таких приложений технология Java Servlet определяет HTTP-специфичные сервлет классы. Пакеты `javax.servlet` и `javax.servlet.http` обеспечивают интерфейсы и классы для создания сервлетов. Первая спецификация сервлетов была создана в Sun Microsystems (версия 1.0 была закончена в июне 1997). ...». В сочетании с апплетами, которые были созданы еще в 1995 году, появилась возможность проектировать полноценные графические приложения, используя только функциональные возможности браузеров.

Несмотря на предоставление новых возможностей, технологии Java встретили и недружественное противодействие:

- разработчикам браузеров не нравилось, что апплеты, имея своё собственное окно, делают ненужным и сам рендеринг HTML-страниц; поэтому, обосновывая сложную поддержку «громоздкой» Java-машины (JRE), которая ещё и может нарушать безопасность компьютера, они «спускали на тормозах» нормальную работу апплетов;
- необходимость создавать HTML-страницы на языке Java не повышало популярности предложенной технологии;
- использование апплетами пакета `javax.swing` делает их графическое представление «чужеродным».

С 1999 года, Sun Microsystems стала формировать отдельную платформу для разработки приложений уровня предприятия — J2EE (Java EE), а также выпускать новый сервер [55]: «**Tomcat** (в старых версиях – **Catalina**) — контейнер сервлетов с открытым исходным кодом, разрабатываемый Apache Software Foundation. Реализует спецификацию сервлетов, спецификацию JavaServer Pages (JSP) и JavaServer Faces (JSF). Написан на языке Java. **Tomcat** позволяет запускать веб-приложения и содержит ряд программ для самоконфигурирования. **Tomcat** используется в качестве самостоятельного веб-сервера, в качестве сервера контента в сочетании с веб-сервером Apache HTTP Server, а также в качестве контейнера сервлетов в серверах приложений JBoss и GlassFish. ...». Таким образом, был создан полный набор инструментов, позволяющих развивать web-технологии Java, даже без ориентации на технологию апплетов.

Чтобы иметь более полное представление о сказанном выше, мы рассмотрим ряд конкретных примеров:

- краткое описание классов ***Servlet*** и ***HttpServlet***;
- контейнер сервлетов Apache Tomcat;
- технологию JSP-страниц;
- модель проектирования MVC, - применительно к технологии сервлетов.

4.3.1 Классы *Servlet* и *HttpServlet*

Официальную документацию на описание сервлетов можно найти, например, в источнике [56], где указано, что в пакете ***javax.servlet*** имеется публичный интерфейс ***Servlet***, содержащий описание пяти методов:

- `void destroy()` - вызывается контейнером сервлета, чтобы указать сервлету, что он выводится из эксплуатации;
- `ServletConfig getServletConfig()` - возвращает объект *ServletConfig*, который содержит параметры инициализации и запуска для этого сервлета.
- `String getServletInfo()` - возвращает информацию о сервлете, такую как автор, версия и авторские права.
- `void init(ServletConfig config)` - вызывается контейнером сервлета, чтобы указать сервлету, что он вводится в эксплуатацию.
- `void service(ServletRequest req, ServletResponse res)` - вызывается контейнером сервлета, чтобы сервлет мог ответить на запрос.

Используемый в программировании сервлет ***HttpServlet*** является расширением абстрактного класса ***GenericServlet***, включающего интерфейс ***Servlet***, как это показано на рисунке 4.5.

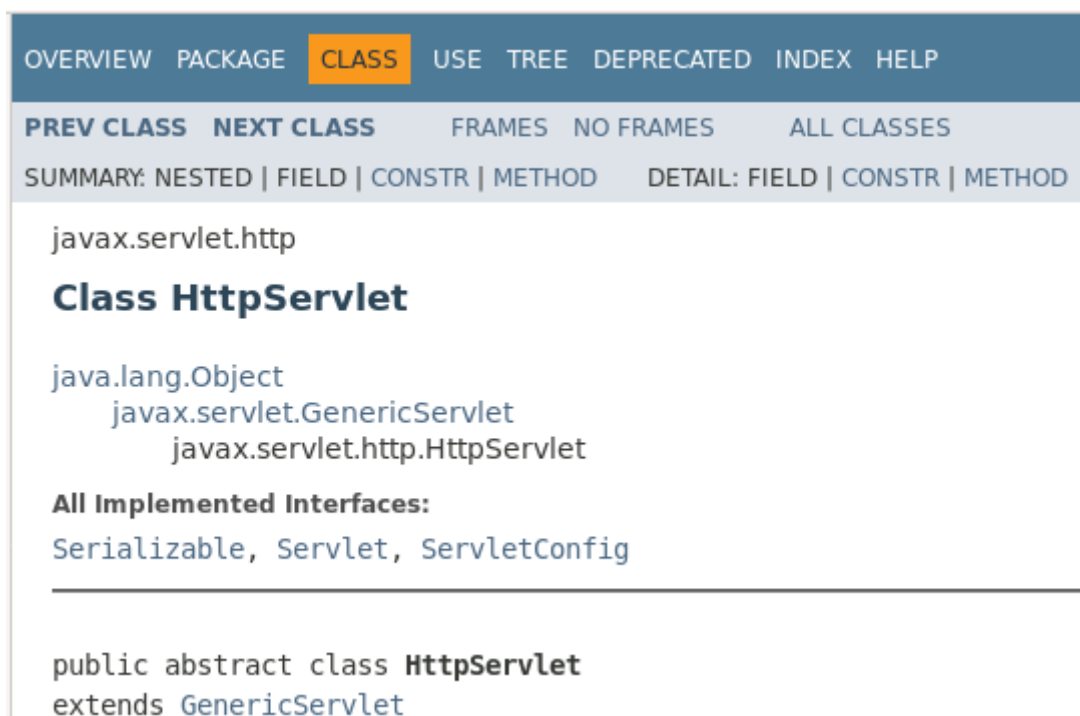


Рисунок 4.5 — Структура зависимостей для абстрактного класса `HttpServlet`

Документация [56] так характеризует **HttpServlet**: «... предоставляет абстрактный класс, который можно разделить на подклассы для создания HTTP-сервлета, подходящего для веб-сайта. Подкласс HttpServlet должен переопределить хотя бы один метод, обычно один из следующих:

- **doGet(...)**, если сервлет поддерживает запросы HTTP GET;
- **doPost(...)**, для запросов HTTP POST;
- **doPut(...)**, для запросов HTTP PUT;
- **doDelete(...)**, для запросов HTTP DELETE;
- **init(...)** и **destroy(...)**, чтобы управлять ресурсами, которые управляют жизненным циклом сервлета;
- **getServletInfo(...)**, когда необходимо предоставить информацию о себе.

Нет необходимости переопределять метод **service(...)**. Он обрабатывает стандартные HTTP-запросы, отправляя их методам-обработчикам для каждого типа HTTP-запроса (перечисленные выше методы **doXXX**). Аналогично, почти нет причин переопределять методы **doOptions(...)** и **doTrace(...)**.

Сервлеты обычно работают на многопоточных серверах, поэтому имейте в виду, что сервлет должен обрабатывать параллельные запросы, и соблюдайте осторожность при синхронизации доступа к общим ресурсам. К общим ресурсам относятся данные в памяти, такие как переменные экземпляра или класса, и внешние объекты, такие как файлы, соединения с базой данных и сетевые соединения. ...».

Любой сервлет, который создает проектировщик является обычным публичным Java-классом, который расширяет абстрактный класс **HttpServlet**. Его «жизненный цикл» состоит из трёх периодов:

1. Когда сервер стартует, то загружает доступные ему сервлеты. При этом, каждый сервлет выполняет метод **init(...)**. При необходимости, проектировщик может переопределить этот метод.
2. В процессе работы сервера, сервлет выполняет методы, которые запрашивает клиент, кроме методов **init(...)** и **destroy(...)**. При необходимости, проектировщик переопределяет нужные методы, обычно — методы **doGet(...)** и **doPost(...)**.
3. Когда сервер завершает работу, он для каждого сервлета вызывает его метод **destroy(...)**. При необходимости, проектировщик может переопределить этот метод.

В процессе работы, сервер параллельно обслуживает запросы клиентов. Если необходимо, чтобы сервлет обслуживал только одного клиента в определённый момент времени, нужно реализовать интерфейс **SingleThreadModel** в дополнение к наследованию абстрактного класса **HttpServlet**. При этом, нет необходимости вносить каких либо изменений в реализацию самого сервлета.

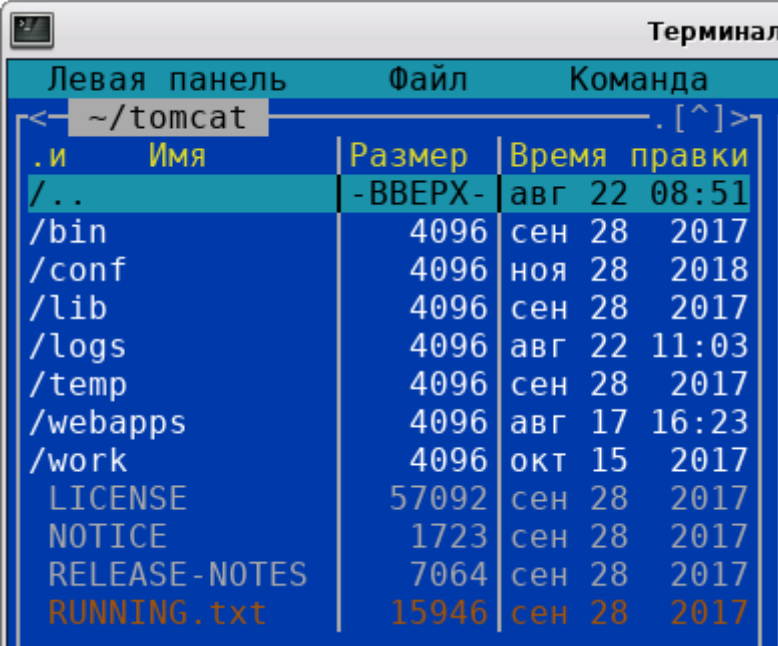
На практике обычно переопределяются два метода: **doGet(...)** и **doPost(...)**. Оба они имеют одинаковые аргументы:

- `ServletRequest req` — объект запроса, получающий информацию от клиента (браузера);
- `ServletResponse res` — объект ответа, передаваемый клиенту (браузеру).

Каждый из этих объектов имеет достаточно много собственных методов, которые нужно изучать по представленной в [56] документации. Мы, когда будем демонстрировать конкретные примеры, рассмотрим только наиболее важные из них и выполним их в среде разработки Eclipse EE.

4.3.2 Контейнер сервлетов Apache Tomcat

Как было отмечено ранее, классическим представителем контейнера сервлетов является Apache Tomcat [55]. Его можно загрузить с сайта [57]. Для учебных целей данного курса используется **Tomcat v8.5 Server**, установленный в домашней директории пользователя: **\$HOME/tomcat**. На рисунке 4.6 показано содержимое этой директории.



Имя	Размер	Время правки
./	-ВВЕРХ-	авг 22 08:51
/bin	4096	сен 28 2017
/conf	4096	ноя 28 2018
/lib	4096	сен 28 2017
/logs	4096	авг 22 11:03
/temp	4096	сен 28 2017
/webapps	4096	авг 17 16:23
/work	4096	окт 15 2017
LICENSE	57092	сен 28 2017
NOTICE	1723	сен 28 2017
RELEASE-NOTES	7064	сен 28 2017
RUNNING.txt	15946	сен 28 2017

Рисунок 4.6 — Структура каталогов дистрибутива Apache Tomcat

В лучших традициях ОС UNIX, назначение основных каталогов дистрибутива — следующее:

- **bin** — содержит служебные сценарии для администрирования сервера; причём, сценарии, имеющие расширение **.sh**, предназначены для ОС UNIX/Linux, а сценарии, имеющие расширение **.bat**, — для ОС MS Windows;
- **conf** — содержит различные файлы конфигурации сервера, включая обеспечение безопасности;
- **lib** — содержит JAR-архивы библиотек;

- **webapps** — базовое расположение реализованных на сервере проектов.

Минимальная настройка дистрибутива Apache Tomcat требует задания системной переменной `CATALINA_HOME`, что для нашего варианта установки имеет значение показанное на рисунке 4.7.

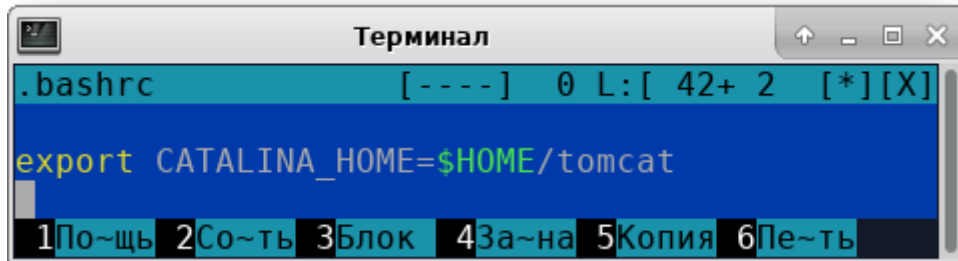


Рисунок 4.7 — Задание системной переменной `CATALINA_HOME`

Минимальная проверка работоспособности дистрибутива Apache Tomcat осуществляется с помощью сценария **startup.sh**, который запускается из каталога `$CATALINA_HOME/bin`, как это демонстрируется рисунком 4.8.

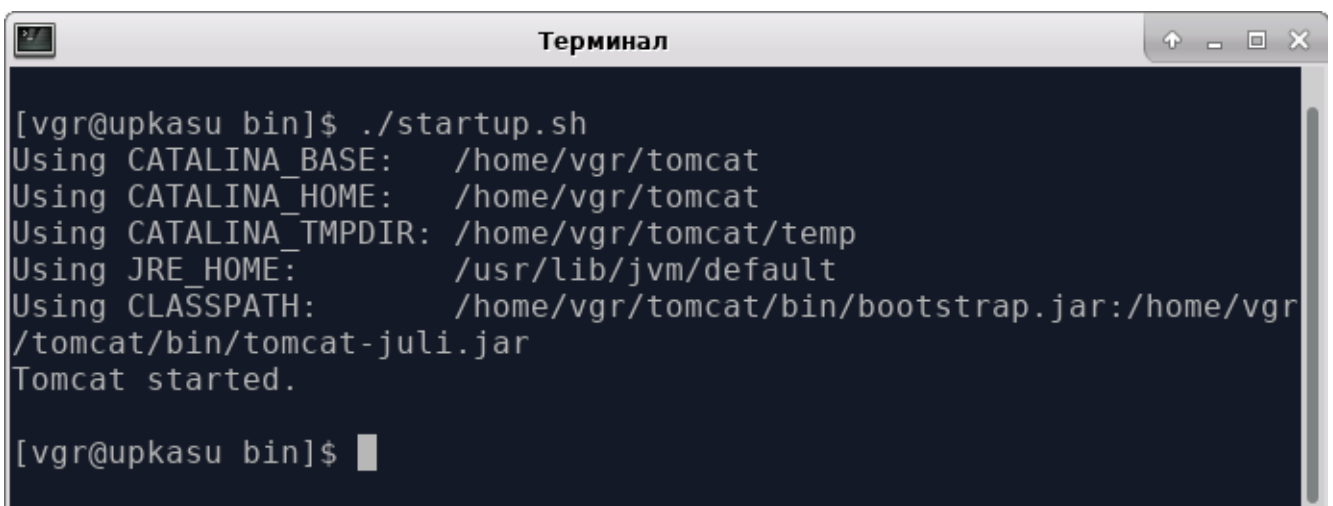


Рисунок 4.8 — Нормальный старт сервера Apache Tomcat

После старта, сервер Apache Tomcat начинает прослушивать порт 8080. Это отличает его от обычных web-серверов, которые по умолчанию прослушивают порт 80. Соответственно, для остановки запущенного сервера используется сценарий **shutdown.sh**.

В учебных примерах (при создании и тестировании сервлетов), мы будем использовать инструментальные средства среды Eclipse EE. Эта среда сама стартует сервер Apache Tomcat, отображая его дистрибутив в своём адресном пространстве реализации проектов. Стартуемый в Eclipse EE сервер Apache также использует порт 8080, поэтому, перед началом использования среды разработки, Apache Tomcat должен быть остановлен!

Демонстрацию запуска сервера из среды Eclipse EE проведём в рамках отдельного проекта с именем **proj14**. Для этого, выберем **File** → **New** → **Dynamic Web Project** и, в появившемся окне, укажем имя проекта, используемую версию Apache Tomcat и местоположение его дистрибутива. Правильный результат установок показан на рисунке 4.9.

New Dynamic Web Project

Dynamic Web Project

Create a standalone Dynamic Web project or add it to a new or existing Enterprise Application.

Project name:

Project location

☒ Use default location

Location:

Target runtime

Dynamic web module version

Configuration

A good starting point for working with Apache Tomcat v8.5 runtime. Additional facets can later be installed to add new functionality to the project.

EAR membership

☐ Add project to an EAR

EAR project name:

Working sets

☐ Add project to working sets

Working sets:

Рисунок 4.9 — Открытие первого проекта Dynamic Web Project и привязка к нему дистрибутива сервера Apache Tomcat

После нажатия кнопки «*Finish*», проект **proj14** откроется с необходимой привязкой к используемому контейнеру сервлетов. На рисунке 4.10 показана базовая структура этого вновь созданного проекта. Эту структуру необходимо изучить и знать назначение ее основных каталогов.

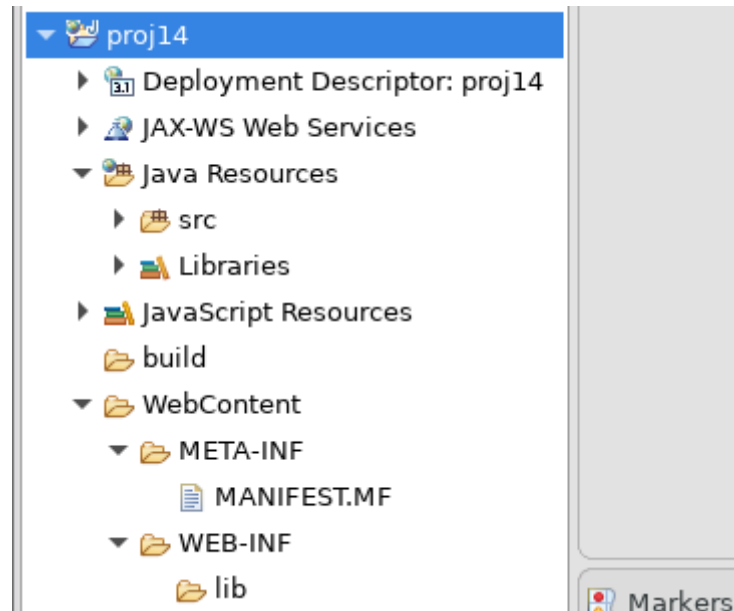


Рисунок 4.10 — Начальная структура каталогов проекта proj14

Общее назначение основных каталогов — следующее:

- **src** — каталог, предназначенный для хранения исходных текстов сервлетов проекта;
- **WebContent** — корневой (*root*) каталог проекта; помещённые в него каталоги и файлы задаются в абсолютной адресации, например, файл **index.html**, помещённый в него, будет адресоваться как **/index.html**;
- **META-INF** — внутренний для проекта каталог, предназначенный для хранения его манифеста; созданный после завершения разработки JAR-архив проекта будет содержать этот манифест;
- **WEB-INF** - внутренний для проекта каталог, предназначенный для хранения JSP-страниц и других файлов, доступных в относительной адресации и только для программного обеспечения этого проекта;
- **lib** — каталог, в который помещают дополнительные библиотеки, используемые только самим проектом.

Проведём демонстрацию работы Apache Tomcat как обычного web-сервера, для чего выделим мышкой каталог WebContent и правой кнопкой активируем меню, в котором выберем **New** → **HTML File**. Далее:

- указываем имя файла **Title.html** и нажимаем кнопку «**Next** >» (см. рисунок 4.11);
- выбираем тип HTML-файла: **New HTML File (5)** (см. рисунок 4.12);

- нажимаем кнопку «**Finish**» и получаем шаблон HTML-страницы, показанный на рисунке 4.13.

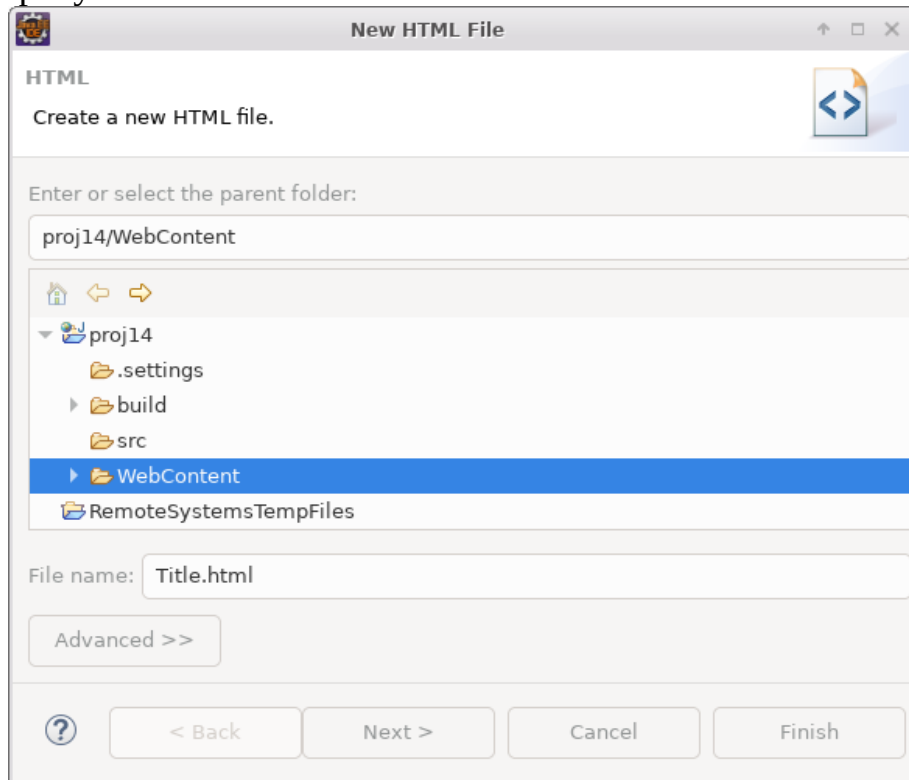


Рисунок 4.11 — Задание имени создаваемого HTML-файла

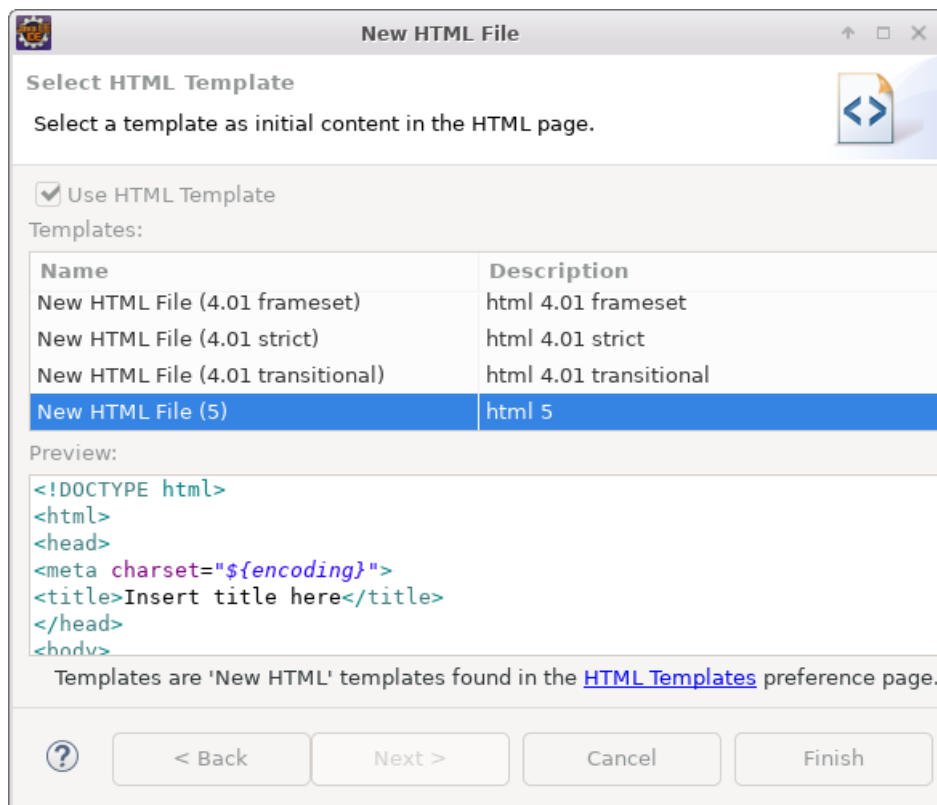


Рисунок 4.12 — Выбор версии шаблона создаваемого HTML-файла

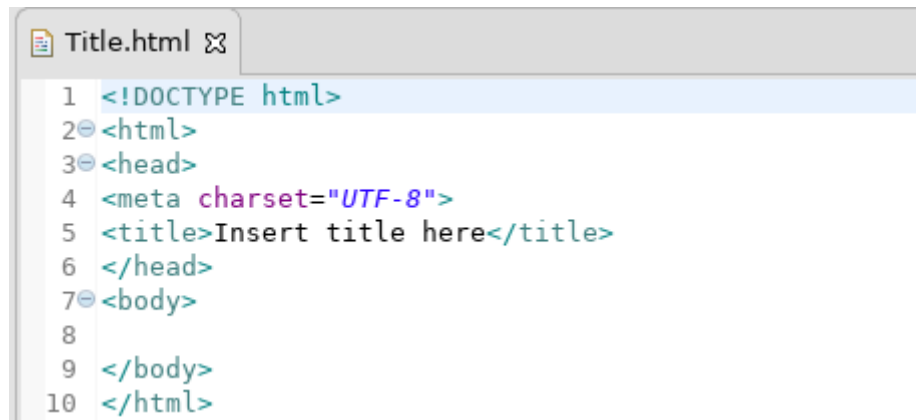


Рисунок 4.13 — Вкладка редактора Eclipse EE с шаблоном файла Title.html

Таким образом создаются и размещаются все статические HTML-страницы, адресуемые и доступные для просмотра из любого браузера.

Теперь, заменим содержимое данного шаблона на текст листинга 4.1, а затем запустим проект на выполнение.

Сначала появится окно с предложением выбрать и запустить сервер. Здесь нужно согласиться и появится отображение содержимого файла во встроенном браузере Eclipse EE. Результат такого запуска показан на рисунке 4.14.

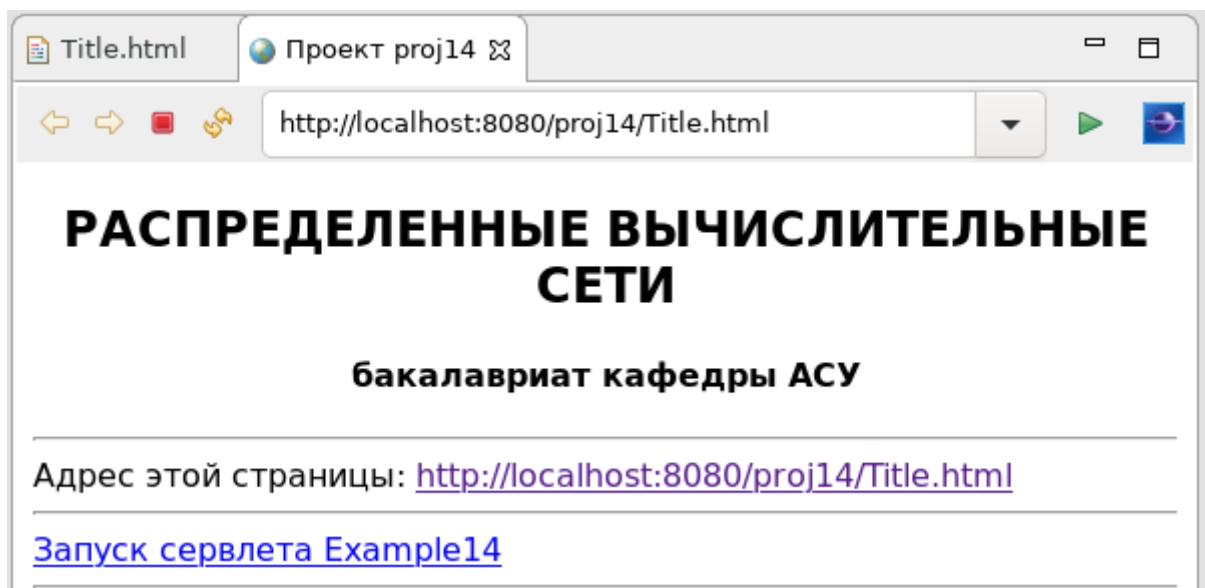


Рисунок 4.14 — Отображение файла Title.html во встроенном браузере Eclipse EE

Попытка запустить сервлет по второй ссылке закончится неудачей, поскольку сервлет **Example14** — ещё не создан. Браузер выведет стандартное сообщение, показанное на рисунке 4.15.

Таким образом, мы научились создавать статический контент реализуемого проекта и запускать его из среды Eclipse EE.

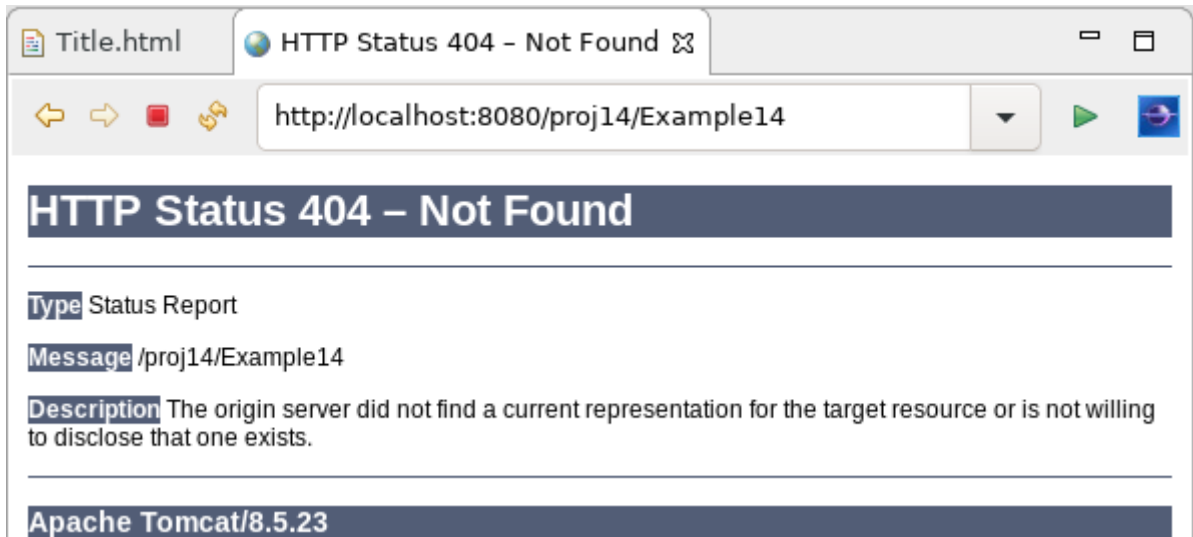


Рисунок 4.15 — Сообщение встроенного браузера об отсутствии адресуемого ресурса

Теперь создадим первый сервлет с именем **Example14**. Для этого, в проекте **proj14** выделим каталог **src**, а затем правой кнопкой мышки активируем меню: **New** → **Servlet**. В появившемся окне укажем нужный пакет и имя сервлета, как это показано на рисунке 4.16.

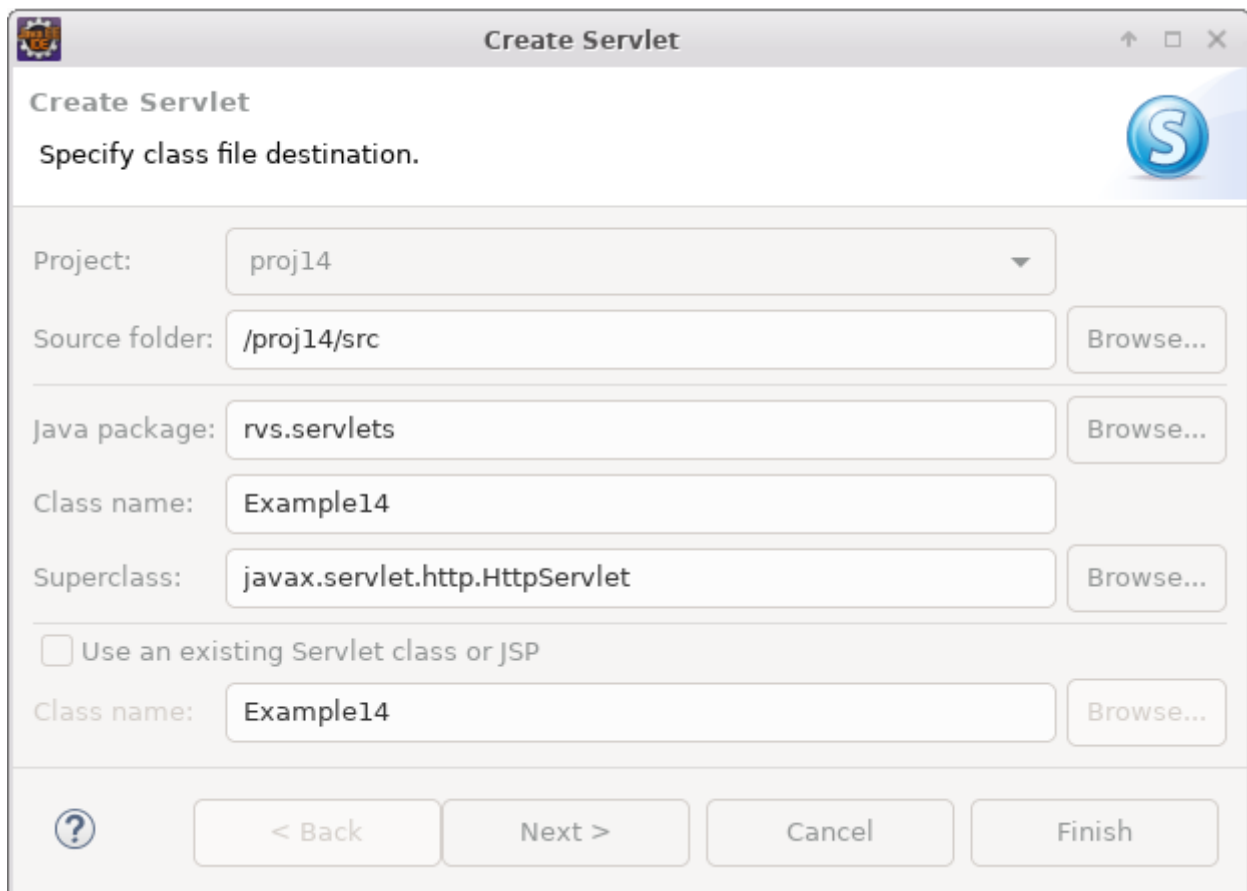


Рисунок 4.16 — Окно открытия сервлета Example14

Два раза нажав кнопку «*Next >*», мы переходим к окну, показанному на рисунке 4.17, где происходит выбор используемых сервлетом методов.

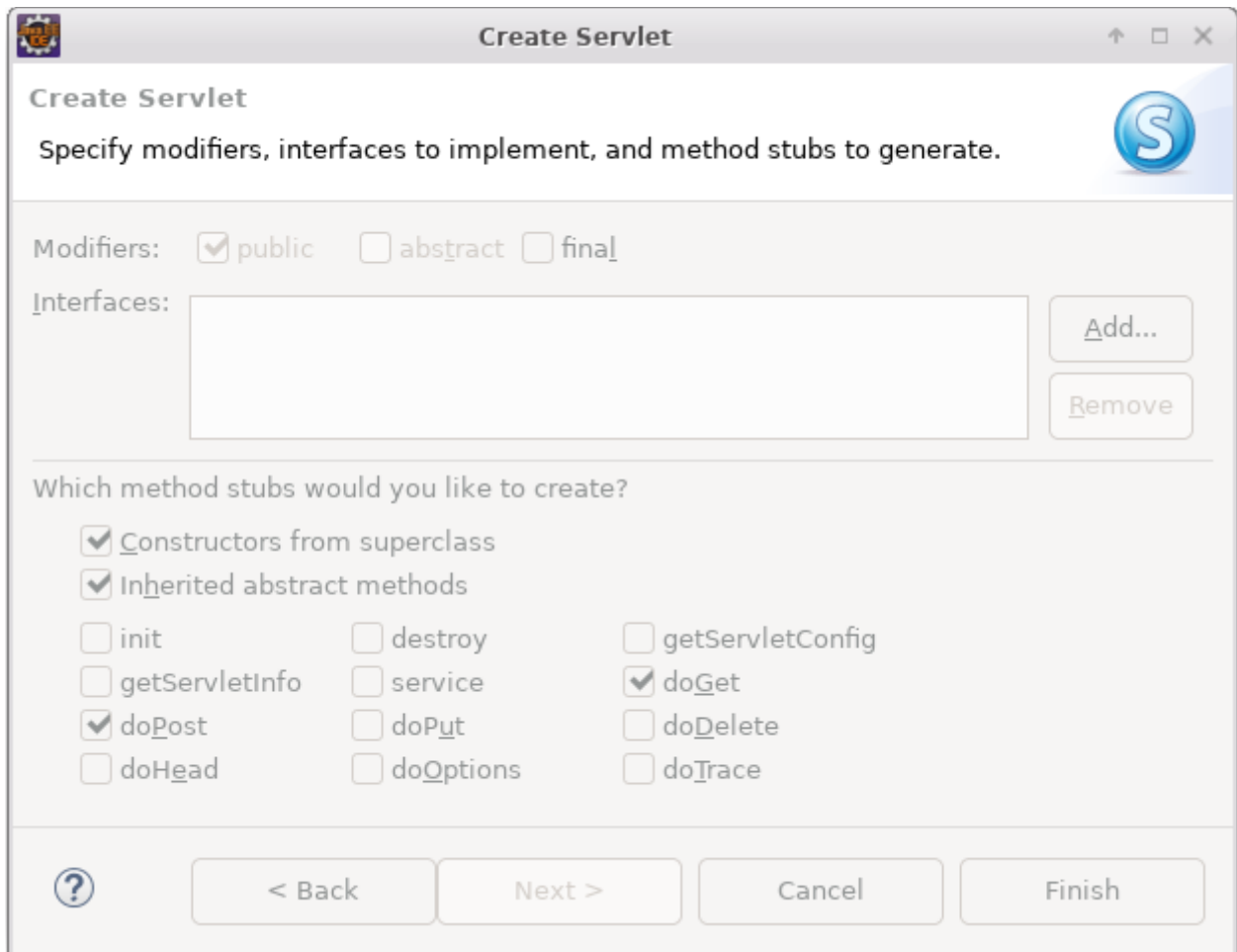


Рисунок 4.17 — Методы сервлета, предлагаемые по умолчанию

Обычно используются два метода ***doGet(...)*** и ***doPost(...)***, которые запрашивают все браузеры, поэтому — нажимаем кнопку «*Finish*» и получаем результат, представленный на листинге 4.2.

Листинг 4.2 — Стандартный шаблон сервлета *Example14*

```
package rvs.servlets;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * Servlet implementation class Example14
 */
@WebServlet("/Example14")
public class Example14 extends HttpServlet {
    private static final long serialVersionUID = 1L;
```

```

/**
 * @see HttpServlet#HttpServlet()
 */
public Example14() {
    super();
    // TODO Auto-generated constructor stub
}

/**
 * @see HttpServlet#doGet(HttpServletRequest request,
 * HttpServletResponse response)
 */
protected void doGet(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException
{
    // TODO Auto-generated method stub
    response.getWriter().append("Served at: ").append(
        request.getContextPath());
}

/**
 * @see HttpServlet#doPost(HttpServletRequest request,
 * HttpServletResponse response)
 */
protected void doPost(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException {
    // TODO Auto-generated method stub
    doGet(request, response);
}
}

```

Первоначальный шаблон сервлета содержит: имя пакета, импортируемые по умолчанию классы и определение публичного класса **Example14**, расширяющего абстрактный класс **HttpServlet**. Тело шаблона сервлета содержит:

- статическую константу **serialVersionUID**, используемую для идентификации сервлета;
- конструктор **Example14()**;
- метод **doGet(...)**, посылающий в качестве ответа текстовое сообщение;
- метод **doPost(...)**, просто вызывающий метод **doGet(...)**.

Если запустить этот сервер на выполнение, то он выдаст сообщение, показанное на рисунке 4.18.

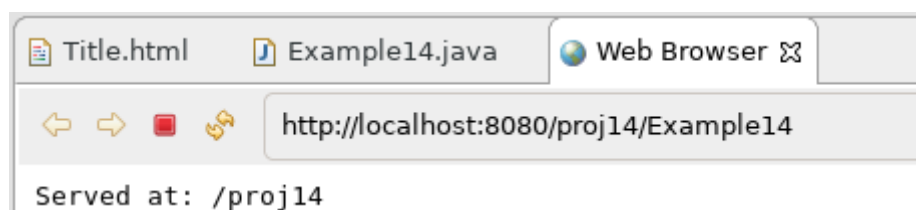


Рисунок 4.18 — Ответ шаблона сервлета Example14

Для дальнейшего изучения сервлетов необходимо учитывать следующие правила взаимодействия браузера и web-сервера:

1. Основным методом запроса браузера является метод GET, поэтому первым в сервлете нужно реализовать метод ***doGet(...)***.
2. Браузеры разных ОС настроены на разные кодировки символов: ***Cp1251*** — для MS Windows и ***UTF-8*** — для UNIX/Linux. Сервлеты должны самостоятельно учитывать используемую браузерами кодировку, для этого, объекты запроса (***request***) и ответа (***response***) методов сервлета имеют соответствующие методы ***getCharacterEncoding()*** и ***setCharacterEncoding(String str)***, обрабатывающие эту ситуацию.
3. Сервлет должен возвращать браузеру HTML-страницу с правильным типом контента, обычно - ***text/html***. Поэтому объект ***response*** должен использовать метод ***setContentType(String str)***.
4. При первом обращении к сервлету он компилируется в новый Java-класс, поэтому после изменения его исходного кода и запуске на выполнение, среда Eclipse EE предлагает перезапустить сервер и подключить к нему новый вариант сервлета.

Демонстрацию важности перечисленных правил провести очень просто, для этого в методе ***doGet(...)*** шаблона сервлета можно заменить текст "Served at: " на русский текст "Запрашивает: ". В результате, запрос к сервлету будет выглядеть как показано на рисунке 4.19.

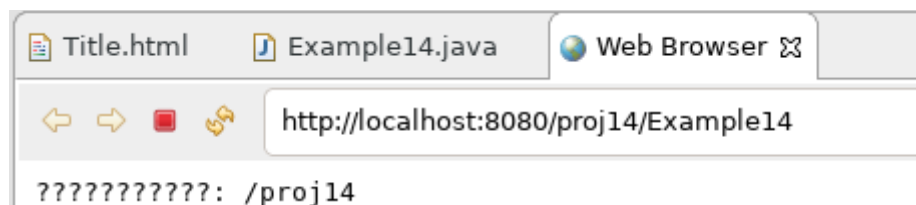


Рисунок 4.19 — Ответ шаблона сервлета, содержащего русскоязычный тест

4.3.3 Диспетчер запросов - *RequestDispatcher*

Сервлет получает запрос от браузера в виде объекта ***request*** определённого интерфейсом ***HttpServletRequest*** и проводит его анализ. Для этого объект запроса содержит множество методов, определяющих детали протокола HTTP:

- ***String getCharacterEncoding()*** – определение символьной кодировки запроса;
- ***String getContentType()*** – определение MIME-типа (Multipurpose Internet Mail Extension) пришедшего запроса;

- *String getProtocol()* – определение названия и версии протокола;
- *String getServerName()*, *getServerPort()* – определение имени сервера, принявшего запрос, и порта, на котором запрос был соответственно принят сервером;
- *String getRemoteAddr()*, *getRemoteHost()* – определение IP-адреса и имени клиента, от которого пришел запрос;
- *String getRemoteUser()* – определение имени пользователя, выполнившего запрос;
- *ServletInputStream getInputStream()*, *BufferedReader getReader()* – получение ссылки на поток, ассоциированный с содержимым полученного запроса.

После анализа запроса, разработчик должен привязать к объекту **request** некоторый ресурс сервера, который должен быть передан клиенту. Это делается с помощью реализации объекта интерфейса **RequestDispatcher**:

```
RequestDispatcher disp =
    request.getRequestDispatcher(String path);
```

где **path** — абсолютный путь (в пределах сервлета) к подключаемому ресурсу.

Сама передача ресурса клиенту осуществляется методом **forward(...)** в виде:

```
disp.forward(request, response);
```

Для демонстрации этого стандартного решения, подключим к запросу уже имеющийся ресурс — файл **Title.html**. Для этого, преобразуем метод **doGet(...)** из листинга 4.2 к виду показанному на листинге 4.3.

Листинг 4.3 — Подключение Title.html в методе doGet(...) сервлета Example14

```
/**
 * @see HttpServlet#doGet(HttpServletRequest request,
 * HttpServletResponse response)
 */
protected void doGet(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException
{
    /**
     * Явная установка кодировок объектов запроса и ответа.
     * Стандартная установка контекста ответа.
     */
    request.setCharacterEncoding("UTF-8");
    response.setCharacterEncoding("UTF-8");
    response.setContentType("text/html");
}
```

```

/**
 * Стандартное подключение ресурса сервлета.
 */
RequestDispatcher disp =
    request.getRequestDispatcher("/Title.html");
disp.forward(request, response);
}

```

После внесенных изменений и перезапуска сервлета, обращения по адресам <http://localhost:8080/proj14/Title.html> и <http://localhost:8080/proj14/Example14> выдают одинаковый результат показанный ранее на рисунке 4.14.

Указанный пример демонстрирует доступ сервлета к общедоступному ресурсу. Для внутренних ресурсов, недоступных прямой адресации из браузеров, в архитектуре сервлета имеется директория **WEB-INF**. Чтобы показать это, создадим в этом каталоге файл с именем **post1.html**, содержащий форму запроса к приложению ведения записей в базе данных, как это делалось в примерах использования технологий CORBA и RMI. Содержимое такого файла показано на листинге 4.4.

Листинг 4.4 — Исходный текст файла *post1.html* сервлета *Example14*

```

<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Форма запроса</title>
</head>
<body>
    <hr>
    <b>Запрос к таблице ведения записей</b>
    <hr>
    <form action="Example14" method="post" accept-charset="UTF-8">
        <p> Введи ключ :
            <input type="text" size="10" name="key">
        </p>
        <p> Введи текст: <br>
            <textarea rows="10" cols="40" name="text"></textarea>
        </p>
        <p>
            <input type="submit">
        </p>
    </form>
    <hr>
</body>
</html>

```

Если мы напрямую обратимся к файлу **post1.html**, то получим ответ, показанный на рисунке 4.20, а если мы в методе **doGet(...)** создадим объект диспетчера в виде:

```

RequestDispatcher disp =
    request.getRequestDispatcher("/WEB-INF/post1.html");

```

то вызов сервлета покажет нужную страницу, приведённую на рисунке 4.21.

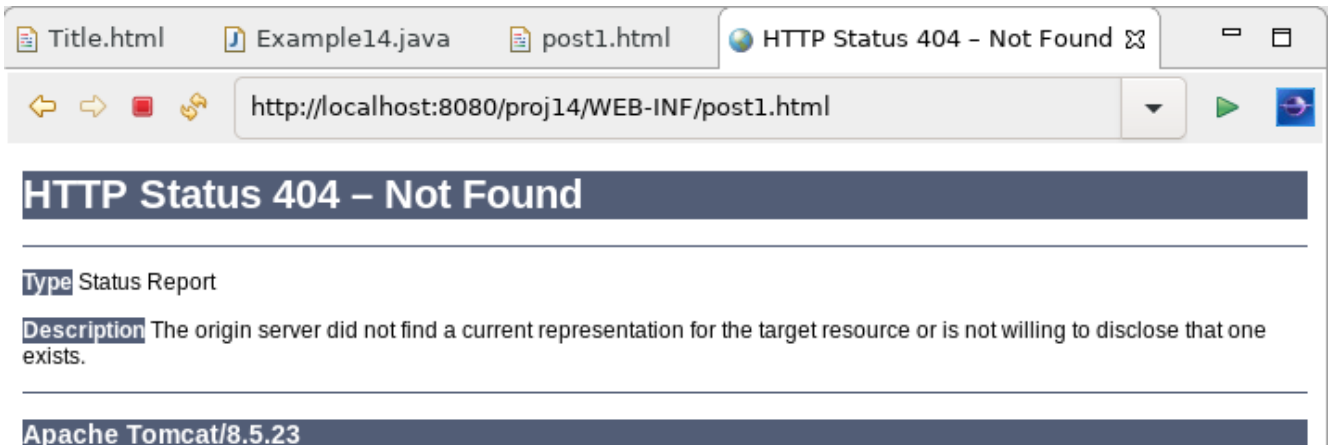


Рисунок 4.20 — Прямое обращение к файлу post1.html

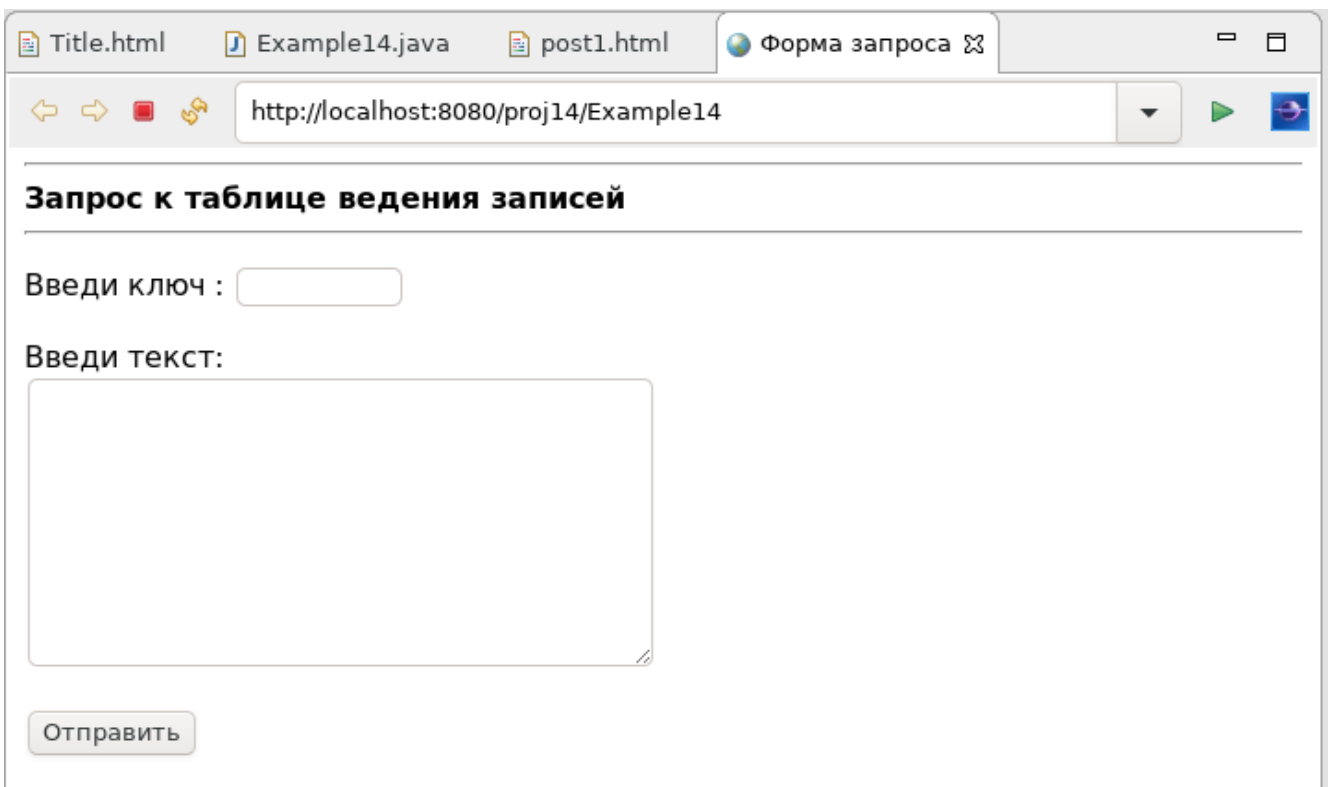


Рисунок 4.21 — Обращение к файлу post1.html из метода doGet(...)

4.3.4 Технология JSP-страниц

Наиболее существенным свойством сервера Apache Tomcat является реализация в нем технологии JSP-страниц [58]: «**JSP (JavaServer Pages)** — технология, позволяющая веб-разработчикам создавать содержимое, которое имеет как статические, так и динамические компоненты. Страница JSP содержит текст двух типов: статические исходные данные, которые могут быть оформлены в одном из текстовых форматов HTML, SVG, WML, или XML, и JSP-элементы, которые

конструируют динамическое содержимое. Кроме этого могут использоваться библиотеки JSP-тегов, а также Expression Language (EL), для внедрения Java-кода в статичное содержимое JSP-страниц. Код JSP-страницы транслируется в Java-код сервлета с помощью компилятора JSP-страниц Jasper, и затем компилируется в байт-код виртуальной машины Java (JVM). Контейнеры сервлетов, способные исполнять JSP-страницы, написаны на платформенно-независимом языке Java. JSP-страницы загружаются на сервере и управляются из структуры специального Java server packet, который называется Jakarta EE Web Application. Обычно страницы упакованы в файловые архивы .war и .ear. Технология JSP, является платформенно-независимой, переносимой и легко расширяемой, для разработки веб-приложений. ...».

Создадим в нашем проекте JSP-страницу с именем **post2.jsp**, используя имеющиеся шаблоны среды Eclipse EE. Для этого, в проекте выделим каталог **WEB-INF** и правой кнопкой мыши активируем меню: **New** → **JSP File**. В появившемся окне (см. рисунок 4.22) введём нужное имя файла и нажмём кнопку «**Next** >».

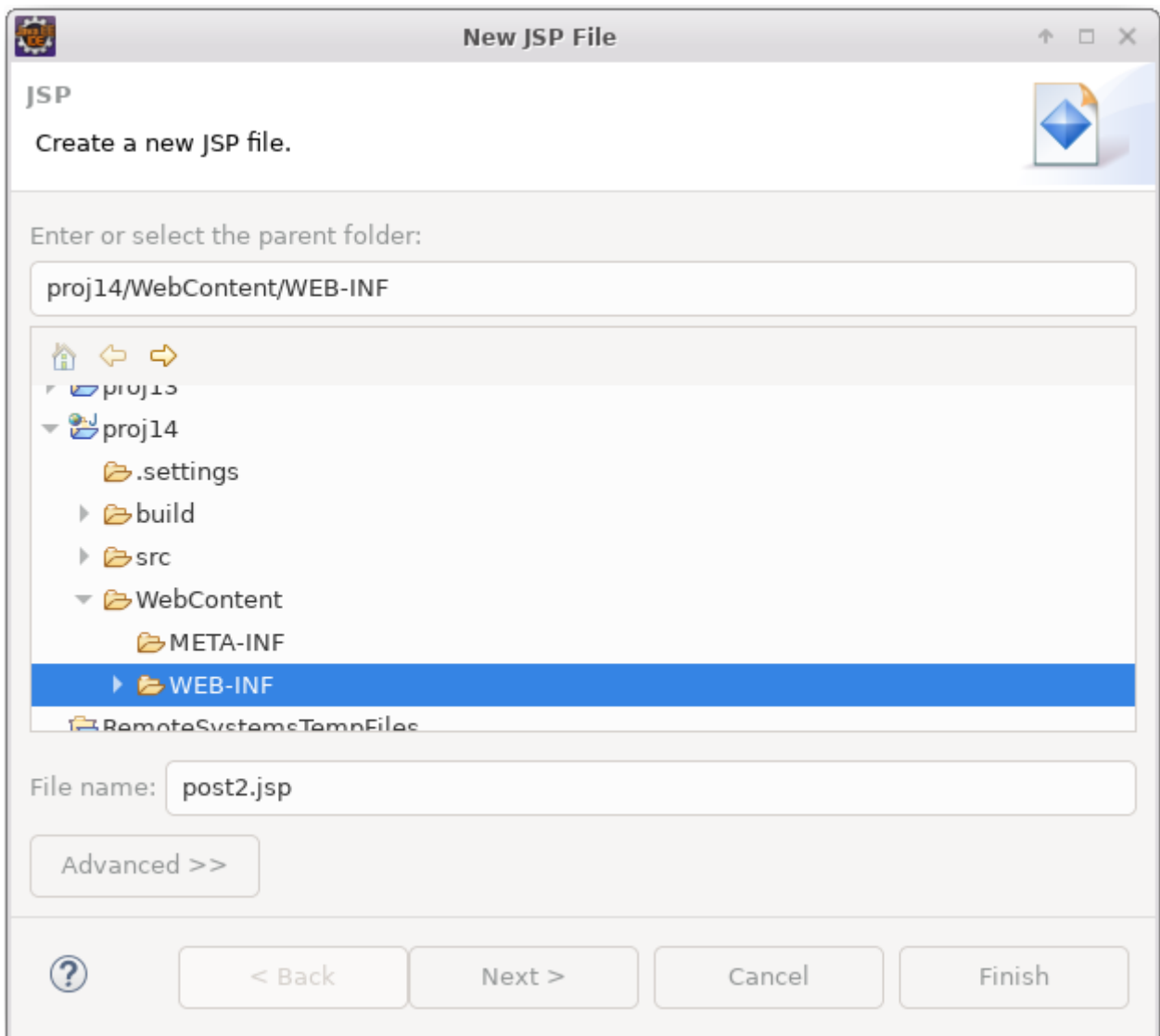


Рисунок 4.22 — Задание имени файла JSP-страницы

Появится новое окно, показанное на рисунке 4.23, в котором выбирается один из доступных шаблонов JSP-страниц. Выберем предложенное по умолчанию и нажмём кнопку «**Finish**».

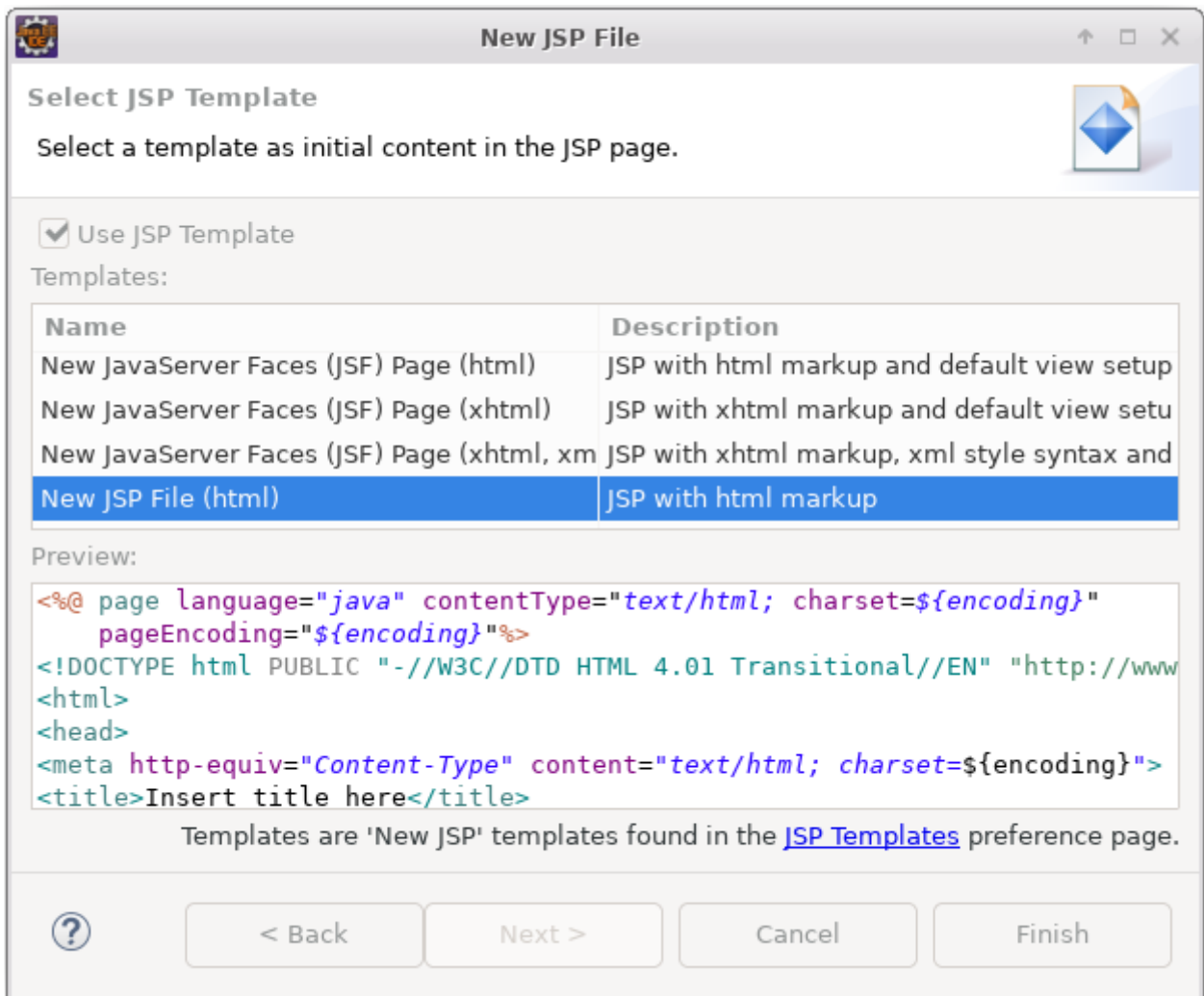


Рисунок 4.23 — Выбор шаблона файла JSP-страницы

В результате, в редакторе Eclipse EE появится новая вкладка с текстом выбранного шаблона для файла **post2.jsp**, показанного на листинге 4.5.

Листинг 4.5 — Исходный текст JSP-файла post2.jsp сервлета Example14

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>
</head>
<body>
</body>
</html>
```

Представленный шаблон JSP-страницы, является типичным шаблоном страницы HTML версии 4.01 дополненный в начале текста директивой, указывающей на используемый язык программирования, тип контента и используемую кодировку символов, которую можно заменить на любую другую. Далее, в этот текст можно вставлять конструкции языка HTML и синтаксические конструкции JSP-кода, представленные в таблице 4.1.

Таблица 4.1 — Синтаксические конструкции JSP-кода страниц

Элемент JSP	Синтаксис	Описание
Выражение JSP	<code><%= выражение %></code>	Выражение обрабатывается и направляется на вывод.
Скриплет JSP	<code><% код %></code>	Код добавляется в метод service.
Объявление JSP	<code><%! код %></code>	Код добавляется в тело класса сервлета, вне метода service.
Директива JSP page	<code><%@ page att="значение" %></code>	Директивы для движка сервлета с информацией об основных настройках.
Директива JSP include	<code><%@ include file="url" %></code>	Файл в локальной системе, подключаемый при трансляции JSP в сервлет.
Комментарий JSP	<code><%-- комментарий --%></code>	Комментарий; игнорируется при трансляции JSP страницы в сервлет.
Действие jsp:include	<code><jsp:include page="относительный URL" flush="true"/></code>	Подключает файл при запросе страницы.
Действие jsp:useBean	<code><jsp:useBean att=значение*/> or <jsp:useBean att=значение* > ... </jsp:useBean></code>	Найти или создать Java Bean.
Действие jsp:setProperty	<code><jsp:setProperty att=значение*/></code>	Устанавливает свойства bean, или явно, или указанием на соответствующее значение параметра, передаваемое при запросе.
Действие jsp:getProperty	<code><jsp:getProperty name="ИмяСвойства" value="значение"/></code>	Получение и вывод свойств bean.
Действие jsp:forward	<code><jsp:forward page="относительныйURL"/></code>	Передаёт запрос другой странице.
Действие jsp:plugin	<code><jsp:plugin attribute="значение"* > ... </jsp:plugin></code>	Генерирует тэги OBJECT или EMBED, в зависимости от типа браузера, в котором будет выполняться апплет, использующий Java Plugin.

В целом, все синтаксические конструкции таблицы 4.1 подразделяются на пять групп: **директивы, объявления, выражения, скриплеты и действия.**

Первая группа — **директивы JSP** распространяются на всю структуру класса, в который компилируется страница. Общий формат этой группы:

```
<%@ директива атрибут1="значение1"
    атрибут2="значение2"
    ...
    атрибутN="значениеN" %>
```

Существуют два основных типа директив:

- **page**, которая позволяет совершать такие операции, как импорт классов, изменение суперкласса сервлета и другие;
- **include**, которая даёт возможность вставлять файлы в тело JSP-страницы, при трансляции JSP файла в сервлет; эта директива имеет проблемы преобразования символов, поэтому ей нужно пользоваться с осторожностью.

Конкретный вариант использования директивы **page** показан в начале листинга 4.5. Общий список вариантов директивы **page** — следующий:

- `import="nakem.class1,...,nakem.classN"`.

Позволяет задать пакеты, которые должны быть импортированы. Например:

```
<%@ page import="java.util.*" %>
```

`import` - единственный атрибут, допускающий многократное применение.

- `contentType="MIME-Tun"` или
`contentType="MIME-Tun; charset=Кодировка-Символов"`

Задаёт тип MIME для вывода. По умолчанию используется text/html. К примеру, директива:

```
<%@ page contentType="text/plain" %>
```

приводит к тому же результату, что и использование скриплет:

```
<% response.setContentType("text/plain"); %>
```

- `isThreadSafe="true|false"`.

Значение **true** - "истина", принимается по умолчанию, задаёт нормальный режим выполнения сервлета, когда множественные запросы обрабатываются одновременно с использованием одного экземпляра сервлета. Значение **false** ("ложь") сигнализирует о том, что сервлет должен наследовать *SingleThreadModel* (однопоточную модель), при которой последовательные или одновременные запросы обрабатываются отдельными экземплярами сервлета.

- `session="true|false"`.

Значение **true** - "истина", принимается по умолчанию, сигнализируя, что заранее определённая переменная `session` типа `HttpSession` должна быть при-

вязана к существующей сессии, если таковая имеется. В противном случае, создаётся новая сессия, к которой и осуществляется привязка. Значение **false** ("ложь") определяет, что сессии не будут использоваться, и попытки обращения к переменной *session* приведут к возникновению ошибки при трансляции JSP страницы в сервлет.

- *buffer="размерkb|none"*.
Задаёт размер буфера для JspWriter out. Значение принимаемое по умолчанию зависит от настроек сервера, но должно превышать 8kb.
- *autoflush="true|false"*.
Значение **true**, принимаемое по умолчанию, устанавливает, что при переполнении буфер должен автоматически очищаться. Значение **false**, которое крайне редко используется, устанавливает, что переполнение буфера должно приводить к возникновению исключительной ситуации. При установке значения атрибута *buffer="none"*, установка значения **false** для этого атрибута недопустимо.
- *extends="накет.class"*.
Задаёт суперкласс для генерируемого сервлета. Этот атрибут следует использовать с большой осторожностью, поскольку возможно что сервер уже использует какой-нибудь суперкласс.
- *info="сообщение"*.
Задаёт строку, которая может быть получена при использовании метода *getServletInfo*.
- *errorPage="url"*.
Задаёт JSP страницу, которая вызывается в случае возникновения каких-либо событий Throwables, которые не обрабатываются на данной странице.
- *isErrorPage="true|false"*.
Сигнализирует о том, может ли эта страница использоваться для обработки ошибок для других JSP страниц. По умолчанию принимается значение **false** ("ложь").
- *language="java"*.
Данный атрибут предназначен для задания используемого языка программирования. По умолчанию принимается значение "java", поскольку на сегодняшний день это единственный поддерживаемый язык программирования.

Директива **include** позволяет включать файлы в процессе трансляции JSP страницы в сервлет. Ее использование имеет следующий формат:

```
<%@ include file="абсолютный или относительный url" %>
```


Рассмотрим четыре — наиболее часто используемые конструкции языка JSP, которые представлены таблицей 4.2.

Таблица 4.2 — Часто используемые конструкции языка JSP

Группа	Пояснение
<code><jsp:include page="url" /></code>	Подключение внешних файлов к странице JSP, в процессе обращения к ней.
<code><%! код %></code>	Объявление: объявление глобальных типов языка Java, в пределах JSP-страницы.
<code><%= выражение %></code>	Выражение языка Java, которое вычисляется и направляется на вывод в текстовом виде.
<code><% код %></code>	Скриплет: любой код на языке Java.

JSP-действие (`<jsp:include .../>`) является удобным, когда у нас имеются уже готовые HTML-страницы, которые можно включить в JSP-страницу. Например, файлы *Title.html* и *post1.html* можно включить в созданный шаблон двумя **действиями**:

```
<jsp:include page="/Title.html" />
<jsp:include page="post1.html"/>
```

Остальные три конструкции таблицы 4.2 вставляют код языка Java в JSP-страницу. Для эффективности их использования, в странице доступны четыре predefined типа объектов:

- **request** (тип `HttpServletRequest`) — объект запроса к сервелету;
- **response** (тип `HttpServletResponse`) — объект ответа клиенту;
- **session** (тип `HttpSession`) — ассоциируется с запросом, если таковой имеется;
- **out** (тип `PrintWriter`) - используется для отсылки выводимых клиенту данных.

Кроме predefined объектов, в JSP-странице могут быть объявлены любые типы языка Java. Например, если мы хотим подсчитывать число обращений к странице, то можем создать переменную *accessCount* в виде **объявления**:

```
<%! private int accessCount = 0; %>
```

затем, использовать **выражение** для самого подсчёта:

```
<%= ++accessCount %>
```

Если необходимо производить более сложные расчёты, используются конструкции **скриплетов**, например, вывод текущего времени и параметров

запроса для нашего проекта будет выглядеть:

```
<%
    out.println("Текущее время: " + new java.util.Date() + "<br>");

    out.println("Параметры запроса: <br>"
        + "key = " + request.getParameter("key") + "<br>"
        + "text = " + request.getParameter("text") );
%>
```

Перенесём эти примеры в созданный шаблон **post2.jsp**, как это показано на листинге 4.6.

Листинг 4.6 — Изменённый текст JSP-файла post2.jsp сервлета Example14

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Файл post2.jsp</title>
</head>
<body>
    <!-- Первое действие include -->
    <jsp:include page="/Title.html" />
    <p>
        Это тело JSP-страницы post2.jsp, <br>
    </p>

    <!-- Объявление -->
    <%! private int accessCount = 0; %>

    <!-- Выражение -->
    Количество обращений к странице:
    <%= ++accessCount %><br>

    <!-- Скриплет -->
    <%
        out.println("Текущее время: " + new java.util.Date()
            + "<br>");

        out.println("Параметры запроса:<br>"
            + "key = " + request.getParameter("key") + "<br>"
            + "text = " + request.getParameter("text"));
    %>

    <!-- Второе действие include -->
    <jsp:include page="post1.html" />
</body>
</html>
```

Теперь учтём, что форма, которая первоначально предоставляется клиенту методом **doGet(...)**, вызывает метод сервлета **doPost(...)**. Поэтому метод **doPost(...)** можно преобразовать так, чтобы он вызвал JSP-страницу **post2.jsp** (см. листинг 4.7).

Листинг 4.7 — Измененный метод **doPost(...)** сервлета *Example14*

```
/**
 * @see HttpServlet#doPost(HttpServletRequest request,
 * HttpServletResponse response)
 */
protected void doPost(HttpServletRequest request,
                        HttpServletResponse response)
                        throws ServletException, IOException {
    /**
     * Явная установка кодировок объектов запроса и ответа.
     * Стандартная установка контекста ответа.
     */
    request.setCharacterEncoding("UTF-8");
    response.setCharacterEncoding("UTF-8");
    response.setContentType("text/html");

    /**
     * Стандартное подключение ресурса сервлета.
     */
    RequestDispatcher disp =
        request.getRequestDispatcher("/WEB-INF/post2.jsp");
    disp.forward(request, response);
}
```

Теперь, после запуска сервлета, можно заполнить форму запроса, например, как показано на рисунке 4.18.

Example14.java post2.jsp Форма запроса

http://localhost:8080/proj14/Example14

Запрос к таблице ведения записей

Введи ключ :

Введи текст:

Рисунок 4.24 — Заполнение формы запроса при первом запуске сервлета

Нажав кнопку «*Отправить*», мы получим ответ (см. рисунок 4.25).

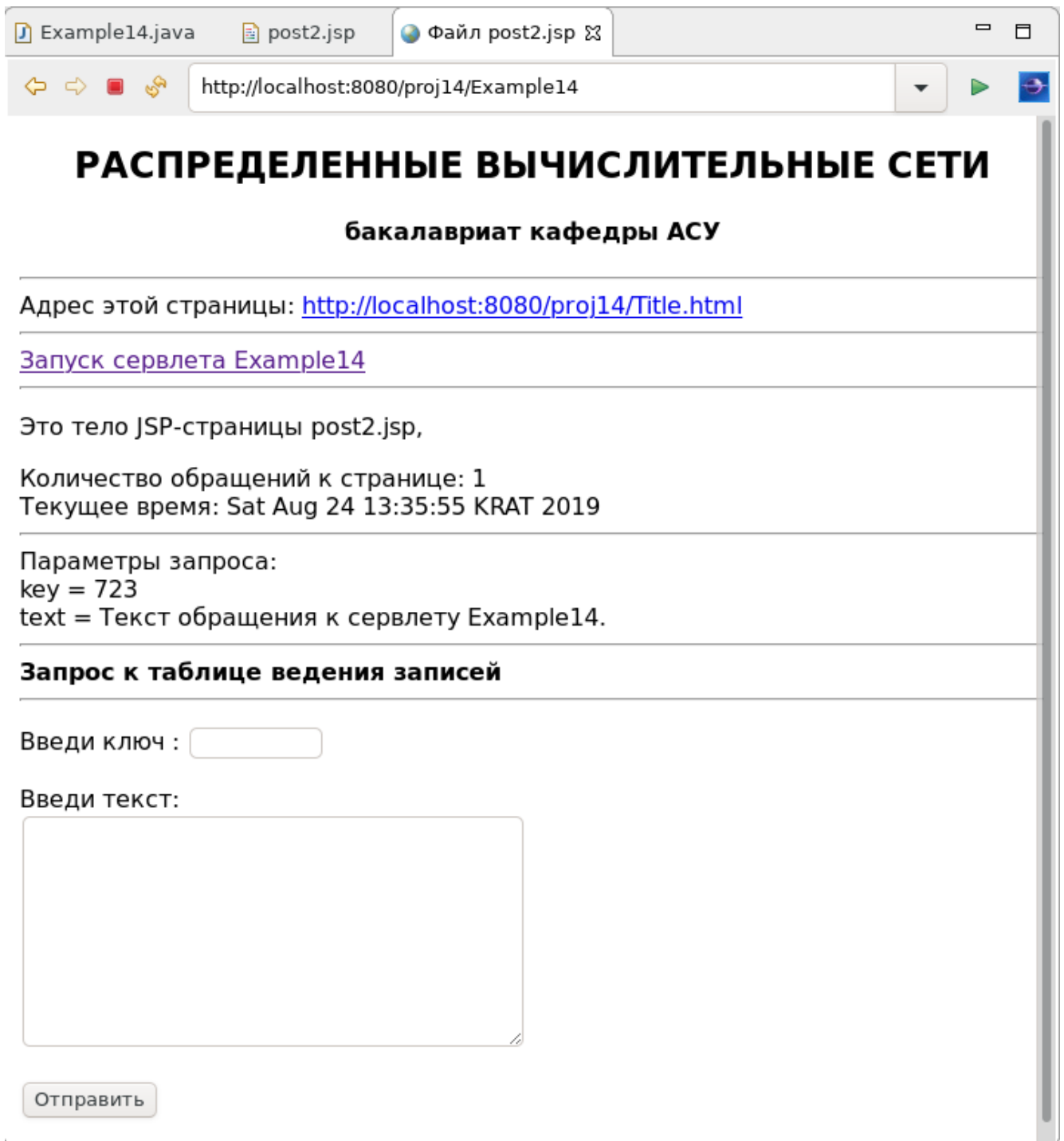


Рисунок 4.25 — Ответ сервлета методом doPost(...)

Напомним, что сама JSP-страница ***post2.jsp***, при первом обращении к ней, компилируется в сервлет и кэшируется сервером Apache Tomcat. Это делает технологию JSP-страниц очень мощной и приемлемой для разработки приложений уровня предприятий.

4.3.5 Модель MVC

Практическая реализация любой РВ-сети начинается с создания некоторой первичной схемы проекта, которая бы создавала основу для дальнейшего функционального наполнения системы. Такая схема должна создаваться на основе принципов, обеспечивающих простоту и надёжность последовательного процесса реализации окончательного варианта системы.

Теоретическим универсальным принципом разработки сложных систем является подход, предполагающий разделение ее на ряд специализированных подсистем, удовлетворяющих следующим двум признакам:

- большая функциональная часть таких подсистем может реализовываться и развиваться независимо от других частей;
- имеется простое и понятное взаимодействие между подсистемами.

Таким признакам удовлетворяет проектная схема MVC, которую мы и рассмотрим в данном пункте, связав ее с технологией сервлетов:

- **MVC** [*Model View Controller*] - это *шаблон проектирования*, который впервые был опубликован в 1970-х годах. Он представляет собой образец архитектуры программного обеспечения, основанный на *принципе разделения представления данных и функционала*, где эти данные формируются и обрабатываются.
- **MVC** — это *аббревиатура*, состоящая из трёх слов:
 - *модель* [Model];
 - *вид* (представление) [View];
 - *контроллер* [Controller].

Модель — некое хранилище (поставщик) данных в рамках всего проекта. Главные задачи модели заключаются в предоставлении доступа к данным для их просмотра или актуализации (добавления, редактирования, удаления).

Представление — это компонент MVC, где выполняется вывод данных на экран. При классическом подходе к web-разработкам в представлении будет формироваться HTML код.

Контроллер (контроллеры) — это компонент MVC, предназначенный для связи между моделями и представлениями, а также для обработки данных, которые пришли от пользователя к серверу через формы запроса и другие источники. После того, как контроллер получил информацию, в зависимости от необходимости задачи, он передаст данные в представление для вывода или в модель для актуализации (добавления, редактирования, удаления).

Взаимодействие работы перечисленных компонент, данного шаблона, можно представить рисунком 4.26, на котором каждая из компонент реализуется средствами сервера Apache Tomcat:

- **модель** (model) – некоторое приложение, поставляющее данные в сервлет;
- **представление** (view) – набор JSP-страниц, подготавливающий для сервлета ответ клиенту (браузеру);
- **контроллер** (controller) – сервлет, который организует доступ к приложению модели, принимает от него данные, передает на подготовку HTML-страниц в службу представления и возвращает результат клиенту (браузеру).

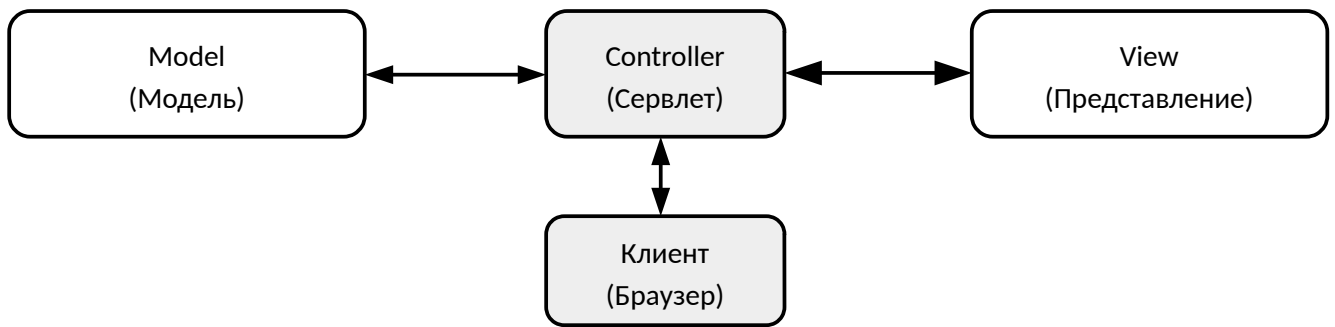


Рисунок 4.26 — Трёхзвенная архитектура, реализуемая моделью MVC

Представленная трёхзвенная архитектура РВ-сети легко может быть реализована в нашем проекте. Для этого воспользуемся, например, классом *NotePad*, который находится в JAR-архиве *\$HOME/lib/notepad.jar*, а чтобы наш сервлет имел доступ к этому архиву, мы:

- выделим мышкой проект *proj14*;
- правой кнопкой мышки активируем меню и выберем *Build Path* → *Configure Build Path*;
- в появившемся окне добавим внешний JAR-архив, как это показано на рисунке 4.27.

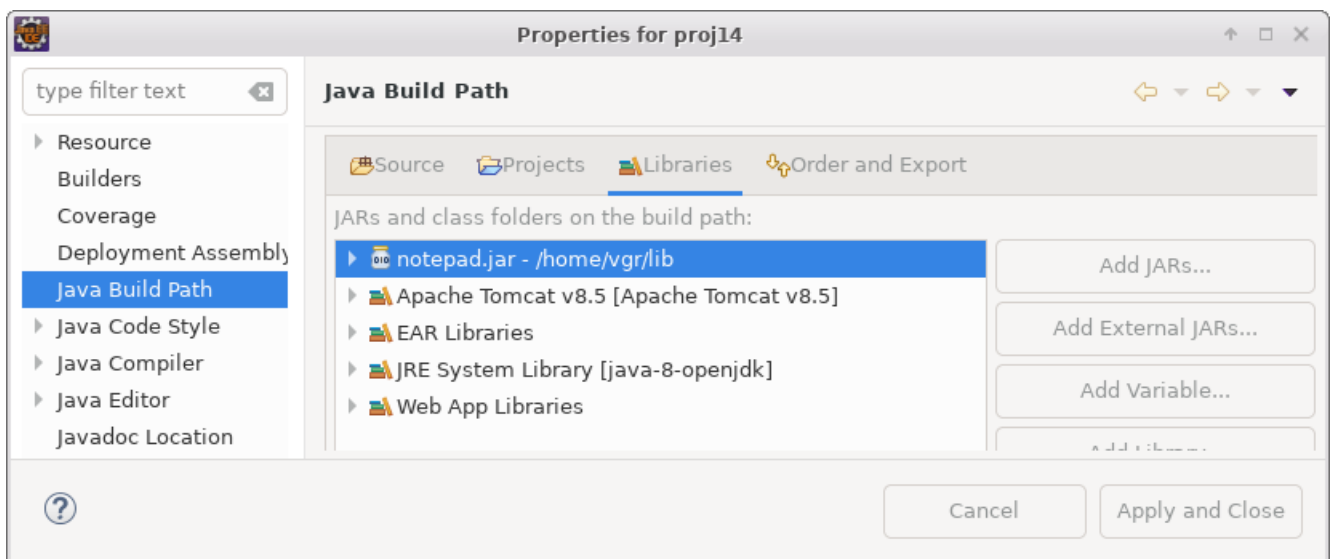


Рисунок 4.27 — Подключение архива notepad.jar к проекту proj14

Дополнительно, необходимо подключить архив *\$HOME/lib/notepad.jar* к серверу Apache Tomcat, иначе он не сможет стартовать по причине невозможности загрузить класс *NotePad*. Для этого необходимо:

- в Project Explorer выделить используемый сервер;
- из главного меню Eclipse EE выбрать *Run* → *Run Configurations...*;
- в появившемся окне (см. рисунок 4.28), на вкладке *Classpath* подключить внешний архив *notepad.jar*.

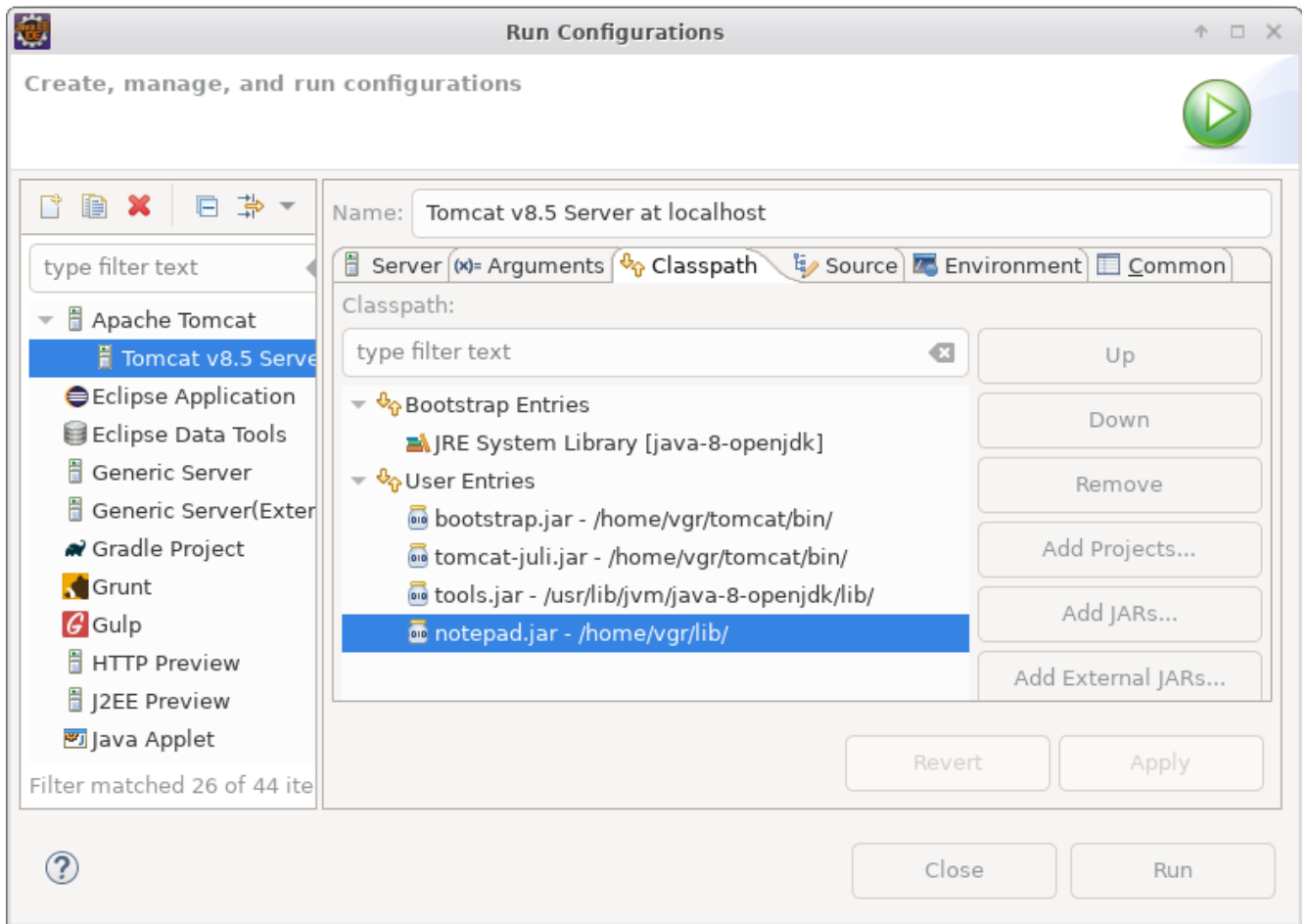


Рисунок 4.28 — Подключение архива notepad.jar к среде запуска сервера Apache Tomcat

Теперь все готово для реализации проекта по схеме трёхзвенной РВ-сети. Исключительно для целей демонстрации, ограничим объем реализации нашего примера только чтением, с помощью класса **NotePad**, содержимого таблицы **notepad** базы данных **exampleDB** и предоставлением его пользователю в виде списка отдельных записей.

Действуя по схеме MVC, сначала обеспечим доступ к модели, что реализуется с помощью следующих статических конструкций языка Java, добавляемых в начало сервлета **Example14**:

- **NotePad notepad** — ссылка на объект, реализующий доступ к БД;
- **boolean isOpen** — статус состояния класса **NotePad**;
- **boolean isError** — статус ошибки создания объекта класса **NotePad**;
- **boolean dbOpen()** — метод создания объекта класса **NotePad** и контроля его статуса, который возвращает значение **true**, если дальнейшие действия с БД являются допустимыми;
- **void dbClose()** — закрывает объект класса **NotePad** и устанавливает статус доступа, что — необходимо, если будет переопределяться метод сервлета **destroy()**;
- **String[] dbList()** — метод получения списка записей БД.

Использование указанных статических конструкций обосновано тем, что запросы могут осуществляться многими пользователями параллельно, а открытый объект *notepad* должен быть один. Указанные объекты добавлены в начало сервлета *Example14* и представлены на листинге 4.8.

Листинг 4.8 — Новые статические конструкции сервлета *Example14*

```
/**
 * Servlet implementation class Example14
 */
@WebServlet("/Example14")
public class Example14 extends HttpServlet {
    private static final long serialVersionUID = 1L;

    /**
     * Объекты доступа к классу Notepad
     */
    private static Notepad notepad = null;
    private static boolean isOpen = false;
    private static boolean isError = false;

    /**
     * Открытие объекта Notepad.
     * @return
     */
    protected static boolean dbOpen()
    {
        if(isError)
            return false;
        if(!isOpen)
            notepad = new Notepad();
        if(notepad == null)
        {
            isError = true;
            System.out.println("Не могу открыть Notepad");
            return false;
        }else
        {
            isOpen = notepad.isConnected();
            if(isOpen)
                System.out.println("Notepad - открыта!");
            else
                System.out.println("Notepad - не открыта!");

            return isOpen;
        }
    }

    /**
     * Закрытие объекта Notepad.
     */
    protected static void dbClose()
    {
        if(isError || !isOpen)
            return;
        notepad.setClose();
        isOpen = false;
        notepad = null;
    }
}
```



```

/**
 * Получение списка записей объекта Notepad.
 * @return String[]
 */
protected static String[] dbList()
{
    if(!dbOpen())
        return null;

    Object[] obj = notepad.getList();

    if(obj == null)
    {
        System.out.println("getList() вернул null");
        return null;
    }
    int ns =
        obj.length;
    System.out.println("Получено " + ns + " строк(и)");
    String[] ss =
        new String[ns];

    for(int i=0; i < ns; i++)
        ss[i] = obj[i].toString();

    return ss;
}
// Далее идут стандартные методы класса Example14.

```

Теперь перейдём к реализации проектной части представления. Для этого учтём, что методы запроса **request** позволяют взаимодействовать сервлету и JSP-странице посредством передачи значений атрибутов:

- `request.setAttribute(String name, Type value)` — устанавливает значение атрибута с именем **name**;
- `request.getAttribute(String name)` — читает значение атрибута с именем **name**.

Новое представление для метода **doPost(...)** создадим в виде JSP-страницы **post3.jsp**, показанной на листинге 4.9.

Листинг 4.9 — Исходный текст post3.jsp сервлета Example14

```

<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Страница post3.jsp</title>
</head>
<body>
    <!-- Первое действие include -->
    <jsp:include page="/Title.html" />
    <p>

```

Список сообщений таблицы notepad:


```

</p>
<ul>

<!-- Объявление --%>
<%! private int ns = 0;%>

<!-- Скриплет --%>
<%
    String[] ss =
        (String[])request.getAttribute("list");
    if(ss == null)
        out.println("<li>Нет данных, возможно - ошибка БД!</li>");
    else
    {
        ns = ss.length;

        for(int i=0; i<ns; i++)
            out.println("<li>" + ss[i] + "</li>");
    }
%>
</ul><hr>

Получено <%=ns %> строк(и)<br>

<!-- Второе действие include --%>
<jsp:include page="post1.html" />

</body>
</html>

```

Завершаем реализацию данного примера посредством изменения метода **doPost(...)**, показанного на листинге 4.10.

Листинг 4.10 — Изменённый метод doPost(...) сервлета Example14

```

/**
 * @see HttpServlet#doPost(HttpServletRequest request,
 * HttpServletResponse response)
 */
protected void doPost(HttpServletRequest request,
                        HttpServletResponse response)
                        throws ServletException, IOException {

    System.out.println("Вызывается метод doPost(...)");
    /**
     * Явная установка кодировок объектов запроса и ответа.
     * Стандартная установка контекста ответа.
     */
    request.setCharacterEncoding("UTF-8");
    response.setCharacterEncoding("UTF-8");
    response.setContentType("text/html");
    /**
     * Обращение к модели и установка результата
     * в виде атрибута к JSP-странице:
     */
    request.setAttribute("list", dbList());
    /**
     * Далее, на основе анализа request должны
     * реализовываться методы добавления и

```

```

    * удаления строк таблицы notepad БД exampleDB.
    */
/**
    * Стандартное подключение ресурса сервлета.
    */
    RequestDispatcher disp =
        request.getRequestDispatcher("/WEB-INF/post3.jsp");
    disp.forward(request, response);
}

```

Начальный результат запуска изменённого сервлета **Example14** соответствует форме запроса приведённого ранее на рисунке 4.24, а после отправки сообщения, браузер покажет список записей таблицы **notepad**, подобно рисунку 4.29.

Example14.java Страница post3.jsp

http://localhost:8080/proj14/Example14

РАСПРЕДЕЛЕННЫЕ ВЫЧИСЛИТЕЛЬНЫЕ СЕТИ

бакалавриат кафедры АСУ

Адрес этой страницы: <http://localhost:8080/proj14/Title.html>

[Запуск сервлета Example14](#)

Список сообщений таблицы notepad:

- 124 Разработал программу для технологии RMI.
- 125 Исправил исходные тексты.
- 126 Стартовал RMI-сервер из архива ~/lib/rmipadserver.jar.

Получено 3 строк(и)

Запрос к таблице ведения записей

Введи ключ :

Введи текст:

Рисунок 4.29 — Форма отображения с помощью JSP-страницы post3.jsp

Таким образом, мы завершили заявленный объем реализации приложения с трёхзвенной архитектурой РВ-сети, которое естественным образом может быть развито для поддержки остальных функций класса **NotePad**: добавление и удале-

ние записей таблицы *notepad* базы данных *exampleDB*. И, в плане этой реализации, мы ограничимся только следующими замечаниями:

- для использования объекта класса *NotePad*, в сервлет *Example14* были включены дополнительные статические объекты, которые необходимы для синхронизации обращений многих клиентов к встроенной базе данных; принципиально, можно было бы создать новый класс, решающий как задачу синхронизации, так и предоставляющий нужные методы без «загромождения» текста самого сервлета; такой класс должен быть помещён в поддерево каталога *WEB-INF*, чтобы он был недоступен для прямого вызова программой клиента;
- представление *post1.jsp* метода *doGet(...)* преследует чисто учебные цели, поэтому его можно и нужно заменить на более информативное содержание, описывающее и рекламирующее само распределенное приложение; в любом случае, оно должно содержать форму, хотя бы в виде одной кнопки, переводящей запросы клиента на метод *doPost(...)*;
- полная реализация метода *doPost(...)* предполагает чтение параметров запроса и их анализ, а, как следствие, это порождает различные обращения к объекту класса *NotePad* и вызов различных JSP-страниц, соответствующих протоколу диалога клиента и сервлета.

Рассмотренным примером завершается учебный материал главы, посвящённой web-технологиям распределенных систем. Наряду с общим обзором, включающим мотивы ее появления и тенденции развития, нами были изучены и подкреплены конкретными примерами: модели развития базовой архитектуры «Клиент-сервер», а также мощная и перспективная технология Java-сервлетов.

В процессе своего развития, web-технологии всегда претендовали на нечто большее, чем простой информационный обмен HTML-страницами. Технология Java-сервлетов наглядно продемонстрировала возможность создания распределённых РВ-сетей уровня предприятия с использованием «Тонкого клиента». И хотя язык Java представляет лишь часть инструментальных средств создания РВ-сетей, он занимает достойное место среди других и может быть более модных, инструментальных технологий.

В заключении данной главы отметим, что рассмотренные нами инструменты создания РВ-сетей лишь актуализировали проблему адресации все возрастающего объёма распределенных приложений. Теоретические и практические подходы к решению этой проблемы рассмотрим в следующей главе.

Вопросы для самопроверки

1. Что такое — URI и его формы представления?
2. Чем отличается URI от URL?
3. В чем состоит назначение языка HTML и его отличие от языка XML?
4. Что такое - «Тонкий клиент» и как это понятие связано с АРМ?
5. Когда были сформированы стандарты на технологию World Wide Web?
6. Какой уровень стека протоколов TCP/IP описывает протокол HTTP?
7. Объясните назначение и соотношение терминов JavaScript, Applet и Servlet?
8. Какие методы запросов клиента к серверу обеспечивает протокол HTTP?
9. Какие три уровня предлагает вертикальная модель распределения классической архитектуры «Клиент-сервер»?
10. В чем состоит назначение горизонтальной модели распределения классической архитектуры «Клиент-сервер»?
11. Что такое — двухзвенная архитектура «Клиент-сервер»?
12. Что такое — трёхзвенная архитектура «Клиент-сервер» и в чем состоит ее отличие от двухзвенной архитектуры?
13. Что такое — технология CGI?
14. В чем состоит назначение Apache Tomcat?
15. Что такое Servlet и какой абстрактный класс он порождает?
16. Какой пакет Java содержит ПО сервлетов?
17. Какие два основных метода запросов формирует браузер к web-серверу?
18. Какие два базовых метода использует класс HttpServlet?
19. Что такое — JSP-страница и для чего она предназначена?
20. Что делает сервер с JSP-страницей, когда она вызывается сервлетом?

5 Тема 5. Сервис-ориентированные архитектуры

В подразделе 1.3 первой главы, уже была дана достаточно ёмкая характеристика понятию «*Сервис-ориентированные технологии*». В частности, на рисунке 1.11 была показана обобщённая схема соотношения *объектно-ориентированного* и *сервисного* подходов в создании прикладных систем. Дополнительно, было дано разъяснение понятию «*Сервис-ориентированная архитектура*» и была отмечена значимость систем типа **Middleware**, использующих модель **SOA** и обеспечивающих взаимодействие распределённых приложений на основе протокола **SOAP**.

В общем случае, тема сервисно-ориентированных технологий, далее — **COT**, требует отдельного учебного пособия, раскрывающего все многообразие подходов и методов реализованных в них. В данной главе мы отметим, что COT является дальнейшим продолжением и развитием ранее изученной концепции «*Объектные распределённые системы*», а также основным трендом современного развития компьютерных технологий.

Учебный материал данной главы ограничен теоретическим описанием двух аспектов COT: общей концепции SOA и частных подходов к реализации сервисных технологий.

Общая концепция SOA охватывает теоретические аспекты рассматриваемой технологии, которые детализируются в трёх направлениях:

- в первом направлении рассматриваются *вопросы связывания* распределённых программных систем;
- во втором направлении обсуждаются web-сервисы, которые условно подразделяются на первое и второе поколения;
- в третьем направлении рассматриваются три брокерные архитектуры, которые в настоящее время используются web-сервисами.

В подразделе, посвящённом частным подходам к реализации сервисных технологий, основное внимание уделяется следующим направлениям:

- технология одноранговых сетей становится все более популярной в различных общественных сервисах ориентированных на культуру, личные контакты и индустрию развлечений; первичным здесь является отказ от централизованных средств учёта и управления адресами ресурсов Интернет;
- технология GRID, которая ориентирована на создание некоторого большого «*виртуального суперкомпьютера*», обслуживающего множество организаций и научных коллективов;
- технология облачных вычислений, ориентированных на учёт используемых ресурсов ЭВМ и обеспечивающих построение как коммерческих, так и общественных распределённых сервисных ресурсов.

В целом, обсуждаемые технологии рассматриваются в теоретическом плане и не привязываются к конкретным языкам программирования или конкретным фреймворкам.

5.1 Концепция SOA

Суммируя уже изученный материал подраздела 1.3, можно утверждать, что **SOA** или «Сервис-ориентированная архитектура» - это парадигма организации и использования распределенных возможностей приложений, которые принадлежат различным владельцам сервисов.

SOA — это парадигма, которая расширяет архитектуру объектных распределённых систем, выделяя модель независимого компонента. Этот компонент не обязан присутствовать на каждой стороне участников распределенного взаимодействия, как это должен делать класс, реализующий используемый удалённый объект.

Базовыми составляющими SOA, как модели, являются: *сервисные компоненты* (сервисы), *интерфейсы сервисов*, *соединители сервисов* и *механизмы обнаружения сервисов*.

Сервисные компоненты (или сервисы) описываются программными компонентами, которые обеспечивают прозрачную сетевую адресацию.

Интерфейс сервиса обеспечивает описание возможностей и качества предоставляемых сервисом услуг. В таком описании определяется формат сообщений, используемых для обмена информацией, а также входные и выходные параметры методов, поддерживаемых сервисным компонентом.

Соединитель сервисов — это транспорт, обеспечивающий обмен информацией между отдельными сервисными компонентами.

Механизмы обнаружения сервисов предназначены для поиска сервисных компонентов, обеспечивающих требуемую функциональность сервиса. Среди всего множества вариантов, обеспечивающих обнаружения сервисов, выделяются две основные категории: системы *динамического* и *статического* обнаружения.

Статические системы обнаружения сервисов, например UDDI, ориентированы на хранение информации о сервисах в редко изменяющихся системах.

Динамические системы обнаружения сервисов ориентированы на системы, в которых допустимо частое появление и исчезновение сервисных компонентов.

Исторически, взаимодействие между поставщиками и потребителями сервисов основывалось на протоколе **SOAP** [22], который, в свою очередь, опирается на язык **XML** [50]. Сам протокол SOAP считается независимым от языка и платформы, хотя само взаимодействие проходит путь *XML->HTTP/HTML->TCP/IP*. Такая ситуация привела к понятию «*Веб-сервисы*», которое и рассматривается в большинстве литературных источников. Подобную интерпретацию будем использовать и мы, хотя сама теория не ограничивает базу реализации сервисов, допуская применение моделей CORBA, RMI и другие технологии сетевого взаимодействия. В частности, большинство современных веб-сервисов для передачи своих сообщений применяют протокол HTTP, поскольку он экономит трафик передачи данных по сравнению с простой передачей текста на языке XML.

5.1.1 Связывание распределённых программных систем

Одной из характеристик РВ-сетей является **связанность** их сервисов. По этому признаку распределённые программные системы подразделяются на два типа: «Сильносвязанные системы» (Strong coupling) и «Слабосвязанные системы» (Loose coupling).

Сильная связанность возникает при использовании объектных распределённых сервисов, когда зависимый класс содержит ссылку на конкретный класс, предоставляющий сервис.

Слабая связанность возникает, когда зависимый класс содержит ссылку на интерфейс, который может быть реализован одним или многими различными классами.

Очевидно, что использование концепции слабосвязанных программных систем уменьшает количество зависимостей между сервисными компонентами. Это, в свою очередь, уменьшает объем возможных последствий, порождённых сбоями или модификациями распределённых систем. В таблице 5.1, заимствованной из источника [5], приведено сравнение ряда общих характеристик слабосвязанных и сильносвязанных систем.

Таблица 5.1 — Сравнение слабосвязанных и сильносвязанных систем

	Сильносвязанные системы	Слабосвязанные системы
Физические соединения	Точка-точка	Через посредника
Стиль взаимодействий	Синхронные	Асинхронные
Модель данных	Общие сложные типы	Простые типы
Связывание	Статическое	Динамическое
Платформа	Сильная зависимость от базовой платформы платформы	Независимость от базовой платформы платформы
Развертывание	Одновременное	Постепенное

Общий вывод, следующий из анализа связанности, проектируемых систем, состоит в том, что:

- *традиционный подход* разработки распределённых приложений обычно значительно усложняет создание и сопровождение РВ-сетей;
- *веб-сервисы*, используя слабую связанность сервисов, позволяют значительно упростить координацию распределённых систем и их реконфигурацию.

5.1.2 Web-сервисы первого и второго поколений

Первое поколение веб-сервисов (до 2007 года) опиралось на парадигму XML веб-служб, позволяющих создавать независимые масштабируемые слабосвязанные приложения. Для этого использовались три основных стандарта: WSDL, UDDI и SOAP, образующие так называемый «Треугольник SOA», показанный на рисунке 5.1.



Рисунок 5.1 - Взаимодействие между клиентом и поставщиком веб-сервиса [5]

Согласно общим представлениям [59]: «... **UDDI** (англ. *Universal Description Discovery & Integration*, ...) — инструмент для расположения описаний веб-сервисов (WSDL) для последующего их поиска другими организациями и интеграции в свои системы. UDDI это кроссплатформенное программное обеспечение, основанное на XML. UDDI является открытым проектом, спонсируемым OASIS, который позволяет организациям публиковать описания веб-сервисов (WSDL) для последующего их поиска другими организациями и интеграции в свои системы, а также определять, как сервисы или приложения взаимодействуют через Internet. UDDI был первоначально предложен в качестве основного веб-сервис стандарта. Он предназначен для опроса SOAP сообщениями и для обеспечения доступа к ... документам, описывающим привязки протоколов и форматов сообщений, необходимых для взаимодействия с веб-услугами, перечисленными в его каталоге. ...».

Соответственно [60]: «... **WSDL** (англ. *Web Services Description Language*) — язык описания веб-сервисов и доступа к ним, основанный на языке XML. Последняя официальная спецификация на момент написания статьи версия 2.0 (WSDL Version 2.0 от 26 июня 2007 года), которая имеет статус рекомендации, и версия 1.1 (WSDL Version 1.1 от 15 марта 2001 года), которая имеет статус заметки (note). ...».

Протокол SOAP [22] обеспечивает непосредственную передачу сообщений между клиентом сервиса поставщиком сервиса. Сообщения передаются XML-конвертами (**Envelope**), а общая структура которых показана на рисунке 5.2, где:

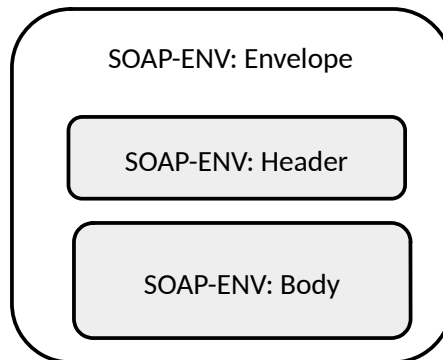


Рисунок 5.2 — Структура сообщения протокола SOAP

- **Envelope** – Корневой элемент, определяющий сообщение и пространство имен, использованное в документе.
- **Header** – Содержит атрибуты сообщения, такие как: информация о безопасности или о сетевой маршрутизации.
- **Body** – Содержит сообщение, которым обмениваются приложения.
- **Fault** – Необязательный элемент, который предоставляет информацию об ошибках, произошедших при обработке сообщений.

Качественные характеристики веб-сервисов первого поколения состоят из следующих положений:

- контент сервисов *формируется поставщиками сервисов*;
- *отсутствуют единые стандарты* протоколов авторизации и аутентификации пользователей, что требовало от поставщиков сервиса проводить самостоятельные разработки;
- *отсутствует понятие состояния* сервиса и после обращения клиента к серверу, его состояние на сервере не сохраняется.

Начиная с 2004 года начинают публиковаться и внедряться различные стандарты, повышение качество работы веб-сервисов. К ним относятся:

- **WS-Security** – обеспечение безопасности веб-сервисов;
- **WS-Addressing** – маршрутизация и адресация SOAP-сообщений;
- **WSRF, WS-Notification** – работа с состоянием веб-сервисов.

Внедрение перечисленных стандартов позволило многим разработчикам говорить о «**Втором поколении веб-сервисов**», поскольку теперь:

- контент сервисов *формируется пользователями сервисов*;
- *используются единые стандарты* протоколов авторизации и аутентификации пользователей;
- в разработках сервисов начинает использоваться «*Состояние сервиса*».

5.1.3 Брокерные архитектуры web-сервисов

Излишне напоминать, что «Треугольник SOA», показанный на рисунке 5.1, демонстрирует взаимодействие клиента и сервера с помощью посредника (брокера). В целом, выделяется три разновидности такого взаимодействия: *прямой вызов* через UDDI, *синхронный вызов* через посредника и *асинхронный вызов* через посредника. Рассмотрим каждый из них.

Прямой вызов через UDDI предполагает развёртывание «Службы адресов» по хорошо известному потребителю адресу. Потребитель сервиса может использовать UDDI для поиска нужных ему Web-служб (Провайдеров сервиса), демонстрируется рисунком 5.3.

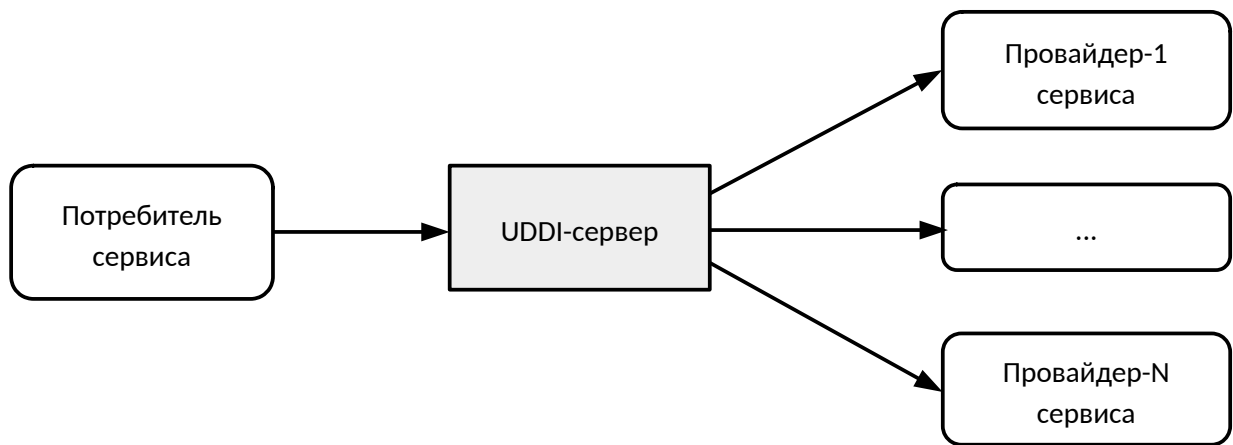


Рисунок 5.3 — Прямой вызов через UDDI

Получив адрес, потребитель самостоятельно обращается к провайдеру сервиса и ведёт с ним собственный диалог, а такой подход имеет ряд недостатков, которые можно сформулировать следующим образом:

- Потребитель *должен знать **URI конечной точки*** провайдера для вызова службы. Он использует UDDI как каталог для поиска этого URI.
- Если имеется несколько провайдеров, UDDI содержит несколько URI, и потребитель должен выбрать один из них.
- Если провайдер сервиса меняет URI конечной точки, он должен повторно зарегистрироваться на сервере UDDI для того, чтобы UDDI хранил новый URI. Потребитель должен повторно запросить UDDI для получения нового URI.

В сущности, использование прямого вызова UDDI означает, что каждый раз, когда потребитель хочет вызвать службу, он должен запросить URI конечных точек в службе UDDI. Такой подход также вынуждает потребителя самостоятельно оценивать качество провайдера каким-либо способом.

Синхронный вызов через посредника предполагает использование «интеллектуального брокера», который способен самостоятельно проводить оценку провайдеров сервиса на предмет их присутствия в сети, качества обслуживания пользователей и загруженности запросами.

Потребитель вызывает прокси-службу такого брокера, который, в свою очередь, оценивает загруженность провайдеров, выбирает одного из них и передаёт UDDI. UDDI возвращает только один URI, и потребитель не должен делать выбор. Потребитель даже может и не знать, что окончательная точка является прокси-службой. Он знает только о том, что может использовать этот URI для вызова Web-службы.

Взаимодействие потребителя, UDDI-службы и провайдеров такого сервиса показано на рисунке 5.4.

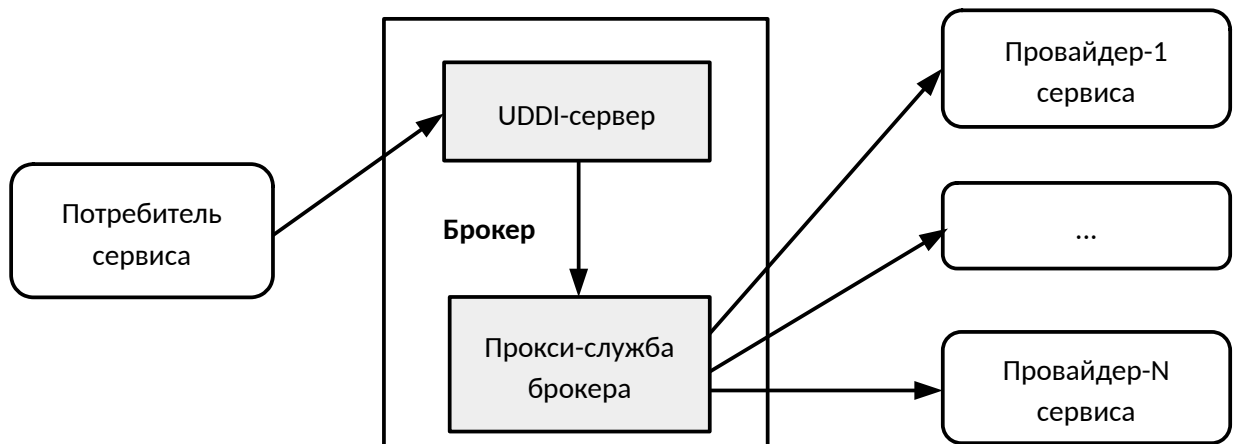


Рисунок 5.4 — Синхронный вызов через посредника

В РВ-сети слабосвязанных приложений использование синхронного вызова, также имеет ряд недостатков:

- Потребитель должен ждать окончания выполнения службы провайдера. На это время поток исполнения клиента должен быть заблокирован.
- Если служба провайдера выполняется длительное время, то потребитель может прекратить ожидание ответа.
- Имеются случаи, когда потребитель выполняет запрос, но не может ждать ответ провайдера сервиса.
- Если у потребителя возникает аварийная ситуация во время блокировки его работы, то ответ будет потерян и вызов сервиса нужно будет повторить.

Асинхронный вызов через посредника решает многие проблемы, связанные как с реализацией брокера, так и с организацией взаимодействия потребителя (клиентское приложение) провайдера (поставщика сервиса).

Брокер, используя свои оценки провайдеров сервиса, разрешает или ограни-

чивает им доступ к двум очередям запросов и ответов. Поэтому, провайдеры сервисов получают доступ к запросам клиентов, конкурируя между собой и с учётом своих возможностей обслуживать клиента, что и показано на рисунке 5.5.

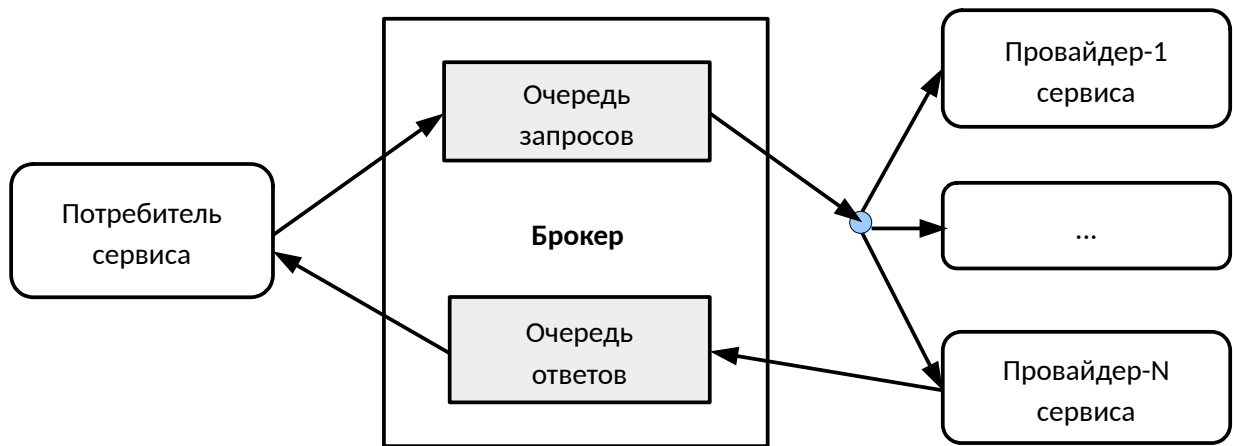


Рисунок 5.5 — Асинхронный вызов через посредника

При таком подходе, потребитель использует *два асинхронных канала* связи с брокером. По одному каналу передаёт запрос брокеру, который ставит его в свою очередь запросов. По другому каналу, потребитель асинхронно забирает ответ из очереди ответов брокера.

Преимущества асинхронного подхода является наиболее предпочтительным для создания РВ-сетей:

- Потребитель *не должен блокировать* свою работу при ожидании ответа и может в это время выполнять другую работу, поэтому потребитель намного менее чувствителен к продолжительности работы службы у провайдера.
- Брокер, предоставляющий возможность потребителю вызывать Web-службу асинхронно, реализуется при помощи технологии хорошо развитых систем обмена сообщениями.
- Пара очередей сообщений выступает как один адрес, использующийся любым потребителем сервиса независимо от количества обслуживающих их провайдеров.

Завершая краткий теоретический обзор концепции SOA, отметим, что «Сервис-ориентированные технологии» не ограничиваются только веб-службами. Просто, веб-службы являются наиболее «модным» современным направлением, который пока ещё не достиг пика своих возможностей. Тем не менее, мы должны помнить, что слабо связанные приложения и сервисы, не сохраняющие состояние запросов клиентов, больше соответствуют публичной сфере использования РВ-сетей. Большинство же приложений требуют учёта поставщиками сервисов своих состояний, что значительно усложняет как их разработку, так и сопровождение.

5.2 Частные подходы к реализации сервисных технологий

В этом подразделе приведён краткий обзор трех технологических направлений, которые напрямую не ассоциируют себя с веб-сервисами, но в практическом плане являются реализациями сервисных технологий. К ним относятся: *технологии одноранговых сетей*, *технологии Grid* и *облачные вычисления*. Более подробный обзор можно найти, например, в источнике [5].

5.2.1 Технология одноранговых сетей

Технология одноранговых или P2P сетей обеспечивает взаимодействие приложений РВ-сетей на базе принципа децентрализации, когда разделение вычислительных ресурсов и сервисов производится напрямую посредством прямого взаимодействия между участниками сети друг с другом. В этом отношении «Одноранговые вычислительные сети» являются противоположностью «строгим» клиент-серверным архитектурам, таким как CORBA, RMI, RPC и другими. Согласно [61]: «... **Одноранговая, децентрализованная, или пиринговая** (англ. *peer-to-peer*, *P2P* — равный к равному) **сеть** — оверлейная компьютерная сеть, основанная на равноправии участников. Часто в такой сети отсутствуют выделенные серверы, а каждый узел (peer) является как клиентом, так и выполняет функции сервера. В отличие от архитектуры клиент-сервера, такая организация позволяет сохранять работоспособность сети при любом количестве и любом сочетании доступных узлов. Участниками сети являются все пиры. ...». Считается, что положительные качества P2P-сетей определяются свойствами:

- отсутствия зависимости от централизованных сервисов и ресурсов;
- возможностью системы пережить серьёзное изменение в структуре сети;
- высокой масштабируемостью модели одноранговых вычислений.

Основным элементом является **Пир** (Peer), который считается фундаментальным составляющим блоком, который может быть двух типов:

- **простым пиром**, обеспечивающим работу конечного пользователя, предоставляя ему сервисы других пиров и обеспечивая предоставление ресурсов другим участникам сети;
- **роутером**, который реализует механизм взаимодействия между пирами, отделёнными от сети брандмауэрами или NAT-системами.

Сам пир:

- имеет уникальный идентификатор;
- принадлежит одной или нескольким группам;
- может взаимодействовать с другими пирами как в своей, так и в других группах.

Согласно источнику [5]: «... **Группа пиров** – это набор пиров, сформированный для решения общей задачи или достижения общей цели. Группы пиров могут предоставлять членам своей группы такие наборы сервисов, которые недоступны пирам, входящим в другие группы. **Сервисы** – это функциональные возможности, которые может привлекать отдельный пир для полноценной работы с удалёнными пирами. В качестве примера сервисов, которые может предоставлять отдельный пир можно указать сервисы передачи файлов, предоставления информации о статусе, проведения вычислений и др. **Сервисы пира** – это такие сервисы, которые может предоставить конкретный узел P2P. Каждый узел в сети P2P предоставляет определённые функциональные возможности, которыми могут воспользоваться другие узлы. Эти возможности зависят от конкретного узла и доступны только тогда, когда узел подключён к сети. Как только узел отключается, его сервисы становятся недоступны. **Сервисы группы** – это функциональные возможности, предоставляемые группой входящим в неё узлам. Возможности могут предоставляться несколькими узлами в группе, для обеспечения избыточного доступа к этим возможностям. Как только к группе подключается узел, обеспечивающий необходимый сервис, он становится доступным для всей группы. ...».

P2P-сети имеют свои протоколы. Один из таких протоколов реализовала компания Sun Microsystems в рамках проекта JXTA [62]: «... **JXTA (Juxtapose)** — спецификации протоколов по обслуживанию P2P-сетей для обмена данными различного типа. Проект был запущен корпорацией Sun Microsystems в 2001 для решения проблем, стоящих на пути развития пиринговых сетей. JXTA использует открытые протоколы XML и может быть реализован на любом современном языке программирования. В настоящий момент JXTA реализован на J2SE, J2ME и Си/Си++/Си#. JXTA распространяется под лицензией, производной от Apache License. ...».

Несмотря на значительные преимущества децентрализованного взаимодействия участников РВ-сетей, выделяются также недостатки технологии одноранговых сетей [5]: «...

- в одноранговых сетях не может быть обеспечено гарантированное качество обслуживания: любой узел, предоставляющий те или иные сервисы, может быть отключён от сети в любой момент;
- индивидуальные технические характеристики узла могут не позволить полностью использовать ресурсы P2P сети (каждый из узлов обладает индивидуальными техническими характеристиками что, возможно, будет ограничивать его роль в P2P-сети и не позволять полностью использовать ее ресурсы: низкий рейтинг в torrent-сетях, LowID в eDonkey могут значительно ограничить ресурсы сети, доступные пользователю);
- при работе того или иного узла через брандмауэр может быть значительно снижена пропускная способность передачи данных в связи с необходимостью использования специальных механизмов обхода;
- участниками одноранговых сетей в основном являются индивидуальные пользователи, а не организации, в связи с чем возникают вопросы безопасности предоставления ресурсов: владельцы узлов P2P-сети, скорее всего, не

знакомы друг с другом лично, предоставление ресурсов происходит без предварительной договорённости;

- при увеличении числа участников P2P сети может возникнуть ситуация значительного возрастания нагрузки на сеть (как с централизованной, так и с децентрализованной структурой);
- в случае применения сети типа P2P приходится направлять значительные усилия на поддержку стабильного уровня ее производительности, резервное копирование данных, антивирусную защиту, защиту от информационного шума и других злонамеренных действий пользователей. ...».

5.2.2 Технологии GRID

Другим подходом, относящимся к РВ-сетям, являются Grid-вычисления появившиеся в начале 1990-х годов, как метафора, демонстрирующая возможность простого доступа к вычислительным ресурсам. Согласно [63]: «... **Грид-вычисления** (англ. *grid* — решётка, сеть) — это форма распределенных вычислений, в которой «виртуальный суперкомпьютер» представлен в виде кластеров, соединённых с помощью сети, слабосвязанных гетерогенных компьютеров, работающих вместе для выполнения огромного количества заданий (операций, работ). Эта технология применяется для решения научных, математических задач, требующих значительных вычислительных ресурсов. Грид-вычисления используются также в коммерческой инфраструктуре для решения таких трудоемких задач, как экономическое прогнозирование, сейсмоанализ, разработка и изучение свойств новых лекарств. Грид с точки зрения сетевой организации представляет собой согласованную, открытую и стандартизованную среду, которая обеспечивает гибкое, безопасное, скоординированное разделение вычислительных ресурсов и ресурсов хранения информации, которые являются частью этой среды, в рамках одной виртуальной организации. ...».

Не вдаваясь в детали реализации виртуальных суперкомпьютеров и виртуальных организаций, можно выделить следующие уровни архитектуры **Grid**:

1. **Базовый уровень** (Fabric) – содержит различные ресурсы, такие как компьютеры, устройства хранения, сети, сенсоры и другие.
2. **Связывающий уровень** (Connectivity) – определяет коммуникационные протоколы и протоколы аутентификации.
3. **Ресурсный уровень** (Resource) – реализует протоколы взаимодействия с ресурсами РВС и их управления.
4. **Коллективный уровень** (Collective) – управление каталогами ресурсов, диагностика, мониторинг.
5. **Прикладной уровень** (Applications) – инструментарий для работы с **Grid** и пользовательские приложения.

С 2001 года, в качестве базы для создания стандарта архитектуры **Grid** была выбрана технология веб-сервисов. Этот выбор был обусловлен двумя основными причинами:

- язык описания интерфейсов веб-сервисов WSDL (Web Service Definition Language) поддерживает стандартные механизмы для определения интерфейсов отдельно от их реализации, что в совокупности со специальными механизмами связывания обеспечивает возможность динамического поиска и компоновки сервисов в гетерогенных средах;
- широко распространённая адаптация механизмов веб-сервисов означает, что инфраструктура, построенная на базе веб-сервисов, может использовать различные утилиты и другие существующие сервисы, такие как различные процессоры WSDL, системы планирования потоков задач и среды для размещения веб-сервисов.

5.2.3 Облачные вычисления и «виртуализация»

Концепция облачных вычислений является одной из новомодных современных концепций компьютерных технологий, так называемый - *Cloud computing* [64].

Облачные вычисления (*cloud computing*) — технология распределённой обработки данных, в которой компьютерные ресурсы и мощности предоставляются пользователю как Интернет-сервис. Сам термин «Облако» используется как метафора, основанная на изображении *сложной инфраструктуры*, за которой скрываются все технические детали.

Многие авторы, на период 2008 года, указывали, что «Облачная обработка данных — это парадигма, в рамках которой информация постоянно хранится на серверах в Интернет и временно кэшируется на клиентской стороне, например, на персональных компьютерах, игровых приставках, ноутбуках, смартфонах и тому подобных устройствах». Для обеспечения согласованной работы ЭВМ, которые предоставляют услугу облачных вычислений, используется специализированное ПО, обобщенно называемое «*Middleware control*». Это ПО обеспечивает:

- мониторинг состояния оборудования,
- балансировку нагрузки,
- обеспечение ресурсов для решения задачи.

Для облачных вычислений основным предположением является неравномерность запроса ресурсов со стороны клиента. Для сглаживания этой неравномерности для предоставления сервиса между реальным железом и **Middleware** помещается ещё один слой - *виртуализация серверов*. Серверы, выполняющие приложения, виртуализируются и балансировка нагрузки осуществляется как средствами ПО, так и средствами распределения виртуальных серверов по реальным серверам, что соответственно порождает различные модели развёртывания. Приведём конкретные примеры таких моделей.

Частное облако (*private cloud*) — инфраструктура, предназначенная для использования одной организацией, включающей несколько потребителей, например, подразделений клиентами или подрядчиками данной организации. Частное облако может находиться в собственности, управлении и эксплуатации как самой

организации, так и третьей стороны или какой-либо их комбинации, а также может физически существовать как внутри, так и вне юрисдикции владельца.

Публичное облако (*public cloud*) — инфраструктура, предназначенная для свободного использования широкой публикой. Публичное облако может находиться в собственности, управлении и эксплуатации коммерческих, научных и правительственных организаций или какой-либо их комбинации. Публичное облако физически существует в юрисдикции владельца - поставщика услуг.

Гибридное облако (*hybrid cloud*) — это комбинация из двух или более различных облачных инфраструктур - частных, публичных или коммунальных, остающихся уникальными объектами, но связанных между собой стандартизованными или частными технологиями переносимости данных и приложений, например, кратковременное использование ресурсов публичных облаков для балансировки нагрузки между облаками.

Общественное облако (*community cloud*) — вид инфраструктуры, предназначенный для использования конкретным сообществом потребителей из организаций, имеющих общие задачи, например, миссии, требования безопасности, политики, и соответствия различным требованиям. Общественное облако может находиться в кооперативной (совместной) собственности, управлении и эксплуатации одной или более из организаций сообщества или третьей стороны или какой-либо их комбинации, и может физически существовать как внутри так и вне юрисдикции владельца.

Завершая данный подраздел и главу в целом, отметим, что проведённый краткий обзор теоретических концепций СОР, а также ряда их практических реализаций не обеспечивает студента полным набором знаний, необходимых для реализации таких систем. Главное, на что следует обратить внимание, это — интенсивное развитие указанного направления, которое в будущем должно обязательно прийти к своему насыщению и естественному теоретическому завершению рассматриваемой тематики.

Вопросы для самопроверки

1. В чем состоит основное отличие модели SOA от известных технологий CORBA и RMI?
2. Какое значение имеет ПО Middleware для систем построенных по моделям SOA?
3. Что такое - «*Интерфейс сервиса*» и где он применяется?
4. Что такое — UDDI и какое отношение к нему имеет понятие брокера?
5. На чем основан протокол SOAP и какие языки его поддерживают?
6. В чем отличие сильносвязанных программных систем от слабосвязанных?
7. В чем состоит парадигма первого поколения веб-сервисов?
8. Чем отличается второе поколение веб-сервисов от первого?
9. Соотношение каких схематических компонент предполагает «*Треугольник SOA*»?
10. Что означает аббревиатура WSDL и каково ее соотношение с архитектурами объектных распределенных систем?
11. Что подразумевается под сокращением РВ-сети?
12. Использование каких брокерных архитектур подразумевают веб-сервисы?
13. Какую схему подразумевает асинхронное взаимодействие через посредника?
14. В чем состоят недостатки синхронных методов взаимодействия программных систем?
15. Что такое — Р2Р-сети?
16. В чем состоят преимущества децентрализованных пиринговых сетей?
17. Для чего предназначена технология GRID?
18. Какие уровни архитектуры выделяются в технологии GRID?
19. Что такое - «Виртуальный суперкомпьютер»?
20. Перечислите виды «Облачных» инфраструктур?

Заключение

Распределенные вычислительные сети остаются примером основных современных архитектурных концепций построения сложных программных систем, объединяющих множество отдельных компьютеров, различных по составу и качеству исполнения приложений, а также производителей и потребителей информации, генерируемой такими системами. Излишне утверждать, что проектирование, реализация и сопровождение подобных систем является занятием, требующим разносторонних знаний, высокопрофессиональных умений и передовых технологий, специально созданных для этих целей.

В целом, распределенные системы имеют, пусть небольшую, но насыщенную историю. Она неразрывно связана с развитием возможностей аппаратных средств вычислительной техники и последующей наработкой программных решений, позволяющих создавать все более сложные программно-аппаратные комплексы и приложения. Одновременно развивались и концепции построения таких систем.

Первоначально, распределенные системы строились по принципу объединения отдельных приложений, размещённых на небольшом числе отдельных компьютеров и взаимодействующих между собой через сеть посредством некоторого доступного числа протоколов. Концептуальная идея таких систем опиралась на классические средства создания программного обеспечения, функциональные языки программирования, библиотеки программ для передачи данных по сети и специализированные средства доступа к базам данных (СУБД). В перспективе, такие системы виделись как распределенные вычислительные среды (DCE), что хорошо описано в фундаментальном труде Эндрю Таненбаума [3].

Последующее развитие теории распределенных систем проходило параллельно с развитием технологий самих средств создания программного обеспечения компьютеров. Действительно, бурное развитие языков объектно-ориентированного программирования (ООП) создало концептуальную базу для идеи объектных распределенных систем. Развитие web-технологий создало соответствующую базу для распределенных сервис-ориентированных систем. Развитие программных средств электронной коммерции породило технологию облачных вычислений. Учебный материал данного издания в целом следует указанной логике, раскрывая начальные понятия и концепции в плане обучения студентов уровня бакалавриата и направления подготовки *«Информатика и вычислительная техника»*.

В основной части учебного материала последовательно изложены основные технологические подходы, обозначенные выше. Для практического закрепления учебного материала использована единая общедоступная платформа, представленная инструментальными средствами языка Java. Такая позиция автора обеспечивает учебный процесс необходимым количеством нужных примеров, а также готовит студентов к работе с современными средствами автоматизации разработки программных систем уровня предприятий Eclipse EE, СУБД Apache Derby и контейнера сервлетов Apache Tomcat. Для демонстрации изучаемых технологий использована единая простейшая задача сохранения произвольных тестовых запи-

сей в базе данных и доступа к ним по уникальному ключу. В процессе изучения теоретических вопросов, эта задача трансформируется в соответствующие примеры, доведённые до уровня проектов на языке Java.

По главам, познавательная нагрузка учебного материала распределена следующим образом.

Вводная часть — Раздел 1 базируется на обсуждении концепций, изложенных в источниках [1-5]. Здесь, в обзорном плане, проведено сравнение различных подходов, интерпретирующих предметную область объекта изучения, выявлены обнаруженные противоречия и обоснована тематика изучаемой дисциплины, которая вынесена в название учебного пособия как *«Распределенные вычислительные сети»*. По результатам изучения этой главы, студент начинает ментально выделять классический подход реализации распределенных систем, основанный на знаниях уже изученных дисциплин, и подходы, которые ещё необходимо изучить.

Раздел 2 готовит студента основам использования языка Java. Обучение проводится как с привлечением стандартных инструментов языка, так и с привлечением интегрированных инструментальных средств Eclipse EE. Результирующий уровень подготовки студентов обеспечивает последующее использование классических средств управления сетевыми соединениями, а также организацию доступа к базам данных. В практическом плане формулируется и реализуется задача хранения записей в базе данных СУБД Apache Derby, которая в последующих главах трансформируется в набор демонстрационных примеров для реализации изучаемых технологий.

Раздел 3 раскрывает тему объектных распределенных систем, основанных на общей модели «Клиент-сервер». Теоретические построения этой главы закрепляются примером разбиения демонстрационной задачи на компоненты клиента и сервера. Далее, эти компоненты реализуются общим решением в рамках технологии CORBA. Завершается глава реализацией распределенного приложения в рамках технологии RMI.

Раздел 4 развивает тематику распределенных систем, включая в процесс обучения достижения web-технологий. Студент осваивает инструментальные средства контейнера сервлетов Apache Tomcat, базовые возможности JSP-страниц и шаблона MVC, входящие в стандартный набор разработчика распределенных приложений уровня Enterprise Edition. Результат применения этих технологий демонстрируется соответствующим вариантом реализации тестовой задачи.

Раздел 5 завершает изложение учебного материала дисциплины уровня бакалавриата. В ней кратко описаны концепции сервис-ориентированных архитектур, включая основные положения концепции SOA, а также - ориентированные на частные подходы облачных вычислений и технологии GRID. Целевое назначение этого учебного материала — начальная теоретическая подготовка студента для последующего более углублённого изучения предметной области под общим названием *«Распределенные сервис-ориентированные системы»*.

Подводя общий итог данной работы, автор считает, что учебное пособие, дополненное учебно-методическим материалом по практическим и лабораторным занятиям, полностью соответствует учебным планам процесса обучения студента уровня бакалавриата по направлению подготовки *«Информатика и вычислитель-*

ная техника».

Дальнейшее изучение предметной области РВ-сетей связано с технологиями, нашедшими своё воплощение в концепции сервис-ориентированных архитектур (SOA). Это — бурно развивающееся направление, которое во многом сформировалось под натиском достижений web-технологий и неуклонно стремится к своему коммерческому оформлению в виде облачных вычислений. Именно web-технологии заложили практический базис создания первых сервис-ориентированных систем, предложив технологию публикации интерфейсов удалённых систем, которую ранее выполняли «закрытые» брокерные архитектуры.

Здесь следует высказать ряд существенных замечаний. Дело в том, что первые сервис ориентированные системы создавались по принципу наименьшей связанности (см. содержание подраздела 5.1). Это считалось позитивным явлением, противодействующим всевозрастающей сложности распределённых систем. В частности, наметилось стремление уходить от централизованного управления такими системами, используя методы управления, которые развивались в рамках технологии одноранговых сетей. Со временем, эта тенденция привела к снижению надёжности и безопасности практических реализаций таких распределённых приложений. В результате, пришло понимание невозможности дальнейшего развития практики реализации РВ-сетей без существенной инструментальной технологической базы, нейтрализующей перечисленные выше вызовы.

С учётом высказанного замечания, недолгая история развития web-сервисов разделяется на первое и второе поколения. Начиная с 2004 года, начинают публиковаться и внедряться различные стандарты, повышающие качество работы веб-сервисов: WS-Security, обеспечивающие безопасность веб-сервисов, WS-Addressing, связанные с обеспечением маршрутизации и адресации SOAP-сообщений, а также — WSRF и WS-Notification, поддерживающие использование состояний web-сервисов. Все это требует изучения специальных инструментальных средств, которые хоть и реализованы на платформе языка Java, но могли быть представлены в начальном курсе подготовки бакалавра.

Хочется надеяться, что студенты, освоившие материал данной дисциплины, продолжат изучение современных технологий РВ-сетей, успешно используя полученные знания и повышая свой уровень теоретической и практической подготовки, совершенствуя свои умения на базе платформы инструментальных средств языка Java.

Список использованных источников

1. Бройдо В. Л., Ильина О. П. Вычислительные системы, сети и телекоммуникации: Учебник для вузов. 4-е изд. — СПб.: Питер, 2011. — 560 с.: ил. - ISBN 978-5-49807-875-5
2. Орлов С. А., Цилькер Б. Я. Организация ЭВМ и систем: Учебник для вузов. 3-е изд. — СПб.: Питер, 2015. — 688 с.: ил. (Серия «Учебник для вузов»). - ISBN 978-5-496-01145-7
3. Распределенные системы. Принципы и парадигмы / Э. Таненбаум, М. ван Стеен. — СПб.: Питер, 2003. — 877 с.: ил. — (Серия «Классика computer science»). - ISBN 5-272-00053-6
4. Ларионов А. М., Майоров С.А., Новиков Г. И. ВЫЧИСЛИТЕЛЬНЫЕ КОМПЛЕКСЫ, СИСТЕМЫ И СЕТИ. - Ленинград, ЭНЕРГОАТОМИЗДАТ, 1987. - 178 с.
5. Радченко, Г.И. Распределенные вычислительные системы / Г.И. Радченко. – Челябинск:: Фотохудожник, 2012. - 184 с. - ISBN 978-5-89879-198-8.
6. Учебный программный комплекс кафедры АСУ на базе ОС ArchLinux [Электронный ресурс]: Учебно-методическое пособие для студентов направления 09.03.01, направление подготовки "Программное обеспечение средств вычислительной техники и автоматизированных систем" / В.Г. Резник - 2016. 33 с. — Режим доступа: <https://edu.tusur.ru/publications/6238>.
7. Резник В.Г. Распределенные вычислительные системы. Лабораторные работы по направлению подготовки бакалавриата 09.03.01. Учебно-методическое пособие. – Томск, ТУСУР, 2019.
8. Резник В.Г. Распределенные вычислительные системы. Практические занятия по направлению подготовки бакалавриата 09.03.01. Учебно-методическое пособие. – Томск, ТУСУР, 2019.
9. ГОСТ 33707-2016 (ISO/IEC 2382:2015) Информационные технологии (ИТ). Словарь [Электронный ресурс]: Режим доступа: <http://docs.cntd.ru/document/1200139532>.
10. Ноутон П., Шилдт Г. JAVA 2. Наиболее полное руководство в подлиннике. СПб.: БХВ-Петербург, 2008. - 1072 с. - ISBN 978-5-94157-012-6.
11. Таксономия Флинна - Википедия [Электронный ресурс]: Режим доступа: https://ru.wikipedia.org/wiki/Таксономия_Флинна.
12. Распределенная система - Википедия [Электронный ресурс]: Режим доступа: https://ru.wikipedia.org/wiki/Распределённая_система.
13. Сетевая модель OSI - Википедия [Электронный ресурс]: Режим доступа: https://ru.wikipedia.org/wiki/Сетевая_модель_OSI.

14. Клиент-сервер - Википедия [Электронный ресурс]: Режим доступа: https://ru.wikipedia.org/wiki/Клиент_—_сервер.
15. Распределённая вычислительная среда - Википедия [Электронный ресурс]: Режим доступа: https://ru.wikipedia.org/wiki/Распределённая_вычислительная_среда.
16. Удалённый вызов процедур - Википедия [Электронный ресурс]: Режим доступа: https://ru.wikipedia.org/wiki/Удалённый_вызов_процедур.
17. Object Management Group - Википедия [Электронный ресурс]: Режим доступа: https://ru.wikipedia.org/wiki/Object_Management_Group.
18. CORBA - Википедия [Электронный ресурс]: Режим доступа: <https://ru.wikipedia.org/wiki/CORBA>.
19. SOA. Архитектурные особенности и практические аспекты [Электронный ресурс]: Режим доступа: http://www.tadviser.ru/index.php/Статья:SOA_Архитектурные_особенности_и_практические_аспекты.
20. BPM (управленческая концепция) - Википедия [Электронный ресурс]: Режим доступа: [https://ru.wikipedia.org/wiki/BPM_\(управленческая_концепция\)](https://ru.wikipedia.org/wiki/BPM_(управленческая_концепция)).
21. Сервис-ориентированная архитектура - Википедия [Электронный ресурс]: Режим доступа: https://ru.wikipedia.org/wiki/Сервис-ориентированная_архитектура.
22. SOAP - Википедия [Электронный ресурс]: Режим доступа: <https://ru.wikipedia.org/wiki/SOAP>.
23. OASIS - Википедия [Электронный ресурс]: Режим доступа: <https://ru.wikipedia.org/wiki/OASIS>.
24. Сервисная шина предприятия - Википедия [Электронный ресурс]: Режим доступа: https://ru.wikipedia.org/wiki/Сервисная_шина_предприятия.
25. Виртуальная машина - Википедия [Электронный ресурс]: Режим доступа: https://ru.wikipedia.org/wiki/Виртуальная_машина.
26. Java Virtual Machine - Википедия [Электронный ресурс]: Режим доступа: https://ru.wikipedia.org/wiki/Java_Virtual_Machine.
27. Java - Википедия [Электронный ресурс]: Режим доступа: <https://ru.wikipedia.org/wiki/Java>.
28. Sun Microsystems - Википедия [Электронный ресурс]: Режим доступа: https://ru.wikipedia.org/wiki/Sun_Microsystems.
29. Oracle - Википедия [Электронный ресурс]: Режим доступа: <https://ru.wikipedia.org/wiki/Oracle>.
30. Java SE 6 Documentation [Электронный ресурс]: Режим доступа: <https://docs.oracle.com/javase/6/docs/>.

31. Бруннер Р. Введение в Apache Derby [Электронный ресурс]: Режим доступа: <https://www.ibm.com/developerworks/ru/library/os-ad-trifecta1/>.
32. Apache Derby - Википедия [Электронный ресурс]: Режим доступа: https://ru.wikipedia.org/wiki/Apache_Derby.
33. Apache Derby [Электронный ресурс]: Режим доступа: <http://db.apache.org/derby/>.
34. Java Database Connectivity - Википедия [Электронный ресурс]: Режим доступа: https://ru.wikipedia.org/wiki/Java_Database_Connectivity.
35. Прокси-сервер - Википедия [Электронный ресурс]: Режим доступа: <https://ru.wikipedia.org/wiki/Прокси-сервер>.
36. Язык описания интерфейсов - Википедия [Электронный ресурс]: Режим доступа: https://ru.wikipedia.org/wiki/Язык_описания_интерфейсов.
37. Java IDL: Glossary [Электронный ресурс]: Режим доступа: <https://docs.oracle.com/javase/8/docs/technotes/guides/idl/jidlGlossary.html>.
38. Interface Definition Language — PDF [Электронный ресурс]: Режим доступа: <https://www.omg.org/spec/IDL/4.2/PDF>.
39. Naming (Java Platform SE 8) [Электронный ресурс]: Режим доступа: <https://docs.oracle.com/javase/8/docs/api/java/rmi/Naming.html>.
40. LocateRegistry (Java Platform SE 8) [Электронный ресурс]: Режим доступа: <https://docs.oracle.com/javase/8/docs/api/java/rmi/registry/LocateRegistry.html>.
41. Всемирная паутина - Википедия [Электронный ресурс]: Режим доступа: https://ru.wikipedia.org/wiki/Всемирная_паутина.
42. URI - Википедия [Электронный ресурс]: Режим доступа: <https://ru.wikipedia.org/wiki/URI>.
43. URL - Википедия [Электронный ресурс]: Режим доступа: <https://ru.wikipedia.org/wiki/URL>.
44. URN - Википедия [Электронный ресурс]: Режим доступа: <https://ru.wikipedia.org/wiki/URN>.
45. HTML - Википедия [Электронный ресурс]: Режим доступа: <https://ru.wikipedia.org/wiki/HTML>.
46. Элементы HTML - Википедия [Электронный ресурс]: Режим доступа: https://ru.wikipedia.org/wiki/Элементы_HTML.
47. JavaScript - Википедия [Электронный ресурс]: Режим доступа: <https://ru.wikipedia.org/wiki/JavaScript>.
48. Java-апплет - Википедия [Электронный ресурс]: Режим доступа: <https://ru.wikipedia.org/wiki/Java-апплет>.
49. AJAX - Википедия [Электронный ресурс]: Режим доступа: <https://ru.wikipedia.org/wiki/AJAX>.

50. XML - Википедия [Электронный ресурс]: Режим доступа:
<https://ru.wikipedia.org/wiki/XML>.
51. HTTP - Википедия [Электронный ресурс]: Режим доступа:
<https://ru.wikipedia.org/wiki/HTTP>.
52. Apache HTTP Server - Википедия [Электронный ресурс]: Режим доступа:
https://ru.wikipedia.org/wiki/Apache_HTTP_Server.
53. PHP - Википедия [Электронный ресурс]: Режим доступа:
<https://ru.wikipedia.org/wiki/PHP>.
54. Сервлет (Java) - Википедия [Электронный ресурс]: Режим доступа:
[https://ru.wikipedia.org/wiki/Сервлет_\(Java\)](https://ru.wikipedia.org/wiki/Сервлет_(Java)).
55. Apache Tomcat - Википедия [Электронный ресурс]: Режим доступа:
https://ru.wikipedia.org/wiki/Apache_Tomcat.
56. javax.servlet (Java (TM) EE 7 Specification APIs) [Электронный ресурс]: Режим доступа: <https://docs.oracle.com/javaee/7/api/javax/servlet/package-summary.html>.
57. Apache Tomcat — Welcome! [Электронный ресурс]: Режим доступа:
<https://tomcat.apache.org/>.
58. JavaServer Pages - Википедия [Электронный ресурс]: Режим доступа:
https://ru.wikipedia.org/wiki/JavaServer_Pages.
59. UDDI - Википедия [Электронный ресурс]: Режим доступа:
<https://ru.wikipedia.org/wiki/UDDI>.
60. WSDL - Википедия [Электронный ресурс]: Режим доступа:
<https://ru.wikipedia.org/wiki/WSDL>.
61. Одноранговая сеть - Википедия [Электронный ресурс]: Режим доступа:
https://ru.wikipedia.org/wiki/Одноранговая_сеть.
62. JXTA - Википедия [Электронный ресурс]: Режим доступа:
<https://ru.wikipedia.org/wiki/JXTA>.
63. Грид - Википедия [Электронный ресурс]: Режим доступа:
<https://ru.wikipedia.org/wiki/Грид>.
64. Облачные вычисления - Википедия [Электронный ресурс]: Режим доступа:
https://ru.wikipedia.org/wiki/Облачные_вычисления.

Алфавитный указатель

А

Автоматизированное рабочее место.....	152, 158
Адресация ресурсов.....	153
АРМ.....	152, 158
Архитектура компьютера.....	8

Б

БД.....	10
Браузер.....	152
Брокер.....	99
Брокерная архитектура CORBA.....	105
Брокерные архитектуры.....	99

В

Веб-сервисы.....	199
Виртуальная машина.....	30
Виртуальные системы.....	10, 30
VM.....	8
ВС.....	10
Втором поколении веб-сервисов.....	202
Вызов удалённых процедур.....	101
Вычислительная машина.....	8
Вычислительные комплексы.....	12, 13, 35
Вычислительные сети.....	14, 17, 36
Вычислительные системы.....	9, 12, 13, 35

Г

Гибридное облако.....	210
-----------------------	-----

Д

Динамические системы обнаружения сервисов.....	199
--	-----

И

Интерфейс передачи сообщений.....	32
Интерфейс сервиса.....	199

К

Клиент-сервер.....	21, 98, 99, 157, 158
Компилятор IDL.....	103

Л

ЛВС.....	10
----------	----

М

Межброкерный протокол для Интернет.....	24
---	----

Механизмы обнаружения сервисов.....	199
Многомашинные вычислительные комплексы.....	13
Многопроцессорные вычислительные комплексы.....	13

О

Облачные вычисления.....	209
Общая классификация систем обработки данных.....	10
Общественное облако.....	210
Объектные распределенные системы.....	98, 198
Одноранговые вычислительные сети.....	206
ООП.....	103
ОС.....	10

П

Первое поколение веб-сервисов.....	201
Передача ресурсов.....	153
Представление ресурсов.....	153
Прокси-сервер.....	100
Публичное облако.....	210

Р

распределенная обработка данных.....	16
распределенная система базы данных.....	16
Распределенные вычислительные сети.....	17, 35
Распределенные вычислительные системы.....	9, 19
Распределенные системы.....	11, 14, 16, 18, 35
Распределенный объект.....	104
Распределительная вычислительная среда.....	22
РВ-сети.....	35, 36

С

Сервис-ориентированная архитектура.....	28, 198, 199
Сервис-ориентированные технологии.....	10, 25, 26, 198, 205
Сервисная шина предприятия.....	29
Сервисно-ориентированная архитектура.....	27
Сервисные компоненты.....	199
Сетевая модель OSI.....	20
Сетевые объектные системы.....	10
Сильносвязанные системы.....	200
Системы обработки данных.....	11
Системы телеобработки.....	14, 36
Слабосвязанные системы.....	200
Службы промежуточного уровня.....	25
СОД.....	9, 11
Соединитель сервисов.....	199
СОИ.....	9
Сосредоточенные системы.....	11-14, 18, 35

Состояние сервиса.....	202
COT.....	198, 210
Статические системы обнаружения сервисов.....	199
СУБД.....	9, 82-84, 86
СУБД Derby.....	83-86
Т	
Технология CORBA.....	105
Технология RMI.....	136
Тонкий клиент.....	152, 158, 159
Треугольник SOA.....	201, 203
У	
Удалённый вызов процедур.....	23
Ф	
Файл определения интерфейса.....	103
Ч	
Частное облако.....	209
Э	
ЭВМ.....	9, 12, 35
А	
AJAX.....	156
AOP.....	28
В	
BPM.....	28
С	
CGI.....	162
Client stub.....	23
Cloud computing.....	209
Common Gateway Interface.....	162
Common Object Request Broker Architecture.....	24
Computer architecture.....	8
CORBA.....	24, 25, 105, 108, 136
Д	
DCE.....	22, 101
DCE RPC.....	102, 103
DCOM.....	105
Distributed object.....	104
Е	
EAI.....	28
Enterprise service bus.....	29
ESB.....	29

G

General Inter-ORB Protocol.....	24
GIOP.....	24, 105
Globe.....	105
Grid.....	208

H

HTIOP.....	25, 105
HTML.....	152-156
HTTP.....	153, 154, 156, 157, 199
HyperText InterORB Protocol.....	25

I

IDL.....	24, 101
IOP.....	24, 105
Interface Definition Language.....	23, 101
Interface Description Language.....	101
Internet InterORB Protocol.....	24

J

Java Remote Method Protocol.....	136
Java Runtime Environment.....	33
Java Virtual Machine.....	33
JavaScript.....	156
JDBC.....	86
JRE.....	33
JRMP.....	136
JVM.....	33
JXTA.....	207

L

Loose coupling.....	200
---------------------	-----

M

Message Passing Interface.....	32
Middleware.....	99, 100, 158, 198, 209
Middleware control.....	209
MPI.....	32

O

OASIS.....	28, 201
Object Management Group.....	24
Object Resource Broker.....	105
OMG.....	24
ORB.....	105
OSI.....	21

R

Remote Method Invocation.....	25, 136
Remote object.....	103
Remote Procedure Call.....	23
RMI.....	25, 136
RPC.....	23, 100-102

S

Server stub.....	23
Simple Object Access Protocol.....	28
SOA.....	27, 28, 198, 199
SOAP.....	28, 198, 199, 201, 202
SSL InterORB Protocol.....	24
SSLIOP.....	24, 105
Strong coupling.....	200

U

UDDI.....	201
Universal Description Discovery & Integration.....	201
URI.....	153, 154, 157
URL.....	153-155
URN.....	153-155
Uuidgen.....	102

V

Virtual Machine.....	30
VM.....	30

W

Web Services Description Language.....	201
WSDL.....	201

X

XML.....	155, 156, 199, 201
----------	--------------------