

Министерство науки и высшего образования Российской Федерации

Томский государственный университет
систем управления и радиоэлектроники

В.Г. Резник

РАСПРЕДЕЛЕННЫЕ ВЫЧИСЛИТЕЛЬНЫЕ СЕТИ

Учебное пособие

Тема 3. Объектные распределенные системы

Томск
2020

Резник, Виталий Григорьевич

Распределенные вычислительные сети. Учебное пособие. Тема 3. Объектные распределенные системы / В.Г. Резник. – Томск : Томск. гос. ун-т систем упр. и радиоэлектроники, 2019. – 61 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 подкрепляются 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».

Одобрено на заседании каф. АСУ протокол №_____ от _____

УДК 004.75

© Резник В. Г., 2019

© Томск. гос. ун-т систем упр. и радиоэлектроники, 2019

Оглавление

3 Тема 3. Объектные распределенные системы.....	4
3.1 Брокерные архитектуры.....	5
3.1.1 Вызов удалённых процедур.....	7
3.1.2 Использование удалённых объектов.....	9
3.2 Технология CORBA.....	11
3.2.1 Брокерная архитектура CORBA.....	12
3.2.2 Проект серверной части приложения NotePad.....	14
3.2.3 Проект клиентской части приложения Example12.....	21
3.2.4 Генерация распределенного объекта OrbPad.....	25
3.2.5 Реализация серверной части ORB-приложения.....	33
3.2.6 Реализация клиентской части ORB-приложения.....	39
3.3 Технология RMI.....	45
3.3.1 Интерфейсы удалённых объектов.....	46
3.3.2 Реализация RMI-сервера.....	47
3.3.3 Реализация RMI-клиента.....	53
3.3.4 Завершение реализации RMI-проекта.....	58
Вопросы для самопроверки.....	61

3 Тема 3. Объектные распределенные системы

В подразделе 1.2 уже была дана общая характеристика сетевых объектных систем. Она касалась классических приложений модели OSI, распределённых 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:

- 1) **подраздел 3.1** описывает брокерные архитектуры, которые составляют идейную основу объектных распределенных систем;
- 2) **подраздел 3.2** посвящён технологии CORBA, которая стандартизирует общую архитектуру брокерных систем, обеспечивая сетевые приложения абстрактным протоколом взаимодействия GIOP и теоретическим набором стандартных служб;
- 3) **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 для практической реализации приложений с использованием языка Java.

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

Главной причиной, потребовавшей новых идей, является *интенсивное развитие сетевых технологий*, которое объективно *увеличило количество ЭВМ* и различных вычислительных систем, участвующих в создании распределенных систем.

Выделим следующие аспекты указанной причины:

- а) *количество потенциальных связей* квадратично зависит от числа взаимодействующих элементов, что соответственно увеличивает и нагрузка на программирование сетевых взаимодействий между ними;
- б) *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;
- в) необходима *отдельная сетевая объектная адресация*, позволяющая именовать элементы распределенных приложений независимо от особенностей реализации самой сети; это 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 первом разделе, на рисунке 1.5, или как посредник в передаче сообщений и объектных запросов, что рассматривается в данном разделе. В любой интерпретации, 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.

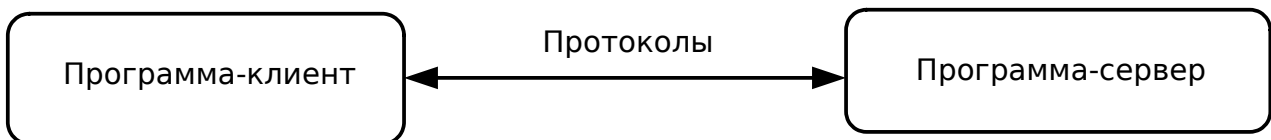


Рисунок 3.1 — Классическая модель взаимодействия «Клиент-сервер»

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

- а) обе программы *соответствуют классической модели «Клиент-сервер»*, реализованной на базе протокола TCP пакета *java.net*;
- б) 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;
- в) 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;
- г) 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 её схема взаимодействия демонстрируется рисунком 3.2.

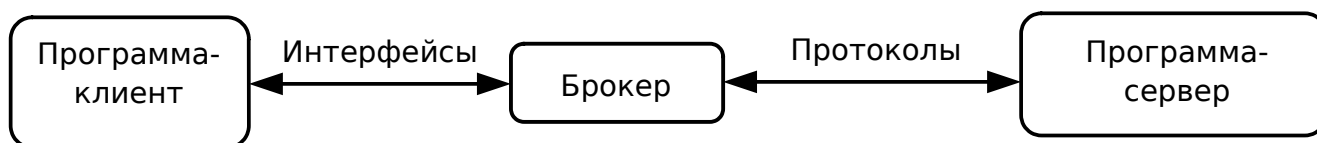


Рисунок 3.2 — Брокерная модель взаимодействия «Клиент-сервер»

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. При этом, 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++.

Широко известны следующие реализации IDL:

- а) **AIDL**: Реализация IDL на Java для Android, поддерживающая локальные и удалённые вызовы процедур. Может быть доступна из нативных приложений посредством JNI.
- б) **CORBA IDL** — *язык описания интерфейсов* распределённых объектов, разработанный рабочей группой OMG. Создан в рамках обобщённой архитектуры CORBA.
- в) **IDL DCE**, *язык описания интерфейсов* спецификации межплатформенного взаимодействия служб, которую разработал консорциум Open Software Foundation (теперь The Open Group).
- г) **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.
- д) **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 ранее на рисунке 1.9. Соответственно, как показано далее на рисунке 3.3, ПО брокера распределяется между программами клиента и сервера в виде программ-*заглушек*

(*Client stub* и *Server stub*), которые и реализуют 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.

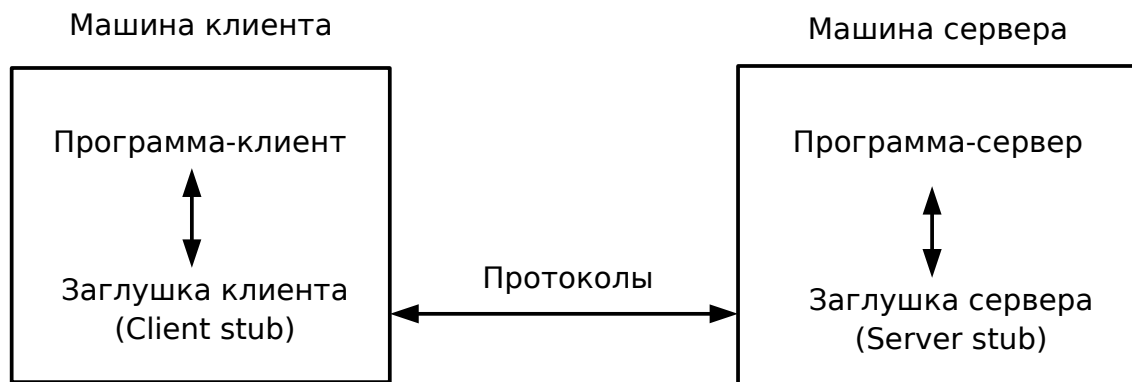


Рисунок 3.3 — Брокерная модель RPC

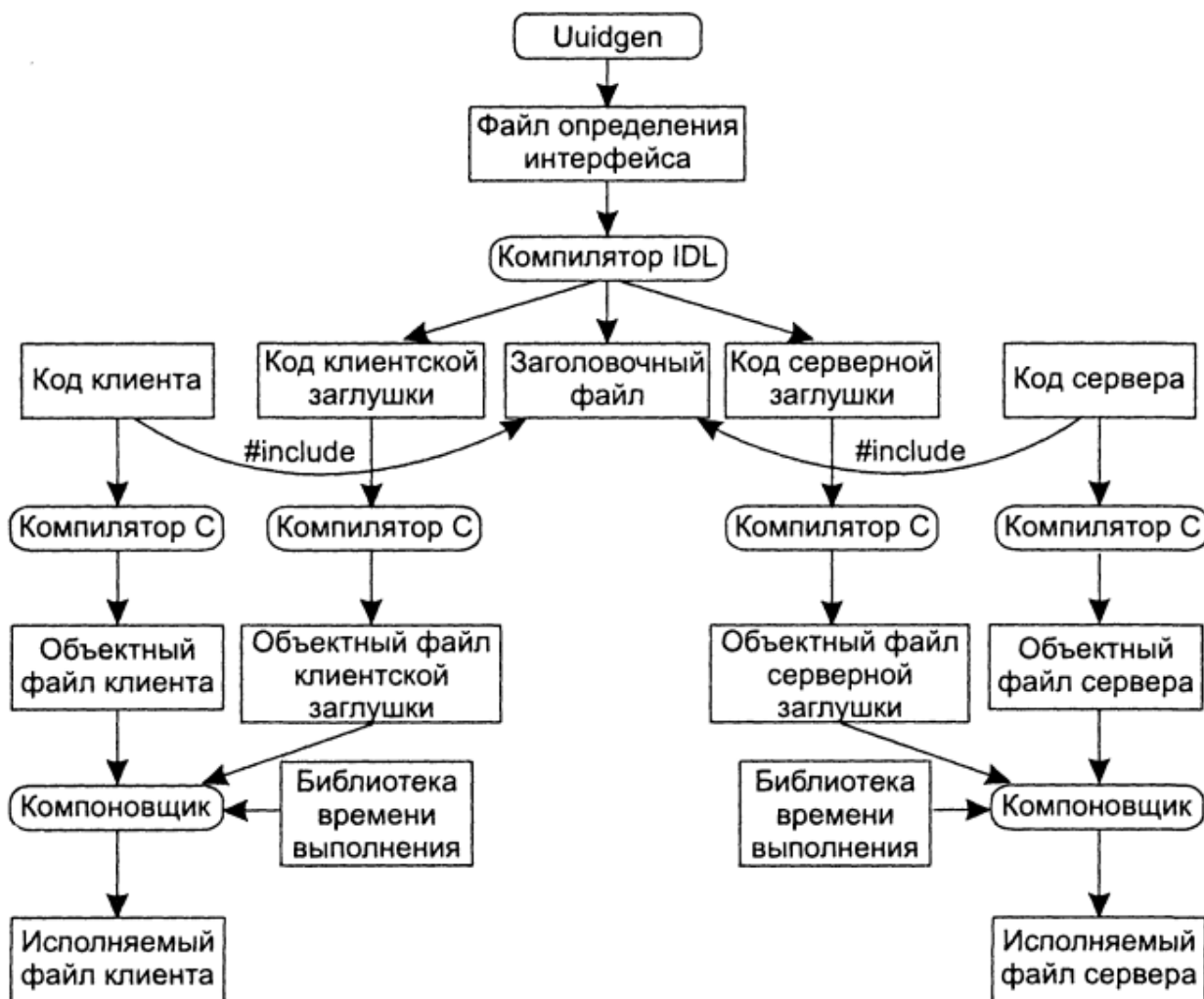


Рисунок 3.4 — Этапы реализации DCE RPC [3]

Поясним только начальные этапы представленной технологии:

- а) ***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.
- б) ***Файл определения интерфейса*** — исходный текст описания интерфейса на языке IDL DCE.
- в) ***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*.
- г) 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. Желаящим подробно изучить эту проблематику следует обратиться к разделу 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:

- а) 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;
- б) на стороне программы-**сервера** создаётся 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:

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

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

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

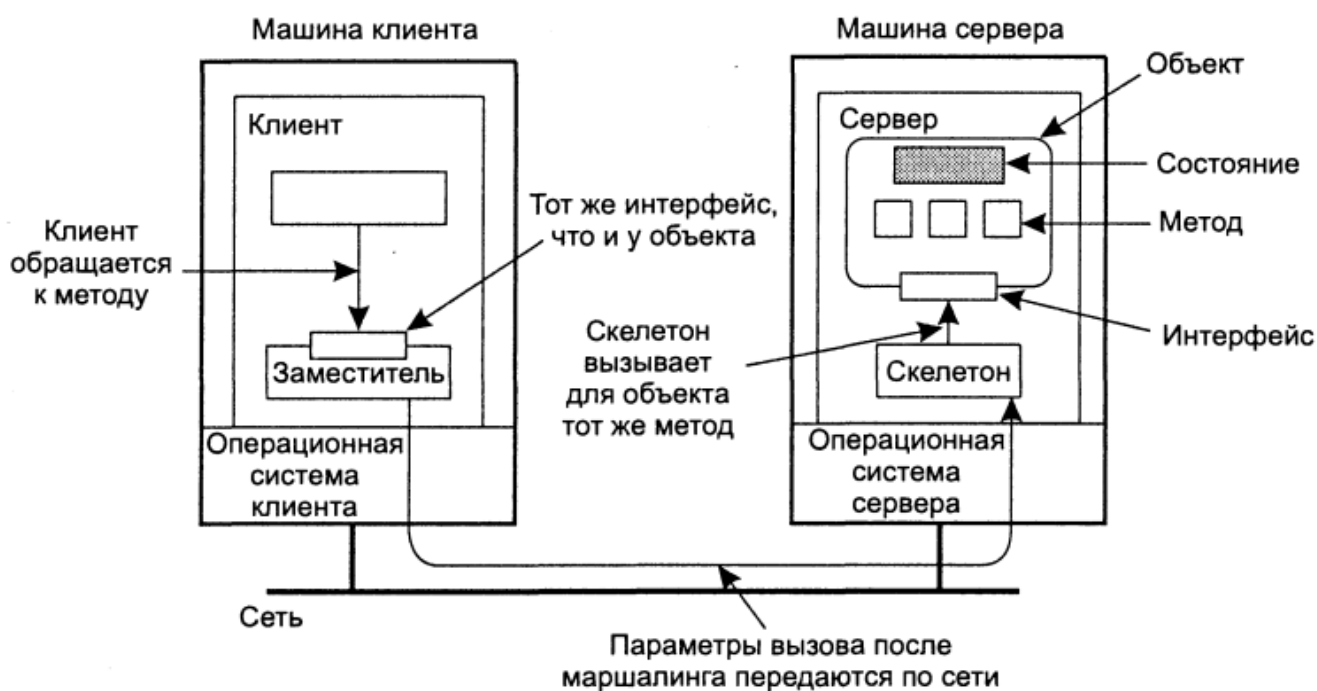


Рисунок 3.5 — Обобщённая организация удалённых объектов [3]

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 компиля-

цию 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 первом разделе (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 распределённую систему Globe.

Технология CORBA даёт описание распределённых приложений *независимо от языков их реализации*. Мы будем интерпретировать её архитектуру, предполагая реализацию на языке Java. Исходя из этого, общая последовательность учебного материала разделена на шесть этапов, которых студент должен придерживаться при проектировании распределённых 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 распределённого объекта.
3. **Этап 3** (пункт 3.2.3) — *пример приложения* (**Example12**), которое рассматривается как клиентская часть будущего распределённого объекта.
4. **Этап 4** (пункт 3.2.4) — *генерация базового описания*, с помощью специального компилятора, распределённого класса (**OrbPad**) для сервера, который демонстрирует использование инструментального средства языка IDL, применительно к языку Java.
5. **Этап 5** (пункт 3.2.5) — *пример реализации удалённого объекта* по описаниям распределённого 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:

- а) **Прикладные объекты** — части конкретных распределенных приложений.
- б) **Общие объектные службы** — базовые сервисы, которые доступны всем объектам, подключённым к ORB.
- в) **Горизонтальные средства** (*CORBA horizontal facilities*) — высокоуровневые службы общего назначения, независящие от прикладной области использующих их программ: *средства мобильных агентов, средства печати и средства локализации*.
- г) **Вертикальные средства** (*CORBA vertical facilities*) — это домены или прикладные области CORBA, которые первоначально были разделены на одиннадцать рабочих групп, предназначенных для следующих сфер применения: *корпоративные системы, финансы и страхование, электронная коммерция, промышленность* и другие.

Для разработчиков приложений наиболее важными являются шестнадцать *общих объектных служб*, являющихся 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 (*Concurrency 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.

В парадигме «Клиент-сервер», всё ПО ORB разделяют на три части, показанные на рисунке 3.7:

- а) **ORB-сервер** — посредник взаимодействия;
- б) **ORB клиента** — Stub клиента;
- в) **ORB сервера** — Stub сервера.

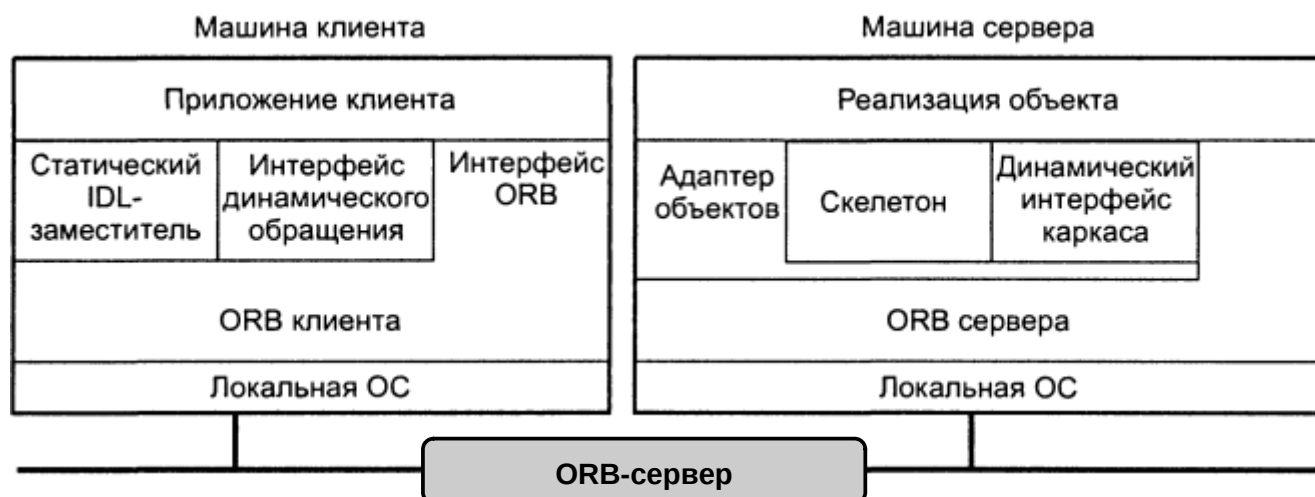


Рисунок 3.7 — Общая организация системы CORBA [3, Таненбаум Э.]

В процессе конкретизации клиент-серверной технологии CORBA к инструментальным средствам языка Java, мы будем опираться на официальный глоссарий источника [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 Managment 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:

- 1) *NotePad()* — конструктор, устанавливающий соединение с уже существующей базой данных *examoleDB*;
- 2) *boolean isConnected()* — метод, проверяющий наличие соединения с БД;
- 3) *Object[] getList()* — метод, получающий содержимое таблицы *notepad* БД *exampleDB* в виде списка текстовых строк;

- 4) `int setInsert(int key, String str)` — метод, добавляющий текст к содержимому таблицы *notepad* БД; также учитывается уникальность ключа *key*;
- 5) `int setDelete(int key)` — метод, удаляющий по заданному ключу *key* запись из таблицы *notepad* БД;
- 6) `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"
            + "-----");

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

        if(obj.isConnect())
            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.

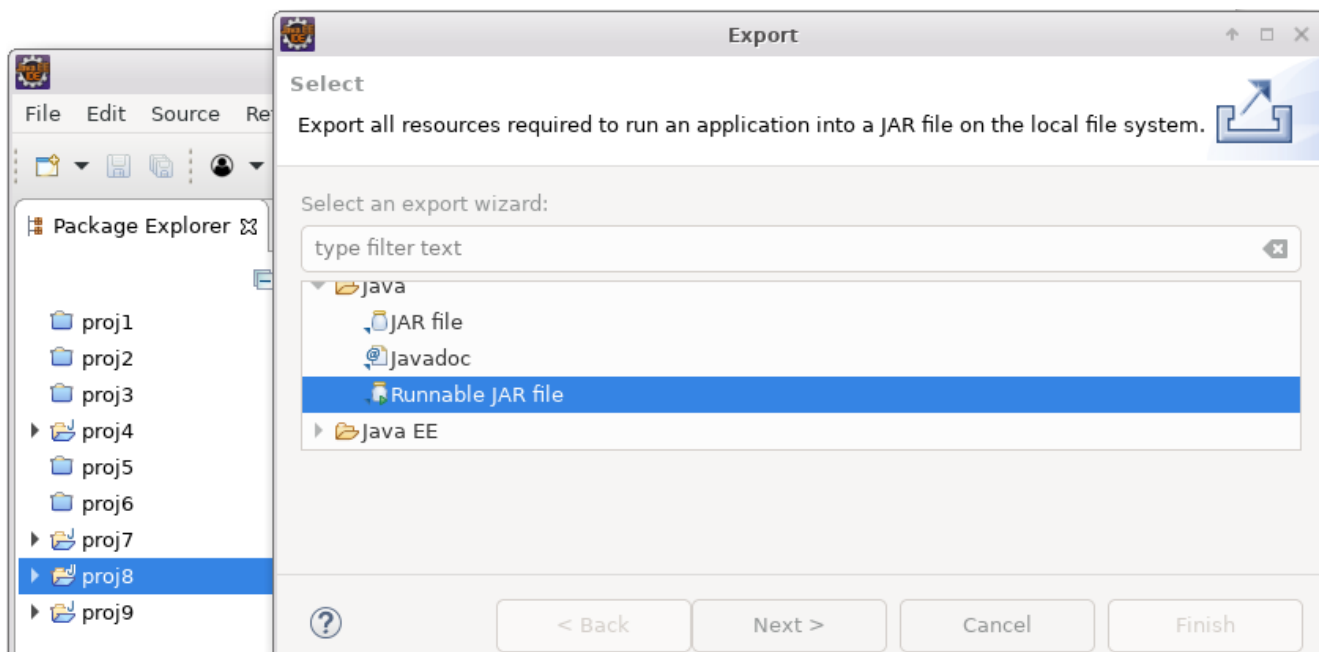


Рисунок 3.8 — Выбор типа экспортируемого проекта

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

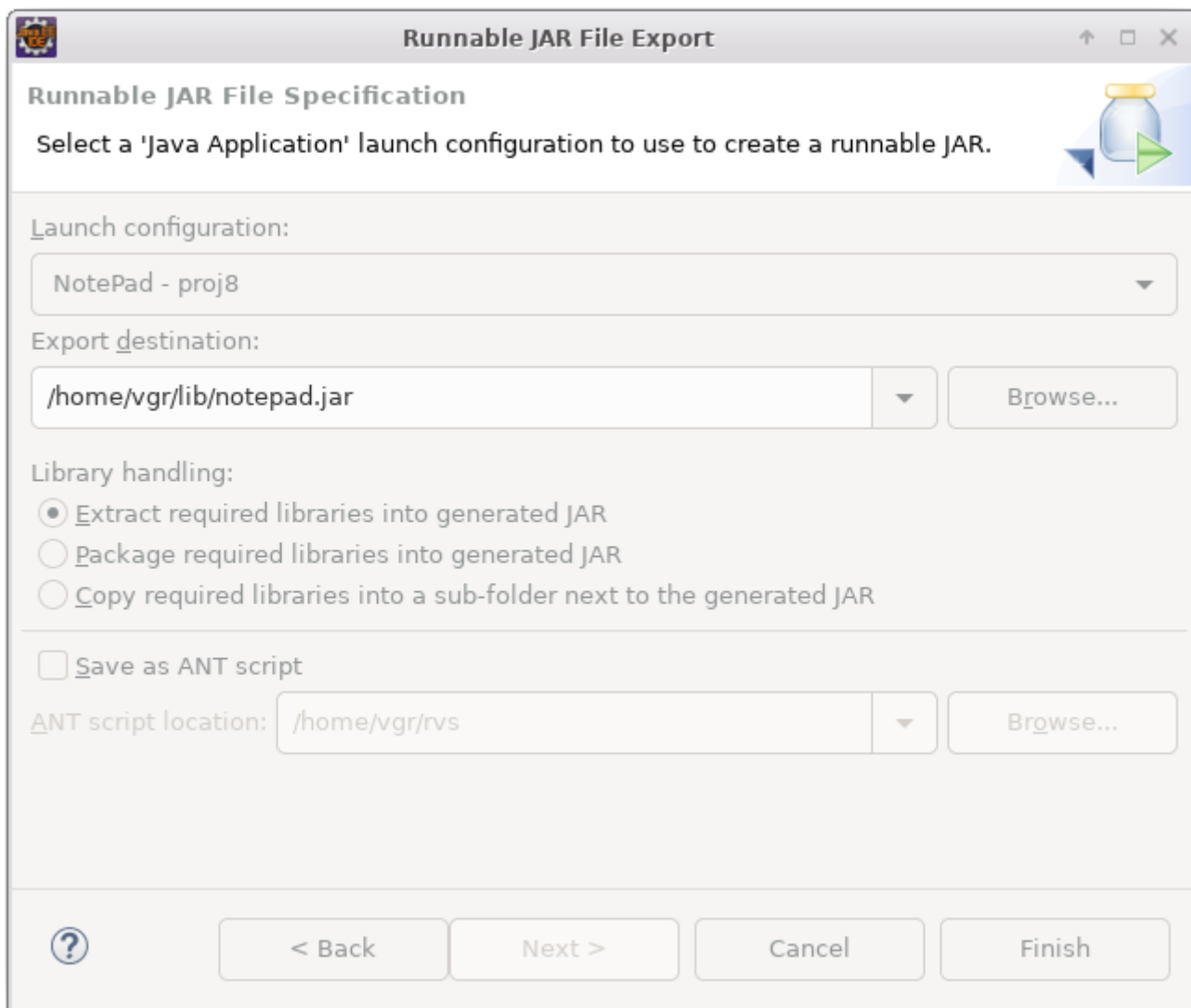


Рисунок 3.9 — Указание имени и места размещения архива

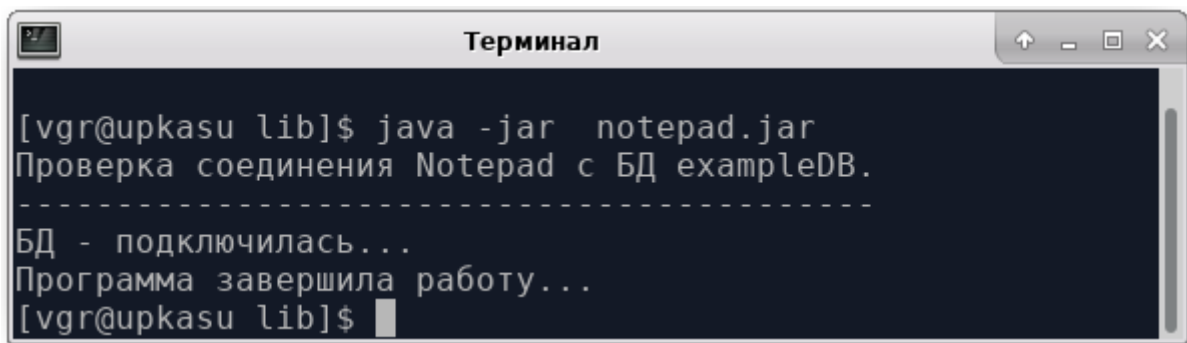
Теперь, активировав кнопку «**Finish**», мы завершаем создание архива класса *NotePad*.

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.



```
[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:

- а) `int getKey()` - чтение целого положительного числа;
- б) `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:

- а) если на предложение программы *ввести ключ* пользователь нажимает клавишу «*Enter*», то программа — завершается;
- б) если на предложение программы *ввести строку текста* пользователь нажимает клавишу «*Enter*», то введённый ранее *ключ* используется в команде удаления строки из таблицы *notepad*;
- в) если *введённая строка текста* не является пустой, то введённый 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("\n0шибка 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:

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

```

Терминал
[vgr@upkasu lib]$
[vgr@upkasu lib]$
[vgr@upkasu lib]$ java -jar example12.jar
Example12 для ведения записей в БД exampleDB.
    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 ...
-----

```

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 распре-

делённой системы **OrbPad**, которая:

- а) для программ-**клиентов** — будет предоставляться реализованный интерфейс с именем **OrbPad**, имеющий те же методы, что и класс **NotePad**;
- б) 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, независимом 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.

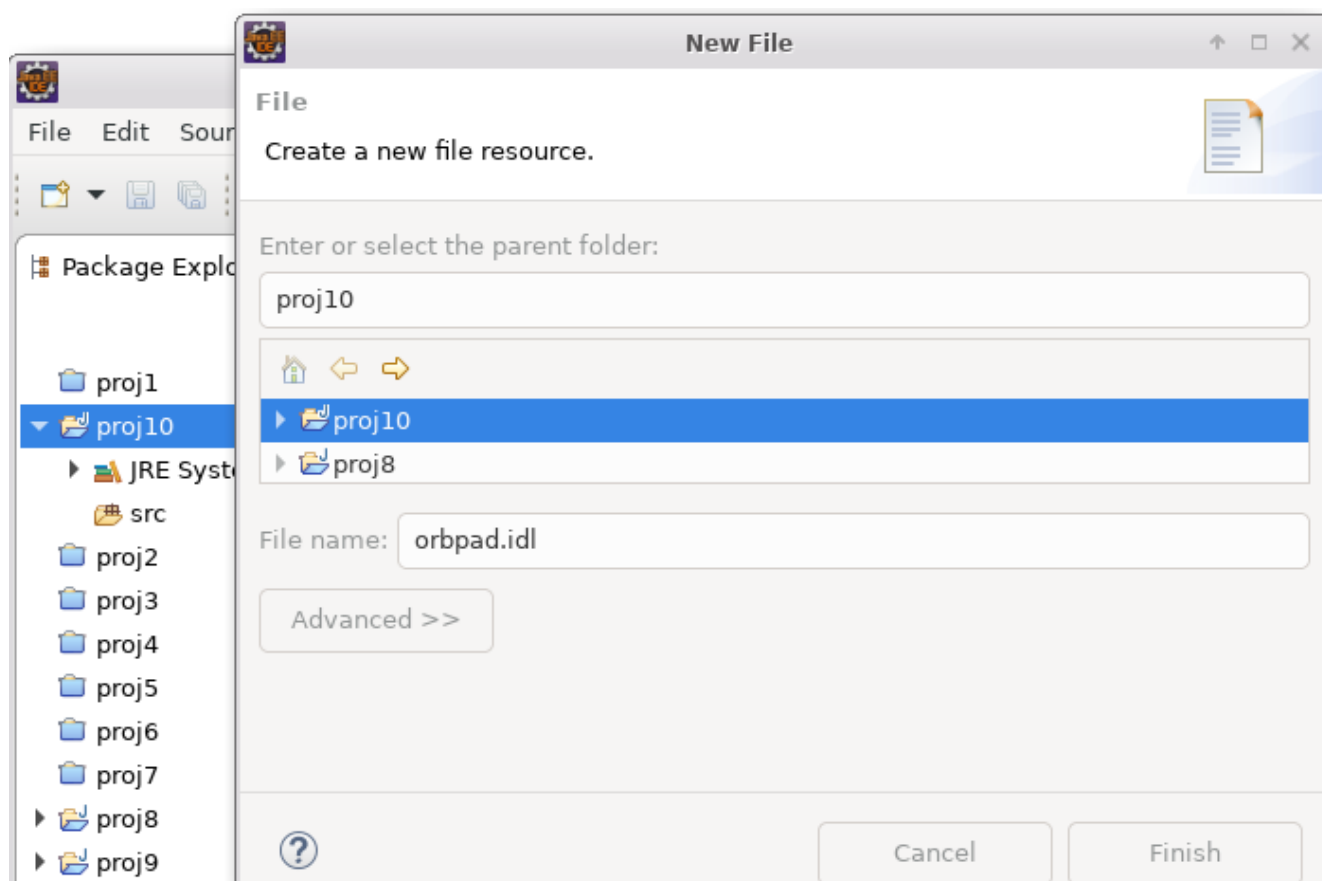


Рисунок 3.12 — Открытие IDL-файла в проекте proj10

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

Само описание интерфейса проведем на основе методов класса *NotePad*, ре-

ализованного в пункте 3.2.2. С учётом преобразования типов таблицы 3.1, оно будет выглядеть как показано на рисунке 3.13.

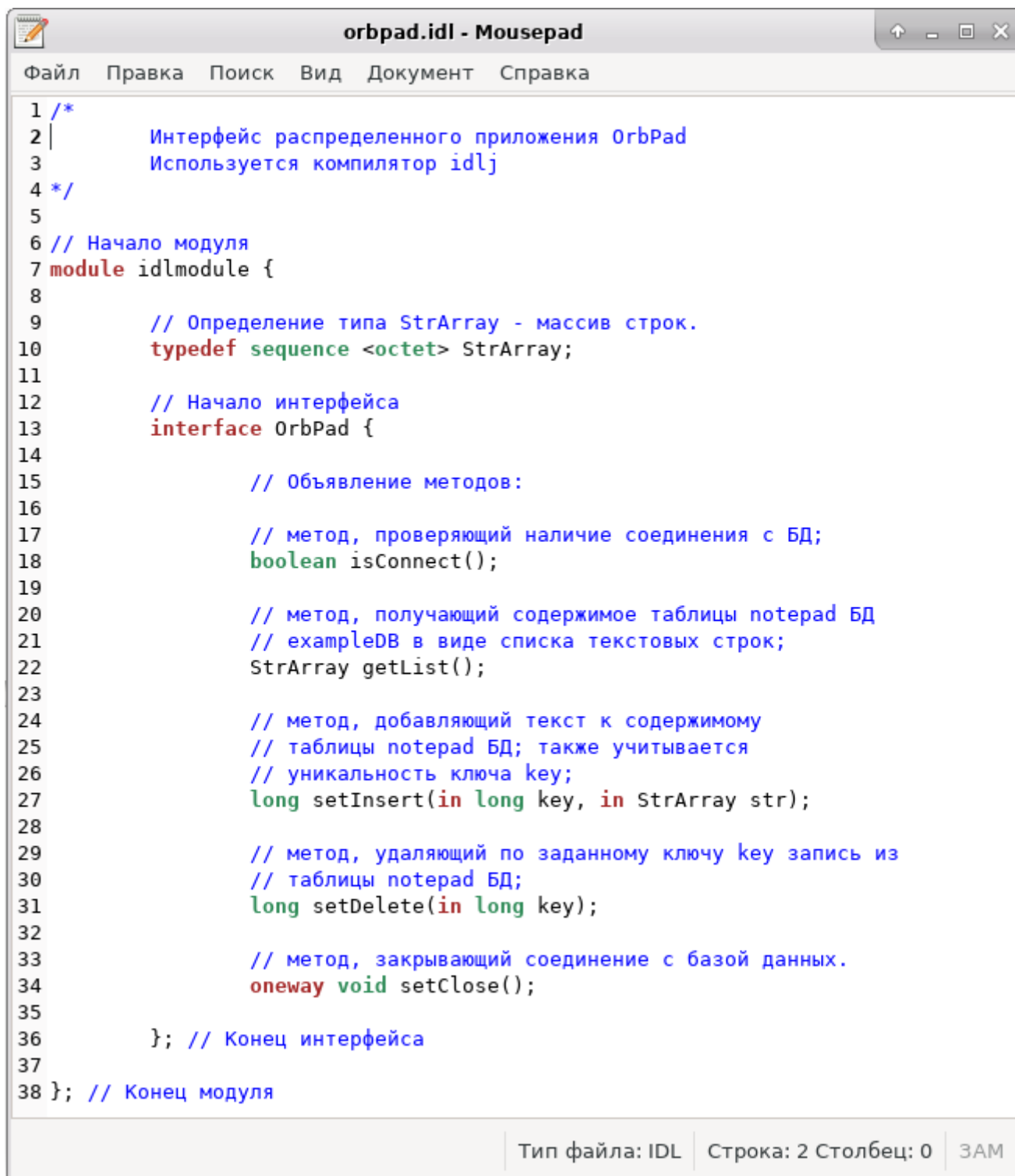
По синтаксису, язык IDL очень похож на язык C++. Его формальное описание можно найти, например, в источнике [38]. Мы лишь отметим, что полное описание любого интерфейса должно быть помещено в синтаксическую конструкцию:

```
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).



```
1 /*
2 |     Интерфейс распределенного приложения OrbPad
3 |     Используется компилятор idlj
4 */
5
6 // Начало модуля
7 module idlmodule {
8
9     // Определение типа StrArray - массив строк.
10    typedef sequence <octet> StrArray;
11
12    // Начало интерфейса
13    interface OrbPad {
14
15        // Объявление методов:
16
17        // метод, проверяющий наличие соединения с БД;
18        boolean isConnect();
19
20        // метод, получающий содержимое таблицы notepad БД
21        // exampleDB в виде списка текстовых строк;
22        StrArray getList();
23
24        // метод, добавляющий текст к содержимому
25        // таблицы notepad БД; также учитывается
26        // уникальность ключа key;
27        long setInsert(in long key, in StrArray str);
28
29        // метод, удаляющий по заданному ключу key запись из
30        // таблицы notepad БД;
31        long setDelete(in long key);
32
33        // метод, закрывающий соединение с базой данных.
34        oneway void setClose();
35
36    }; // Конец интерфейса
37
38 }; // Конец модуля
```

Тип файла: IDL | Строка: 2 Столбец: 0 | ЗАМ

Рисунок 3.13 — Описание интерфейсов OrbPad на языке IDL

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

- 1) *комментарии* в описании интерфейсов переносятся в исходные тексты генерируемых компонент, но компилятор *idlj* не поддерживает национальные языки;

- 2) не следует использовать строки (*string*) и массивы строк (*sequence string*), поскольку *возникают ошибки кодирования/декодирования*; лучше их заменять массивами байт (*sequence octet*);
- 3) все описываемые методы будут иметь модификатор *public*;
- 4) если метод не возвращает данные (имеет тип *void*) и клиент не будет ожидать завершения этого метода, то следует использовать специальный модификатор *oneway*;
- 5) 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.

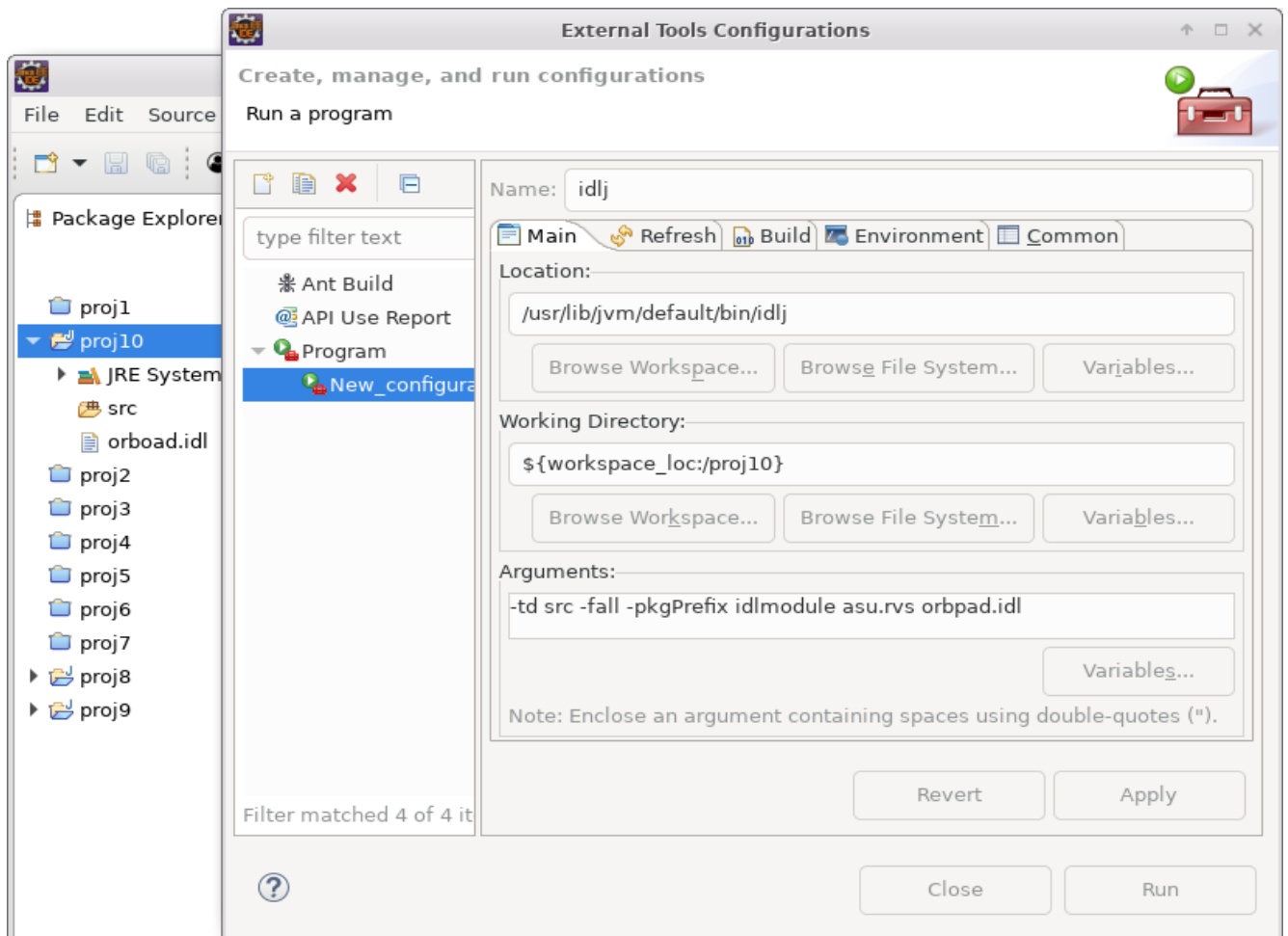


Рисунок 3.14 — Задание параметров вызова компилятора idlj

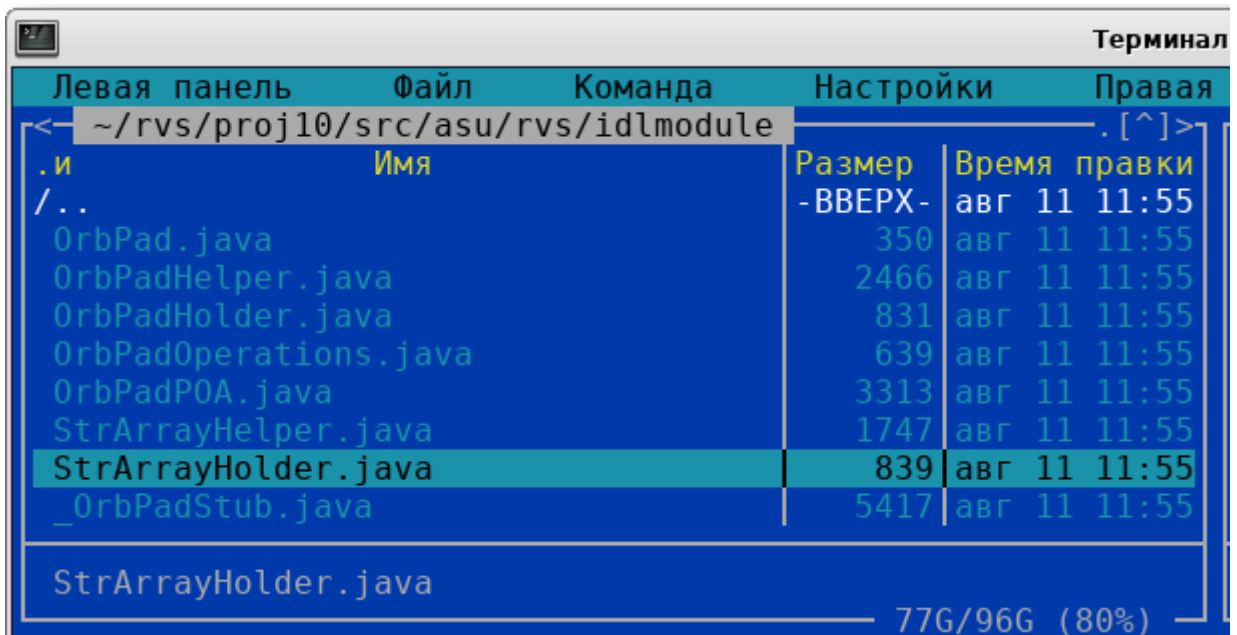


Рисунок 3.15 — Полный список сгенерированных файлов проекта proj10

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 3.3а и являющийся интерфейсом для сервера и клиентских приложений. Он описывает простое объединение трёх интерфейсов:

- а) ***org.omg.CORBA.Object*** и ***org.omg.CORBA.portable.IDLEntity*** — два интерфейса пакета ***org.omg.CORBA***;
- б) ***OrbPadOperations*** — определённый в файле ***OrbPadOperations.java*** интерфейс, объявляющий методы распределенного объекта ***OrbPad*** (см. листинг 3.3б).

Листинг 3.3а — 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 3.3б — 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 3.3б 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:

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

Теперь перейдём к реализации программы сервера.

3.2.5 Реализация серверной части ORB-приложения

На четвёртом шаге проектирования создадим серверную часть ORB-приложения, которая обеспечивает интерфейс удалённого объекта **OrbPad**.

Простейший вариант архитектуры такого приложения состоит из двух классов:

- 1) *собственно класса сервера* (назовём его **OrbPadServer**), имеющего статиче-

ский метод **main()**, организующего приём запросов от клиентов и отправку им результатов выполненных методов;

- 2) 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 notepad 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 notepad 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:

- 1) создать объект класса *NotePad* и проверить его готовность к работе;
- 2) создать и инициализировать объект класса **ORB**, являющийся локальным представителем брокера в программе сервера и показанный ранее на рисунке 3.7 как компонент *ORB сервер*;
- 3) создать объект класса *POA*, являющийся локальным представителем (адаптером) переносимых объектов сервера, и активировать его;
- 4) создать объект класса *OrbPadServer*, являющийся сервантом, и передать ему ссылки на созданные объекты классов *NotePad* и **ORB**;
- 5) с помощью менеджера *POA* создать объект класса *org.omg.CORBA.Object*, являющийся специальной ссылкой на объект серванта;
- 6) 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;
- 7) создать объекты-ссылки на службу имён «*NameService*» отдельного сервера **ORB**, который уже должен быть запущен;
- 8) провести регистрацию удалённого объекта на сервере ORB под именем «*OrbPad*»;
- 9) 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.isConnected())
        {
            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 создаём
             * удалённый 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 удалённый объект
 * в службе имён ORB-сервера
 */
NameComponent path[] =
    ncRef.to_name( "OrbPad" );
ncRef.rebind(path, orbpad);

System.out.println("OrbPadServer готов и ждёт ...");

// 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 приёма 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:

- 1) создадим новый проект;
- 2) скопируем в него файл описания интерфейса *orbpad.idl*;
- 3) проведём генерацию компонент приложения, как это было описано в предыдущем пункте.

Приложение клиента реализуем в виде класса *OrbPadClient*. Для чего выполним следующие действия:

- а) создадим новый Java-класс с указанным именем, где в качестве операнда оператора *package* укажем *asu.rvs.client*;
- б) 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:

- а) создать объект класса *OrbPadClient* — для доступа к собственным методам класса;
- б) создать и инициализировать локальный объект класса *ORB* — для доступа к серверу *ORB*;
- в) создать объекты-ссылки — для доступа к серверу *ORB*;
- г) создать объект класса *OrbPad*, являющийся ссылкой на удалённый объект и зарегистрированный на сервере *ORB* под именем «*OrbPad*», — для доступа к методам интерфейса;
- д) 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 удалённый 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 введённого 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 = opс.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("\nЗавершать работу сервера? (Enter - нет): ");
s = opc.getString();

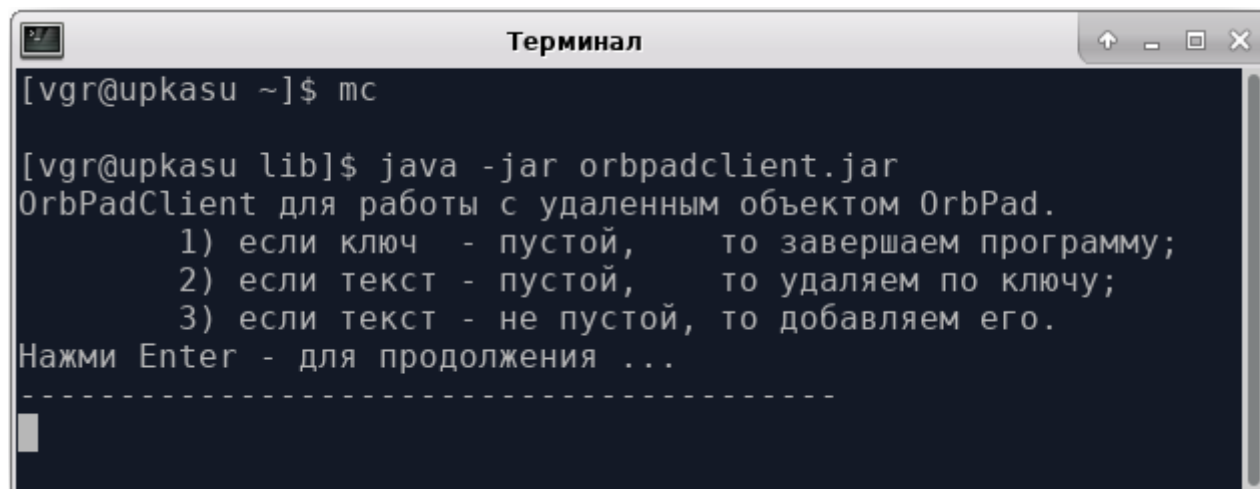
if (s.length() > 0)
    orbpad.setClose(); // Завершаем работу сервера

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.



```
[vgr@upkasu ~]$ mc

[vgr@upkasu lib]$ java -jar orbpadclient.jar
OrbPadClient для работы с удаленным объектом OrbPad.
    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 ...
-----
█
```

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.

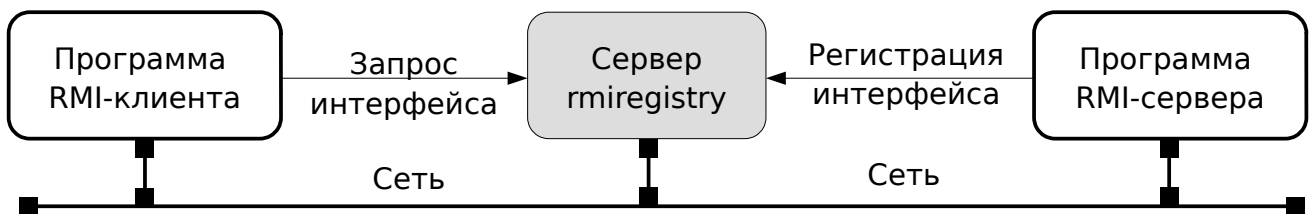


Рисунок 3.17 — Общая организация технологии RMI

Как и технология CORBA, RMI основана на описании интерфейса удалённого объекта, но отличается тем, что:

- а) *использует* стандартное описание интерфейса на языке Java;
- б) *является* расширением интерфейса **java.rmi.Remote**;
- в) *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 локально-

го приложения и обеспечивающей способность получения интерфейса удалённого объекта с сервера *rmiregistry*.

Поскольку теория создания объектных распределённых 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.

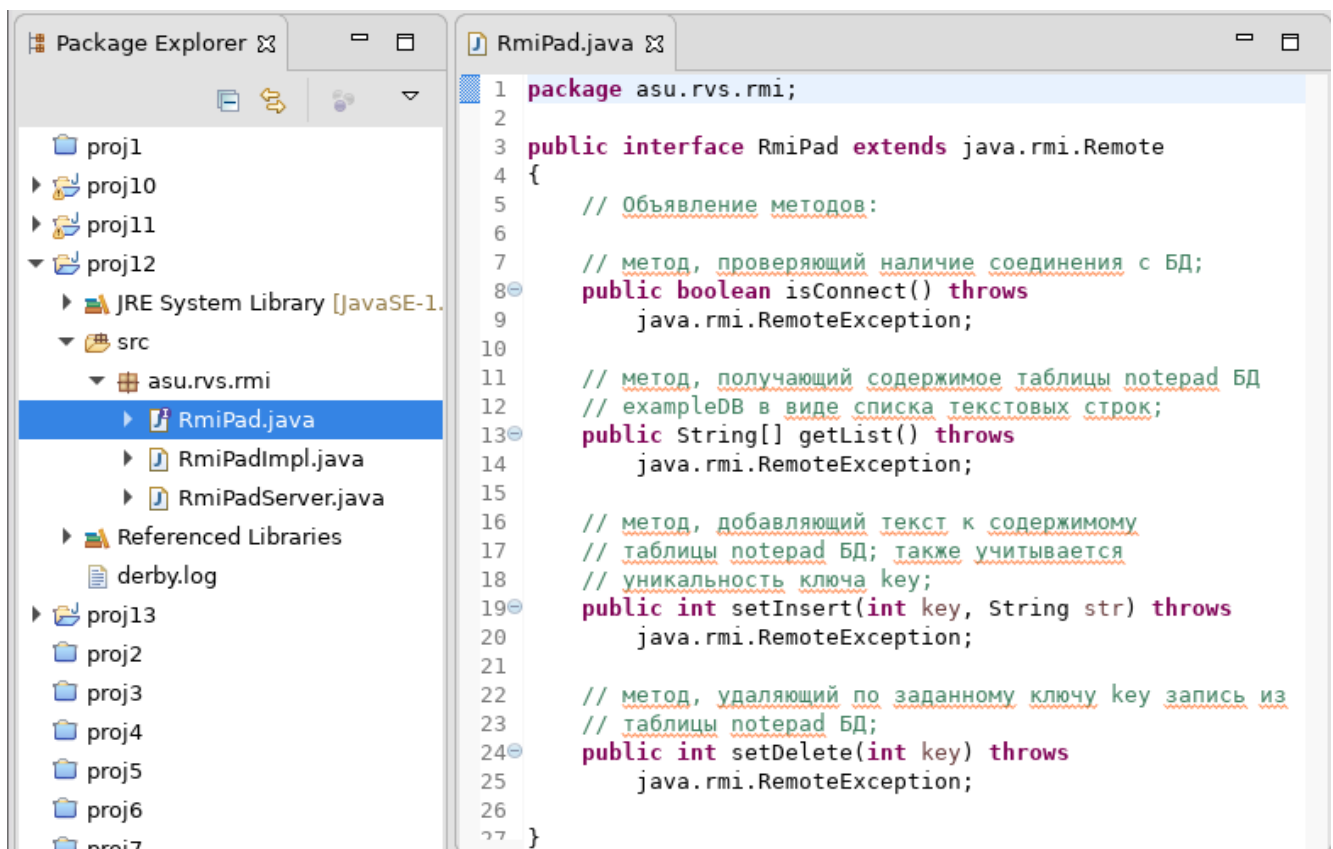


Рисунок 3.18 — Пример интерфейса удалённого объекта в технологии RMI

Представленный интерфейс имеет следующие три особенности:

- 1) удалён метод **`void setClose()`**, поскольку его отсутствие не уменьшает функциональности приложения, а назначение является достаточно одиозным для удалённых объектов, разрешающих клиентским приложениям останавливать сервера;
- 2) аргументы методов, передающих строковые значения, заменены на «родной» для языка Java тип — ***String***;
- 3) 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:

- а) ***RmiPad.java*** — описание интерфейса, уже реализованного в проекте ***proj12***;
- б) ***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;
- в) ***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 удалённым объектом:

- а) должен расширять класс *java.rmi.server.UnicastRemoteObject*, обеспечивающий экспорт удалённого объекта с помощью протокола *JRMP*;
- б) реализовать описанный интерфейс (в нашем случае — *asu.rvs.rmi.RmiPad*);
- в) являться публичным классом (*public*) и имеет статический номер версии (*serialVersionUID*), который генерируется и вставляется средой Eclipse EE;
- г) имеет один или несколько защищённых (*protected*) конструкторов (в нашем случае имеется один конструктор, передающий в класс ссылку на экземпляр класса *NotePad*, созданный компонентой сервера);
- д) 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 {
    /**
     * Создаётся экземпляр удалённого объекта класса RmiPadImpl,
     * которому передаётся ссылка на экземпляр класса NotePad.
     */
    RmiPadImpl impl =
        new RmiPadImpl(obj);

    /**
     * Регистрация экземпляра удалённого 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. Создаёт объект класса *NotePad*, который обеспечивает взаимодействие с таблицей *notepad* базы данных *exampleDB*.
2. Создаёт удалённый объект класса *RmiPadImpl*, передавая ему через конструктор ссылку на объект класса *NotePad*.
3. Регистрирует удалённый 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(...)* ещё 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 удалённые объекты в реестре удалённых 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 (удалённый 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 удалённому объекту — это привязка или регистрация имени для удалённого 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()*...).

Замечание — Приведённая 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 имён — *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:

- а) *void bind(String name, Remote obj)* — регистрирует удалённый объект *obj* под именем *name*;
- б) *String[] list(String name)* — возвращает массив имён, связанных в реестре и представляющих собой строки в формате URL (без компонента схемы); массив содержит снимок имён, присутствующих в реестре на момент вызова;
- в) *Remote lookup(String name)* — возвращает ссылку на удалённый объект, связанный с указанным именем.
- г) *void rebind(String name, Remote obj)* — проводит перерегистрацию удалённого объекта *obj* под именем *name*;
- д) *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).

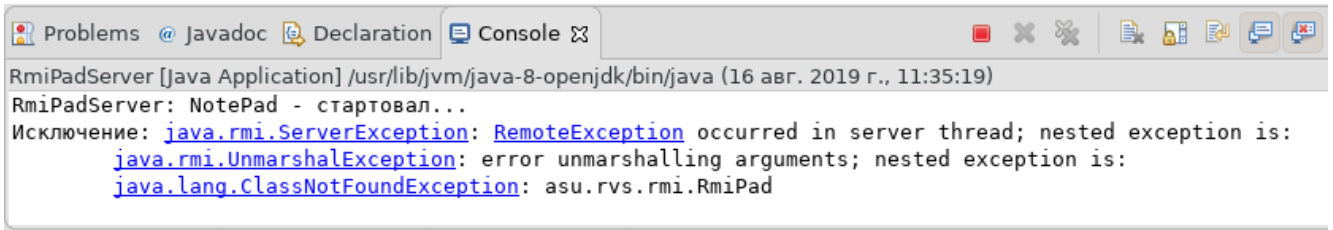


Рисунок 3.19 — Ошибочное сообщение при запуске сервера RmiPadServer

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*:

- а) программа разработана пользователем *vgr*, поэтому *\$HOME=/home/vgr*;
- б) программа разработана в среде Eclipse EE с базовым адресом всех проектов *\$HOME/rvs*;
- в) программа разработана в проекте *proj12*, поэтому адрес искомого файла, с учётом префикса *asu.rvs.rmi*, будет: */home/vgr/rvs/proj12/bin*.

Теперь 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*.

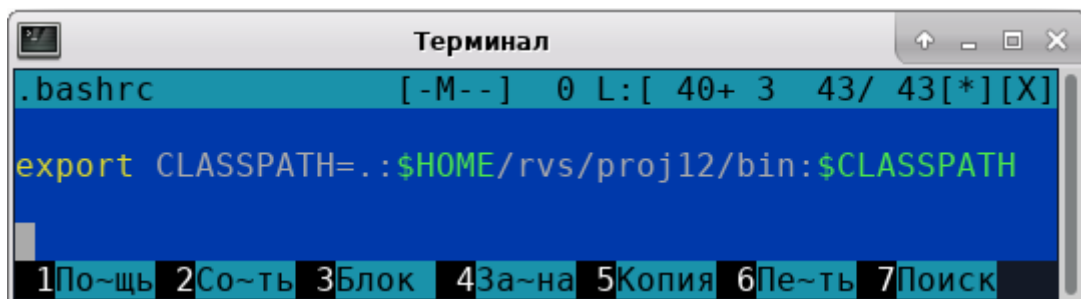


Рисунок 3.20 — Добавление пути в системную переменную CLASSPATH

В результате, запущенный сервер сообщит о нормальной регистрации на сервере имён *rmiregistry*, как показано на рисунке 3.21.

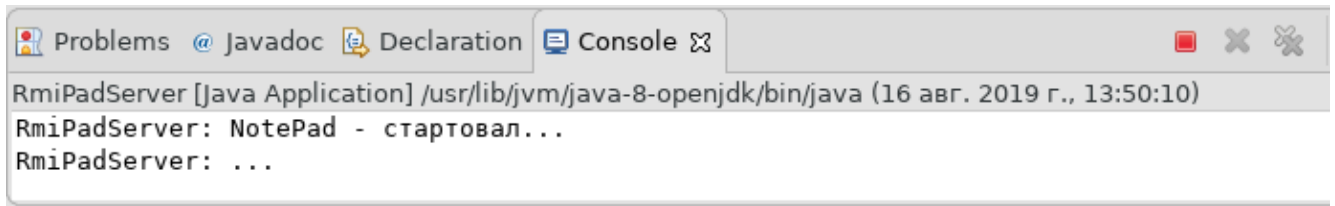


Рисунок 3.21 — Нормальный старт сервера RmiPadServer

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 её 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.rmicient*, 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.rmicient;

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 удалённый 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 её работоспособности, как показано на рисунке 3.22.

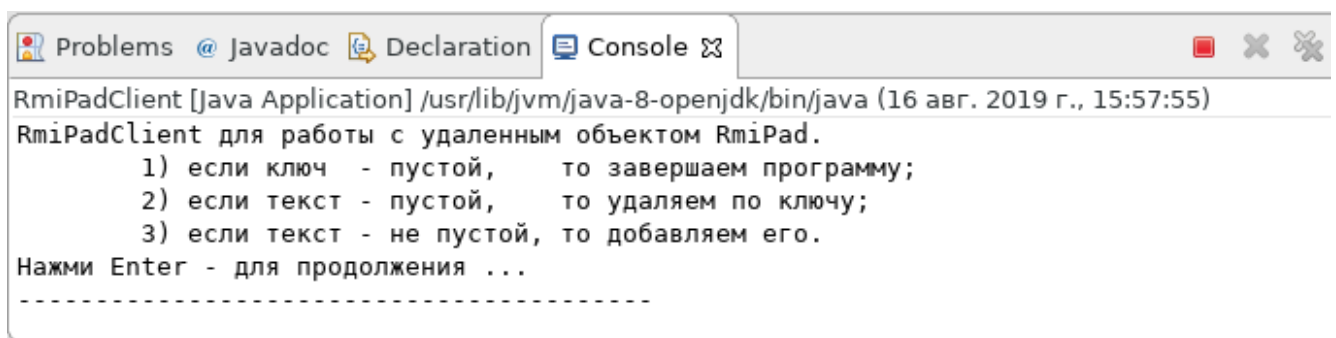


Рисунок 3.22 — Начало диалога приложения RmiPadClient

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:

- а) в среде разработки Eclipse EE: остановить работу RMI-клиента, а затем — RMI-сервера;
- б) 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.

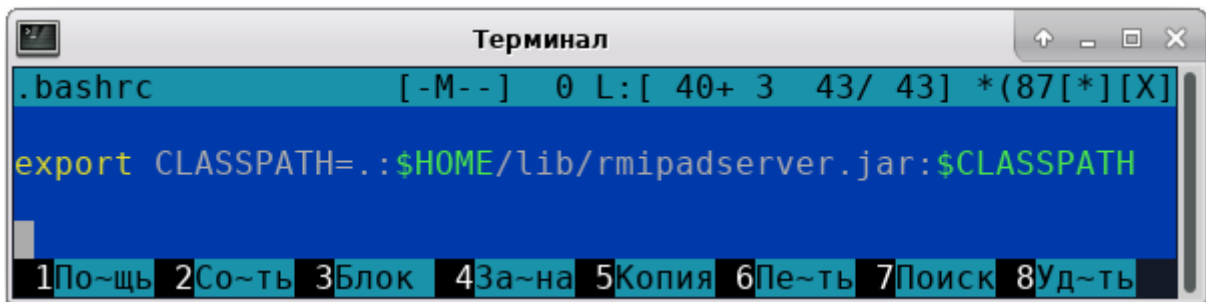


Рисунок 3.23 — Редактирование .bashrc для архива RMI-сервера

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.

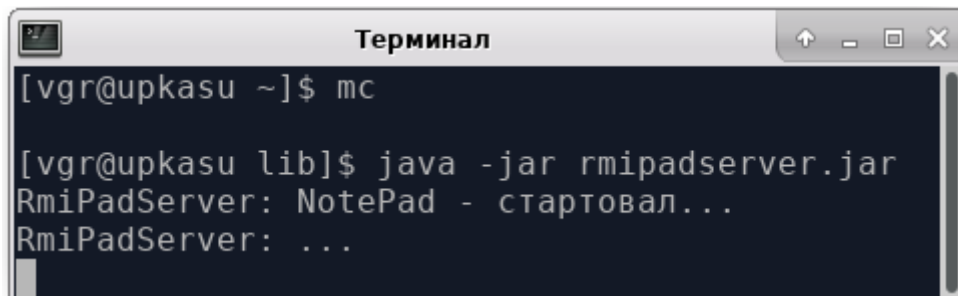


Рисунок 3.24 — Запуск RMI-сервера из архива rmipadserver.jar

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:

- а) тестирование RMI-клиента в среде Eclipse EE;
- б) создание архива проекта *proj13* и размещение его в каталоге *\$HOME/lib/*;
- в) 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 распределенным 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:

- а) первоначально создаётся локальный вариант решения задачи, предполагающий будущее разделение на клиентскую и серверную части; такие части реализованы в архивах *notepad.jar* и *example12.jar*;
- б) решение этой задачи средствами технологии CORBA представлена архивами *orbpadserver.jar* и *orbpadclient.jar*;
- в) решение этой задачи средствами технологии RMI представлена архивами *rmibpadserver.jar* и *rmipadclient.jar*.

```

Терминал

[vgr@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

Терминал			
Левая панель	Файл	Команда	Настр
< ~/lib .[^]>			
.и	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?