

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  
2024

**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, 2024. – 66 с.

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

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

Одобрено на заседании каф. АСУ протокол № \_\_\_\_ от \_\_\_\_\_

УДК 004.75

© Резник В. Г., 2024

© Томск. гос. ун-т систем упр. и радиоэлектроники, 2024

## Оглавление

<b>3 Тема 3. Объектные распределенные системы.....</b>	<b>4</b>
3.1 Брокерные архитектуры.....	5
3.1.1 Критика классической сетевой модели «Клиент-сервер».....	6
3.1.2 Критика классической модели хранения данных.....	6
3.1.3 Критика теоретической брокерной модели проху-серверов.....	7
3.1.4 Брокерная архитектура сильносвязанных систем.....	8
3.2 Программное проектирование распределённой системы.....	9
3.2.1 Общая функциональная модель процесса проектирования распределённого приложения.....	9
3.2.2 Функциональная модель учебного примера.....	11
3.2.3 Проект описания интерфейса NotePad.....	13
3.2.4 Проект серверной части приложения NotePadImpl.....	15
3.2.5 Проект клиентской части приложения Example12.....	21
3.2.6 Завершение проектирования учебного прототипа OPC.....	26
3.3 Технология RMI.....	28
3.3.1 Реализация интерфейса приложения для технологии RMI.....	30
3.3.2 Реализация сервера для технологии RMI.....	32
3.3.3 Реализация клиента для технологии RMI.....	38
3.3.4 Завершение реализации проекта для технологии RMI.....	42
3.4 Технология CORBA.....	44
3.4.1 Брокерная архитектура технологии CORBA.....	45
3.4.2 IDL CORBA и генерация распределённого объекта OrbPad.....	47
3.4.3 Реализация серверной части OPC OrbNotePad.....	53
3.4.4 Реализация клиентской части OPC OrbNotePad.....	59
3.5 Выводы по результатам всей темы.....	65
Вопросы для самопроверки.....	66

### 3 Тема 3. Объектные распределенные системы

**Учебная цель данного раздела** — изучение технологий реализации *объектных распределённых систем (ОРС)* применительно задачам, охватывающим выделенную ранее предметную область «*Сетевые объектные системы*» (см. подраздел 1.2).

Примечание — Данная тема открывает последовательность из *пяти разделов* по систематическому изучению технологий распределённых систем, которые ориентированы на реализацию средствами языка Java.

**Заявленная тематика раздела** охватывает все классические сетевые приложения модели OSI, основанные на общей теоретической парадигме «*Клиент-сервер*». В технологическом плане эти задачи соответствуют моделям *распределённых вычислительных сред (DCE)*, *общей объектной брокерной архитектуре запросов (CORBA)* и *вызовам удалённых методов (RMI)*.

**Учебный материал** данной темы разделён на пять частей, последовательно раскрывающих следующие теоретические и практические аспекты учебной цели:

- 1) *подраздел 3.1* — брокерные архитектуры, составляющие идейную основу объектных распределённых систем;
- 2) *подраздел 3.2* — пример проектирования распределённого приложения, основанного на учебном примере Java-класса *Example11*;
- 3) *подраздел 3.3* — реализация учебного примера средствами технологии *RMI*, что демонстрирует применение «*Межброкерного протокола для Интернет*» (*IIOP, Internet InterORB Protocol*);
- 4) *подраздел 3.4* — реализация учебного примера средствами технологии *CORBA*, что демонстрирует применение протокола *IIOP* для служб *сильносвязанных распределённых систем*, не привязанных к конкретному языку программирования;
- 5) *подраздел 3.5* — краткие выводы по результатам всей темы.

Примечание — Технологии объектных распределённых систем (ОРС) являются естественным этапом развития технологий объектно-ориентированного программирования (ООП).

**Побудительная причина** развития технологий объектных распределённых систем (ОРС) — распространение существующих технологий объектно-ориентированного программирования (ООП) на предметную область классических сетевых технологий.

Применение новых технологий должно обеспечить:

- 1) *увеличение мощности вычислительных ресурсов* необходимых для реализации масштабных приложений;
- 2) *использование разработчиками уже имеющегося сложного ПО* (программного обеспечения), которое реализовано и отлажено на удалённых ЭВМ;
- 3) *реализацию методами ООП задач*, которые предполагают разграничение прямого доступа к необходимым для обработки данным.

Примечание — Технологии *ОРС* — технологии *сильносвязанных распределённых систем (CPC)*.

**Основная технологическая идея** реализации сильносвязанных распределённых систем (CPC) — *использование сетевых брокерных архитектур*, которые бы обеспечили создание «*стандартного*» системного ПО в минимальной степени усложняющего применение классических технологий языков ООП.

### 3.1 Брокерные архитектуры

**Брокерные архитектуры** — разновидность классической сетевой модели «Клиент-сервер», которая предполагает использование специального вида серверов, выполняющих функции *брокера* или *посредника* между ПО программ клиентов и ПО программ серверов прикладного назначения.

**Брокер** — посредник, который:

- 1) принимает запросы от программ клиентов;
- 2) предаёт эти запросы серверам прикладного назначения; ожидает от серверов ответы на запросы;
- 3) передаёт полученные от серверов ответы программам клиентов.

Применительно к учебному материалу изучаемой дисциплины классическую модель «Клиент-сервер» можно упрощённо представить рисунком 3.1, а упрощённую брокерную архитектуру — рисунком 3.2.

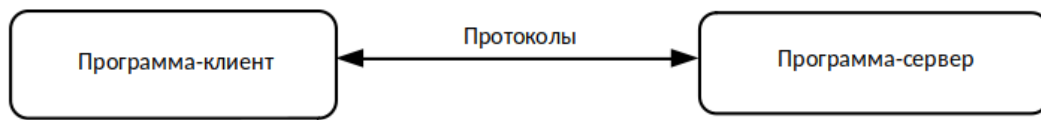


Рисунок 3.1 — Классическая сетевая модель взаимодействия «Клиент-сервер»



Рисунок 3.2 — Брокерная архитектура сетевой модели взаимодействия «Клиент-сервер»

Теоретически показанная на рисунке 3.2 брокерная архитектура предоставляет программам-клиентам *«Интерфейсы»*, тем самым предлагая:

- 1) *стандартизировать* и *упростить* взаимодействие программ-клиентов с серверами приложений;
- 2) *переложить* все сложности реализации протоколов и ПО серверов на разработчиков серверных приложений.

Примечание — Безусловно брокерная архитектура сетевой модели взаимодействия «Клиент-сервер» является частным случаем «Систем (служб) промежуточного уровня» или *Middleware*.

Хотя такое определение брокерных архитектур может вызывать массу нареканий различных специалистов, нам следует учесть, что:

- 1) сам термин «Брокерные архитектуры» является зонтичным и заимствован из сфер финансовой, торговой или страховой деятельности;
- 2) применительно к тематике нашей дисциплины этот термин может пониматься как «Служба промежуточного уровня» (*Middleware*), например, как это определено Эндрю Таненбаумом в источнике [3] и ранее представлено на рисунке 1.6;
- 3) применительно к тематике «Сетевых объектных систем», рассмотренной нами ранее в подразделе 1.2, этот термин можно понимать как *специализированный вариант проху-сервера*, выполняющего посреднические функции по передаче сообщений и объектных запросов между приложениями клиентов и серверов.

Примечание — В любой интерпретации указанной выше, контекст термина «*Брокерные архитектуры*» должен раскрываться дополнительно и рассматриваться как развитие классической модели «Клиент-сервер».

Чтобы более качественно раскрыть технологии ОРС применительно к тематике сильно-носвязанных распределённых систем (СРС), следует хотя бы кратко ответить на следующие вопросы:

1. В чём состоят недостатки классической сетевой модели «Клиент сервер»?
2. В чём состоят недостатки классической модели хранения данных?
3. В чём недостатки теоретической брокерной модели ргоху-серверов?
4. В чём состоит основная суть брокерной архитектуры сильноносвязанных систем?

### 3.1.1 Критика классической сетевой модели «Клиент-сервер»

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

Анализируя пример простейшей задачи, рассмотренной ранее в подразделе 2.5 «Управление сетевыми соединениями» (см. пункт 2.5.5 и листинги 2.13 и 2.14), мы можем констатировать, что реализованные две *Java-программы* клиента и сервера имеют следующие технологические свойства:

- 1) **обе программы** соответствуют классической модели «Клиент-сервер», реализованной на базе протокола TCP пакета *java.net*;
- 2) **сами программы** реализуют собственный *прикладной 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;
- 3) **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;
- 4) **прикладное наполнение** этих программ *требует разработки дополнительных протоколов*; например, проведение структуризации передаваемых сообщений в виде команд и последующей реализации программного обеспечения, поддерживающего эти команды.

В качестве недостатка такой технологии следует отметить, что с ростом функционального наполнения *классической сетевой модели* растёт не только объем прикладного ПО клиента и сервера, но и объем ПО, поддерживающего все новые протоколы взаимодействия, поскольку каждая подобная задача реализуется самостоятельно.

### 3.1.2 Критика классической модели хранения данных

Для анализа заявленного вопроса рассмотрим *типовой пример выборки данных*, который был изучен ранее в подразделе 2.6 «Организация доступа к базам данных». Реализация данного примера описана в пункте 2.6.3 как программа клиента на языке Java (см. листинг 2.15), которая для доступа к базе данных использует инструментальный пакет *java.sql*.

Примечание — В этом примере, как и в классической сетевой модели, также выделяются *клиент, сервер и протоколы*, но две последних составляющих реализуются не самим программистом, а берутся им как законченные продукты или готовые инструментальные средства.

Рассмотренный пример хоть и построен по классической модели «Клиент-сервер», но в плане используемой технологии использует промежуточное ПО (*Middleware*) в виде сервера СУБД. Будет ли взаимодействие клиента и сервера сетевым или «встроенным» зависит от используемого драйвера JDBC и не влияет на прикладную часть приложения клиента.

Примечание — Формально классические модели хранения данных с использованием серверов СУБД не рассматриваются в рамках теории распределённых систем (РВС), поскольку являются сильно специализированными и уже состоявшимися технологиями. Тем не менее их можно рассматривать как ранние прототипы брокерных РВС.

Заметим также, что технология рассмотренного примера не зависит от того, использовался ли язык ООП Java или, например, язык программирования С. Для реляционных моделей хранения данных программисту предоставляется достаточно высокоуровневый язык манипулирования данными (SQL), но он сам вынужден реализовывать прикладные протоколы запросов к СУБД.

**В качестве недостатка** классической модели хранения данных следует отметить:

- 1) *сильная специализация технологии* на простейшую обработку данных, связанную моделью её хранения: иерархическая, реляционная или сетевая;
- 2) *специализация языков запросов* к серверу, которая сильно отличается от синтаксиса языков программирования прикладных систем.

### 3.1.3 Критика теоретической брокерной модели проху-серверов

**Брокерная модель проху-серверов** — теоретическое и прикладное направление создания программных посредников нацеленное на упрощение технологий создания *программ-клиентов*.

Обратимся за комментариями к Википедии [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...».

Примечание — Представленное выше описание проху-серверов настолько расширяет само понятие брокера (посредника), что низводит само определение до семантики таких понятий как *Middleware* и *ESB*, рассмотренных ранее в разделе 1.

**В качестве недостатка** модели проху-серверов и в подтверждение к сказанному в примечании отметим наличие различных типов «*прокси*», ключевыми из которых являются:

- 1) пересылающие прокси (forward proxies);
- 2) прозрачные прокси (transparent proxies);
- 3) кеширующие прокси (caching proxies);
- 4) прокси обеспечения безопасности (security proxies);
- 5) обратные прокси (reverse proxies).

### 3.1.4 Брокерная архитектура сильносвязанных систем

**Брокерная архитектура сильносвязанных систем** — технология проектирования и создания прокси-серверов, в основе которой лежит технология использования языков описания интерфейсов (IDL).

Википедия даёт следующее определение понятию IDL [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:

1. **AIDL**: Реализация IDL на Java для Android, поддерживающая локальные и удалённые вызовы процедур. Может быть доступна из нативных приложений посредством JNI.
2. **CORBA IDL** — язык описания интерфейсов распределённых объектов, разработанный рабочей группой OMG. Создан в рамках обобщённой архитектуры CORBA.
3. **IDL DCE**, язык описания интерфейсов спецификации межплатформенного взаимодействия служб, которую разработал консорциум Open Software Foundation (теперь The Open Group).
4. **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.
5. **COM IDL** — язык описания интерфейсов между модулями COM. Является преемником языка IDL в технологии DCE («среда распределённых вычислений») — спецификации межплатформенного взаимодействия служб, которую разработал консорциум Open Software Foundation (теперь The Open Group)...

Примечание — Обратите внимание, что различных IDL — много и они ориентированы как на использование вызова удалённых процедур (RPC), среду распределённых вычислений (DCE), так и на запросы к удалённым объектам.

Согласно тематике данного раздела мы рассмотрим только IDL распределённых объектов, архитектурная модель которых показана ранее на рисунка 3.2. Среда исполнения языка Java (JRE), используемая в данном пособии для демонстрации примеров распределённых систем (RBC), предоставляет нам две реализации брокеров:

- 1) брокер **orbd** из пакета **org.omg**, поддерживающий интерфейс CORBA IDL;
- 2) брокер **rmiregistry** из пакета **java.rmi**, поддерживающий классическое описание интерфейса на языке Java.

Примечание — В целом, брокерная архитектура сильносвязанных систем ориентирована на создание технологии RBC, вносящей минимальные изменения в традиционные технологии создания сосредоточенных систем.

Действительно, в дополнение к архитектуре рисунка 3.2, программы клиента и сервера дополняются следующим системным ПО:

- 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** (**stub**, **skeleton**), реализующий методы **демаршалинга**: преобразования входного потока данных в запрос к объекту сервера.



## 3.2 Программное проектирование распределённой системы

**Типовая задача** проектирования объектной распределённой системы — преобразование сосредоточенной объектной системы в распределённую, когда единый функционал приложения разделяется на программные компоненты:

- 1) *описание интерфейса* распределённой системы на языке программирования сосредоточенной объектной системы, обеспечивающей внешние требования по разделению функционала клиентской и серверной частей программного обеспечения;
- 2) *создание и тестирование классов программы-сервера*, обеспечивающих функциональность описанного интерфейса распределённого приложения;
- 3) *создание и тестирование классов программы-клиента*, которые будут подключаться к удалённым объектам через сервер брокера, согласно известным и зарегистрированным интерфейсам.

Примечание — Решение указанной типовой задачи создаст *компоненты прототипа* целевой распределённой системы, поэтому её необходимо осуществить до выбора конкретной технологии реализации сильносвязанной распределённой системы.

**Решение типовой задачи** проектирования распределённой системы (РВС) проведём на примере сосредоточенного приложения доступа к базам данных, описанного и реализованного на языке Java в подразделе 2.6 данного пособия, как класс *Example11*.

**Последовательность процесса решения** указанной задачи излагается в следующих пяти пунктах:

- 1) пункт 3.2.1 — описывает общую *функциональную модель проектирования* распределённого приложения;
- 2) пункт 3.2.2 — описывает функциональную модель учебного примера, конкретизирующего общую функциональную модель проектирования распределённого приложения;
- 3) пункт 3.2.3 — описывает проект *интерфейса* серверной части распределённого приложения, представленного классом *NotePad* на языке Java;
- 4) пункт 3.2.4 — описывает проект *серверной прикладной части* распределённого приложения, представленного классом *NotePadImpl* на языке Java;
- 5) пункт 3.2.5 — описывает прототип *клиентской прикладной части* распределённого приложения, представленного классом *Example12* на языке Java;
- 6) пункт 3.2.6 — подводит итог учебного материала всего подраздела.

### 3.2.1 Общая функциональная модель процесса проектирования распределённого приложения

Преобразование сосредоточенной программной системы в прототип распределённой системы это — *процесс преобразования* исходных текстов программного обеспечения (ПО), который необходимо провести перед началом реализации самой целевой системы. В дальнейшем такое преобразование должно конкретизироваться для конкретных систем РВС.

**Учебная тема данного пункта** — структурный функциональный анализ процесса решения типовой задачи проектирования объектной распределённой системы.

**Моделирование процесса решения типовой задачи** проведём на основе методологии IDEF0 при следующих условиях:

- 1) требования к моделированию определяются «Заказчиком» модели;
- 2) моделирование проводится специалистом IT названным «Проектировщик»;
- 3) цель: «Структурное функциональное описание методики построения прототипа рас-

пределённого приложения»;

4) точка зрения: «Заказчик РВС».

При заданных условиях контекстная диаграмма модели может быть представлена рисунком 3.3 созданным в инструментальной среде Ramus Educational.

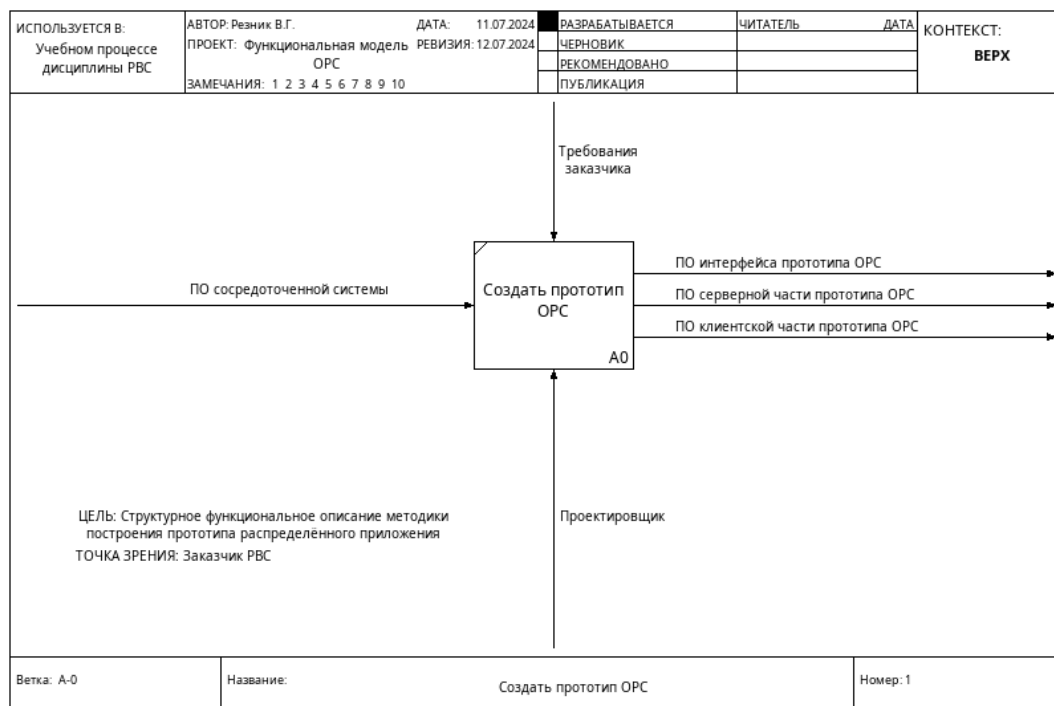


Рисунок 3.3 — Контекстная диаграмма общей функциональной модели проектирования прототипа ОРС

Примечание — Используемое сокращение ОРС подчёркивает, что функциональная модель строится для сильносвязанной объектной распределённой системы (ОРС).

В общем случае для детализации общей функциональной модели достаточно провести один уровень декомпозиции, состоящей из трёх блоков представленных в таблице 3.1.

Таблица 3.1 — Описание блоков первого уровня декомпозиции контекстной диаграммы

№	Название блока	Описание функции блока
A1	Создать ПО интерфейса прототипа ОРС	Исходный текст интерфейса создается на языке разработки ПО и включает те методы сосредоточенной системы, которые определил «Заказчик РВС».
A2	Создать ПО серверной части прототипа ОРС	Создать и протестировать ПО серверной части ОРС (имплементацию класса ОРС), реализующую методы описанные в интерфейсе прототипа ОРС.
A3	Создать ПО клиентской части прототипа ОРС	Создать и протестировать ПО клиентской части ОРС, используя ПО интерфейса и серверной части прототипа ОРС как библиотеку объектов.

Примечание — В общем случае прототип ОРС может содержать несколько интерфейсов и классов серверной части удалённого приложения.

С учётом синтаксиса и семантики требований, изложенных в таблице 3.1, декомпозиция блока A0 контекстной диаграммы может быть представлена рисунком 3.4.

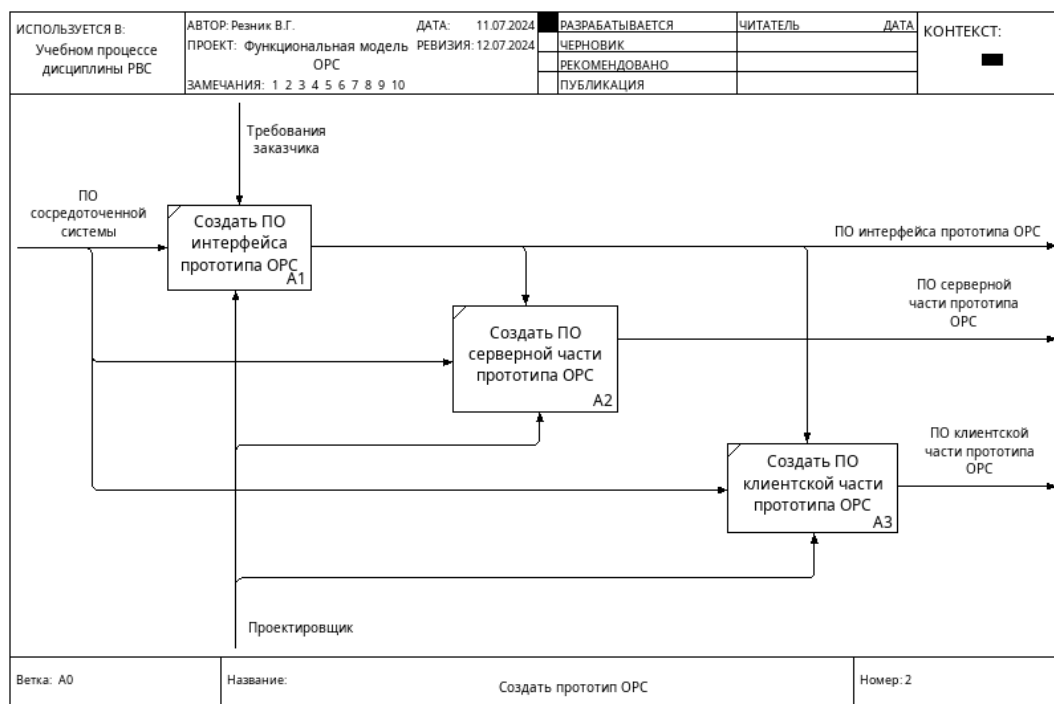


Рисунок 3.4 — Первый уровень декомпозиции общей функциональной модели ОРС

Примечание — Методология IDEF0 не ограничивает количество уровней декомпозиции контекстной диаграммы, но необходимость такого проектирования определяется требованиями «Заказчика».

Представленная на рисунке 3.4 декомпозиция контекстной диаграммы вполне покрывает потребности учебной темы данного пункта. В общем случае решение о развитии функциональной модели проектирования определяется следующей конкретизацией:

- 1) *содержимого и масштаба ПО* исходной сосредоточенной системы;
- 2) «Требованиями заказчика» системы и другими ограничивающими условиями, связанными с технологией разработки самой ОРС.

В дальнейших пунктах данного подраздела приводится конкретизация общей функциональной модели для учебного примера на основе сосредоточенной системы **Example11**.

### 3.2.2 Функциональная модель учебного примера

**Учебная тема данного пункта** — конкретизация общей функциональной модели процесса проектирования ОРС до семантики модели учебного примера.

Примечание — Конкретизация общей функциональной модели необходима для систематической последовательной регистрации всех проектных решений выбранной типовой задачи проектирования ОРС.

**На первом этапе** проводится конкретизация всех внешних материальных и информационных объектов, соответствующих контекстной диаграмме рисунка 3.3.

**Результат такой конкретизации** сначала отображается в таблице переопределения имён внешних объектов представленных в таблице 3.2.

**Окончательно**, результат переопределения внешних переносится в контекстную диаграмму нового конкретизированного проекта.

Контекстная диаграмма нового проекта показана ниже на рисунке 3.5.

Таблица 3.2 — Переопределение имён внешних объектов контекстной диаграммы

№ п/п	Исходные имена объектов	Имена объектов учебного примера
1	ПРОЕКТ: Функциональная модель ОРС	ПРОЕКТ: Прототип ОРС учебного примера
2	ЦЕЛЬ: Структурное функциональное описание методики построения прототипа распределённого приложения	ЦЕЛЬ: Структурное описание методики решения типовой проектной задачи для учебного примера
3	ТОЧКА ЗРЕНИЯ: Заказчик РВС	ТОЧКА ЗРЕНИЯ: Преподаватель дисциплины РВС
4	Требования заказчика	Требования преподавателя
5	ПО сосредоточенной системы	ПО класса Example11
6	ПО интерфейса прототипа ОРС	ПО интерфейса NotePad
7	ПО серверной части прототипа ОРС	ПО класса NotePadImpl
8	ПО клиентской части прототипа ОРС	ПО класса Example12
9	Проектировщик	Студент

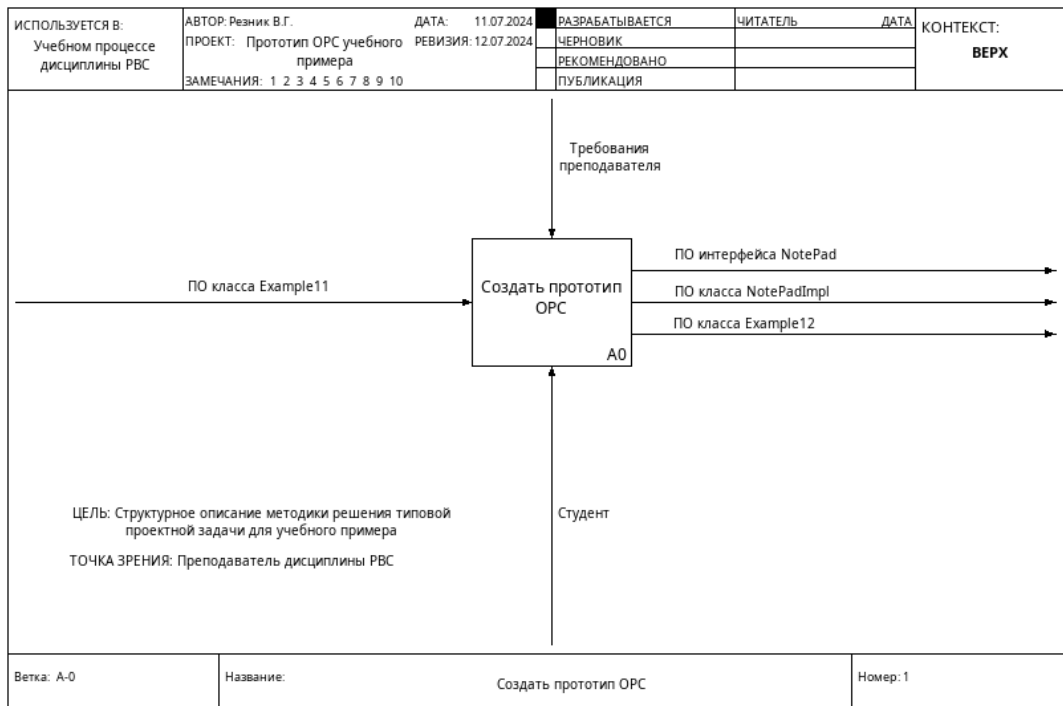


Рисунок 3.5 — Контекстная функциональная диаграмма модели учебного примера

Примечание — По правилам методологии IDEF0 каждый проект дополнительно к диаграммам должен содержать «Глоссарий» и «Текстовый документ» подробно описывающий все имена блоков, материальных и информационных объектов.

Поскольку учебная задача является максимально простой, мы не будем выполнять полное документирование модели по методологии IDEF0, а сразу рассмотрим главные особенности диаграммы декомпозиции *A0*, которая представлена ниже на рисунке 3.6:

- 1) *все три блока* на входе обрабатывают «ПО класса Example11» и на выходе выдают конкретный результат;
- 2) *главным блоком* является A1, создающий «ПО интерфейса NotePad» согласно управлению «Требования преподавателя»;
- 3) *управляющим входом* для блоков A2 и A3 является информационный объект «ПО интерфейса NotePad».

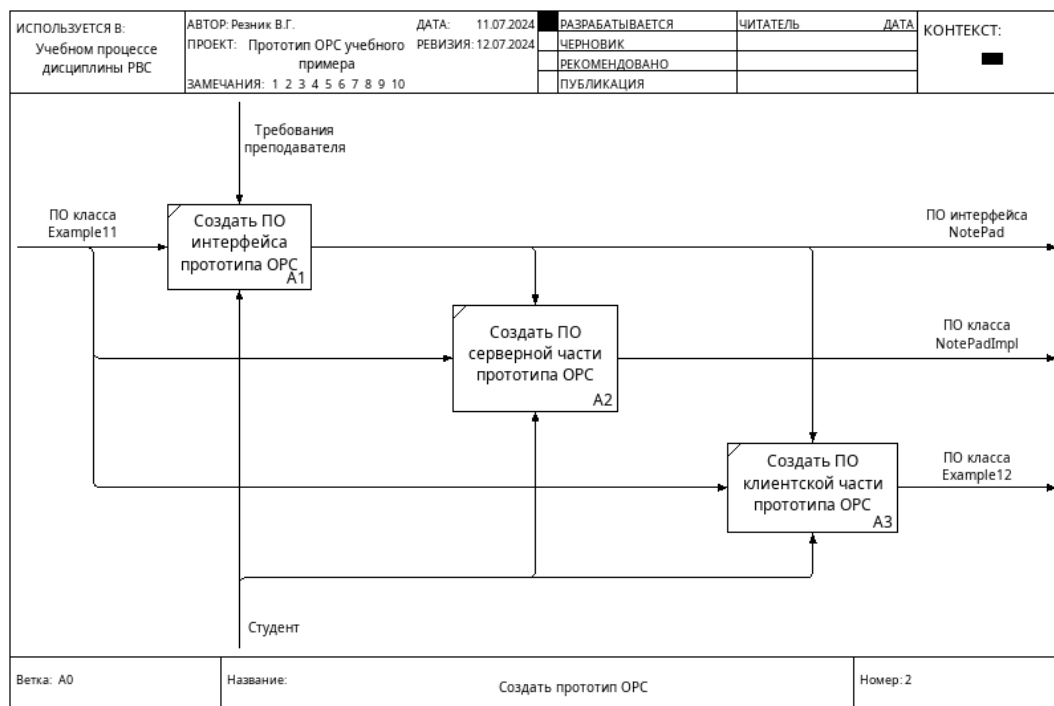


Рисунок 3.6 — Декомпозиция контекстной диаграммы учебного примера

Примечание — «Проектировщик» ОРС (*Студент*) обязан выполнить все «Требования заказчика» (*Требования преподавателя*).

Диаграммы рисунков 3.5 и 3.6 наглядно показывают, что функциональная модель учебной задачи нуждается в конкретизации управления «Требования преподавателя».

**Требования преподавателя** — управляющие ограничения на реализацию прототипа ОРС, которые определяются следующими ограничениями:

- 1) «ПО интерфейса *NotePad*» должно обеспечить полную функциональность клиентской части ОРС — «ПО класса *Example12*», эквивалентное «ПО класса *Example11*»;
- 2) «ПО класса *NotePadImpl*» должно содержать функциональность работы серверной части прототипа ОРС извлечённую из «ПО класса *Example11*», которая обеспечивает работу с таблицей *notepad* базы данных *exampleDB*, во встроенном (*embedded*) режиме работы с СУБД Derby.

Примечание — «Требования заказчика» (*Требования преподавателя*) не определяют последующую однозначную реализацию прототипа ОРС.

Конкретная реализация ПО прототипа ОРС выполнена в следующих трёх пунктах данного подраздела.

### 3.2.3 Проект описания интерфейса *NotePad*

**Учебная тема данного пункта** — реализация интерфейса *NotePad* на языке разработки Java в соответствии с ограничением «Требования преподавателя», которые декларированы в предыдущем пункте данного подраздела.

Примечание — По определению интерфейс *NotePad* должен содержать *описание методов*, которые будет использовать прототип клиентского приложения *Example12*.

**Описание методов** интерфейса *NotePad* содержит две группы:

- 1) методы, *реализующие прикладные запросы* к приложению сервера, определённые в та-

блице 3.3;

- 2) методы, реализующие обслуживающие запросы к приложению сервера, определённые в таблице 3.4.

Таблица 3.3 — Методы интерфейса NotePad, реализующие прикладные запросы

№ п/п	Определение метода	Назначение метода
1	<code>String[ ] getList()</code>	Метод, получающий содержимое таблицы <i>notepad</i> БД <i>exampleDB</i> в виде списка текстовых строк, причём первое слово каждой строки соответствует значению ключа <i>key</i> таблицы <i>notepad</i> .
2	<code>int setInsert(int key, String str)</code>	Метод, добавляющий текст <i>str</i> к содержимому таблицы <i>notepad</i> БД при условии задания уникального значения ключа <i>key</i> .
3	<code>int setDelete(int key)</code>	Метод, удаляющий по заданному ключу <i>key</i> запись из таблицы <i>notepad</i> БД <i>exampleDB</i> .

Примечание — Обратите внимание, что в таблице 3.3 отсутствует метод, модифицирующий таблицу *notepad* БД *exampleDB*.

Это связано с двумя причинами:

- 1) строгим следованием ограничениям «Требования преподавателя»;
- 2) стремлением преподавателя минимизировать объём кода ПО учебного примера.

Таблица 3.4 — Методы интерфейса NotePad, реализующие обслуживающие запросы

№ п/п	Определение метода	Назначение метода
1	<code>boolean setOpen()</code>	Метод, устанавливающий соединение с БД.
2	<code>void setClose()</code>	Метод, закрывающий соединение с базой данных.
3	<code>boolean isConnect()</code>	Метод, проверяющий наличие соединения с БД.

Примечание — Наличие обслуживающих запросов к удалённой части прототипа ОРС не нарушает общие «Требования заказчика», но содержат субъективные решения «Проектировщика».

Необходимость методов, определённых в таблице 3.4, обосновывается самим режимом (*embedded*) взаимодействия ПО серверной части ОРС с СУБД Derby. Этот режим предполагает *монопольный доступ приложений* к таблицам используемых баз данных (БД).

Выполнив формальное табличное определение методов интерфейса *NotePad*, выполним функционал блока *AI* диаграммы рисунка 3.6, реализующего его выход в виде «ПО интерфейса *NotePad*». Содержимое целевого интерфейса приведено в тексте листинга 3.1.

Листинг 3.1 — Исходный текст интерфейса NotePad из среды Eclipse EE

```
/**
 * Общий package для проекта учебного примера
 */
package asu.rvs;
/**
 * Интерфейс учебного примера
 *
 */
public interface NotePad
{
    /**
     * Методы, реализующие прикладные запросы
     */
    // Получить содержимое таблицы notepad
    String[ ] getList();

    // Добавить запись с ключом key и текстом str
    int setInsert(int key, String str);
}
```

```

// Удалить запись по заданному ключу key
int setDelete(int key);

/**
 * Методы, реализующие обслуживающие запросы
 */
// Установить соединение с БД exampleDB
boolean setOpen();

// Закрыть соединение с БД exampleDB
void setClose();

// Проверить наличие соединения с БД exampleDB
boolean isConnect();
}

```

Примечание — Разработчик приложения всегда должен помнить, что разработка интерфейса приложения всегда допускает различные интерпретации, поэтому следует максимально строго следовать внешним ограничениям, соответствующим «Требованиям заказчика».

### 3.2.4 Проект серверной части приложения NotePadImpl

**Учебная тема данного пункта** — реализация прототипа *серверной части ОРС*, заявленной на рисунке 3.6 как «ПО класса *NotePadImpl*».

**Основное требование** к реализации класса *NotePadImpl* — обеспечение функционала серверной части ОРС в соответствии с требованиями интерфейса *NotePad*.

**Дополнительные требования** реализации класса *NotePadImpl* — учёт следующих ограничивающих факторов:

- 1) учёт условий среды разработки целевого приложения;
- 2) учёт ограничений режима взаимодействия «*embedded*» между целевым приложением класса *NotePadImpl* и ПО СУБД Derby;
- 3) ограничения, связанные с последующим использованием целевой разработки;
- 4) ограничения, связанные с универсальностью целевой разработки.

Примечание — Удовлетворение всех дополнительных требований обычно обеспечивается условиями проведения лабораторных работ.

Прикладной характер учебной темы данного пункта требует обязательной конкретизации условий среды разработки целевого приложения, которое представлено в таблице 3.5.

Таблица 3.5 — Ограничивающие требования к среде разработки целевого приложения

№ п/п	Объект ограничения	Требование
1	Среда разработки	Eclipse EE
2	Имя проекта	proj8
3	Имя пакета	asu.rvs
4	Каталог целевого приложения	\$HOME/lib
5	Библиотечный файл целевого приложения	notepad.jar

Учёт ограничений режима взаимодействия «*embedded*» реализуется заданием статических переменных, контролирующих местоположение и уникальный доступ к БД *exampleDB*.

Это поясняется комментариями исходного текста целевого приложения, а использование объектов ограничения целевой разработки определяется требованиями таблицы 3.5.

**Универсальность целевой разработки** — минимальна, что обеспечивается статическим характером задания исходных данных целевого приложения.

Примечание — Реальное проектирование целевого приложения требует тщательного документирования дополнительных требований к реализации целевого приложения.

В данном пункте основные и дополнительные требования к целевому приложению *NotePadImpl* реализованы в виде комментариев его исходного теста представленного ниже на листинге 3.2.

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

```
/**
 * Общий package для проекта учебного примера
 */
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;

/**
 * Класс реализации серверной части учебного примера OPC,
 * реализующий интерфейс NotePad
 */
public class NotePadImpl implements NotePad
{
    /**
     * Объектные переменные класса в статическом исполнении
     */
    static boolean flag = false; // Нет соединения
    static Connection conn = null; // Нет объекта соединения
    // Адрес БД для режима embedded СУБД Derby
    static String url = "jdbc:derby:/home/upk/databases/exampleDB";
    static String user = "upk"; // Имя владельца БД
    static String password = "upkasu"; // Пароль владельца БД

    /**
     * Конструктор класса
     */
    public NotePadImpl() {}

    /**
     * Методы, реализующие ПРИКЛАДНЫЕ запросы к БД
     */

    /**
     * Метод ПОЛУЧЕНИЯ всех строк из таблицы notepad
     */
    @Override
    public String[] getList()
    {
        // Строка запроса к БД
        String sql = "SELECT * FROM notepad ORDER BY notekey";

        // Защита от неправомерного соединения
        if(!flag)
            return null;

        // Временное хранилище списка записей
        String al = "";

        // Обработка запроса с контролем исключения
        try
        {
            // Формирование объекта запроса
            Statement st =
                conn.createStatement();
```



```

        // Сам запрос и получение результата
        ResultSet rs =
            st.executeQuery(sql);

        // Проверка, что ответ - не пустой
        if(rs == null)
            return null;

        // Цикл обработки результата запроса
        while(rs.next())
        {
            // Преобразование значений двух столбцов в строку
            // и добавление текущей строки в список строк
            if ( al.length() > 0 )
                al += "#";
            al += rs.getString(1) + "\t" + rs.getString(2);
        }

        // Возвращение массива строк
        return al.split("#");
    }
    catch (SQLException e2)
    {
        System.out.println(e2.getMessage());
        return null;
    }
}

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

    // Строка запроса к БД
    String sql = "INSERT INTO notepad values( ? , ? )";

    // Выполнение запроса к БД
    try
    {
        PreparedStatement pst =
            conn.prepareStatement(sql);

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

        // Установка второго параметра запроса
        pst.setString(2, str);

        // Возврат количества обработанных строк
        return pst.executeUpdate();
    }
    catch (SQLException e)
    {
        System.out.println(e.getMessage());
        return -1;
    }
}

/**
 * Метод УДАЛЕНИЯ записи по ключу
 * Метод, реализующий DELETE
 * @param key - ключ записи.
 * @return число измененных строк или -1, если - ошибка.
 */
@Override
public int setDelete(int key)

```

```

{
    // Защита от неправомерного соединения
    if(!flag)
        return -1;

    // Строка запроса к БД
    String sql = "DELETE FROM notepad WHERE notekey = ?";

    // Выполнение запроса к БД
    try
    {
        PreparedStatement pst =
            conn.prepareStatement(sql);

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

        // Возврат количества обработанных строк
        return pst.executeUpdate();
    }
    catch (SQLException e)
    {
        System.out.println(e.getMessage());
        return -1;
    }
}

/**
 * Методы, реализующие ОБСЛУЖИВАЮЩИЕ запросы к БД
 * -----
 */

/**
 * Метод ОТКРЫТИЯ соединения с БД exampleDB
 */
@Override
public boolean setOpen()
{
    if ( isConnect() )
        return flag;

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

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

        // Проверяем соединение на значение null
        if ( conn == null )
        {
            flag = false;
            return flag;
        }

        // Проверяем соединение на корректность
        if(conn.isValid(0))
            flag = true;

        return flag;
    }
    catch (ClassNotFoundException e1)
    {
        System.out.println("ClassNotFoundException: "
            + e1.getMessage());

        conn = null;
        flag = false;
        return flag;
    }
    catch (SQLException e2)
    {
        System.out.println("SQLException: "
            + e2.getMessage());
    }
}

```

```

        conn = null;
        flag = false;
        return flag;
    }
}

/**
 * Метод ЗАКРЫТИЯ соединения с БД exampleDB
 */
@Override
public void setClose() {
    try
    {
        if ( conn != null )
            conn.close();

        flag = false;
        conn = null;

    } catch (SQLException e)
    {
        flag = false;
        conn = null;
        e.printStackTrace();
    }
}

/**
 * Метод ПРОВЕРКИ соединения с БД exampleDB
 * @return true, если соединение присутствует.
 */
@Override
public boolean isConnect()
{
    flag = false;
    if ( conn == null )
        return flag;

    // Проверка соединения
    try {
        if ( conn.isValid(0) )
        {
            flag = true;
            return flag;
        }

        conn = null;
        return flag;
    } catch (SQLException e)
    {
        conn = null;
        e.printStackTrace();
        return false;
    }
}
}

```

Примечание — Проектирование серверной части приложения *NotePadImpl* предполагает и его тестирование.

**Полное тестирование** работы приложения *NotePadImpl* возможно только после реализации клиентской части ОРС, поэтому ограничимся частичным тестированием методов, реализующих обслуживающие запросы к БД.

**Частичное тестирование** работы приложения *NotePadImpl* предполагает проверку работоспособности методов *setOpen()*, *setClose()* и *isConnect()*.

**Тестовое приложение**, обеспечивающее частичное тестирование, реализуем в виде класса *TestNotePadImpl*, исходный текст которого представлен на листинге 3.3.

Листинг 3.3 — Исходный текст класса *TestNotePadImpl* из среды Eclipse EE

```
/**
 * Общий package для проекта учебного примера
 */
package asu.rvs;

public class TestNotePadImpl
{
    public static void printState(String text, boolean bb)
    {
        System.out.print("\t" + text + "\t");
        if( bb )
            System.out.println("БД - подключена...");
        else
            System.out.println("БД - не подключена!!!");
    }

    public static void main(String[] args)
    {
        System.out.println("Приложение TestNotePadImpl:\n"
            + "Проверка соединения NotePadImpl с БД exampleDB.\n"
            + "-----");

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

        // Последовательность тестирования
        printState("Начало: ", obj.isConnected());

        printState("setOpen():", obj.setOpen());

        obj.setClose();
        printState("setClose: ", obj.isConnected());

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

После создания приложения *TestNotePadImpl* частичное тестирование серверной части OPC выполняется в три этапа:

- 1) запуск и анализ работы тестового приложения в среде Eclipse EE;
- 2) создание архива проекта в библиотеке */home/upk/lib/notepad.jar*;
- 3) тестирование библиотечного архива серверного приложения в терминальной среде операционной системы.

Результат запуска тестового приложения в среде Eclipse EE приведён на рисунке 3.7.

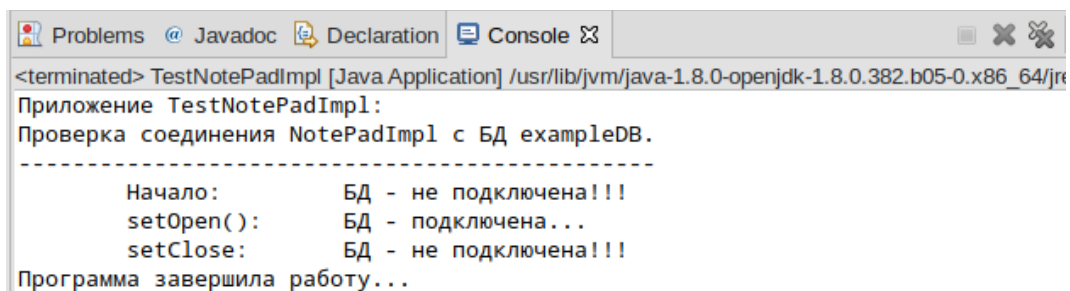
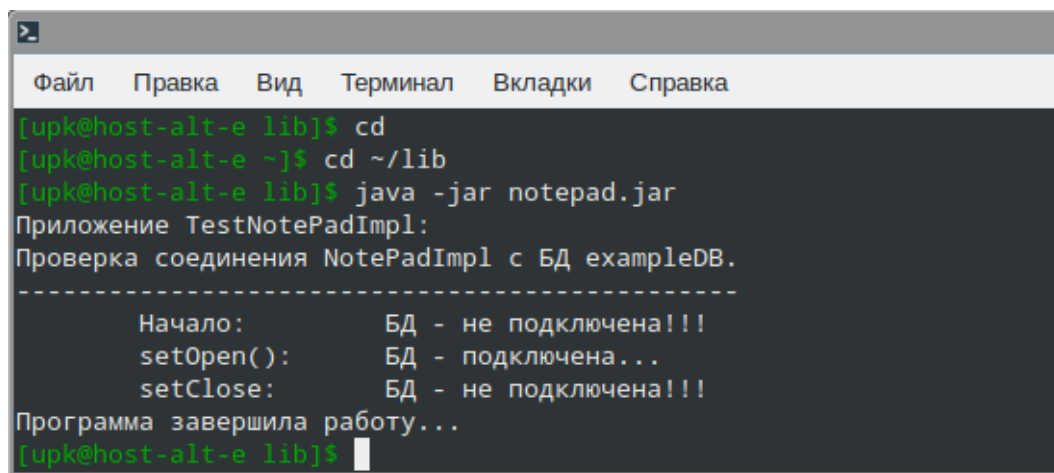


Рисунок 3.7 — Результат запуска тестового приложения в среде Eclipse EE

Без особых комментариев видно, что все обслуживающие функции тестируемого класса *NotePadImpl* работают нормально.

Примечание — Создание библиотеки архива проекта *proj8* может проводиться любыми средствами, которые уже изучены студентами при изучении материала второго раздела.

**Тестирование библиотечного архива** серверного приложения OPC в терминальной среде операционной системы показано на рисунке 3.8 и демонстрирует качественное создание целевого архива.



```
Файл  Правка  Вид  Терминал  Вкладки  Справка
[upk@host-alt-e lib]$ cd
[upk@host-alt-e ~]$ cd ~/lib
[upk@host-alt-e lib]$ java -jar notepad.jar
Приложение TestNotePadImpl:
Проверка соединения NotePadImpl с БД exampleDB.
-----
Начало:          БД - не подключена!!!
setOpen():       БД - подключена...
setClose:        БД - не подключена!!!
Программа завершила работу...
[upk@host-alt-e lib]$
```

Рисунок 3.8 — Результат запуска тестового приложения из библиотеки архива в среде ОС

### 3.2.5 Проект клиентской части приложения Example12

**Учебная тема данного пункта** — реализация прототипа *клиентской части OPC*, заявленной на рисунке 3.6 как «*ПО класса Example12*».

**Основное требование** к реализации класса *Example12* — обеспечение функционала клиентской части OPC в соответствии с ограничениями:

- 1) на функциональность приложения *Example11*;
- 2) на функциональность интерфейса *NotePad*.

Примечание — Дополнительные требования к реализации класса *Example12* соответствуют дополнительным требованиям к реализации класса *NotePadImpl*.

**Особенности реализации** клиентской части приложения *Example12* имеют следующие аспекты проектных решений:

- 1) *диалоговая часть проектного решения*, которая опирается на использование достаточно универсальных методов приложения *Example11*: методов *getKey()*, *getString()* и части функционала метода *main(...)*, обеспечивающих чтение многострочного текста с терминала ОС;
- 2) *серверная часть проектного решения*, которую желательно использовать в виде библиотеки *\$HOME/lib/notepad.jar*.

Примечание — Каждый аспект проектных решений требует своего индивидуального подхода и обоснования его применения.

**Диалоговая часть проектного решения** реализуется в проекте *notedialog* среды разработки Eclipse EE, согласно *спецификации* представленной в таблице 3.6.

Таблица 3.6 — Спецификация проекта *notedialog*

№ п/п	Объект спецификации	Семантика объекта
1	<i>notedialog</i>	Имя проекта среды разработки Eclipse EE.
2	<i>NoteDialog</i>	Имя класса проекта.
3	<i>package asu.rvs</i>	Имя пакета проекта.
4	<i>int getKey()</i>	Метод чтения целого положительного числа.
5	<i>int getKey(String prompt)</i>	Метод чтения целого положительного числа со строкой приглашения ввода данных — <i>prompt</i> .
6	<i>String getString()</i>	Метод чтения строки произвольной длины.
7	<i>String getString(String prompt)</i>	Метод чтения строки произвольной длины со строкой приглашения ввода символов — <i>prompt</i> .
8	<i>String getText(String prompt)</i>	Метод чтения строки произвольной длины со строкой приглашения ввода символов — <i>prompt</i> .
9	<i>\$HOME/lib/notedialog.jar</i>	Целевая библиотека проектного решения.

Примечание — Реализация отдельного проекта *notedialog* — *первое типичное проектное решение* локальной разработки приложения, которое направлено на создание необходимого количества обслуживающих проект библиотек.

Исходный текст класса *NoteDialog* проекта *notedialog* представлен на листинге 3.4.

Листинг 3.4 — Исходный текст класса *NoteDialog* из среды Eclipse EE

```
/**
 * Общий package для проекта учебного примера
 */
package asu.rvs;

import java.io.IOException;

/**
 * Класс, реализующий ОБСЛУЖИВАЮЩИЕ методы диалога
 * для клиентской части ОРС
 * -----
 */
public class NoteDialog {
    /**
     * 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 целого числа со стандартного ввода
 * метод getKey() и выводящий строку подсказки prompt.
 * @return целое число или -1, если - ошибка.
 */
public int getKey(String prompt)
{
    System.out.print(prompt);
    return getKey();
}

/**
 * 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...";
    }
}

/**
 * Метод чтения строки текста со стандартного ввода,
 * предварительно выводя подсказку prompt.
 * @return строка текста.
 */

```

```

public String getString(String prompt)
{
    System.out.print(prompt);
    return getString();
}

/**
 * Метод чтения многостроного текста со стандартного ввода
 * с предварительным выводом подсказки prompt.
 * @return строка текста.
 */
public String getText(String prompt)
{
    // Диалог ввода строк текста
    String s = getString(prompt);
    String text = s;

    // Ввод текста в цикле
    while (s.length() > 0)
    {
        s = getString(prompt);

        // Условие завершения ввода текста
        if(s.length() <= 0)
            break;

        // Объединяем введенные строчки текста
        text += ("\n" + s);
    } // Конец диалога ввода текста

    return text;
}
}

```

На основе проекта *notedialog* создается библиотека *notedialog.jar*, которая помещается в общий каталог библиотек: *~/HOME/lib*. Эта библиотека с успехом может быть использована в различных диалоговых проектах.

Примечание — Второе типичное проектное решение локальной разработки приложения — реализация целевого приложения в виде отдельного проекта, использующего уже созданные ранее библиотеки его программных компонент.

Целевой класс *Example12* клиентского приложения реализуем в среде Eclipse EE, в виде проекта *example12* с использованием библиотек: *notepad.jar* и *notedialog.jar*.

Исходный текст реализации класса *Example12* приведён на листинге 3.5.

*Листинг 3.5 — Исходный текст класса Example12 из среды Eclipse EE*

```

/**
 * Общий package для проекта учебного примера
 */
package asu.rvs;

/**
 * Класс реализации клиентской части учебного примера OPC,
 * использующей интерфейс NotePad
 */
public class Example12
{
    /**
     * Реализация алгоритма работы клиентской части прототипа OPC
     * @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"
        + "-----");

// Создание объекта диалога класса NoteDialog
NoteDialog dial =
    new NoteDialog();
dial.getKey(); // Ожидание для чтения заголовочного текста

// Объект серверной части OPC, доступный через интерфейс NotePad
NotePad obj =
    (NotePad)new NotePadImpl();

// Установка и проверка соединения с БД
if (!obj.setOpen())
{
    System.out.println(
        "Не могу открыть доступ к БД...");
    System.exit(1);
}
System.out.println("БД - открыта...");

/**
 * Служебные переменные
 */
int ns; // Число прочитанных строк
int nb; // Число прочитанных байт
// Строки введенного текста
String      text;
String[]    ls;

/**
 * Основной цикл обработки запросов
 */
while(true)
{
    //Печатаем заголовок для списка записей БД
    System.out.println(
        "-----\n"
        + "Ключ\tТекст\n"
        + "-----");

    // Читаем массив записей из БД в виде массива строк
    ls =
        obj.getList();
    if (ls == null )
    {
        System.out.println(
            "Нет записей в базе 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 на список строк из БД
    System.out.println(
        "-----\n"
        + "Прочитано " + ls.length + " строк\n"
        + "-----\n"
        + "Формируем новый запрос!");

    // Ожидание для продолжения диалога
    nb = dial.getKey("\nВведи ключ или нажми Enter: ");

    // Если ключ не введен, то завершаем работу программы
    if (nb == -1)
        break; // Завершаем работу программы
}

```

```

// Диалог ввода строк текста
text = dial.getText("Строка текста или Enter: ");

// Проверяем общую длину введенного текста
if (text.length() <= 0)
{
    // Запрос на удаления записи по ключу
    ns = obj.setDelete(nb);
    if (ns == -1)
        System.out.println(
            "\nОшибка удаления строки !!!");
    else
        System.out.println(
            "\nУдалено " + ns + " строк...");
}
else
{
    // Запрос на вставку новой записи
    ns = obj.setInsert(nb, text);
    if (ns == -1)
        System.out.println(
            "\nОшибка добавления строки !!!");
    else
        System.out.println(
            "\nДобавлено " + ns + " строк...");
}

dial.getKey("Нажми Enter ...");
}

//Закрываем все объекты и разрываем соединение
obj.setClose();
System.out.println("Программа завершила работу...");
} // Конец main()
} // Конец класса Example12

```

Примечание — Обратите внимание, что исходный текст класса *Example12* не содержит собственных методов, а использует только методы классов, определенных в подключаемых библиотеках: *notepad.jar* и *notedialog.jar*.

После тестирования приложения в среде Eclipse EE, учебная тема данного пункта считается выполненной.

### 3.2.6 Завершение проектирования учебного прототипа OPC

В предыдущих пяти пунктах, на примере конкретной задачи, продемонстрирована технология преобразования сосредоточенного типа приложения (класс *Example11*) в набор компонент для распределённого приложения, включающего класс *Example12* и библиотеки: *notepad.jar* и *notedialog.jar*.

Завершение проектирования учебного примера OPC состоит из двух операций:

- 1) создание архива приложения с именем *example12.jar* на основе проекта *example12* среды разработки Eclipse EE, который следует разместить в директории *\$HOME/lib* ;
- 2) тестирование работы приложения *example12.jar* в командной среде виртуального терминала, как это показано ниже на рисунке 3.9.

```
Terminal
Файл  Правка  Вид  Терминал  Вкладки  Справка
[upk@host-alt-e lib]$ cd
[upk@host-alt-e ~]$ cd lib
[upk@host-alt-e 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 3.9 — Запуск на тестирования приложения example12.jar

**Методологическая ценность** изученного в данном подразделе учебного материала — разделение всего процесса программного проектирования объектной распределённой системы (ОРС) на три последовательных этапа:

- 1) *описание интерфейса* ОРС;
- 2) *выделение классов* серверной части приложения ОРС;
- 3) *выделение классов* клиентской части приложения ОРС и *тестирование работы* всего приложения в варианте сосредоточенного приложения.

**Практическая ценность** освоенного учебного материала — умение создавать заготовки ОРС, которые позволят применять уже известные технологии создания ОРС.

Примечание — Фактически, при наличии качественного прототипа ОРС, применение конкретных технологий создания ОРС сводится к набору формальных операций, которые слабо зависят от прикладного содержания целевого приложения.

Подтверждение заявленному выше высказыванию студент получит при изучении двух широко известных технологий, изложенных в следующих двух подразделах:

- 1) подраздел 3.3 — технология RMI;
- 2) подраздел 3.4 — технология CORBA.

### 3.3 Технология RMI

**Учебная тема** данного подраздела — изучение технологии **RMI** (*Remote Method Invocation*) для целей создания сильносвязанных объектных распределённых систем (ОПС).

Первоначально технология RMI ориентировалась на «прямую реализацию» протоколов *IIOP* и *SSLIOP*, что требовало использование специального компилятора дополнительного ПО («стабов») обеспечивающих коммуникацию между частями ОПС:

- 1) *client stub* — для клиентской части ПО ОПС;
- 2) *server stub, skeleton* — для серверной части ПО ОПС.

**Современная реализация** технологии RMI, основанная на собственном протоколе **JRMP** (*Java Remote Method Protocol*), не требует генерации «стабов» или «скелетонов», а использует специальный пакет **java.rmi**, входящий в стандартный состав ПО виртуальной машины языка Java.

Таким образом, в современной интерпретации, общая организационная сетевая структура технологии RMI может быть представлена рисунком 3.10.

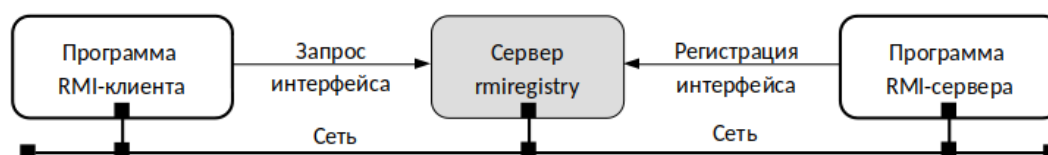


Рисунок 3.10 — Общая организация технологии RMI

**Центральное место** в организационной структуре технологии RMI занимает *сервер службы имён RMI* — **rmiregistry**, который в среде операционной системы Linux можно найти в каталоге **\$JRE\_HOME/bin/**. Такой сервер запускается командой (3.1) и может регистрировать имена объектов множества серверных программ, а также обеспечивать подключение к серверам множества клиентских программ.

**rmiregistry [ port ]** (3.1)

где параметр **port** — не является обязательным (по умолчанию используется значение **1099**).

Примечание — Современная технология RMI может обходиться и без сервера **rmiregistry**, но тогда теряется универсальность общей организационной архитектуры целевой системы.

Согласно общей методики проектирования объектной распределенной системы (ОПС), рассмотренной в предыдущем подразделе, *методика реализации ОПС* также сводится к следующим трём операциям:

- 1) написание *интерфейса* ОПС согласно языку IDL;
- 2) написание *программы сервера*, реализующего функциональность удалённого приложения, в соответствии с заданным интерфейсом, и способного регистрировать свои объекты на сервере **rmiregistry**;
- 3) написание *программы клиента*, реализующего свою функциональность, способного подключиться к серверу **rmiregistry** и использовать функциональность удалённого приложения.

Примечание — Общая *методика реализации ОПС* требует своей конкретизации в соответствии с выбранной *методологией создания ОПС*.

Учебная методика данного подраздела — **проектная реализация** учебного примера прототипа OPC, изложенного в предыдущем подразделе, на основе методологии технологии **RMI**.

Учебное проектирование проведём в соответствии со спецификацией на реализацию OPC представленной в таблице 3.7.

Таблица 3.7 — Спецификация на реализацию проекта учебной OPC

№ п/п	Объект спецификации	Семантика объекта
1	<i>RmiNotePad</i>	Общее название объекта проектирования OPC.
2	<i>java.rmi</i>	Инструментальный пакет языка Java технологии RMI.
3	<i>RmiPad.java</i>	Интерфейс целевой системы.
4	<i>rmipadserver.jar</i>	Приложение удалённого сервера целевой системы.
5	<i>rmipadclient.jar</i>	Приложение клиента целевой системы.
6	<i>notepad.jar</i>	Библиотека серверной части приложения.
7	<i>notedialog.jar</i>	Библиотека клиентской части приложения.
8	<i>NotePad.java</i>	Интерфейс прототипа учебной задачи OPC.
9	<i>Example12.java</i>	Исходный текст прототипа клиентской части OPC.
10	<i>proj9</i>	Проект разработки серверной части приложения в среде Eclipse EE.
11	<i>proj10</i>	Проект разработки клиентской части приложения в среде Eclipse EE.
12	<i>package asu.rvs.rmi</i>	Имя пакета для реализации целевых приложений.

На основании спецификации, представленной в таблице 3.7, создадим контекстную диаграмму процесса реализации учебной OPC, в формате методологии IDEF0, которая представлена на рисунке 3.11.

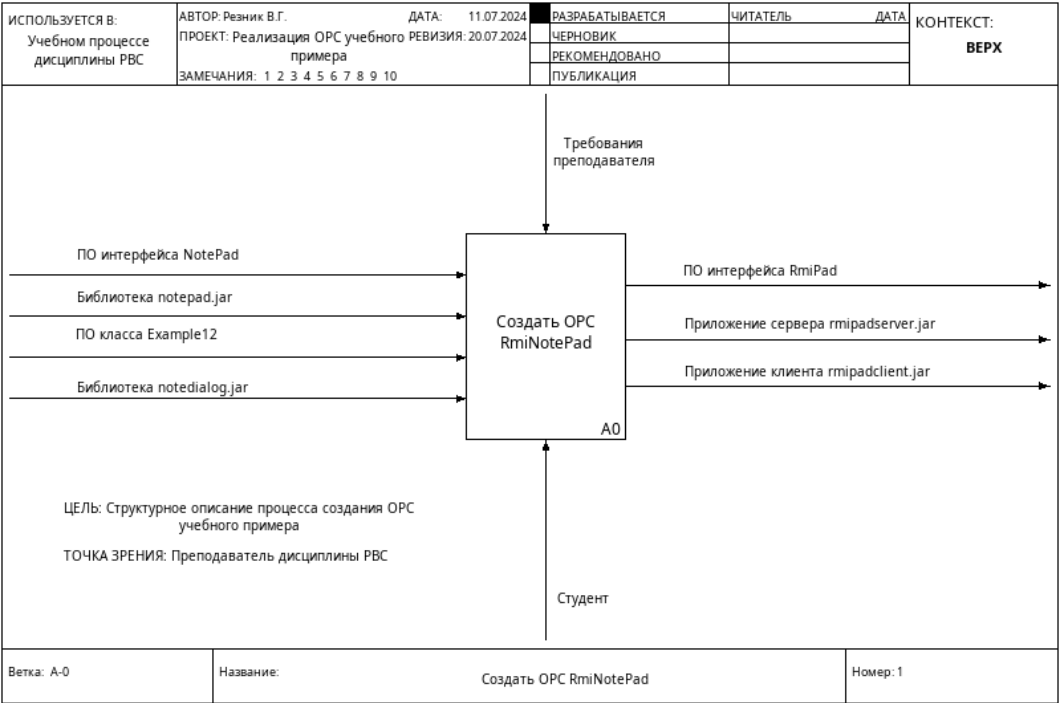


Рисунок 3.11 — Контекстная диаграмма процесса реализации учебной OPC

Примечание — Контекстная диаграмма процесса создания OPC должна дополняться документами *гlossария* и *текстового описания* всех отображённых на рисунке 3.10 графических элементов диаграммы.

Мы не будем рассматривать полное описание глоссария и текстового описания графических элементов диаграмм, поскольку это — занятие для дисциплины «Проектирование информационных систем». Отметим лишь следующие семантические стрелки «Требования преподавателя» и «Студент».

**Требования преподавателя** — стрелка управления, предполагающая подчинение проекта всем требованиям спецификации на реализацию проекта учебной OPC, представленным в таблице 3.7.

**Студент** — стрелка механизма, которая конкретизирует разработчика целевой системы *RmiNotePad* и предполагает наличие стрелки вызова системы разработки Eclipse EE.

Примечание — Стрелка вызова не отображена на рисунке 3.11 в силу ограничений инструментального средства Ramus Educational.

Декомпозиция контекстной диаграммы, детализирующая процесс реализации целевой системы *RmiNotePad*, представлена на рисунке 3.12.

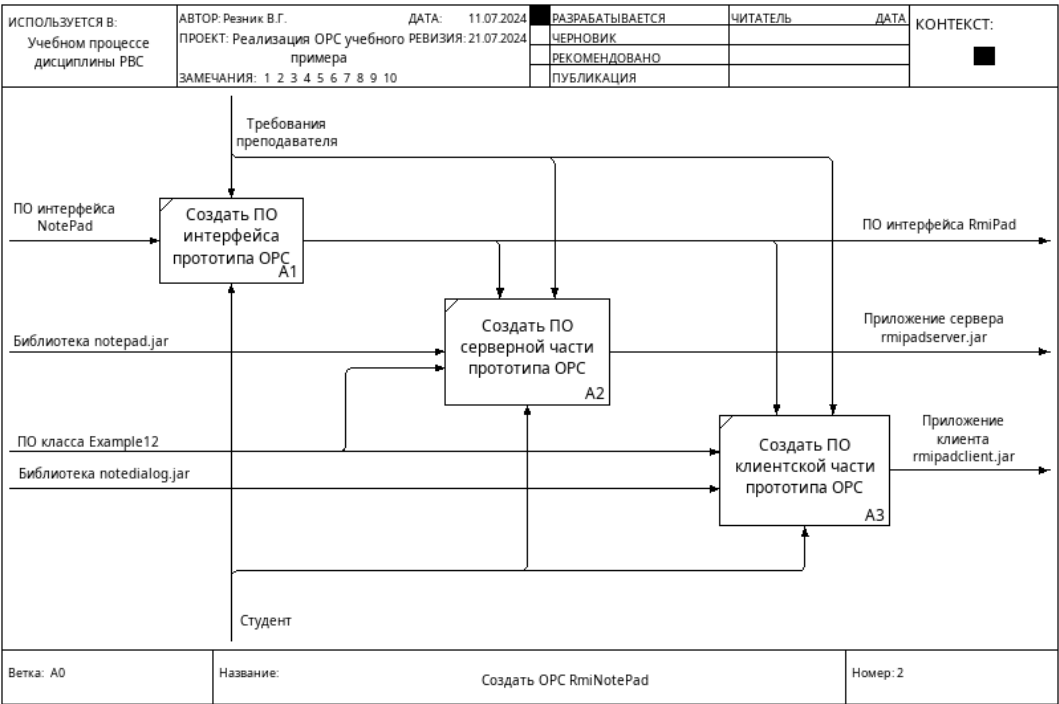


Рисунок 3.12 — Декомпозиция контекстной диаграммы процесса реализации системы RmiNotePad

**Последовательность процесса реализации системы *RmiNotePad*** излагается в следующих четырех пунктах:

- 1) пункт 3.3.1 — Реализация интерфейса приложения для технологии RMI;
- 2) пункт 3.3.2 — Реализация сервера для технологии RMI;
- 3) пункт 3.3.3 — Реализация клиента для технологии RMI;
- 4) пункт 3.3.4 — Завершение реализации проекта для технологии RMI;

### 3.3.1 Реализация интерфейса приложения для технологии RMI

**Учебная тема данного пункта** — проектирование и реализация «ПО интерфейса *RmiPad*», что соответствует функциональности блока *A1* показанного на рисунке 3.12.

При наличии прототипа целевой OPC задача создания целевого интерфейса *RmiPad* становится формальной интеллектуальной процедурой состоящей из двух операций:

- 1) из интерфейса прототипа *NotePad* выбираются только те методы, которые будут необходимы для создания целевого приложения *rmipadclient.jar*;
- 2) исходный текст интерфейса *RmiPad* записывается в соответствии с требованиями синтаксиса пакета *java.rmi* языка Java.

Примечание — реализацию интерфейса *RmiPad*, как и реализацию сервера целевого приложения, проведём в рамках проекта *proj9* среды разработки Eclipse EE.

**Методика выбора методов** целевого интерфейса *RmiPad* опирается на следующую логику рассуждений:

- 1) целевому интерфейсу *нужны* все методы интерфейса *NotePad*, которые описывают прикладные запросы к серверу и представлены ранее в таблице 3.3;
- 2) целевому интерфейсу *нужен* метод *isConnect()* интерфейса *NotePad*, который описывает обслуживающий запрос к серверу для целей проверки его готовности к работе и представлен ранее в таблице 3.4;
- 3) клиентскому приложению *rmipadclient.jar* *не нужны* другие обслуживающие запросы, поскольку операции открытия соединения с базой данных и закрытие такого соединения осуществляются самим целевым сервером.

На основании проведённых рассуждений спецификация целевого интерфейса *RmiPad* может быть представлена таблицей 3.8.

Таблица 3.8 — Спецификация целевого интерфейса *RmiPad*

№ п/п	Определение метода	Назначение метода
1	<i>String[ ] getList()</i>	Метод, получающий содержимое таблицы <i>notepad</i> БД <i>exampleDB</i> в виде списка текстовых строк, причём первое слово каждой строки соответствует значению ключа <i>key</i> таблицы <i>notepad</i> .
2	<i>int setInsert(int key, String str)</i>	Метод, добавляющий текст <i>str</i> к содержимому таблицы <i>notepad</i> БД при условии задания уникального значения ключа <i>key</i> .
3	<i>int setDelete(int key)</i>	Метод, удаляющий по заданному ключу <i>key</i> запись из таблицы <i>notepad</i> БД <i>exampleDB</i> .
4	<i>boolean isConnect()</i>	Метод обслуживающего запроса, проверяющий наличие соединения с БД.

**Синтаксис описания интерфейсов** в пакете *java.rmi* языка Java определяется следующими двумя ограничениями:

- 1) целевой интерфейс *должен* расширять интерфейс *java.rmi.Remote*;
- 2) методы интерфейса *не должны* обрабатывать исключение *java.rmi.RemoteException*.

После выполнения базовых двух операций, целевой интерфейс *RmiPad* может быть представлен исходным текстом на языке Java, как это представлено на листинге 3.6.

Листинг 3.6 — Исходный текст интерфейса *RmiPad* из среды Eclipse EE

```
/**
 * Общий package для технологии RMI
 */
package asu.rvs.rmi;

import java.rmi.Remote; // Импорт стандартного пакета

/**
 * Интерфейс целевого проекта OPC: RmiNotePad
 * @author upk
 */
public interface RmiPad extends Remote
{
    /**
```

```

    * Методы, реализующие прикладные запросы
    */
    // Получить содержимое таблицы notepad
    String[ ] getList();

    // Добавить запись с ключом key и текстом str
    int setInsert(int key, String str);

    // Удалить запись по заданному ключу key
    int setDelete(int key);

    /**
     * Методы, реализующие обслуживающие запросы
     */
    // Проверить наличие соединения с БД exampleDB
    boolean isConnected();
}

```

Таким образом, интерфейс **RmiPad** для технологии RMI — описан и готов к использованию как на стороне сервера, так и на стороне клиента.

### 3.3.2 Реализация сервера для технологии RMI

**Учебная тема данного пункта** — проектирование и реализация «Приложение сервера *rmipadserver.jar*», что соответствует функциональности показанному на рисунке 3.12 блока **A2**, который:

- 1) *управляется* ограничениями интерфейса **RmiPad** и «Требованиями преподавателя»;
- 2) *использует* функциональность серверной части приложения OPC, представленное как «Библиотека *notepad.jar*».

**Типовое решение целевой задачи** сводится к созданию и тестированию двух классов на языке Java:

- 1) **RmiPadImpl** — класс реализации методов, описанных в интерфейсе **RmiPad**, на основе функциональности библиотеки *notepad.jar*;
- 2) **RmiPadServer** — класс реализации самого сервера, создающий необходимое количество объектов класса **RmiPadImpl** и регистрирующий их на сервере *rmiregistry*, с различными именами и типом интерфейса **RmiPad**.

Примечание — Оба класса типового решения целевой задачи должны удовлетворять ограничениям пакета *java.rmi* языка Java.

Рассмотрим последовательные этапы решения целевой задачи.

**Этап 1. Типовая реализация** класса **RmiPadImpl** основана на использовании функциональности класса **NotePadImpl**, в пределах функциональности интерфейса **NotePad**, и должна учитывать следующие ограничения языка Java и пакета *java.rmi*:

- 1) *должен* расширять класс *java.rmi.server.UnicastRemoteObject*;
- 2) *не должен* обрабатывать исключение *java.rmi.RemoteException*.

Примечание — При выполнении заданных выше «жестких» ограничений, возможны разные варианты реализации класса **RmiPadImpl**.

**Реализация** класса **RmiPadImpl** допускает два базовых варианта реализации:

- 1) *монопольное владение* объектами класса **NotePadImpl**, не допускающее использование их другими классами;
- 2) *раздельное использование* объектов класса **NotePadImpl**, допускающее создание для них различных интерфейсов и классов реализации этих интерфейсов.



Примечание — Мы рассмотрим второй вариант, как более универсальный. Он требует от целевого класса **RmiPadImpl** иметь хотя бы один конструктор, содержащий ссылку на уже созданный объект класса **NotePadImpl**.

Дальнейшие пояснения по реализации класса **RmiPadImpl** кратко представлены в исходном коде на языке Java, представленном текстом листинга 3.7.

*Листинг 3.7 — Исходный текст класса RmiPadImpl из среды Eclipse EE*

```
/**
 * Общий package для технологии RMI
 */
package asu.rvs.rmi;

import asu.rvs.NotePadImpl;

/**
 * Класс реализации методов, описанных в интерфейсе RmiPad
 * @author upk
 */
public class RmiPadImpl
    extends java.rmi.server.UnicastRemoteObject
    implements asu.rvs.rmi.RmiPad
{
    /**
     * Версия реализации (наличие - обязательно!)
     */
    private static final long serialVersionUID = 1L;

    /**
     * Сохраняемая ссылка на класс NotePadImpl, передаваемая
     * экземпляру класса RmiPadImpl, через его конструктор.
     */
    private NotePadImpl obj = null;

    /**
     * КОНСТРУКТОР класса RmiPadImpl:
     * 1) реализующий интерфейс RmiPad;
     * 2) передающий ссылку на класс NotePadImpl.
     */
    protected RmiPadImpl(NotePadImpl note)
        throws java.rmi.RemoteException
    {
        // База класса UnicastRemoteObject
        super();

        /**
         * Сохранение ссылки на объект класса NotePadImpl
         * с интерфейсом NotePad
         */
        obj = note;

        /**
         * Проверка состояния внешнего объекта obj
         * класса NotePadImpl, предполагающего
         * наличие соединения с БД
         */
        if( !obj.isConnected() )
            System.out.println("RmiPadImpl: БД не открыта!!!");
        System.out.println("RmiPadImpl: БД открыта...");
    }

    /**
     * Методы, реализующие ПРИКЛАДНЫЕ ЗАПРОСЫ
     * интерфейса RmiPad
     */

    /**
     * Метод, получающий содержимое таблицы notepad:
     *
     * Возвращает null, если нет соединения с БД
     * или нет записей в БД.
     */
}
```

```

    */
    public String[] getList()
        throws java.rmi.RemoteException
    {
        return obj.getList();
    }

    /**
     * Метод, добавляющий запись в БД по новому
     * ключу key и тексту str:
     *
     * Возвращает -1, если нет соединения с БД.
     */
    public int setInsert(int key, String str)
        throws java.rmi.RemoteException
    {
        if ( !obj.isConnected() )
            return -1;

        return obj.setInsert(key, str);
    }

    /**
     * Метод, удаляющий запись из БД по заданному
     * ключу key.
     *
     * Возвращает -1, если нет соединения с БД.
     */
    public int setDelete(int key)
        throws java.rmi.RemoteException
    {
        if ( !obj.isConnected() )
            return -1;

        return obj.setDelete(key);
    }

    /**
     * Методы, реализующие ОБСЛУЖИВАЮЩИЕ ЗАПРОСЫ
     */
    /**
     * Метод, проверяющий наличие соединения с БД exampleDB:
     *
     * Возвращает false, если нет соединения.
     */
    public boolean isConnected()
        throws java.rmi.RemoteException
    {
        if ( !obj.isConnected() )
            return false;

        return true;
    }
}

```

Примечание — Обратите внимание, что *тестирование работоспособности* класса **RmiPadImpl** возможно только после реализации второй части серверного приложения **RmiPadServer**.

**Этап 2. Типовая реализация** класса **RmiPadServer** основана на использовании функциональности классов **NotePadImpl** и **RmiPadImpl**.

**Минимальный функционал** типовой реализации ПО класса **RmiPadServer** включает последовательность следующих четырёх действий:

- 1) *создание объекта* класса **NotePadImpl**;
- 2) *установка соединения* объекта класса **NotePadImpl** с целевой базой данных;
- 3) *создание объекта* класса **RmiPadImpl**, с учётом уже созданного ранее объекта класса **NotePadImpl**;
- 4) *регистрацию объекта* класса **RmiPadImpl** на сервере **rmiregistry** с выбранным строковым именем (*адресом*), которое будет использоваться приложениями клиентов.

Примечание — Регистрация объектов классов на сервере *rmiregistry* осуществляется методами класса *Naming* пакета *java.rmi* языка Java.

Поскольку первые три действия, определяющие минимальный функционал типовой реализации ПО класса *RmiPadServer*, — вполне понятны студентам, рассмотрим статические методы класса *Naming*, синтаксис и семантика которых представлены в таблице 3.9.

Таблица 3.9 — Статические методы класса *Naming*

№ п/п	Синтаксис метода	Семантика метода
1	<i>void bind(String name, Remote obj)</i>	Регистрирует удалённый объект <i>obj</i> под именем <i>name</i> .
2	<i>String[] list(String name)</i>	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.
3	<i>Remote lookup(String name)</i>	Возвращает ссылку на удалённый объект, связанный с указанным именем.
4	<i>void rebind(String name, Remote obj)</i>	Проводит перерегистрацию удалённого объекта <i>obj</i> под именем <i>name</i> .
5	<i>void unbind(String name)</i>	Уничтожает привязку для указанного имени, связанного с удалённым объектом.

Официальная документация [39], в моём свободном переводе и пунктуации с английского языка, так характеризует класс *Naming*:

«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* (3.2)

где:

- 1) **host** — это хост (удалённый или локальный), где находится реестр;
- 2) **port** — номер порта, на который реестр принимает вызовы;
- 3) **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(...)*...».

Примечание — 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 прово-

дить 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 нашей реализации класса *RmiPadServer*, исходный текст которого представлен листингом 3.8, аргумент *name* из таблицы 3.9 используется в формате выражения (3.3).

"//localhost:1099/RMIPAD" (3.3)

*Листинг 3.8 — Исходный текст сервера RmiPadServer из среды Eclipse EE*

```
/**
 * Общий package для технологии RMI
 */
package asu.rvs.rmi;

import java.rmi.Naming;
import asu.rvs.NotePadImpl;

/**
 * Класс сервера RmiPadServer:
 * 1) создаёт объекты классов NotePadImpl и RmiPadImpl;
 * 2) регистрирует объект класса RmiPadImpl на сервере
 *    rmiregistry с именем "RMIPAD".
 */
public class RmiPadServer
{
    public static void main(String[] args)
    {
        /**
         * Создается экземпляр класса NotePadImpl.
         */
        NotePadImpl obj =
            new NotePadImpl();

        /**
         * Устанавливается соединение объекта obj
         * класса NotePadImpl с БД:
         *
         * Если соединение с БД не устанавливается,
         * то завершаем работу сервера.
         */
        if(!obj.setOpen())
        {
            System.out.print(
                "RmiPadServer: Нет соединения с БД!!!");
            System.exit(1);
        }

        /**
         * Сообщаем, что соединение - установлено.
         */
        System.out.println("RmiPadServer: Есть соединение с БД... ");

        /**
         * Создание и регистрация целевого объекта impl
         * класса RmiPadImpl.
         */
        try
        {
            /**
             * Создаётся экземпляр объекта impl
             * класса RmiPadImpl с функциональностью
             * интерфейса RmiPad.
             */
            RmiPadImpl impl =
                new RmiPadImpl(obj);

            /**
```

```

        * Проверяем соединение с БД с помощью объекта impl,
        * но НЕ ЗАВЕРШАЕМ свою работу, в случае обнаружения
        * ошибки соединения с БД.
        */
        //
        if (impl.isConnected())
            System.out.println(
                "RmiPadServer:impl: Есть соединение...");
        else
            System.out.println(
                "RmiPadServer:impl: Нет соединения...");

        /**
        * Регистрация экземпляра удалённого объекта impl
        * на сервере rmiregistry под именем "RMIPAD".
        */
        Naming.rebind("//localhost:1099/RMIPAD", impl);

        System.out.println("RmiPadServer: Стартровал... ");

        /**
        * ЦИКЛИЧЕСКИЙ КОНТРОЛЬ регистрации сервера.
        * Завершение работы сервера: Ctrl-C.
        */
        boolean flag = true;
        String[] ls;

        while ( flag )
        {
            // Задание тайм-аута 5 секунд
            TimeUnit.SECONDS.sleep(5);

            // Проверяем наличие регистрации сервера
            ls =
                Naming.list("//localhost:1099/RMIPAD");

            if ( ls.length <= 0 )
                Naming.rebind("//localhost:1099/RMIPAD", impl);

            System.out.println(
                "RmiPadServer: Работаю...");
        }
    }
    catch (Exception e)
    {
        System.out.println("RmiPadServer: " + e.getMessage());
    }
}
}

```

Примечание — Сервер может только одну регистрацию с заданным именем, поэтому для регистрации **RmiPadServer** используется метод **rebind(...)**, в формате выражения (3.4).

Naming.rebind("//localhost:1099/RMIPAD", impl); (3.4)

Примечание — каждый сервер должен сам следить за актуальностью своих регистраций, поскольку брокер **rmiregistry** не контролирует применение методов **unbind(...)**.

Сервер **RmiPadServer** проверяет свою регистрацию с помощью выражения (3.5).

ls = Naming.list("//localhost:1099/RMIPAD"); (3.5)

После завершения программы сервера **RmiPadServer** следует провести автономное тестирование *сетевой работоспособности* серверной части приложения **RmiNotePad**.

Примечание — Не следует проводить тестирование сетевой работоспособности серверной части приложения OPC в среде разработки Eclipse EE, поскольку отдельный серверный брокер **rmiregistry** технологии RMI запускается и функционирует в программной среде ОС.

Согласно таблице 3.7, определяющей «*Спецификация на реализацию проекта учебной OPC*», для тестирования *сетевой работоспособности* приложения **RmiNotePad** разработчику необходимо выполнить следующую последовательность операций:

- 1) *создать* в проекте **proj9** среды разработки Eclipse EE автономное архивное приложение **rmipadserver.jar**, размещённое в среде ОС и способное запускать на исполнение объект программы сервера — **RmiPadServer**;
- 2) *настроить* переменную среды ОС — **CLASSPATH**, включив в неё полный путь доступа к архиву **rmipadserver.jar**;
- 3) *запустить* в отдельном окне виртуального терминала сервер **rmiregistry**;
- 4) *запустить* в отдельном окне виртуального терминала сервер **rmipadserver.jar**, как это показано ниже на рисунке 3.13.

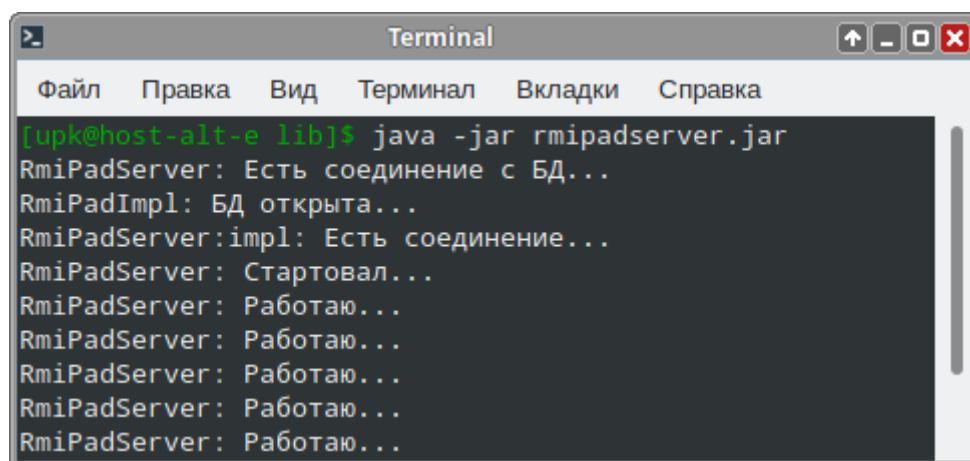


Рисунок 3.13 — Тестирование ПО сервера rmipadserver.jar

Сообщения программы сервера OPC, представленные на рисунке 3.13 запуском на исполнение класса **RmiPadServer**, показывают полную его *сетевую работоспособность* для подключения и обслуживания клиентских программ по сетевому URL, определённом ранее выражением (3.3).

**Особо отметим** следующие правила методологии тестирования серверной части приложения серверной части учебной OPC:

- 1) приложение **rmipadserver.jar** должно быть настроено на запуск класса **RmiPadServer**;
- 2) приложение **rmipadserver.jar** должно содержать все классы приложения **proj9**, внешний архив **notepad.jar**, а также все библиотеки среды исполнения JRE, обеспечивающие работу технологии RMI;
- 3) запуск приложения **rmipadserver.jar** осуществляется из каталога его размещения, с помощью команды, формат которой определяется выражением (3.6).

**java -jar rmipadserver.jar** (3.6)

### 3.3.3 Реализация клиента для технологии RMI

**Учебная тема данного пункта** — проектирование, реализация и тестирование «*Приложение клиента rmipadclient.jar*», что соответствует функциональности показанному на рисунке 3.12 блока **A3**, который:

- 1) управляется ограничениями интерфейса **RmiPad** и «*Требованиями преподавателя*»;
- 2) использует функциональность клиентской части приложения OPC, представленное как «*Библиотека notedialog.jar*», и исходный код приложения «ПО класса **Example12**».

Примечание — Согласно требованию «Таблица 3.7 — Спецификация на реализацию проекта учебной OPC» целевой класс приложения *rmipadclient.jar* реализуется в проекте *proj10* среды разработки Eclipse EE.

**Типовое решение целевой задачи** сводится к созданию и тестированию на языке Java класса *RmiPadClient*, который выполняется на основе исходного текста класса *Example12* в среде системы разработки Eclipse EE.

Это требует от разработчика целевой системы следующей последовательности проектных решений:

- 1) *создание* в среде разработки Eclipse EE отдельного проекта с именем *proj10* и *копирование* в него файла *RmiPad.java*, содержащего исходный текст интерфейса *RmiPad*;
- 2) *создание* в проекте шаблона класса *RmiPadClient* и *копирование* в этот шаблон содержимое метода *main(...)* из исходного текста класса *Example12*;
- 3) *замена* в исходном тексте метода *main(...)* оператора создания класса *NotePad* на оператор создания объекта класса *RmiPad*, посредством использования метода *lookup(...)* класса *Naming*;
- 4) *запуск* в отдельных виртуальных терминалах серверов *rmiregistry* и *RmiPadSever*;
- 5) *тестирование* в среде Eclipse EE работоспособности класса *RmiPadClient*.

Примечание — Все перечисленные выше пункты проектных решений, кроме пункта 3), — вполне доступны для самостоятельного исполнения студентами.

**Проектное решение** пункта 3) требует использование метода *lookup(...)* (см. таблицу 3.9 «Статические методы класса *Naming*»), который в качестве аргумента должен использовать адрес URL, определённый ранее выражением (3.3).

**Рабочий вариант** исходного текста класса *RmiPadClient* представлен листингом 3.9.

*Листинг 3.9 — Исходный текст класса RmiPadClient из среды Eclipse EE*

```
/**
 * Оригинальный package для класса RmiPadClient
 */
package asu.rvs.rmi.rmiclient;

import java.rmi.Naming;
import asu.rvs.NoteDialog;
import asu.rvs.rmi.RmiPad;

public class RmiPadClient
{
    /**
     * Реализация алгоритма работы клиентской части прототипа OPC
     * @param args
     */
    public static void main(String[] args)
    {
        System.out.println(
            "RmiPadClient 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"
            + "-----");

        /**
         * Создание объекта диалога класса NoteDialog
         */
        NoteDialog dial =
            new NoteDialog();
        dial.getKey(); // Ожидание для чтения заголовочного текста
    }
}
```

```

* Служебные переменные приложения
*/
int ns; // Число прочитанных строк
int key; // Значение введённого ключа
String text; // Строка введённого текста
int nn;
String[] ls;

/**
* Область контроля исключений
*/
try {
    /**
    * Получаю и печатаю список всех регистраций
    * на сервере брокера rmiregistry.
    */
    ls =
        Naming.list("//localhost:1099/RMIPAD");

    // Проверяю доступность регистраций сервера.
    ns =
        ls.length;

    System.out.println("RmiPad: Доступна " + ns
        + " регистрация сервера.");

    if( ns <= 0 )
    {
        System.out.println("RmiPad: Отсутствуют доступные "
            + "регистрации сервера.\n"
            + "Программа завершает свою работу...");

        // Аварийное завершение программы.
        System.exit(1);
    }

    for(int i=0; i < ls.length; i++)
        System.out.println("Доступная регистрация клиента: "
            + ls[i]);

    /**
    * Создаю объект класса RmiPad посредством чтения
    * нужной регистрации на сервере rmiregistry.
    */
    RmiPad obj =
        (RmiPad)Naming.lookup("//localhost:1099/RMIPAD");

    /**
    * Проверяю доступность соединения с БД
    */
    if (!obj.isConnected())
    {
        System.out.println(
            "RmiPad: Нет соединения с БД...");
        System.exit(1);
    }
    System.out.println("RmiPad: БД - доступна...");

    /**
    * Основной цикл обработки запросов к БД
    */
    while(true)
    {
        //Печатаем заголовок для списка записей БД
        System.out.println(
            "\nСписок записей таблицы notepad БД exampleDB\n"
            + "-----\n"
            + "Ключ\tТекст\n"
            + "-----");

        // Читаем массив записей из БД в виде массива строк
        ls =
            obj.getList();
        if (ls.equals(null))
        {

```



```

        System.out.println(
            "Нет записей в базе данных...");
        ns = 0;
    }
    else
        ns = ls.length;

    //Выводим (построчно) результат запроса к БД
    nn = 0;

    while(nn < ns){
        System.out.println(ls[nn] + "\n"
            + "-----");
        nn++;
    }

    // Выводим итог запроса на список строк из БД
    System.out.println(
        "-----\n"
        + "Прочитано " + ns + " строк\n"
        + "-----\n"
        + "Формируем новый запрос!");

    // Ожидание для продолжения диалога
    key =
        dial.getKey("\nВведи ключ или нажми Enter: ");

    // Если ключ не введён, то завершаем работу программы
    if (key == -1)
        break; // Завершаем работу программы

    // Диалог ввода строк текста
    text = dial.getText("Строка текста или Enter: ");

    // Проверяем общую длину введённого текста
    if (text.length() <= 0)
    {
        // Запрос на удаления записи по ключу
        nn = obj.setDelete(key);
        if (nn == -1)
            System.out.println(
                "\nОшибка удаления строки !!!");
        else
            System.out.println(
                "\nУдалено " + nn + " строк...");
    }
    else
    {
        // Запрос на вставку новой записи
        nn = obj.setInsert(key, text);
        if (nn == -1)
            System.out.println(
                "\nОшибка добавления строки !!!");
        else
            System.out.println(
                "\nДобавлено " + nn + " строк...");
    }

    dial.getKey("Нажми Enter ...");

    } // Конец цикла обработки запросов к БД

    /**
     * Закрываем объект obj класса RmiPad
     */
    obj = null;
    System.out.println("Программа завершила работу...");
}
catch (Exception e)
{
    // Вывод сообщений об исключениях.
    System.out.println("RmiPadClient Exception:\n" + e);

    System.out.println("Трассировка:");
}

```

```

        e.printStackTrace();
    }

    } // Конец метода main()
} // Конец класса RmiPadClient

```

Примечание — Хотя класс *RmiPadClient* создан на основе исходного программного текста класса *Example12*, он имеет свои особенности реализации, которые следует изучить более внимательно.

Прежде всего программа клиента должна проверить наличие регистрации целевого сервера на сервере брокера *rmiregistry*, что обеспечивается чтением списка целевых регистраций, согласно представленному ранее выражению (3.5).

При наличии регистрации сервера, создаётся целевой объект *obj* класса *RmiPad*, всего лишь одним методом *lookup()* класса *Naming*, как это показано выражением (3.7).

```

/**
 * Создаю объект класса RmiPad посредством чтения
 * нужной регистрации на сервере rmiregistry.
 */
RmiPad obj = (RmiPad)Naming.lookup("//localhost:1099/RMIPAD");

```

(3.7)

После завершения исходного текста класса *RmiPadClient*, его работу можно протестировать прямо в среде Eclipse EE, как это показано на рисунке 3.14.

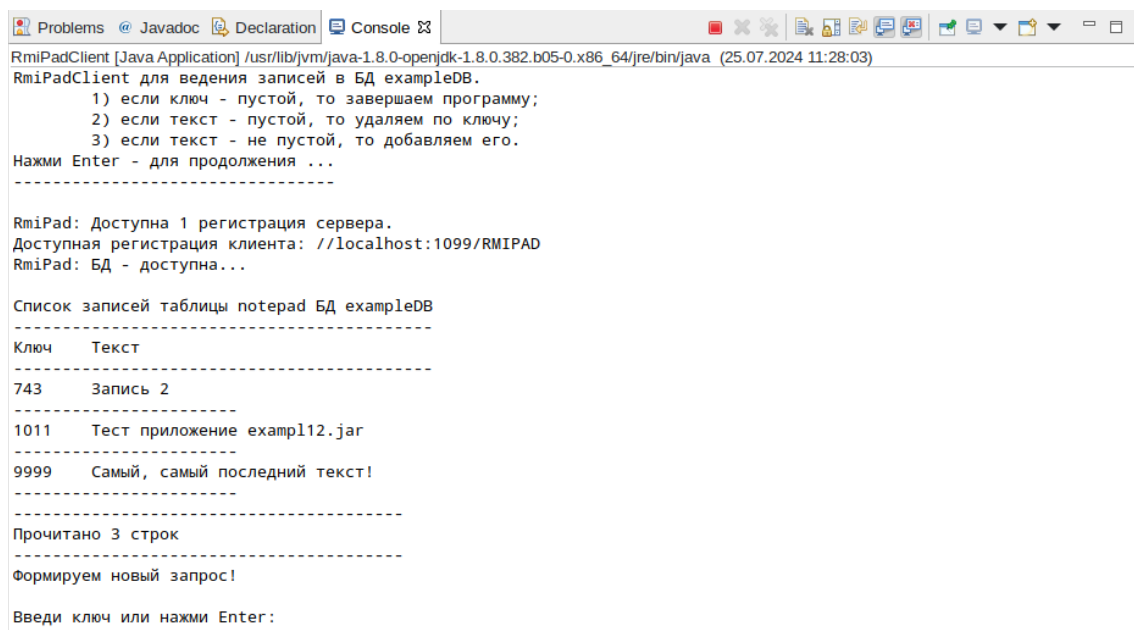


Рисунок 3.14 — Начало диалога приложения RmiPadClient

### 3.3.4 Завершение реализации проекта для технологии RMI

**Учебная тема данного пункта** — полное функциональное тестирование и оформление документации на целевое приложение *RmiNotePad*, что предполагает:

- 1) *создание* в среде Eclipse EE архивного исполняемого файла *rmipadclient.jar*, способного запускать приложение класса *RmiPadClient*;
- 2) *добавление* в переменную среды ОС *CLASSPATH* полного пути к архивному файлу *rmipadclient.jar*;
- 3) *тестирование* всего OPC в среде трёх виртуальных терминалов, предполагая, что

- приложение клиентской части OPC запускается командой заданной выражением (3.8);
- 4) *оформление документации* на проект **RmiNotePad**, согласно требованиям спецификации «Таблица 3.7 — Спецификация на реализацию проекта учебной OPC».

***java -jar rmipadclient.jar*** (3.8)

Примечание — Поскольку оформление документации не является основной темой нашей дисциплины, то ограничимся замечанием, основное внимание должно быть уделено описанию программных продуктов, которые представлены в таблице 3.10.

Таблица 3.10 — Основные документы для проекта OPC RmiNotePad

№ п/п	Объект описания	Характеристика документа
1	<b><i>RmiPad.java</i></b>	Руководство программиста, включающий описание всех входящих в интерфейс методов и их аргументов.
2	<b><i>rmipadserver.har</i></b>	Руководство администратора по установке и эксплуатации сервера приложения, включая инструкцию по установке и эксплуатации приложения <b><i>rmipadclient.jar</i></b> .
3	<b><i>rmipadclient.jar</i></b>	Руководство пользователя по эксплуатации клиентской части приложения, включая демонстрационные примеры его применения.

Примечание — Каждое руководство, заявленное в таблице 3.10, должно включать общие концептуальные вопросы применения технологии RMI, которые в явном виде отражают как *положительные*, так и *отрицательные аспекты* её внедрения и эксплуатации.

**Положительные аспекты** применения технологии RMI — единая языковая среда реализации всех компонент OPC, включая встроенную в единый пакет ***java.rmi*** реализацию протокола ***JRMP***.

Указанные положительные свойства технологии RMI позволяют:

- 1) *выполнять описание* интерфейсов OPC на языке разработки самой системы;
- 2) *не использовать компиляторы* ПО интерфейсов, преобразующих оригинальные описания интерфейсов на языках IDL в программное обеспечение стабов для клиентской и серверной частей OPC;
- 3) *максимально формализовать* реализацию клиентской и серверной частей OPC при наличии качественного прототипа OPC.

**Отрицательные аспекты** применения технологии RMI — специализация на проектирование и реализацию сильносвязанных РВС (распределённых вычислительных систем), все компоненты которых должны программироваться только на языке Java.

Указанные отрицательные свойства технологии RMI ограничивают:

- 1) *использование уже готовых компонент* OPC, реализованных на языках программирования отличных от языка Java;
- 2) *раздельное (параллельное) концептуальное проектирование и отладку* ПО клиентских и серверных частей OPC, в силу прямого вызова программами клиентов методов объектов удалённых приложений;
- 3) *модификацию функционала* уже реализованной OPC, в силу сильной связанности её компонент, приводящих к рассогласованию работы всей системы.

Примечание — Многие отрицательные свойства технологии RMI порождены сильной связанностью компонент целевой OPC.

### 3.4 Технология CORBA

**Учебная тема данного подраздела** — изучение технологии **CORBA** (*Common Object Request Broker Architecture* — *общей архитектуры брокера объектных запросов*) для целей создания сильносвязанных объектных распределённых систем (ОРС).

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 протокола Internet: 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.

Примечание — В данном подразделе рассмотрены только общие концепции архитектуры технологии CORBA, которые стандартизированы консорциумом OMG [17]. Желаящих более подробно изучить эту тему, отправляем к фундаментальному труду Э. Таненбаума [3, глава 9], где он подробно описывает три варианта общих брокерных архитектур: CORBA, DCOM и экспериментальную распределённую систему Globe.

**Технология CORBA** — стандарт консорциума OMG [17], определяющий *брокерную архитектуру* компонент ОРС и *протоколы взаимодействия* компонент этой архитектуры в предположении, что сами компоненты ОРС *могут быть реализованы* на любых различных языках программирования.

**Учебная методика** изучения технологии CORBA соответствует уже использованной в предыдущем подразделе методике изучения технологии RMI и предполагает следующую структуру учебного материала данного подраздела:

- 1) пункт 3.4.1 — *общее описание* брокерной архитектуры технологии CORBA;
- 2) пункт 3.4.2 — *изучение языка IDL CORBA и реализация интерфейса OrbPad* для ОРС учебного примера, который уже был рассмотрен и реализован в предыдущем подразделе с применением технологии RMI;
- 3) пункт 3.4.3 — *пример реализации серверной части* учебного примера средствами технологии CORBA;
- 4) пункт 3.4.4 — *пример реализации клиентской части* учебного примера средствами технологии CORBA.

**Учебное проектирование** целевой ОРС проведём в соответствии со спецификацией представленной в таблице 3.11.

Таблица 3.11 — Спецификация на реализацию проекта учебной ОРС

№ п/п	Объект спецификации	Семантика объекта
1	<b>OrbNotePad</b>	Общее название объекта проектирования ОРС.
2	<b>org.omg.CORBA</b>	Инструментальный пакет языка Java технологии CORBA.
	<b>orbpad.idl</b>	Интерфейс целевой системы на языке IDL CORBA.
3	<b>OrbPad.java</b>	Интерфейс целевой системы на языке Java.
4	<b>orbpadserver.jar</b>	Приложение удалённого сервера целевой системы.
5	<b>orbpadclient.jar</b>	Приложение клиента целевой системы.
6	<b>notepad.jar</b>	Библиотека серверной части приложения.
7	<b>notedoalog.jar</b>	Библиотека клиентской части приложения.
10	<b>orbserver</b>	Проект разработки серверной части приложения в среде Eclipse EE.
11	<b>orbclient</b>	Проект разработки клиентской части приложения в среде Eclipse EE.

Примечание — Для технологии CORBA также можно создать контекстную диаграмму и её декомпозицию процесса реализации учебной OPC, в формате методологии IDEF0, как это было сделано для технологии RMI (см. рисунки 3.11 и 3.12).

Учитывая, что общая структура процесса проектирования OPC нам уже — известна, ограничимся перечнем файлов реализации целевой OPC, которые отражены в таблице 3.12.

Таблица 3.12 — Файлы спецификации, соответствующие реализации проекта OrbNotePad

№ п/п	Файл спецификации	Семантика объекта спецификации
1	<i>orbpad.idl</i>	Интерфейс целевой системы на языке IDL CORBA.
2	<i>orbpadserver.jar</i>	Приложение удалённого сервера целевой системы.
3	<i>orbpadclient.jar</i>	Приложение клиента целевой системы.

### 3.4.1 Брокерная архитектура технологии CORBA

**Учебная тема данного пункта** — изучение общей архитектуры технологии CORBA, использующей специальный сервер: *брокер объектных запросов (ORB)*.

**ORB (Object Resource Broker)** — *брокер ресурсов объектов*, являющийся компонентой глобальной архитектуры технологии CORBA, показанной на рисунке 3.15, и формирующий своеобразный «мост» между приложениями OPC.



Рисунок 3.15 — Глобальная архитектура технологии CORBA [3]

Брокер объектных запросов (ORB), показанный на рисунке 3.15, объединяет следующие специализированные компоненты:

1. **Прикладные объекты** — *части конкретных распределённых приложений* (клиентские приложения и сервера), включённые в структуры конкретных OPC и взаимодействующие через конкретный ORB.
2. **Общие объектные службы** — *базовые сервисы* технологии CORBA, которые доступны всем объектам, подключённым к ORB.
3. **Горизонтальные средства (CORBA horizontal facilities)** — *высокоуровневые службы общего назначения, независимые от прикладной области использующих их программ: средства мобильных агентов, средства печати и средства локализации.*
4. **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.

Примечание — Технология CORBA, в отличие от технологии RMI, ориентирована на концептуально завершённую инструментальную среду разработки и эксплуатации различных OPC.

Для разработки OPC технология CORBA предлагает компонент «Общие объектные службы», включающий инструментальные средства, оформленные как *сервисные службы*.

**Официально** стандартом технологии CORBA зафиксировано шестнадцать сервисных служб или *сервисов*:

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

Примечание — Хотя не все из перечисленных сервисов необходимы для разработки отдельных приложений, хорошо видно, что технология CORBA предоставляет *сервисные инструменты* для реализации самых сложных распределённых систем.

**В плане общей парадигмы «Клиент-сервер»** Эндрю Таненбаум предлагает следующую общую схему модульной организации OPC, вариант которой показан на рисунке 3.16:

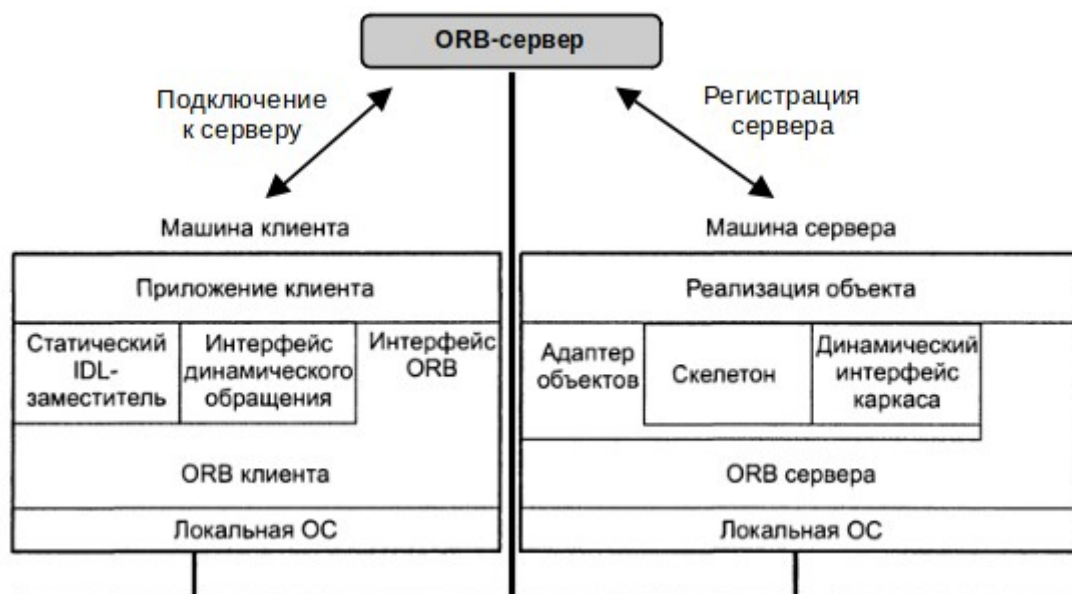


Рисунок 3.16 — Модифицированная общая схема модульной организации OPC CORBA [3]

Примечание — Модификация оригинальной схемы модульной организации OPC CORBA, показанной на рисунке 3.16, состоит в явном указании компонента **ORB-сервер** (сервера *orbd*), выполняющего те же функции, что и сервер *rmiregistry* в технологии RMI.

**Важный прикладной аспект** модульной организации OPC CORBA, показанной на рисунке 3.16, — демонстрация наличия трёх программных компонент, отвечающих за сетевое взаимодействие ПО машины сервера и ПО машины клиента:

- 1) **ORB-сервер** — *универсальный отдельный сервер*, обеспечивающий *регистрацию* серверов приложений и предоставляющий *ссылки* на эти сервера всем клиентским приложениям;
- 2) **ORB клиента** — *Stub клиента*, который генерируется компилятором IDL CORBA на основе интерфейса удалённого объекта, специально для языка программирования ПО приложения клиента;
- 3) **ORB сервера** — *Stub сервера*, который генерируется компилятором IDL CORBA на основе интерфейса удалённого объекта, специально для языка программирования ПО приложения сервера.

Примечание — В пределах изучаемой дисциплины ПО «*Приложение клиента*» и ПО «*Реализация объекта*» разрабатываются на конкретном языке Java, поэтому ORB клиента и сервера тоже должны быть реализованы на языке Java.

Для языка Java поддержка технологии CORBA реализована в пакете **org.omg.CORBA** и требует использования полного инструментального дистрибутива JDK:

- 1) *ORB-сервер* — приложение **orbd**, входящее в состав дистрибутива JRE;
- 2) *компилятор IDL CORBA* — приложение **idlj**, входит только в дистрибутив JDK.

Примечание — Дистрибутивы Java (**OpenJDK**), включая версии 11, 17 и далее, не поддерживают технологию CORBA, поэтому следует использовать JDK версии 8 (1.8)!

Далее предполагается, что все настройки среды ОС используются по умолчанию:

- 1) системные переменные языка Java (**JAVA\_HOME** и **JRE\_HOME**) настроены согласно выражениям (3.9, 3.10);
- 2) ORB-сервер запускается командой выражения (3.11).

**export JAVA\_HOME=/usr/lib/jvm/default** (3.9)

**export JAVA\_JRE=\$JAVA\_HOME/jre** (3.10)

**\$JAVA\_JRE/bin/orbd -ORBInitialPort 1050 -ORBInitialHost localhost** (3.11)

Примечание — По умолчанию сервер **orbd** запускается по адресу **localhost** и порту **1050**.

Дальнейшее изложение учебного материала данного раздела проводится на базе учебного примера, спецификация которого ранее представлена таблицей 3.11.

### 3.4.2 IDL CORBA и генерация распределённого объекта OrbPad

**Учебная тема данного пункта** — основы синтаксиса и семантики IDL CORBA применительно к учебному примеру данного раздела, который реализуется на языке Java.

Примечание — В качестве справочного материала по технологии CORBA рекомендуется пользоваться официальными источниками Интернет [37] и [38].

**Методика изучения данной темы** — решение следующих вопросов:

- 1) изучение соответствия типов объектов языков IDL CORBA и Java, представленных ниже в таблице 3.13;
- 2) реализация интерфейса **OrbPad** на языке IDL CORBA, взяв в качестве прототипа исходный тест интерфейса **RmiPad**, реализованный ранее в предыдущем подразделе;
- 3) генерация стабов для проектов **orbserver** и **orbclient**, на основе интерфейса **OrbPad** на языке IDL CORBA;
- 4) исследование содержимого файлов стабов, сгенерированных компилятором **idlj**, в виде файлов на языке Java.

Примечание — Содержимое таблицы 3.13 будем воспринимать как справочное пособие, обеспечивающее реализацию интерфейса **OrbPad**.

Таблица 3.13 — Соответствие типов объектов языков IDL CORBA и Java

№ п/п	IDL Type	Java Type
1	<i>module</i>	<i>package</i>
2	<i>boolean, wchar</i>	<i>char</i>
3	<i>octet</i>	<i>byte</i>
4	<i>string, wstring</i>	<i>java.lang.String</i>
5	<i>short, unsigned short</i>	<i>short</i>
6	<i>long, unsigned long</i>	<i>int</i>
7	<i>long long, unsigned long long</i>	<i>long</i>
8	<i>float</i>	<i>float</i>
9	<i>double</i>	<i>double</i>
10	<i>fixed</i>	<i>java.math.BigDecimal</i>
11	<i>void</i>	<i>void</i>
12	<i>enum, struct, union</i>	<i>class</i>
13	<i>sequence, array</i>	<i>array</i>
14	<i>interface (non-abstract)</i>	<i>signature interface u operations interface, helper class, holder class</i>
15	<i>interface (abstract)</i>	<i>signature interface, helper class, holder class</i>
16	<i>exception</i>	<i>class</i>
17	<i>any</i>	<i>org.omg.CORBA.Any</i>
18	<i>typedef</i>	<i>helper classes</i>
19	<i>readonly attribute</i>	<i>accessor method</i>
20	<i>readwrite attribute</i>	<i>accessor and modifier methods</i>
21	<i>operation</i>	<i>method</i>

### Изучение соответствия типов объектов языков IDL CORBA и Java.

Для программистов, изучавших языки C++ и Java, многие типы языка IDL CORBA являются интуитивно понятными, поэтому пояснения будут даны только типам объектов, которые выделены в таблице 3.13.

IDL CORBA описывает интерфейсы приложений в виде модулей, используя ключевое слово **module** и его **имя**, идентифицирующее каждый модуль.

Общая синтаксическая конструкция отдельного модуля задаётся выражением (3.12).

module **имя\_модуля** { ... }; (3.12)



В языке Java *имя\_модуля* будет рассматриваться как нижний уровень домена оператора *package*.

Сам интерфейс описывается именованной синтаксической конструкцией, определённой выражением (3.13).

```
interface имя_интерфейса { ... }; (3.13)
```

В языке Java *имя\_интерфейса* — будет присутствовать и в синтаксической конструкции целевого интерфейса.

**Описания методов** IDL CORBA проводятся внутри синтаксической конструкции (3.13) аналогично языку Java, но следует учитывать следующие особенности их написания:

- 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 они порождают 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 оба типа интерпретации синтаксиса и семантики.

Примечание — Проведённый анализ показывает, что технология CORBA позволяет адекватно передавать только простейшие типы данных языка Java, тип *String* и их массивы. Практика показывает, что типы *String* и *String[]* следует заменять типом *byte[]*.

### Реализация интерфейса *OrbPad* на языке IDL CORBA.

Реализация интерфейса *OrbPad* на языке IDL CORBA создаётся следующей последовательностью действий:

- 1) согласно выражения (3.12) создаётся общая синтаксическая конструкция модуля, например, с именем *idlmodule*;
- 2) внутри конструкции модуля создается синтаксическая конструкция интерфейса, удовлетворяющая выражению (3.13), и именем *OrbPad*;
- 3) в синтаксическую конструкцию интерфейса переносятся определения методов интерфейса *RmiPad*, представленного ранее на листинге 3.6;
- 4) заменяются типы методов, написанные на языке Java, на соответствующие типы языка IDL CORBA, согласно требований таблицы 3.13.

Примечание — При замене типа *String[]* метода *getList()* необходимо сначала задать новые типы массивов строк и байт, например, как это показано выражением (3.14).

```
typedef sequence <octet> OctetArray;  
typedef sequence <string> StrArray; (3.14)
```

Завершив указанную выше последовательность действий, мы получим исходный текст модуля *idlmodule*, который показан на листинге 3.10.

Примечание — Не используйте русскоязычные комментарии в тексте модуля. Вы их не прочитаете после обработки компилятором *idlj*.

*Листинг 3.10 — Исходный текст модуля idlmodule для целевого интерфейса OrbPad*

```
/**
 * OrbPad distributed application interface in IDL CORBA language
 */

// Beginning of the module
module idlmodule {

    // Beginning of the module
    interface OrbPad {

        // Defining the type of a octet array
        typedef sequence <octet> OctetArray;

        // Defining the type of a string array
        typedef sequence <string> StrArray;

        /**
         * Methods implementing application queries
         */

        // Get the contents of a notepad table as a list of rows
        // type OctetArray
        OctetArray getListOctet();

        // Get the contents of a notepad table as a list of rows
        // type StrArray
        StrArray getListStr();

        // Add a record with key key and text str
        long setInsert(in long key, in OctetArray str);

        // Delete a record by the given key
        long setDelete(in long key);

        /**
         * Methods implementing service requests
         */
        // Check for connection to DB: exampleDB
        boolean isConnected();

    }; // End of interface description
}
```

Примечание — Чтобы не жалеть о потраченном времени, в интерфейс *OrbPad* включены два описания исходного метода *getList()*: *getListOctet()* и *getListStr()*.

**Генерация стабов для проектов *orbserver* и *orbclient*.**

Записав исходный текст листинга 3.10 в файл *\$HOME/src/orbpad.idl*, проведём генерацию стабов для серверной и клиентской частей целевого приложения *OrbNotePad*.

Примечание — Генерация стабов для языка Java осуществляется утилитой *idlj*, общий формат запуска которой определён выражением (3.15).

*idlj* <Опции> *имя\_файла\_IDL* (3.15)

Аргумент <Опции> утилиты *idlj* допускает комбинацию следующих вариантов использования:

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

Примечание — Технология CORBA допускает, что приложение сервера само может выступать как приложение клиента для доступа к другим серверам OPC. Поэтому лучше создавать все файлы стабов, например, с помощью выражения (3.16), а затем использовать нужные из них для решения конкретных задач.

Проведём генерацию всех стабов с помощью выражения (3.16) в каталог *\$HOME/src/*.

```
idlj -fall -td ./orbstubs -pkgPrefix idlmodule asu.rvs orbpad.idl
```

(3.16)

В результате получим набор файлов и каталогов показанных на рисунке 3.17.

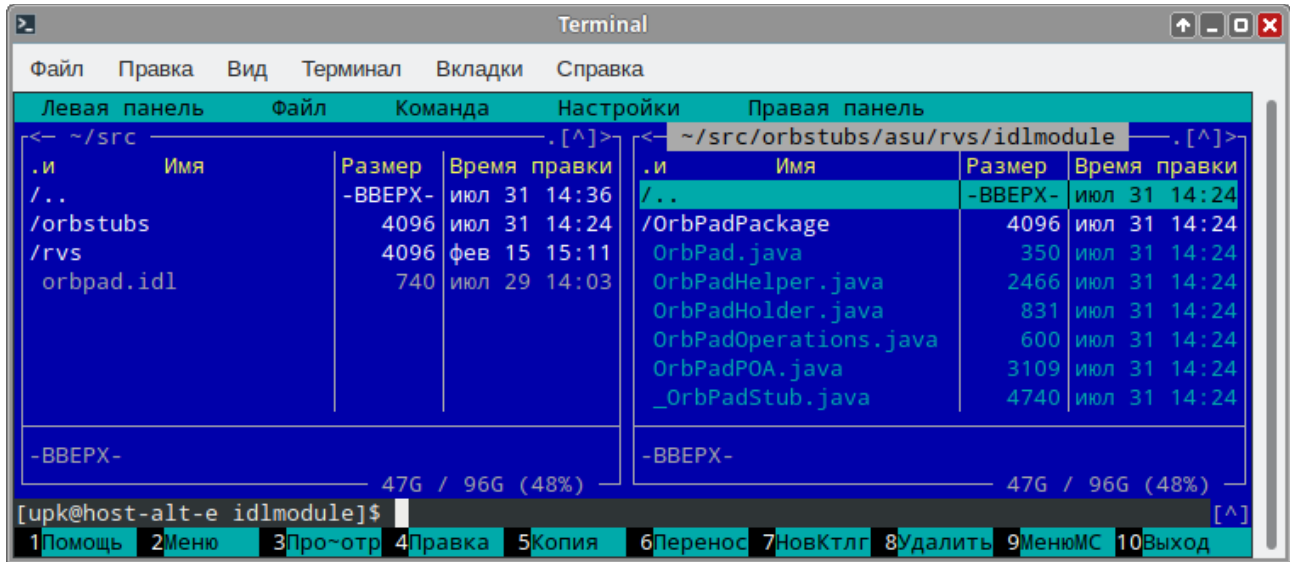


Рисунок 3.17 — Результат генерации стабов для ПО сервера и клиента

Левая часть рисунка 3.17 демонстрирует содержимое каталога, в котором запускался компилятор *idlj*.

Правая часть рисунка 3.17 демонстрирует список основных файлов, входящих в набор ПО серверов и клиентов. Причём каталог **OrbPadPackage** содержит набор файлов для новых объявленных пользователем типов данных: *OctetArrayHelper.java*, *OctetArrayHolder.java*, *StrArrayHelper.java* и *StrArrayHolder.java*.

Примечание — Прежде чем проводить исследование файлов стабов необходимо разобраться с рядом ключевых понятий, которые использует технология CORBA, например, таких как: «Объектный адаптер», *POA*, *Servant* и *Скелетон*.

**Объектный адаптер** (*Адаптер объектов*) — специальный класс (возможно абстрактный) изучаемой технологии CORBA, отвечающий за коммуникацию между другими объектами этой технологии. В реальных приложениях CORBA используется целая древовидная иерархия таких объектных адаптеров.

**POA** (*Portable Object Adapter*) — стандартный объектный адаптер изучаемой технологии CORBA.

**Servant** (*Слуга*) — класс, расширяющий функциональность объектного адаптера POA и предназначенный для реализации методов целевого приложения сервера, описанных его интерфейсами.

**Скелетон** — фактически — синоним понятия *servant*, в плане акцентировании внимания на суперклассе адаптера POA.

Примечание — Приведённые выше ключевые понятия следует правильно ассоциировать с общей схемой модульной организации OPC CORBA, показанной ранее на рисунке 3.16.

### Исследование содержимого файлов стабов.

Прежде всего обратим внимание на общие правила именования файлов:

- 1) *корень имени* соответствует имени целевого интерфейса; в нашем случае это — слово ***OrbPad***;
- 2) *суффикс имени* отражает или тесно связан с одним из перечисленных выше ключевых понятий;
- 3) *если имеются дочерние каталоги*, содержащие дополнительные файлы, то в имени каталога используется суффикс — ***Package***.

Примечание — Ниже в таблице 3.14 представлен список всех основных файлов стабов.

Таблица 3.14 — Список и семантика базовых файлов генерируемых утилитой idlj

№ п/п	Имя файла	Семантика содержимого файла
1	<b><i>OrbPad.java</i></b>	Целевой интерфейс проекта, который расширяет три интерфейса (см. листинг 3.11): 1) OrbPadOperations — см. листинг 3.12; 2) org.omg.CORBA.Object; 3) org.omg.CORBA.portable.IDLEntity.
2	<b><i>OrbPadOperations.java</i></b>	Интерфейс, содержащий целевые методы проекта (см. листинг 3.12).
3	<b><i>OrbPadPOA.java</i></b>	Скелетон ( <i>skeleton</i> ) — абстрактный класс, обеспечивающий базовую функциональность сервера: 1) расширяет класс org.omg.PortableServer.Servant; 2) реализует интерфейс org.omg.CORBA.portable.InvokeHandler.
4	<b><i>_OrbPadStub.java</i></b>	Стаб приложения клиента, обеспечивающий его базовую функциональность.
5	<b><i>OrbPadHelper.java</i></b>	Класс, содержащий статические методы и реализующий вспомогательную функциональность серверов и клиентов: 1) преобразование объектных ссылок — метод narrow(); 2) реализация чтения/записи данных различных типов потоков CORBA.
6	<b><i>OrbPadHolder.java</i></b>	Класс, реализующий интерфейс org.omg.CORBA.portable.Streamable.
7	<b><i>OrbPadPackage</i></b>	Каталог с файлами <b><i>*Helper.java</i></b> и <b><i>*Holder.java</i></b> для каждого типа объекта интерфейса определённого пользователем.

Листинг 3.11 — Исходный текст файла OrbPad.java генерируемый idlj

```
package asu.rvs.idlmodule;

/**
 * asu/rvs/idlmodule/OrbPad.java .
 * Generated by the IDL-to-Java compiler (portable), version "3.2"
 * from orbpad.idl
 * 3 августа 2024 г. 9:32:22 KRAT
 */

// Beginning of the module
public interface OrbPad extends OrbPadOperations,
    org.omg.CORBA.Object, org.omg.CORBA.portable.IDLEntity
{
} // interface OrbPad
```

Листинг 3.12 — Исходный текст файла OrbPadOperations.java генерируемый idlj

```
package asu.rvs.idlmodule;

/**
 * asu/rvs/idlmodule/OrbPadOperations.java .
```

```

* Generated by the IDL-to-Java compiler (portable), version "3.2"
* from orbpad.idl
* 3 августа 2024 г. 9:32:22 KRAT
*/

// Beginning of the module
public interface OrbPadOperations
{

    // type OctetArray
    byte[] getListOctet ();

    // type StrArray
    String[] getListStr ();

    // Add a record with key key and text str
    int setInsert (int key, byte[] str);

    // Delete a record by the given key
    int setDelete (int key);

    // Check for connection to DB: exampleDB
    boolean isConnect ();
} // interface OrbPadOperations

```

Анализ содержимого таблицы 3.14 и листингов 3.11, 3.12 позволяет сделать следующие выводы:

- 1) интерфейс **OrbPadOperations** включает в себя все методы интерфейса **RmiPad**, который выбран в качестве прототипа для данного подраздела;
- 2) интерфейс **OrbPad** расширяет интерфейс **OrbPadOperations**, поэтому может быть использован для реализации целевого приложения.

Примечание — Подробный анализ содержимого всех файлов таблицы 3.14 не входит в программу изучаемой дисциплины. Необходимые комментарии их практического применения будут даны в следующих двух пунктах данного подраздела.

### 3.4.3 Реализация серверной части OPC OrbNotePad

**Учебная тема данного пункта** — проектирование и реализация *серверной части OPC OrbNotePad*, на основе технологии CORBA, функционально соответствующее серверной части OPC *RmiNotePad*.

**Общая задача**, обеспечивающая изучение заявленной темы, предполагает:

- 1) *создание проекта ПО сервера*, обеспечивающего реализацию целевого интерфейса **OrbPadOperations** и функциональность ORB сервера технологии CORBA;
- 2) *реализацию ПО сервера* с использованием ПО приложения **notepad.jar**;
- 3) *создание файла удалённого сервера* целевой системы с именем **orbpadserver.jar**.

Примечание — Простейшая типовая архитектура ПО сервера предполагает реализацию двух классов: *класса скелетона*, реализующего целевой интерфейс, и собственно *класса сервера*, обеспечивающего сетевую коммуникацию объектов скелетона средствами технологии CORBA.

**Типовое решение целевой задачи** — последовательное исполнение следующих этапов учебного процесса, включающего конкретизацию имён целевых объектов:

- 1) *подготовка среды разработки сервера* выполняется в виде проекта **orbserver** инструментальной системы Eclipse EE;
- 2) *реализация серванта* приложения выполняется в виде класса **OrbPadServant**, методы которого используют функциональность библиотеки **notepad.jar**;
- 3) *реализация сервера приложения* выполняется в виде класса **OrbPadServer**;

- 4) *создание архива приложения проекта* выполняется в виде файла ***orbpadserver.jar***;
- 5) *тестирование работы целевого сервера* осуществляется на базе архивного приложения ***orbpadserver.jar***.

Примечание — Оба класса типового решения целевой задачи (***OrbPadServant*** и ***OrbPadServer***) должны удовлетворять ограничениям пакета ***org.omg.CORBA*** языка Java. Все особенности применения этого пакета комментируются в тексте соответствующих этапов реализации указанных классов.

### Этап 1. Подготовка среды разработки сервера, в виде проекта ***orbserver***.

Подготовка среды разработки сервера включает проведение следующей последовательности стандартных операций:

- 1) *создание* в среде Eclipse EE проекта с именем ***orbserver***;
- 2) *копирование* в каталог ***src*** созданного проекта всех файлов и каталогов стаба сервера, размещённых в каталоге ***\$HOME/src/orbstubs***;
- 3) *подключение* к проекту внешней библиотеки ***notepad.jar***.

Примечание — Согласно общей парадигме технологии CORBA приложение сервера может выступать в качестве приложения клиента для обращения к другим серверам сети OPC. Такая возможность заложена в ПО стабов, которые генерируются компилятором ***idlj***;

### Этап 2. Реализация серванта ***OrbPadServant***.

Класс ***OrbPadServant*** выполняет в технологии CORBA роль серванта, что соответствует функциям уже известного и исследованного класса ***RmiPadImpl*** в технологии RMI.

**Основные задачи** серванта, исходный текст которого представлен на листинге 3.13:

- 1) реализация целевых методов интерфейса ***OrbPad (OrbPadOperations)***;
- 2) расширение абстрактного класса ***OrbPadPOA***, выполняющего функции скелетона, в среде архитектуры ПО сервера.

Листинг 3.13 — Исходный текст серванта ***OrbPadServant*** из среды Eclipse EE

```
/**
 * Имя пакета серванта OrbPadServant
 */
package asu.rvs.server;

import asu.rvs.NotePadImpl;
import asu.rvs.idlmodule.OrbPadPOA;

/**
 * Класс серванта OrbPadServant, расширяющий абстрактный
 * абстрактный класс скелетона OrbPadPOA
 */
public class OrbPadServant extends OrbPadPOA
{
    /**
     * Адрес хранения объекта реализации
     * удалённого приложения NotePad
     */
    private NotePadImpl obj;

    /**
     * Метод, сохраняющий адрес объекта
     * удалённого приложения NotePad
     */
    public void setNotePad (NotePadImpl notepad)
    {
        obj = notepad;
    }

    /**
     * Область реализации методов
```

```

* интерфейса OrbPadOperations
*/

public // type OctetArray
// Get the contents of a notepad table as a list of rows
byte[] getListOctet ()
{
    String[] ls =
        obj.getList();

    if( ls == null )
        return "null".getBytes();

    String ss = ls[0];

    for( int i = 1; i < ls.length; i++ )
        ss += ( "###" + ls[i] );

    return ss.getBytes();
}

// type StrArray
// Get the contents of a notepad table as a list of rows
public String[] getListStr ()
{
    String[] ls =
        obj.getList();

    if( ls == null )
        return "null".split("###");

    return ls;
}

// Add a record with key key and text str
public int setInsert (int key, byte[] str)
{
    return obj.setInsert(key,
        new String(str));
}

// Delete a record by the given key
public int setDelete (int key)
{
    return obj.setDelete(key);
}

// Check for connection to DB: exampleDB
public boolean isConnect ()
{
    return obj.isConnect();
}

} // Конец области реализации целевых методов серванта

```

Примечание — Среди двух реализованных методов чтения записей из базы данных рекомендуется использовать метод *getListOctet()*.

### Этап 3. Реализация сервера OrbPadServer.

**Основные задачи** целевого сервера:

- 1) создание полноценных объектов *серванта* и *ORB сервера*;
- 2) регистрацию интерфейса *OrbPad* на внешнем ORB-сервере — *orbd*;
- 3) переход в состояние *ожидания запросов* приложений клиентов.

Примечание — Объект *ORB сервера* создаётся на основе пакета *org.omg.CORBA*, содержащего абстрактный класс *org.omg.CORBA.ORB*.

Основные важные для использования методы абстрактного класса ORB представлены в таблице 3.15.

Таблица 3.15 — Важные методы абстрактного класса org.omg.CORBA.ORB

№ п/п	Синтаксис метода	Семантика метода
1	<b>ORB</b> <i>init(String[] args, Properties props)</i>	Метод, создающий объект ORB.
2	<b>org.omg.CORBA.Object</b> <i>resolve_initial_references(String object_name)</i>	Метод, создающий объектную ссылку на конкретную цель в наборе доступных начальных имён службы: 1) «RootPOA» — менеджер объектного адаптера сервера; 2) «NameService» — общая служба имён ORB-сервера.
3	<b>void</b> <i>run()</i>	Метод, используемый сервером для ожидания запросов на выполнение методов, одновременно блокирующий завершение его работы.
4	<b>void</b> <i>shutdown(boolean wait_for_completion)</i>	Метод, используемый сервером для завершения метода <b>run()</b> ; обычно используется в виде: <b>orb.shutdown(false)</b> .
5	<b>void</b> <i>destroy()</i>	Метод, уничтожающий объект <b>orb</b> , созданный ранее методом <b>init()</b> ;

Алгоритм реализации сервера представлен ниже на листинге 3.14.

Текст листинга сервера — достаточно хорошо комментирован и не требует дополнительных пояснений.

Листинг 3.14 — Исходный текст сервера OrbPadServer из среды Eclipse EE

```
/**
 * Имя пакета сервера OrbPadServer
 */
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.NotePadImpl;
import asu.rvs.idlmodule.OrbPad;
import asu.rvs.idlmodule.OrbPadHelper;

/**
 * Класс сервера OrbPadServer - основной класс
 * серверной части проекта OrbNotePad
 */
public class OrbPadServer {

    public static void main(String[] args)
    {
        /**
         * Создаем объект класса NotePadImpl
         */
        NotePadImpl notepad =
            new NotePadImpl();

        // Устанавливаем и проверяем соединение с БД
        if(!notepad.setOpen())
        {
            System.out.println("OrbPadServer: " +
                "Не могу создать объект класса NotePadImpl ...");
            return; // Завершаем работу программы сервера
        }

        /**

```



```

* Задаем параметры сервера по умолчанию
*/
Properties props =
    System.getProperties();

// Адрес сервера orbd
props.put( "org.omg.CORBA.ORBInitialHost",
    "localhost" );

// Порт сервера orbd
props.put( "org.omg.CORBA.ORBInitialPort",
    "1050" );

try
{
    /**
     * Открываем и инициализируем объект класса ORB
     */
    ORB orb =
        ORB.init(args, props);
    // Теперь ORB сервера создан для взаимодействия с
    // сервером orbd, запущенным по умолчанию

    /**
     * Получаем ссылку на главный объект адаптера сервера
     * и активируем его менеджер POAManager
     */
    POA rootpoa =
        POAHelper.narrow(
            orb.resolve_initial_references("RootPOA"));

    rootpoa.the_POAManager().activate();
    // Теперь менеджер POAManager - активирован

    /**
     * Создаем объект серванта и передаём ему
     * объект типа NotePadImpl
     */
    OrbPadServant servant =
        new OrbPadServant();

    servant.setNotePad(notepad);

    /**
     * Получаем ссылку на объект серванта и создаём
     * объект класса OrbPad, как ссылку на сервант
     */
    org.omg.CORBA.Object ref =
        rootpoa.servant_to_reference(servant);
    OrbPad orbpad =
        OrbPadHelper.narrow(ref);
    // Теперь для сервера объект серванта будет рассматриваться
    // как объект с интерфейсом OrbPad

    /**
     * Получаем объектную ссылку на службу NameService,
     * которую обеспечивает ORB-сервер для регистрации
     * интерфейсов удалённых приложений
     */
    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);
    // Теперь имеем ссылку для регистрации интерфейса OrbPad

    /**
     * Регистрируем объект orbpad, указывающий на сервант,
     * в службе имён ORB-сервера orbd
     */
    NameComponent path[] = // Готовим имя для регистрации
        ncRef.to_name( "ORBPAD" );

```

```

ncRef.rebind(path, orbpad); // Регистрируем сервант
// Теперь сервант зарегистрирован на сервере orbd
// с именем ORBPAD

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

/**
 * Запускаем цикл ожидания входящих запросов от клиентов
 */
orb.run();

/**
 * Нормальное завершение работы сервера
 * после завершения цикла ожидания: orb.run();
 */
notepad.setClose();
orb.destroy();
}
catch(Exception e)
{
    System.err.println("OrbPadServer: " + e);
    notepad.setClose();
}

System.out.println("OrbPadServer завершил работу ...");
}

} // Конец класса OrbPadServer

```

Примечание — При нормальном запуске, сервер должен выдать сообщение «*OrbPadServer готов и ждёт ...*» и ожидать приёма запросов от программ клиентов.

#### Этап 4. Создание файла приложения сервера с именем *orbpadserver.jar*.

Приложение архива целевого сервера создаётся штатными средствами среды разработки Eclipse EE для проекта *orbserver*. Файл архива *orbpadserver.jar* следует поместить в каталог *\$HOME/lib*.

Примечание — При создании архива приложения следует правильно выбрать место и тип библиотеки архива сервера OPC *orbpadserver.jar*, как это показано ниже, на рисунке 3.18.

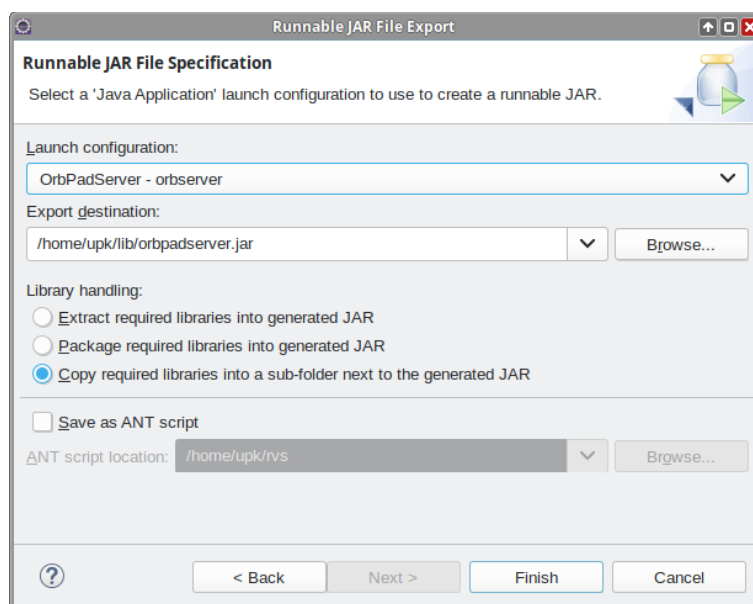


Рисунок 3.18 — Выбор места и типа библиотеки архива сервера OPC — *orbpadserver.jar*

Комментарии по выбору типа целевого архива — следующие:

1. *Extract required libraries into generated JAR* — «**Извлечь необходимые библиотеки в генерированный JAR**», предполагает включение в архив *всех классов* из подключённых к проекту архивных файлов. В результате архив получается большой и возможны противоречия с лицензией подключаемых классов.
2. *Package required libraries into generated JAR* — «**Упаковать необходимые библиотеки в генерированный JAR**», предполагает включение архив *всех файлов архивов*. Дополнительно подключается специальный загрузчик самого Eclipse EE.
3. *Copy required libraries into a sub-folder to the generated JAR* — «**Скопировать необходимые библиотеки в подкаталог генерированного JAR**» упаковывает в целевой архив только классы созданные в проекте пользователем. Все подключённые к проекту, внешние по отношению к JRE, библиотеки копируются в каталог *имя-проекта\_lib*, размещённый в каталоге размещения архива самого проекта.

#### Этап 5. Тестирование работы целевого сервера *orbpadserver.jar*.

Тестирование работы сервера начинается с запуска в отдельном терминале ORB-сервера — *orbd*. С параметрами по умолчанию он запускается согласно выражения (3.17).

```
orbd -ORBInitialHost localhost -ORBInitialPort 1050
```

(3.17)

Примечание — В каталоге выполнения команды (3.17) сервер *orbd* создаёт свой каталог *orb.db* для хранения служебной информации.

Приложение тестируемого сервера запускается в другом терминале. Для этого нужно перейти в каталог *\$HOME/lib* и выполнить команду, заданную выражением (3.18).

```
java -jar orbpadserver.jar
```

(3.18)

Результат запуска сервера показан на рисунке 3.19.

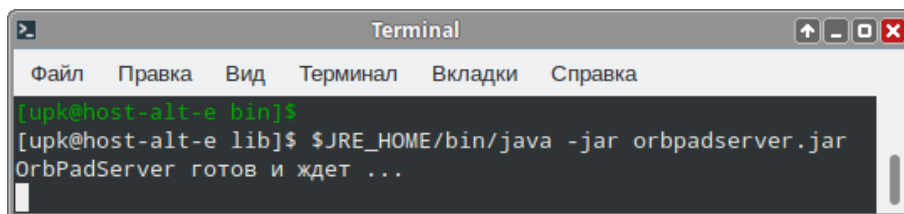


Рисунок 3.19 — Запуск сервера *orbpadserver.jar*

### 3.4.4 Реализация клиентской части OPC OrbNotePad

**Учебная тема данного пункта** — проектирование и реализация *клиентской части OPC OrbNotePad*, на основе технологии CORBA, функционально соответствующее клиентской части OPC *RmiNotePad*.

**Общая задача**, обеспечивающая изучение заявленной темы, предполагает:

- 4) *создание проекта ПО клиента*, обеспечивающего реализацию целевого интерфейса *OrbPad* и функциональность *ORB клиента* технологии CORBA;
- 5) *реализацию ПО клиента* с использованием ПО приложения *notedialog.jar*;
- 6) *создание файла клиента* целевой системы с именем *orbpadclient.jar*.

Примечание — Простейшая типовая архитектура *ПО клиента* предполагает реализацию всего одного класса, реализующего целевой интерфейс, и обеспечивающего сетевую коммуникацию своих объектов средствами стаба технологии CORBA.

**Типовое решение целевой задачи** — последовательное исполнение этапов разработки, которые были определены для реализации сервера, кроме этапа реализации серванта, поэтому сразу приступим к решению задачи.

#### Этап 1. Подготовка среды разработки клиента, в виде проекта *orbclient*.

Подготовка среды разработки клиента CORBA включает проведение следующей последовательности стандартных операций:

- 4) *создание* в среде Eclipse EE проекта с именем *orbclient*;
- 5) *копирование* в каталог *src* созданного проекта всех файлов и каталогов стаба сервера, размещённых в каталоге *\$HOME/src/orbstubs*, кроме файла *OrbPadPOA.java*, потому, что он не нужен приложению клиента;
- 6) *подключение* к проекту внешней библиотеки *notedialog.jar*.

#### Этап 2. Реализация клиента *OrbPadClient*.

**Основные задачи** целевого ПО клиента:

- 1) создание объекта *ORB* клиента и объектной переменной, например, *orbpad* с типом *OrbPad*;
- 2) поиск на внешнем ORB-сервере *orbd* регистрации функционала целевого сервера с именем «*ORBPAD*» и связывание этого функционала с объектом *orbpad*;
- 3) выполнение целевого алгоритма приложения клиента, например, как это уже было реализовано в приложении *RmiPadClient*.

Примечание — Объект *ORB* клиента создаётся на основе пакета *org.omg.CORBA*, содержащего абстрактный класс *org.omg.CORBA.ORB*.

Базовые методы абстрактного класса ORB уже были представлены в таблице 3.15.

Исходный текст приложения клиента представлен на листинге 3.15.

Листинг 3.15 — Исходный текст клиента *OrbPadClient* из среды *Eclipse EE*

```
/**
 * Имя пакета клиента OrbPadClient
 */
package asu.rvs.client;

import java.util.Properties;

import org.omg.CORBA.ORB;
import org.omg.CosNaming.NamingContextExt;
import org.omg.CosNaming.NamingContextExtHelper;

import asu.rvs.NoteDialog;
import asu.rvs.idlmodule.OrbPad;
import asu.rvs.idlmodule.OrbPadHelper;

/**
 * Класс клиента OrbPadServer - основной класс
 * клиентской части проекта OrbNotePad
 */
public class OrbPadClient {

    public static void main(String[] args) {
        System.out.println(
            "OrbPadClient 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"
            + "-----");

        /**
         * Создание объекта диалога класса NoteDialog
         */
    }
}
```

```

NoteDialog dial =
    new NoteDialog();
dial.getKey(); // Ожидание для чтения заголовочного текста

/**
 * Задаем параметры сервера orbd по умолчанию
 */
Properties props =
    System.getProperties();

// Адрес сервера orbd
props.put( "org.omg.CORBA.ORBInitialHost",
    "localhost" );

// Порт сервера orbd
props.put( "org.omg.CORBA.ORBInitialPort",
    "1050" );

/**
 * Область контроля исключений
 */
try
{
    /**
     * Открываем и инициализируем объект класса ORB
     */
    ORB orb =
        ORB.init(args, props);
    // Теперь ORB клиента создан для взаимодействия с
    // сервером orbd, запущенным по умолчанию

    /**
     * Получаем объектную ссылку на службу NameService,
     * которую обеспечивает ORB-сервер для поиска
     * интерфейсов удалённых приложений
     */
    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);
    // Теперь имеем ссылку для поиска интерфейса OrbPad

    /**
     * Создаю объектную ссылку на удалённый объект,
     * читая её с вервера orbd по имени "ORBPAD"
     */
    OrbPad orbpad =
        OrbPadHelper.narrow(ncRef.resolve_str("ORBPAD"));
    // Всё - теперь реализация алгоритма ПО клиента

    /**
     * Выше - область соединения с серверами
     * -----
     * Ниже - область работы целевого алгоритма
     * приложения клиента
     */

    /**
     * Служебные переменные приложения клиента
     */
    int ns; // Число прочитанных строк
    int key; // Значение введённого ключа
    String text; // Строка введённого текста
    int nn;
    String[] ls;
    String ss;

    /**
     * Проверяю доступность соединения с БД
     */
}

```

```

if (!orbpad.isConnected())
{
    System.out.println(
        "OrbPad: Нет соединения с БД...");

    orb.destroy(); // Уничтожаю ORB клиента
    System.exit(1); // Завершаю работу клиента
}
System.out.println("OrbPad: БД - доступна...");

/**
 * Основной цикл обработки запросов к БД
 */
while(true)
{
    //Печатаем заголовок для списка записей БД
    System.out.println(
        "\nСписок записей таблицы notepad БД exampleDB\n"
        + "-----\n"
        + "Ключ\tТекст\n"
        + "-----");

    // Читаем массив записей из БД в виде массива строк
    ss =
        new String(orbpad.getListOchet());

    ls = ss.split("###");

    if (ls[0].equals("null"))
        System.out.println(
            "Нет записей в базе данных...");

    ns = ls.length;

    //Выводим (построчно) результат запроса к БД
    nn = 0;

    while(nn < ns){
        System.out.println(ls[nn] + "\n"
            + "-----");
        nn++;
    }

    // Выводим итог запроса на список строк из БД
    System.out.println(
        "-----\n"
        + "Прочитано " + ns + " строк\n"
        + "-----\n"
        + "Формируем новый запрос!");

    // Ожидание для продолжения диалога
    key =
        dial.getKey("\nВведи ключ или нажми Enter: ");

    // Если ключ не введён, то завершаем работу программы
    if (key == -1)
        break; // Завершаем работу программы

    // Диалог ввода строк текста
    text = dial.getText("Строка текста или Enter: ");

    // Проверяем общую длину введённого текста
    if (text.length() <= 0)
    {
        // Запрос на удаления записи по ключу
        nn = orbpad.setDelete(key);
        if (nn == -1)
            System.out.println(
                "\nОшибка удаления строки !!!");
        else
            System.out.println(
                "\nУдалено " + nn + " строк...");
    }
    else
    {

```

```

        // Запрос на вставку новой записи
        nn = orbpad.setInsert(key, text.getBytes());

        if (nn == -1)
            System.out.println(
                "\nОшибка добавления строки !!!");
        else
            System.out.println(
                "\nДобалено " + nn + " строк...");
    }

    dial.getKey("Нажми Enter ...");

} // Конец цикла обработки запросов к БД

/**
 * Закрываем объект obj класса RmiPad
 */
orbpad = null; // Уничтожаю объект запросов к серверу
orb.destroy(); // Уничтожаю ORB клиента

System.out.println(
    "OrbPadClient: Программа завершила работу...");
}
catch (Exception e)
{
    // Вывод сообщений об исключениях.
    System.out.println("OrbPadClient Exception:\n"
        + e);
    System.out.println("Трассировка:");
    e.printStackTrace();
}

} // Конец метода main()
} // Конец класса OrbPadClient

```

Примечание — Тестирование приложения клиента можно осуществлять прямо в проекте среды разработки Eclipse EE.

### Этап 3. Создание файла приложения клиента с именем **orbpadclient.jar**.

После написания и отладки программы в среде Eclipse EE, её необходимо представить в виде исполняемого архива, например, с именем **orbpadclient.jar** и поместить в общий каталог **\$HOME/lib**.

Технология создания архива клиентского приложения — аналогична технологии создания запускаемого архива для приложения сервера.

Примечание — обратите внимание, что после создания архива клиентского приложения, в каталоге **\$HOME/lib** появился новый каталог **orbpadclient\_lib**, куда был помещен файл архива: **notedialog.jar**.

### Этап 4. Тестирование работы приложения клиента **orbpadclient.jar**.

Для тестирования работы приложения клиента необходимо:

- 1) *сначала*, в отдельных терминалах, запустить сервер **orbd** и приложение сервера, как это показано ранее в выражениях (3.17) и (3.18);
- 2) затем, из каталога **\$HOME/lib**, запустить приложение клиента согласно выражения (3.19).

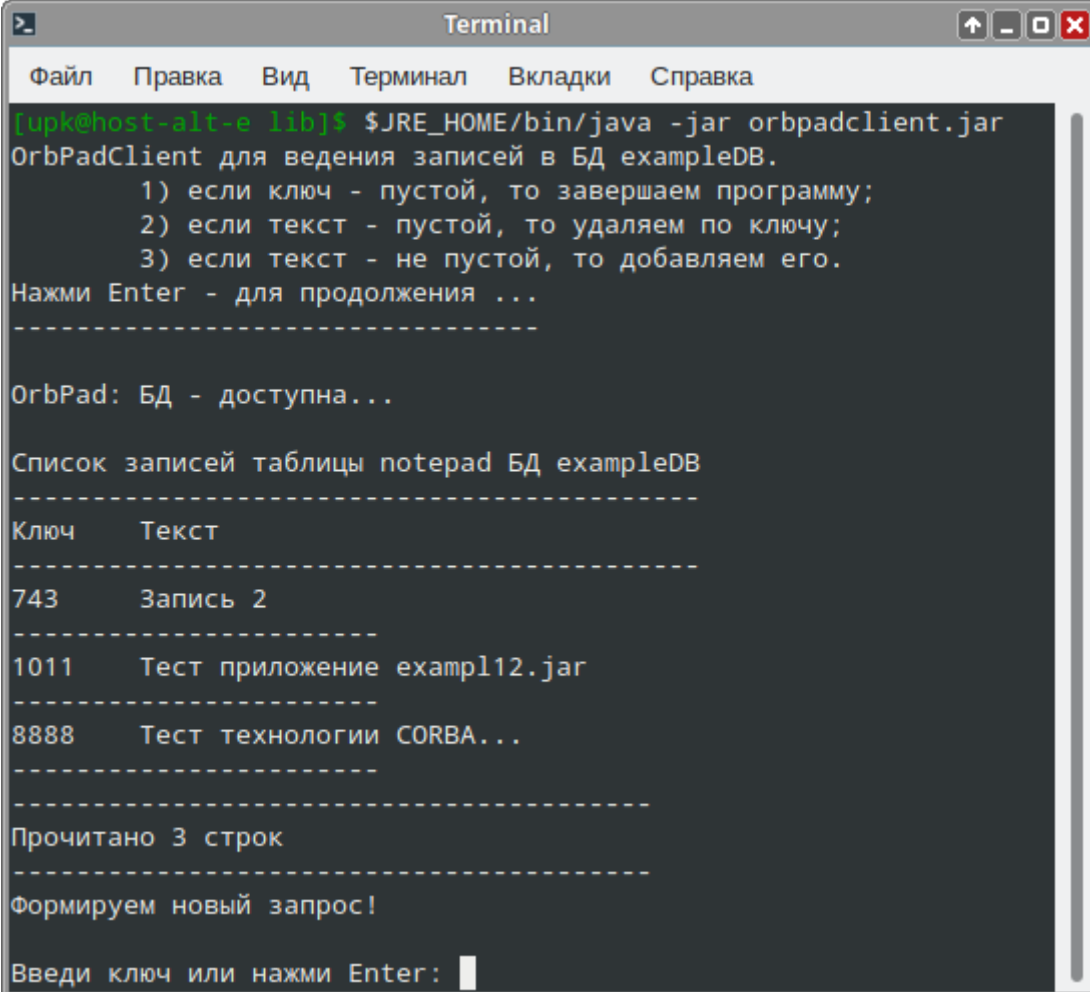
```
java -jar orbpadclient.jar (3.19)
```

Примечание — Если на компьютере установлено несколько версий дистрибутива Java, то необходимо правильно настроить переменную среды ОС — JRE\_HOME и вместо выражения (3.19) использовать выражение (3.20).

```
$JRE_HOME/bin/java -jar orbpadclient.jar
```

(3.20)

Результат запуска приложения клиента показан на рисунке 3.20.



```
Terminal
Файл  Правка  Вид  Терминал  Вкладки  Справка

[upk@host-alt-e lib]$ $JRE_HOME/bin/java -jar orbpadclient.jar
OrbPadClient 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 ...
-----

OrbPad: БД - доступна...

Список записей таблицы notepad БД exampleDB
-----
Ключ      Текст
-----
743       Запись 2
-----
1011      Тест приложение exampl12.jar
-----
8888      Тест технологии CORBA...
-----

Прочитано 3 строк
-----
Формируем новый запрос!

Введи ключ или нажми Enter: 
```

Рисунок 3.20 — Запуск программы приложения клиента

Этим примером завершается учебный материал, как всего подраздела, описывающего технологию CORBA, так и всей темы, описывающей теоретические и практические вопросы технологии объектных распределённых систем (ОРС).



### 3.5 Выводы по результатам всей темы

Подведём краткие итоги по учебному материалу изученной темы «Объектные распределённые системы» (ОРС):

1. **Технологии ОРС** — наиболее консервативный подход проектирования и реализации «Распределённых вычислительных систем» (РВС), предполагающий *сильную связанность взаимодействия* участников сетевой архитектуры «Клиент-сервер».
2. **Основная парадигма ОРС** — использование *проху-серверов в виде серверов-брокеров*, публикующих интерфейсы серверов приложений и предоставляющих программному обеспечению клиентов доступ к методам опубликованных интерфейсов.
3. **Общая архитектура сетевого ПО ОРС** — три сетевых программных компонента: *ПО брокера запросов, ПО стаба сервера, ПО стаба клиента*.
4. **Основное преимущество технологии ОРС** — упрощение процессов проектирования и создания распределённых вычислительных систем (РВС) счёт: стандартизации языков описания интерфейсов (IDL) ОРС, реализации ПО брокеров, стабов сервера и стабов клиента в виде библиотек, обеспечивающих *традиционную технологию реализации* программных систем на языках объектно ориентированного программирования (ООП).
5. **Основной недостаток технологии ОРС** — проектирование и реализация *сильно связанных РВС*, требующих прямой вызов программами клиентов объектов и методов, описанных в интерфейсах ПО серверов. Это сильно усложняет, как процессы реализации РВС, так и последующую их эксплуатацию, модификации и масштабирования работы ПО серверов.
6. **Стандартная архитектура ПО ОРС** — универсальная архитектура технологии *CORBA*, стандартизированная рабочей группой консорциума OMG и поддерживающая: стандартный *IDL CORBA* и три протокола сетевого взаимодействия *GIOP*.
7. **Основное преимущество технологии CORBA** — теоретическая независимость от языков программирования, используемых для реализации ПО серверов и клиентов целевых ОРС.
8. **Основной недостаток технологии CORBA** — теоретическая универсальность реализации *IDL CORBA*, требующая: изучение *преобразований типов* языка IDL в конкретные типы языков программирования, реализации *качественных компиляторов* текстов на языке IDL в исходные тексты стабов для целевых языков программирования, а также общего *понимания структуры и исходных текстов целевых стабов*, как для ПО серверов, так и для ПО клиентов.
9. **Специализированные архитектуры ПО ОРС** — например, *архитектура технологии RMI*, специализированная для языка Java и поддерживающая: IDL языка Java, собственный брокер *rmiregistry*, три реализации абстрактного протокола сетевого взаимодействия *GIOP* и собственный протокол *JRMP*.
10. **Основное преимущество технологии RMI** — использование нативного IDL для языка Java, качественная реализация собственного протокола *JRMP*, не требующая компиляторов для генерации стабов и обеспечивающая реализацию стабов стандартным для языка набором классов.
11. **Основной недостаток технологии CORBA** — требования использования специального брокера запросов *rmiregistry* и реализацию ПО серверов и клиентов только на языке *Java*.

Этим перечнем выводов и заканчивается учебный материал данной темы.

## 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 программы?