Министерство науки и высшего образования Российской Федерации

Томский государственный университет
систем управления и радиоэлектроники

В.Г. Резник

ПРОЕКТИРОВАНИЕ ИНФОРМАЦИОННЫХ СИСТЕМ

Учебное пособие

Тема 4. Объектное проектирование ИС

Томск
2023

УДК 004.8 (004.9)
ББК 65.2-5-05

Резник, Виталий Григорьевич
Проектирование информационных систем. Тема 4. Объектное проектирование ИС. Учебное пособие / В.Г. Резник. – Томск : Томск. гос. ун-т систем упр. и радиоэлектроники, 2023. – 28 с.

Учебное пособие предназначено для обучения дисциплине «Проектирование информационных систем» для студентов направлений подготовки бакалавриата: 09.03.01 — «Информатика и вычислительная техника» и 09.03.03 — «Прикладная информатика».

Одобрено на заседании каф. АСУ протокол №____ от _____

УДК 004.8 (004.9)
ББК 65.2-5-05

Оглавление

4 ОБЪЕКТНОЕ ПРОЕКТИРОВАНИЕ ИС

Напомним, что, закончив стадию концептуального проектирования, написав тактико-техническое задание (ТТЗ) на АС и проведя конкурсный отбор, заинтересованные стороны выполняют третью стадию «Техническое задание», после чего их статус официально изменяется на ***Заказчика*** и ***Исполнителя*** договора на создаваемую систему.

Фактически, после подписания «Технического задания» (ТЗ) на АС (ИС) и «Договора», ***Исполнитель*** начинает четвёртую стадию «Эскизный проект», где он, согласно уже рассмотренной ранее таблице 1.2, выполняет два этапа работ:

а) Разработка предварительных проектных решений по системе и её частям.

б) Разработка документации на АС и её части.

> **Замечание** — Студент, выполняя проектирование ИС по заданию ВКР, не пишет ТТЗ, ТЗ и не заключает «Договор». Тем не менее, он обязан выполнить работы, эквивалентные по содержанию, стадии «Эскизный проект», чтобы наглядно показать: базовые проектные решения, дополняющие архитектурные решения концептуального проектирования, и оценить необходимый объем работ, требующихся для создания программного обеспечения проектируемой ИС.

Безусловно на стадиях «*Эскизного проекта*», «*Технического проекта*» и «*Рабочей документации*» могут использоваться уточняющие методы структурного функционального моделирования, но мы считаем, что нужный уровень декомпозиций уже проведён.

> **Учебная цель** данного раздела — изучение базовых методик, обеспечивающих объектное проектирование ИС.

Основная ошибка, которую допускают большинство студентов — игнорирование результатов или вообще стадии концептуального проектирования.

Обычно, поверхностно осознав условия поставленной задачи, студент начинает объектное проектирование ИС. Для этого он обосновывает выбор языка программирования, инструментальное средство разработки программного обеспечения и СУБД. На основе выбранных инструментов начинает создавать информационную базу данных.

Как правило, выбранные инструменты и проектные действия студента осуществляются спонтанно и на основе тех знаний, которые студент уже получил в процессе обучения.

В результате, при оформлении отчёта по производственной практике или при оформлении выпускной квалификационной работы (ВКР), студент получает некоторый набор программного обеспечения и баз данных, который хоть и связан с исходной постановкой задачи, но не учитывает результаты концептуального проектирования АС.

Окончательно, стремясь удовлетворить требованиям учебного задания, студент «выдумывает» результаты концептуального проектирования и подгоняет обоснование проектных решений под уже полученные практические результаты.

Чтобы устранить описанную выше проблему, необходимо обеспечить прямую логическую связь между функциональными и объектными проектными моделями, опираясь на следующие три аспекта уже известных технологических решений:

1) *Объектные и сервисные распределенные системы* (подраздел 4.1) — как основа выбора архитектуры и инструментальных средств для реализации АС или ИС;
2) *Объектное проектирование процессов на основе языка UML* (подраздел 4.2) — как современные средства проектирования программного обеспечения;
3) *Модель «Сущность-связь»* (подраздел 4.3) — как основа для последующего проектирования баз данных.

4.1 Объектные и сервисные распределённые системы

Структурное функциональное моделирование — теоретический приём построения идеализированной целевой системы, представленной в простейших функциональных форматах, понятных как *Заказчику* так и *Исполнителю* реализуемого проекта.

Действительно, любая система интуитивно воспринимается как объект. Это восприятие не зависит от того, идёт ли речь о полноценной АС или только об отдельной её подсистеме, такой как ИС. Фактически структурное функциональное моделирование выделяет только одно (функциональное) свойство как целой системы, так и её компонент, одновременно обозначая дополнительные материальные и информационные объекты, которые «перерабатываются» обозначенными функциями.

Формально, на стадиях и этапах реализации АС или ИС необходимо «заменить» функции и подфункции целевой системы, полученные на этапе концептуального проектирования, на необходимые объекты программно-технических комплексов.

Фактически заявленный выше тезис и определяет целевую логику проектировщика АС, который на стадии «Эскизный проект» должен обоснованно определить контуры будущих объектов, функциональные свойства которых покрывают функциональные свойства элементов проекта концептуального проектирования.

Проблема реализации целевой логики проектировщика — полисемия самого понятия *объект*.

Поскольку семантика любой системы связана с понятием объект, то источником полисемии объекта является рекурсия декомпозиции системы подсистемы, а также технологические парадигмы использования средств вычислительной техники, кратко описанные в первом разделе данного учебного пособия:

а) *парадигма вычислительных систем* выделяет объектом программу или систему взаимодействующих программ;

б) *парадигма информационного подхода*, интерпретируемая как задача ***системного проектирования предметной области*** (см. рисунок 4.1), декларирует полное разделение семантики «Процессы» и «Объекты», оставляя за последними только две компоненты: «Модели» и «Элементная база»;

в) *парадигма АС* (АСУ), кроме трёхуровневой архитектуры, выделяет девять различных компонент обеспечения, включая информационное обеспечение (ИО АС);

г) *парадигма CALS-технологий* вообще манипулирует масштабными объектами, такими как CAE/CAD/CAM и «*Интегрированные информационные среды*» (ИИС), привязывая их к стадиям жизненного цикла изделия (ЖЦИ);

д) наконец, *парадигма распределённых объектных систем*, показанная на рисунке 4.2, предлагает свой ***шаблон трёхзвенной архитектуры*** «Клиент-сервер», включающий метапонятия «Представление», «Бизнес-логика» и «Данные», которые раскрываются как технологические метаобъекты: «Клиент», «Сервер приложений» и «СУБД».

Примечание — Очевидно, что семантика понятия «Объекты», представленная на рисунке 4.1, ориентирована только на представление и анализ метаобъекта «Данные», который представлен на рисунке 4.2, что не покрывает даже аналогичное представление объектов, принятых в современных языках ООП.

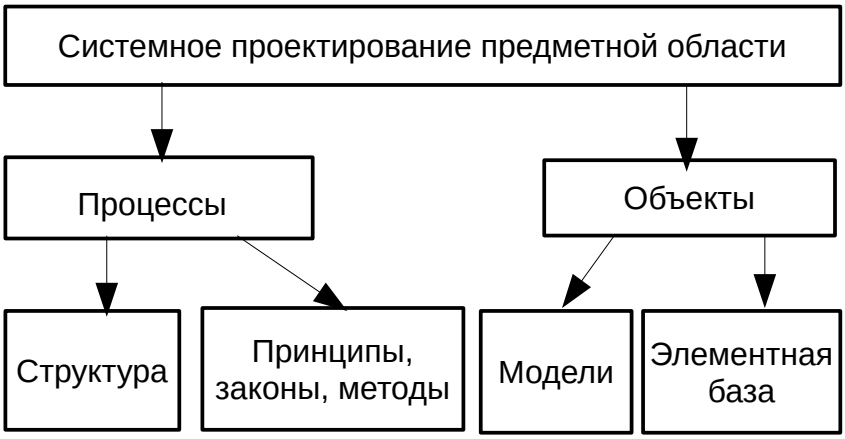

Рисунок 4.1 — Дубль рисунка 1.3, отображающего основные части системного проектирования

Рисунок 4.2 — Дубль рисунка 1.10, отображающего базовую объектную трёхзвенную архитектуру «Клиент-сервер»

Для устранения негативных последствий полисемии понятия «Объект» теория и практика анализа и реализации систем выработала два базовых подхода:

а) структурное функциональное моделирование;

б) шаблоны структурного объектного проектирования.

> **Универсальный шаблон объектного анализа и реализации систем** — «Шаблон проектирования MVC».

Шаблон проектирования MVC — специальный приём проектировщика систем, разделяющий выделенный объект анализа и реализации на три составляющие: *Model-View-Controller*. В технологии проектирования распределённых систем это соответствует метамодели взаимодействия «*Проектировщика*» и «*Проектируемого объекта*» с металшаблоном MVC. Указанное взаимодействие структурно показано на рисунке 4.3, где:

а) **Проектировщик** — актор (субъект), активно воздействующий на исследуемый или проектируемый объект и способный рассматривать себя как «внешняя система» или «пользователь» объекта проектирования;

б) **Проектируемый объект** — объект декомпозиции на составляющие: *Model-View-Controller*;

в) **Model** — сущностная часть анализируемого и реализуемого объекта проектирования, доступная проектировщику только посредством взаимодействия через *Controller*;

г) **View** — преобразователь структуры и типов входной и выходной информации о компоненте *Model* в форму необходимую для проектировщика;

д) **Controller** — компонента объекта проектирования, отвечающая за взаимодействие проектировщика и составляющих *Model* и *View*.

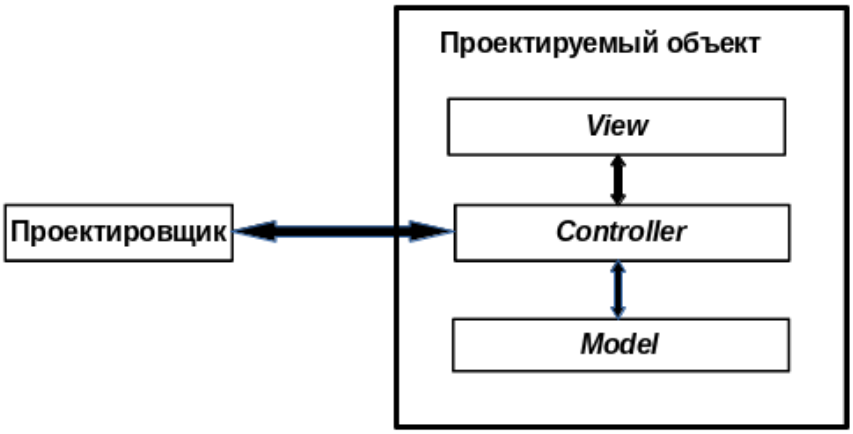

Рисунок 4.3 — Метамодель шаблона проектирования MVC

Представленный шаблон MVC задаёт нам общий теоретический конструктив, который позволяет:

а) *строить конкретизации проектируемого объекта*, уменьшая объём полисемии определения самого понятия объект;

б) *выбирать готовые технологические решения*, которые уже созданы для выделенных в проектируемом объекте компонент: Model, View и Controller.

Примечание — Развитие теории и практики распределённых систем породило множество уже готовых технологических решений, которые студент должен знать и адекватно использовать на стадиях реализации АС и ИС.

В данном подразделе конкретные варианты готовых технологических решений рассмотрим в их исторической последовательности, выделив для обсуждения следующие пять примеров:

а) *Технологии ООП и CORBA* (пункт 4.1.1) описывают зарождение объектного подхода, обеспечившего первоначальную стандартизацию технологий «Объектных распределённых систем»;

б) *Объектная технология RMI и контейнерная технология Java EE* (пункт 4.1.2) раскрывают собственную частную реализацию «Объектных распределённых систем» средствами языка Java, а также создают интегрированные модульные компоненты *сильно связанных* (*Strong coupling*) распределённых систем, получивших название «Контейнерные технологии Java EE»;

в) *Сервис-ориентированная архитектура слабо связанных систем* (пункт 4.1.3) поясняет базовые принципы построения современных масштабных (*Loose coupling*) систем, ориентированных на максимальное разделение этапов проектирования и реализации сетевой парадигмы «Клиент-сервер»;

г) *Контейнерные технологии Web-сервисов* (пункт 4.1.4) описывают частный («облегчённый») вариант реализации компонент «Контейнерных технологий Java EE»;

д) *Web-службы в стиле REST* (пункт 4.1.5) предлагают современный и популярный вариант реализации системсм на базе «Контейнерных технологий Java EE».

Примечание — В целом предполагается, что изложение учебного материала перечисленных выше пунктов опирается на знания, описанные в учебных пособиях [3] и [44].

4.1.1 Технологии ООП и CORBA

Формально шаблон проектирования MVC можно применить к ситуации, когда пользователь ЭВМ, включил компьютер, прошёл аутентификацию, авторизацию и получил доступ к ЭВМ в виде командной строки (терминальный доступ к ЭВМ).

В такой ситуации шаблон проектирования MVC можно конкретизировать распределённым взаимодействием пользователя ОС и ЭВМ, показанным на рисунке 4.4.

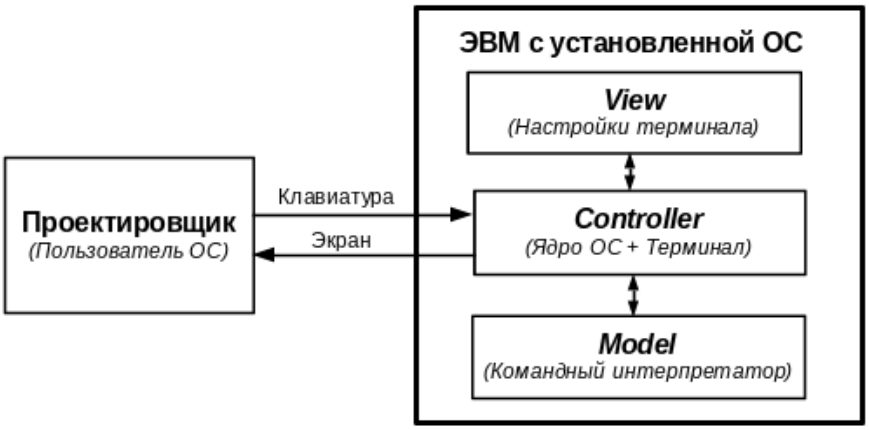

Рисунок 4.4 — Пример конкретизированной модели шаблона проектирования MVC

Представленная конкретизация шаблона MVC интересна только для учебных познавательных целей и легко может быть развита дальше. Например, если в командной строке терминала запустить файловый менеджер *Midnight Commander*, то он заменит собой компоненту *Model* и добавит соответствующий функционал в компоненту *View*.

> **Технология объектно-ориентированного программирования** (ООП) основана на широком использовании уже созданных программных классов, обеспечивающих реализацию нужных объектов для систем АС и ИС.

Если рассматривать целевую систему как *сосредоточенную систему обработки данных* (СОД), реализующую «Автоматизированное рабочее место» (АРМ) средствами технологии ООП, то простейшей конкретизацией шаблона MVC будет архитектура, показанная на рисунке 4.5.

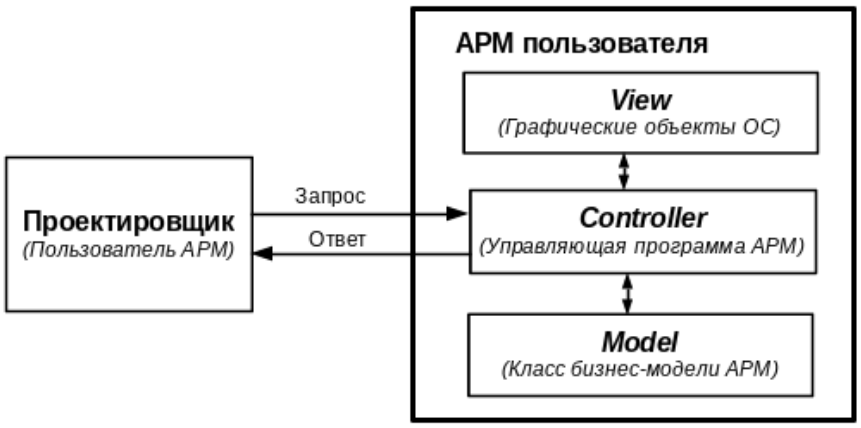

Рисунок 4.5 — Базовая конкретизация шаблона MVC для сосредоточенной объектной системы АРМ

Средства и способы реализации АРМ по шаблону рисунка 4.5 должны быть хорошо известны студенту по результатам изучения дисциплин «Объектно-ориентированное программирование» и «Основы разработки программного обеспечения». Что же касается вопроса «Является ли указанная система АС или ИС?», то ответ зависит от функционала класса *бизнес-модели АРМ*, который представлен компонентой *Model*.

> **В базовой конкретизации шаблона MVC**, компонента ***Model*** может быть реализована отдельно от компоненты ***Controller*** и помещена в конкретную библиотеку для дальнейшего использования.

Выделение компоненты *Model* в отдельный пакет или библиотеку объектов позволяет проводить детальную декомпозицию бизнес-модели АРМ и упрощает реализацию компоненты *Controller*. Это утверждение обосновывается тем, что проектировщику, пишущему управляющую программу АРМ, достаточно знать только *интерфейсы классов*, входящих в компоненту *Model*.

> **Минимальный объём знаний**, необходимый проектировщику для практического использования объектов компоненты *Model*, — ***описание интерфейсов***, входящих в этот компонент классов.

Интерфейс класса — специальное описание класса, в котором определены: весь состав методов, типы аргументов и возвращаемых ими значений, но отсутствует программная реализация этих методов.

> **Современные АС и ИС** являются распределёнными системами.

В случае, когда компонента *Model* должна быть реализована на удалённой ЭВМ или требуется использование баз данных, управляемые удалёнными СУБД, тогда *бизнес-модель* локального АРМ должна включать не только описание интерфейса вызываемой компоненты, но и — соответствующее ***сетевое программное обеспечение*** для связи и доступа к классам реальной бизнес-модели. Очевидно, что в такой ситуации разработка целевой системы значительно усложняется.

В 1989 году был создан консорциум *OMG* (*Object Management Group*), представляющий собой рабочую группу, занимающуюся разработкой и продвижением объектно-ориентированных технологий и стандартов.

Основным достижением консорциума OMG является *проект брокерной архитектуры* взаимодействия программных объектов, получивших сокращённое название *CORBA*.

CORBA (*Common Object Request Broker Architecture*) — общая архитектура брокера объектных запросов, представляющая собой технологический стандарт написания распределённых приложений.

CORBA была создана для целей проектирования, разработки и развёртывания сложных объектно-ориентированных прикладных систем посредством включения в программное обеспечение «механизмов» интеграции различных изолированных систем, написанных на разных языках программирования и работающих в разных узлах сети.

> **Основная идея**, обеспечивающая интеграцию различных изолированных систем, — использование посредника (*Брокера объектных запросов*), позволяющего удалённым объектам посылать *заявки запросов* и получать на них *результаты ответов*, не заботясь о своём местоположении в распределённой среде и способе реализации самих удалённых объектов.

Брокер Объектных Запросов (*Object Request Broker или ORB*) — отдельная, унифицированная, сетевая программная система (*сервер*), поддерживающая конкретные реализации абстрактного протокола *GIOP*.

GIOP (*General Inter-ORB Protocol*) — абстрактный протокол в стандарте CORBA, обеспечивающий интероперабельность (*функциональность взаимодействия*) программного обеспечения брокеров и взаимодействующего с ним программного обеспечения изолированных систем.

Реализация архитектуры протокола GIOP имеет несколько базовых конкретизаций:

а) *Internet InterORB Protocol* (*IIOP или Межброкерный протокол для Интернет*) — протокол опубликованный консорциумом OMG для организации взаимодействия между различными брокерами. IIOP используется GIOP в среде сетей Интернет и обеспечивает отображение сообщений между *протоколом GIOP* и *слоем протокола TCP/IP*;

б) *SSL InterORB Protocol* (*SSLIOP*) — протокол IIOP поверх протокола SSL, поддерживающего шифрование и аутентификацию;

в) *HyperText InterORB Protocol* (*HTIOP*) — протокол IIOP поверх протокола HTTP.

Общая архитектура взаимодействия двух изолированных систем «Клиент» и «Сервер» через «Брокер CORBA» показана на рисунке 4.6.

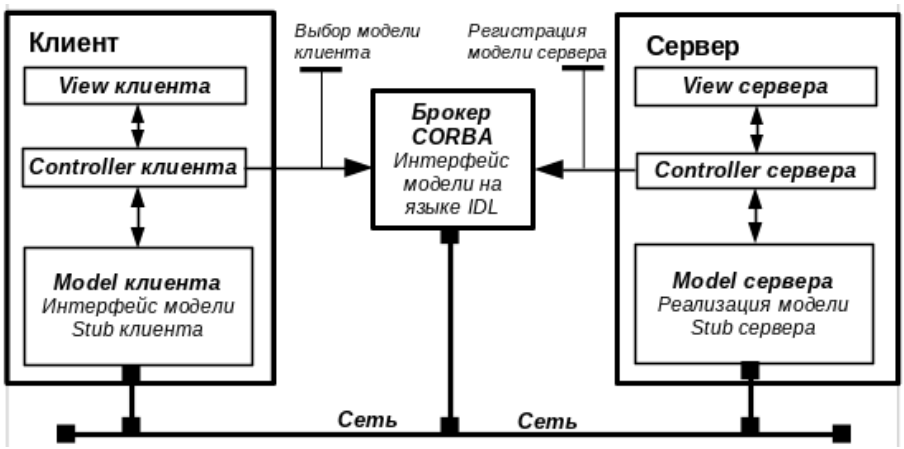

Рисунок 4.6 — Архитектура взаимодействия программ клиента и сервера через брокер CORBA

Центральное место в этой архитектуре взаимодействия программ клиента и сервера занимает сервер брокера, который позволяет программе сервера зарегистрировать на сервере брокера интерфейс реализации компоненты *Model* на языке *IDL CORBA*.

IDL CORBA (*Interface Definition Language CORBA*) — унифицированный язык описания интерфейсов распределённых объектов, разработанный рабочей группой OMG специально для обобщённой архитектуры CORBA.

> **Язык IDL CORBA** не зависит от языков реализации программ клиентов и серверов. В этом отношении он является унифицированным для всех языков программирования изолированных систем.

Применение технологии CORBA, для цели реализации программы удалённого сервера, требует выполнения следующих действий:

а) описать интерфейс компоненты *Model* на языке IDL CORBA;

б) компилировать исходный текст полученного интерфейса специальным компилятором языка IDL CORBA, который сгенерирует исходные тексты необходимого программ-

ного обеспечения на языке реализации программы сервера, включающие: описание интерфейса компоненты *Model* и тексты для «*Stub сервера*»;

в) реализовать программное обеспечение бизнес-модели сервера, согласно полученного в пункте б) исходного текста интерфейса;

г) реализовать остальные компоненты сервера и запустить его на выполнение.

В результате проведённых действий, запущенный сервер должен провести регистрацию своего интерфейса компоненты «*Model сервера*» на сервере «*Брокер CORBA*».

Обратите внимание, что интерфейс реализации «*Model сервера*» регистрируется на сервере брокера, согласно первоначального описания интерфейса на языке IDL CORBA. Это позволяет программам клиентов, реализуемых на других языках программирования чем программа сервера, получить исходные тексты программ, включающих описание собственного интерфейса компоненты *Model* и тексты для «*Stub клиента*».

Примечание — Для реализации программы клиента необходимо выполнить ту же последовательность действий, что и для программы сервера.

После запуска программы клиента, её компонента «*Controller клиента*» обеспечит связь с сервером брокера и произведёт выбор интерфейса, соответствующий интерфейсу компоненты «*Model клиента*». Если нужный интерфейс найден, то далее программа клиента работает с компонентой *Model* так, как будто реализация этой компоненты находится на компьютере клиента.

Технология CORBA предназначена для реализации *сильносвязанных распределенных систем*.

Сильно связанные распределённые системы (*Strong coupling*) — распределённые системы, в которых запрашиваемые программой клиентом методы, объявленные в интерфейсе, напрямую реализуются соответствующими методами программы сервера.

Сильная связанность обеспечивает проектировщика клиентской программы иллюзией локального размещения класса на самой машине клиента.

Считается, что такой подход затрудняет разработку и отладку программного обеспечения распределённых систем.

Завершая краткое описание технологии ООП и CORBA отметим, что изложенный здесь учебный материал студент изучает в дисциплине «Распределённые вычислительные системы» и имеет практические навыки применения этих технологий в проекции языка Java. Для уточнения конкретных неупомянутых здесь деталей, следует воспользоваться учебным пособием [3], в объёме второй и третьей тем (разделов).

4.1.2 Объектная технология RMI и контейнерная технология Java EE

Как было упомянуто в предыдущем пункте, объектно-ориентированный язык программирования (ООП) Java поддерживает реализацию изолированных узлов клиентов и серверов в рамках технологии CORBA. Для этого он:

а) поддерживает реализацию по крайней мере протокола IIOP, поставляемую в рамках системы исполнения (JRE), в виде пакета *org.omg.CORBA*;

б) имеет собственный компилятор *idlj*, обеспечивающий компиляцию исходного текста интерфейса на языке IDL CORBA в набор исходных текстов на языке Java;

в) позволяет создавать программы клиента и сервера, обеспечивающие взаимодействие через брокер CORBA, реализованный в виде программы *orbd*.

На языке Java реализована своя собственная технология объектных распределённых систем, получившая название — *технология RMI*.

RMI (*Remote Method Invocation*) — упрощённый (ориентированный на язык Java) вариант технологии CORBA, появившийся в 1999 году, в первой версии платформы Java EE.

Современная реализация технологии RMI использует собственный протокол *JRMP*, основанный на протоколе IIOP и имеет собственный брокер объектных запросов, реализованный в виде отдельного сервера и представленный в исполнительной среде JRE как программа *rmiregistry*. Общая схема взаимодействия программ клиента и сервера, написанных на языке Java, представлена на рисунке 4.7.

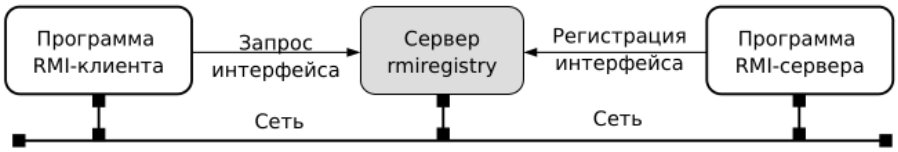

Рисунок 4.7 — Общая схема взаимодействия программ RMI-клиента и RMI-сервера

JRMP (*Java Remote Method Protocol*) — собственный протокол технологии RMI, который не требует описания интерфейсов на языке IDL CORBA, генерации «стабов клиента» (*client stub*) и «стабов сервера» (*server stub, skeleton*).

Для реализации программного обеспечения, использующего технологию RMI, платформа Java EE предоставляет инструментальный пакет *java.rmi*.

Пакет java.rmi — инструментальное средство технологии RMI, содержащее классы и интерфейсы, полностью обеспечивающие объектное проектирование *сильно связанных распределённых систем* на языке Java:

а) *описание интерфейса* компонент *Model* использует синтаксис интерфейса языка Java со следующими требованиями: целевой интерфейс должен расширять базовый интерфейс *java.rmi.Remote*, а все его методы должны быть публичными, но не должны обрабатывать уже заданное исключение *java.rmi.RemoteException*;

б) *класс компоненты Model сервера* должен включать целевой интерфейс и обязательно расширять класс *java.rmi.server.UnicastRemoteObject*;

в) *класс*, реализующий ***программу сервера***, должен использовать статические методы *bind(...)* или *rebind(...)* класса *java.rmi.Naming*, которые обеспечивают регистрацию интерфейса класса компоненты *Model сервера* на сервере брокера *rmiregistry*;

г) *класс*, реализующий ***программу клиента***, должен, при создании объектов класса компоненты Model клиента, использовать имя целевого интерфейса и статический метод *lookup(...)* класса *java.rmi.Naming* для подключения к брокеру сервера *rmiregistry*.

Таким образом, технология RMI, хотя и не является столь универсальной как технология CORBA, но значительно упрощает объём работ по созданию *сильносвязанных* распределённых систем.

Примечание — Изложенный здесь учебный материал по технологии RMI студент подробно изучает в третьей теме дисциплины «Распределённые вычислительные системы» [3] и, кроме теоретических знаний, получает практические навыки применения этой технологии.

Настоящим технологическим прорывом в предметной области создания распределённых систем стала проектная разработка компании Sun Microsystems *контейнерных техноло-*

гий Java EE.

Контейнерные технологии — создание в среде ОС *особых инструментальных программных сред* (*контейнеров*), оснащённых специальными функциональными службами, предназначенными для упрощения разработки, отладки и эксплуатации сложных прикладных программных систем, например, АС или ИС.

Контейнерные технологии Java EE — вариант контейнерных технологий, реализующий компонентный объектно-ориентированный подход на базе виртуальной машины Java (JVM), а также функциональных платформ Java SE (Java Standart Edition) и Java EE (Java Enterprise Edition).

В целом, Java EE поддерживает четыре вида контейнеров (два «клиентских» и два «серверных»), показанных на рисунке 4.8 и предназначенных для обслуживания различных видов компонент:

а) *EJB-контейнер* — создаёт среду для серверных компонент *типа EJB*, предназначенных для реализации бизнес-логики удалённых объектов распределённых приложений; его функционирование обслуживается множеством сервисных служб, обеспечивающих доступ к нему приложений других контейнеров, а также взаимодействие с различными реализациями СУБД;

б) *Веб-контейнер* — среда, предоставляющая базовые серверные службы для управления и исполнения Веб-компонентов: EJB Light, Servlet и JSF;

в) *Контейнер клиентского приложения* — среда, включающая набор Java-классов, библиотек и других файлов, необходимых для реализации клиентских приложений и доступа к компонентам серверных контейнеров по протоколам IIOP, RMI, HTTP и HTTPS;

г) *Контейнеры апплетов* — среды, которые в настоящее время, хоть и ограничено, но реализуются большинством браузеров.

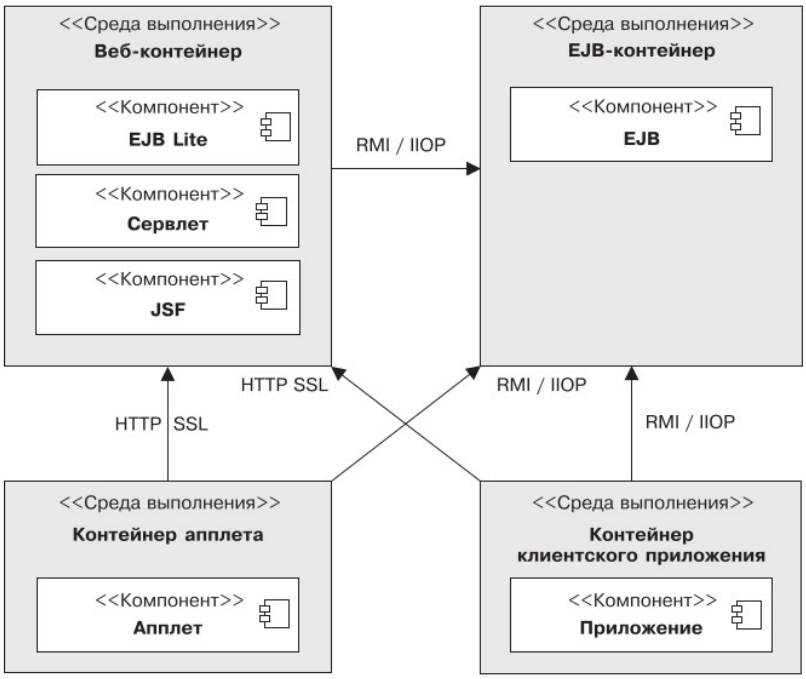

Рисунок 4.8 — Стандартные контейнеры Java EE [45]

Компоненты Java EE — специальные шаблоны классов, предназначенные для разработки распределённых приложений и прозрачно обслуживаемые различными сервисами контейнеров.

Общий набор сервисов контейнеров для Java EE версии 7 показан на рисунке 4.9 и демонстрирует всю мощь служебного программного обеспечения уже доступного проектировщикам ИС и АС для реализации прикладных систем различного назначения.

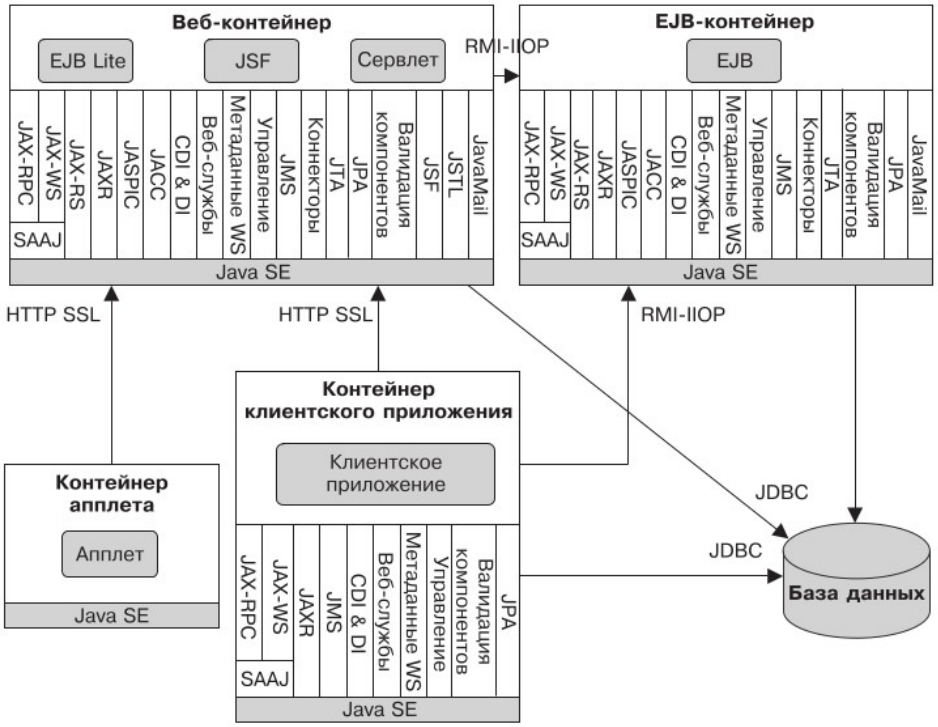

Рисунок 4.9 — Общий набор сервисов контейнеров Java EE версии 7 [45]

Базовым компонентом технологии Java EE безусловно является ***компонента EJB***, обеспечивающая необходимый функционал для работы с базами данных.

EJB (*Enterprise Java Beans*) — корпоративная компонента предназначенная для реализации классов, непосредственно взаимодействующих с СУБД баз данных и использующая в процессе своего функционирования следующие наиболее важные сервисы:

а) **JTA** (*Java Transaction API*) — сервис управления транзакциями при взаимодействии с различными бизнес-приложениями и СУБД;

б) **JPA** (*Java Persistence API*) — сервис, обеспечивающий объектно-реляционные отображения (ORM, Object-Relational Mapping) объектов языка Java в модели реляционных баз данных;

в) **CDI & DI** (*Context and Dependency Injection & Dependency Injection*) — сервис, обеспечивающий для создаваемого компонента *контекст* и *внедрение зависимостей*.

Контейнерные технологии Java EE обеспечивают реализацию не только *сильносвязанных*, но и *слабосвязанных распределённых систем*.

4.1.3 Сервис-ориентированная архитектура слабо связанных систем

Постепенно сильная связанность технологий CORBA и RMI стала создавать проблемы автоматизации процессов принятия решений бизнесом, поскольку сложность реализуемых этими технологиями систем не позволяла проектировать их с нужным качеством и за приемлемое для этого время.

Действительно, *бизнес-парадигма архитектуры предприятия*, показанная на рисунке 4.10 [46], требовала создания уже готовых решений, которыми могли бы воспользоваться бизнес-руководители предприятия, не дожидаясь, пока ИТ-специалисты реализуют нужные классы объектов, протестируют их и создадут приемлемое программное обеспечение.

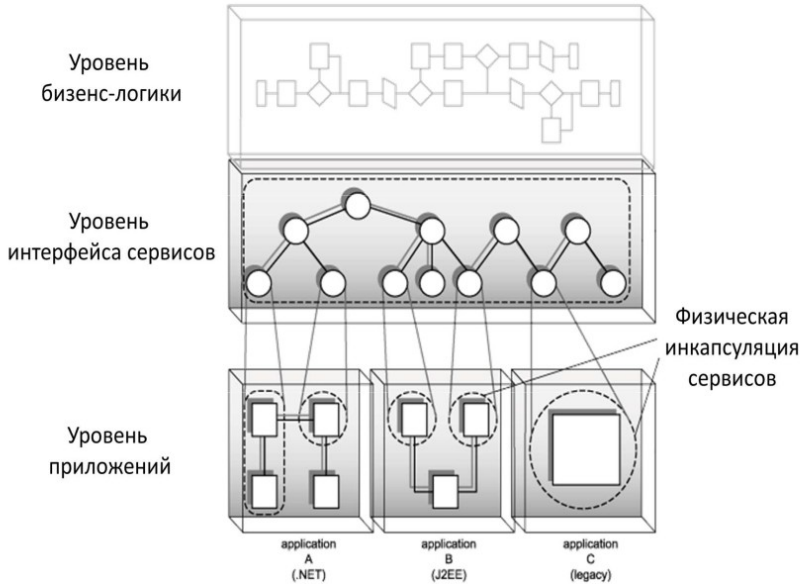

Рисунок 4.10 — Бизнес-парадигма архитектуры предприятия [46]

Бизнес-парадигма архитектуры предприятия — трёхуровневая иерархическая архитектура реализации автоматизированных систем (АС), выделяющая уровни: *бизнес-логики*, *интерфейса сервисов* и *уровень приложений*.

Уровень бизнес-логики — верхний уровень иерархии бизнес-парадигмы АС предприятия, соответствующий уровню принятия решений бизнес-руководителями предприятия, которые генерируют бизнес-процессы и становятся *потребителями сервисов* (***Service Consumer***).

Уровень интерфейса сервисов (*Service Interface*) — средний уровень иерархии бизнес-парадигмы АС предприятия, предназначенный для взаимодействия бизнес-руководителей и ИТ-специалистов предприятия.

На этом уровне обеспечиваются свойства видимости сервисов (***Visibility***), перспектива возможного использования сервисов (***Interaction***) и фактического получения результата услуги сервиса (***Real world effect***). На этом уровне ИТ-специалисты также получают статус провайдеров услуг (***Service Provider***).

Уровень приложений — нижний уровень иерархии бизнес-парадигмы АС предприятия, соответствующий уровню ИТ-специалистов, которые обеспечивают реализацию сервисов.

Этот уровень считается «невидимым» для бизнес-руководителей предприятия, поэтому он может реализовываться различными способами, например:

а) представлять слабосвязанные или сильносвязанные системы;

б) быть распределёнными или сосредоточенными системами;

в) принадлежать самому предприятию или находиться в «облаке» (*cloud computing*) на аутсорсинге (*outsourcing*).

> **Бизнес-парадигма архитектуры предприятия** реализуется теоретическими положениями и практическими решениями *сервис-ориентированных систем*.

Сервис-ориентированные системы — распределённые вычислительные системы, реализованные по теоретическим принципам *сервисно-ориентированной архитектуры* построения прикладных систем.

Сервис-ориентированная архитектура (*Service Oriented Architecture*, *SOA*) — это парадигма для организации и использования распределённых возможностей вычислительных систем, которые могут находиться под контролем различных доменов собственности и предоставлять единые средства для предложения, обнаружения, взаимодействия и использования возможностей для получения желаемых результатов, соответствующих измеримым предварительным условиям и ожиданиям функционирования прикладных систем.

Концептуальная идея сервис-ориентированной архитектуры, представленная автором публикации [47], показана на рисунке 4.11.

Рисунок 4.11 — Концептуальная идея сервис-ориентированной архитектуры [47]

Парадигма SOA предложила разработчикам систем новый уровень абстракции по сравнению с абстракцией классов, предлагающей только уровень объектов. Согласно рисунку 4.11, новый уровень абстракции разделяется на следующие направления конкретизации:

а) **BPM** (*Business process management*, *управление бизнес-процессами*) — концепция процессного управления организацией, рассматривающая бизнес-процессы как ресурсы;

б) **EAI** (*Enterprise Application Integration*) — общее название сервиса интеграции прикладных систем предприятия;

в) **AOP** (*Aspect-Oriented Programming*) — аспектно-ориентированное программирование; парадигма программирования, основанная на идее разделения функциональности программы на модули;

г) **Web-сервисы** (*Web-services*) — web-службы со стандартизированными интерфейсами, адресуемые уникальными URI/URL-адресами.

Дополнительно, в концептуальном плане проектирования и реализации систем, SOA выделяет следующие понятийные составляющие:

а) **Сервисный компонент** (или сервисы), которые описываются программными компонентами и обеспечивают прозрачную сетевую адресацию.

б) **Интерфейс сервиса**, который обеспечивает описание возможностей и качества предоставляемых сервисом услуг. В таком описании определяется формат сообщений для обмена информацией, а также входные и выходные параметры методов, поддерживаемых сервисным компонентом.

в) **Соединитель сервисов** — это транспорт, обеспечивающий обмен информацией между отдельными сервисными компонентами.

г) **Механизм обнаружения сервисов**, который предназначен для поиска сервисных компонентов, обеспечивающих требуемую функциональность сервиса.

> **Сервис-ориентированная архитектура систем** предлагает и реализует *слабую связанность* своих компонент.

Действительно в модели взаимодействия «Клиент-сервер», парадигма SOA предлагает клиенту «*Интерфейс сервиса*», вместо «*Интерфейса классов*», предлагаемых технологиями объектного подхода: CORBA и RMI.

Интерфейс сервиса — это *ссылка на сервис*, а не на удалённый объект, требуемый клиенту. Благодаря такой архитектуре собственно и реализуется идея уровня интерфейса сервисов, соответствующая бизнес-парадигме архитектуры предприятия, показанной ранее на рисунке 4.10.

> **Исторически** и, в прикладном плане, наибольшую популярность получили технологии Web-сервисов.

В 1998 году Дейв Винер, сотрудник компании UserLand Software, опубликовал протокол ***XML-RPC***, предназначенный для вызова удалённых процедур в формате XML-сообщений поверх протокола HTTP. Сам протокол разрабатывался по заказу корпорации Microsoft для обеспечения решения задач в области электронной коммерции. Последующие доработки этого протокола привели к созданию протокола ***SOAP*** (*Simple Object Access Protocol*).

Известны две спецификации этого протокола, опубликованные консорциумом W3C:

а) Simple Object Access Protocol (SOAP) 1.1;

б) SOAP Version 1.2 Part 0: Primer (Second Edition).

В марте 2001 года и в июне 2007 года, консорциум W3C публикует спецификации ***WSDL*** (*Web Service Description Language*), что обеспечивают описание сервисов в формате XML-документов и позволяет их публикацию средствами web-технологий.

Считается, что в августе 2000 года была реализована первая система ***UDDI*** (*Universal Description Discovery & Integration*) — инструмент для размещения описаний WSDL. В настоящее время доступно описание версии UDDI 3.0.2, которое находится под контролем глобального консорциума ***OASIS*** (*Organization for the Advancement of Structured Information Standards*).

> **Примечание** — Более подробно многие аспекты теории и технологии парадигмы SOA изучаются в магистерской дисциплине «Распределённые сервис-ориентированные системы» [44]. Далее мы конкретизируем применение этой парадигмы для нужд реализации информационных систем (ИС).

4.1.4 Контейнерные технологии Web-сервисов

Показанные ранее рисунки 4.8 и 4.9 представляют нам стандартные контейнеры и общий набор служебных сервисов технологии Java EE. Хорошо видно, что проектировщикам и разработчикам автоматизированных и информационных систем предлагаются уже готовые:

а) **Веб-контейнер** — серверная среда для реализации компонент *EJB Lite*, *Сервлет* и *JSF*;

б) **Служебный сервис** — *сервис серверной среды Веб-контейнера*, предоставляющий многообразный функционал реализации прикладных сервисных компонент;

в) **Протоколы HTTP и HTTPS** — *стандартизированный транспортный инструмент* для доступа приложений клиентов к среде серверов приложений.

Уже наличие перечисленного инструментария позволяет с успехом выполнить этап эскизного проектирования АС или ИС.

Простейшие ИС могут быть созданы посредством реализации компонента *Сервлета*.

В 1997 году, компания Sun Microsystems предложила *технологию сервлетов*, опубликовав их спецификацию версии 1.0 и реализовав эту спецификацию на языке Java в виде пакета *javax.servlet*. В дальнейшем был разработан пакет *javax.servlet.http*, обеспечивающий интерфейсы и классы для создания высокопроизводительных Web-серверов.

В 1999 году, Apache Software Foundation реализовала спецификацию HTTP-сервлетов в широко известном Web-сервере *Apache Tomcat*.

Apache Tomcat (в старых версиях – *Catalina*) — контейнер сервлетов с открытым исходным кодом, реализующий спецификации *HTTP-сервлетов* и *JavaServer Pages* согласно общей метамодели шаблона проектирования MVC.

Конкретизация этой архитектуры, применительно к серверу Apache Tomcat, показана на рисунке 4.12.

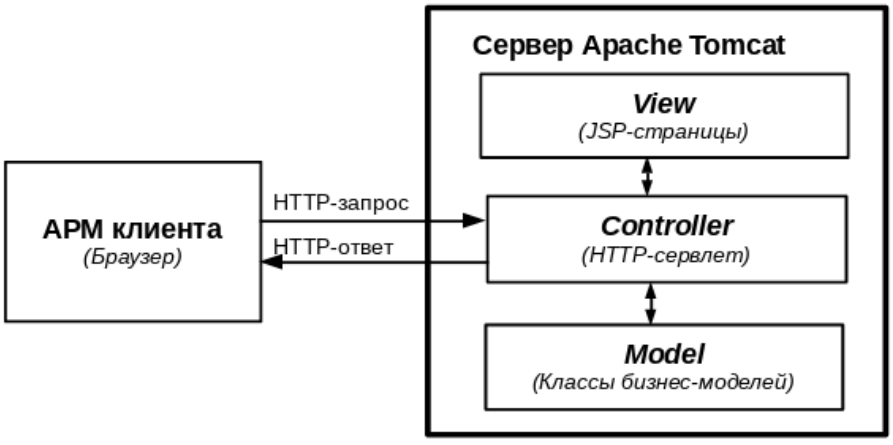

Рисунок 4.12 — MVC-архитектура взаимодействия приложения клиента и сервера Apache Tomcat

Для проектировщика информационных систем (ИС), идея шаблона MVC, реализуемая сервером Apache Tomcat, состоит в следующем:

а) компонента ***Controller*** (сервлет) реализуется публичным Java-классом, который должен расширять класс *HttpServlet*;

б) различные ***HTTP-запросы*** АРМ-клиента (*GET*, *POST*, *DELETE* и другие) могут быть обработаны соответствующими методами сервлета: *doGet(...)*, *doPost(...)*, *doDelete(...)* и другие аналогичные методы, причём в качестве аргументов им передаются объекты

классов *HttpServletRequest* и *HttpServletResponse*;

в) объект типа ***HttpServletRequest*** содержит как данные от АРМ клиента, так и может дополняться данными классов компоненты ***Model***, которые проектировщик должен сам реализовать;

г) объект типа ***HttpServletResponse*** содержит результирующий ответ для АРМ клиента;

д) для обращения к компоненте ***View***, контейнер сервера предоставляет специальный класс типа *RequestDispatcher*, с помощью которого ***Controller*** может обратиться к нужной проектировщику JSP-странице и отправить результат такого обращения клиенту, в виде *HTTP-ответа*;

е) компонента ***View*** реализуется проектировщиком в виде набора *JSP-страниц*, представляющих собой текстовые файлы общего формата XHTML;

ж) ***JSP-страницы*** — специальный формат XHTML, допускающий программные вставки на языке Java и доступ к объектам запроса и ответа, формируемых сервлетом.

Технология сервлетов для каждого АРМ клиента поддерживает отдельную сессию.

Первоначально технология сервлетов предназначалась для создания высокопроизводительных Web-серверов, генерирующих и кеширующих динамические Web-страницы. Apache Tomcat является примером одного из первых таких серверов, который стал использовать контейнерные технологии предлагаемые проектом Java EE.

В частности, инструментальные средства разработки Eclipse EE имеют специальный тип проектов «Dynamic Web Project», способный инкапсулировать в проект дистрибутив сервера конкретного сервера Apache Tomcat, а затем обеспечить разработку и тестирование приложения, созданного по схеме архитектуры MVC.

Примечание — В учебной дисциплине «Распределённые вычислительные системы», студенты подробно изучают технологию сервлетов, поэтому приведённого выше описания — вполне достаточно для самостоятельного для понимания и применения этой технологии в задачах проектирования информационных систем (ИС).

С 1999 года, Sun Microsystems стала формировать отдельную платформу для разработки приложений уровня предприятия — J2EE или Java EE. Новые контейнерные технологии стали добавляться к серверу Apache Tomcat, в результате чего стал формироваться дистрибутив сервера приложений, названный ***Apache TomEE***.

В марте 2004 года, компания Sun Microsystems выпустила первую версию спецификации JSF 1.0, которая должна была усовершенствовать технологию сервлетов и перевести её на компонентную основу.

JSF (*JavaServer Faces*) — спецификация для построения компонентно-ориентированных пользовательских интерфейсов для Web-приложений, написанных на языке Java.

В последующем эта спецификация несколько раз пересматривалась и усовершенствовалась. Общая архитектура JSF версии 2.3, выпущенная в конце марта 2017 года, была подробно описана в монографии [45] и представлена, как показано на рисунке 4.13.

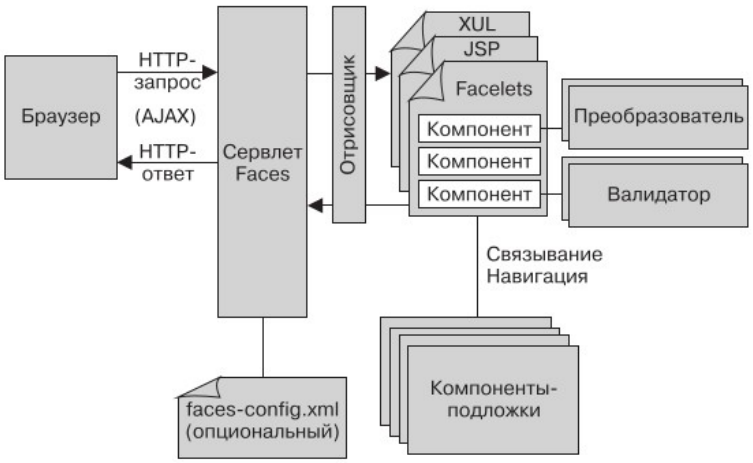

Рисунок 4.13 — Общая архитектура взаимодействия браузера и контейнерной компоненты JSF 2.3 [45]

Показанная на рисунке 4.13 компонента JSF «Сервлет Faces» представляет собой специальную реализацию сервлета, объединяющего в себе служебную функциональность всех трёх компонент шаблона MVC:

а) *клиент* (браузер) с помощью HTTP-запросов адресует набор XHTML-страниц, базирующихся на спецификации HTML 4.01;

б) *адресуемые XHTML-страницы Facelets*, обрабатываемые программным обеспечением языка описания страницы (*PDL*, *Page Description Language*);

в) «*Компоненты-подложки*», на которые адресуются XHTML-страницы и представляют собой «модули» компоненты ***Model*** (в терминологии шаблона MVC);

г) *сам* «*Сервлет Faces*», объединяющий XHTML-страницы *Facelets* и реализующие ***Model*** компоненты-подложки.

Преимущество технологии JSF над технологией HTTP-сервлетов и JSP-страниц состоит в возможности интеграции и отображения множества XHTML-страниц, предоставляя проектировщику возможность многоуровневой реализации компоненты *Model*.

Рекомендуемый спецификацией JSF 2.0 язык описания страниц *Facelets* позволяет объединять множество XHTML-страниц, предоставляя их браузеру как единое изображение. Это повышает повторное использование уже разработанных частей интерфейса приложений.

Что же касается компонент-подложек (Java-классов), то они реализуют *Бизнес-модель* приложения, выделяя как минимум три уровня существования порождаемых ими объектов, что также наглядно показано на рисунке 4.14 и представлено следующим списком:

а) классы, аннотируемые как ***@RequestScoped***, порождают объекты, существующие *только на время осуществления самого HTTP-запроса*;

б) классы, аннотируемые как ***@SessionScoped***, порождают объекты, существующие *на всё время сессии взаимодействия браузера и сервера*;

в) классы, аннотируемые как ***@ApplicationScoped***, порождают объекты, существующие *на всё время существования серверного приложения*.

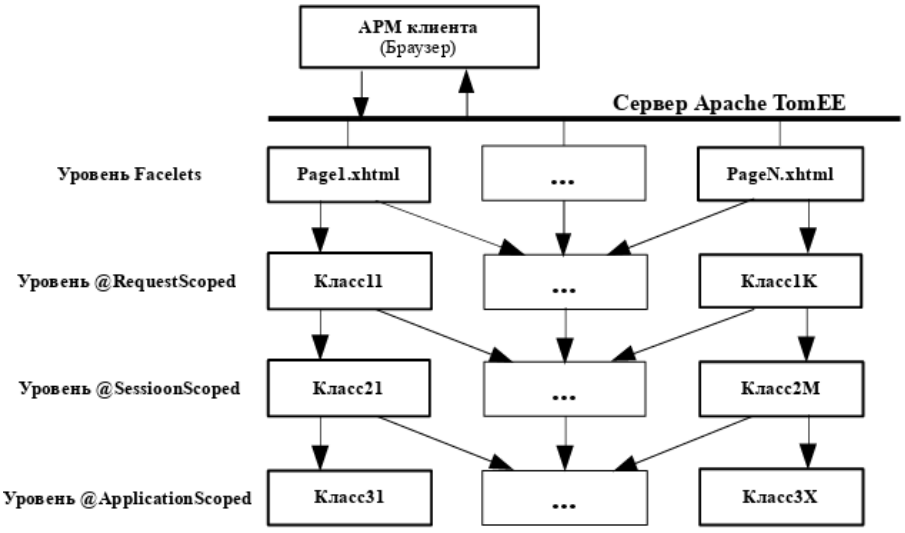

Рисунок 4.14 — MVC-архитектура взаимодействия приложений клиента и сервера по технологии JSF

Инструментальные средства разработки Eclipse EE позволяют подключать сервер приложений Apache TomEE и, в рамках проекта «Dynamic Web Project», реализовать приложения с использованием технологии JSF.

Основные преимущества использования технологии JSF, как шаблона структурного проектирования АС и ИС:

а) *возможность декомпозиции* представления ***View*** отдельными компонентами *Facelets*, с последующей интеграцией этих компонент в едином представлении для *АРМ клиента* (*Браузера*);

б) *возможность декомпозиции* бизнес-приложения ***Model*** на множество классов, уменьшающих алгоритмическую сложность реализации каждого этапа разработки ПО;

в) *возможность последовательной разработки* и отладки программного обеспечения создаваемой системы.

Например, на рисунке 4.15 представлен шаблон интерфейса для уровня *Facelets*, состоящий из пяти HTML-страниц проекта типа «Dynamic Web Project».

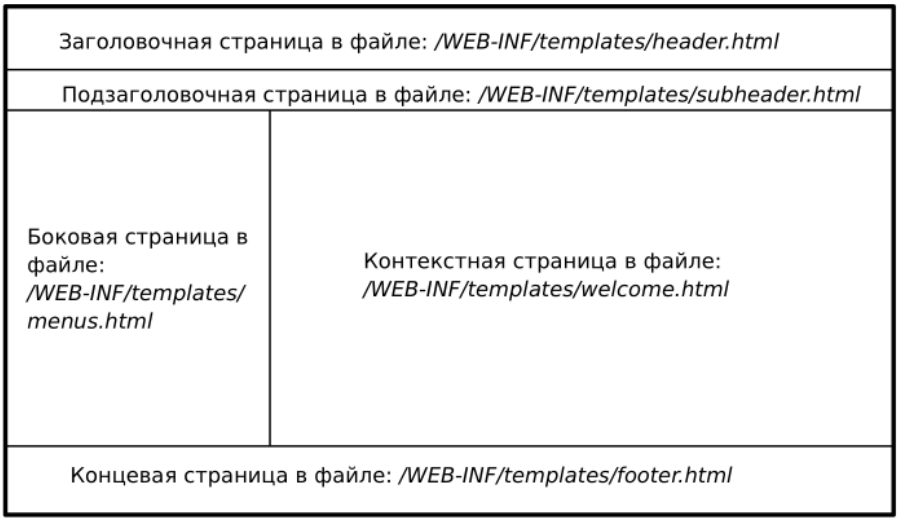

Рисунок 4.15 — Шаблон представления из пяти HTML-страниц [44]

Представленный шаблон интерфейса является *проектом интерфейса* конкретного учебного приложения, описанного источнике [44]. Первоначально он перечисляет наименования файлов, составляющих его HTML-страницы и местоположение этих страниц в пределах окна браузера. В дальнейшем, пользователь браузера получит возможность обращения к серверу Apache TomEE по любой из указанных страниц.

На рисунке 4.16 показана первоначальная реализация спроектированного шаблона, в пределах проекта ***labs*** инструментальной среды Eclipse EE.

Обратите внимание, что:

а) браузер среды Eclipse EE обращается к общедоступной, для всех браузеров, странице XHTML с именем *index.xhtml*;

б) страница ***index.xhtml***, с помощью специальных операторов языка *Facelets*, обращается к невидимой на рисунке 4.16 и недоступной браузерам XHTML-странице-шаблону с именем */WEB-INF/templates/lab2Templ.xhtml*, который и организует, как размещение, так и отображение остальных пяти HTML-страниц;

в) в результате мы имеем структурную заготовку интерфейса пользователя для реализуемого учебного проекта.

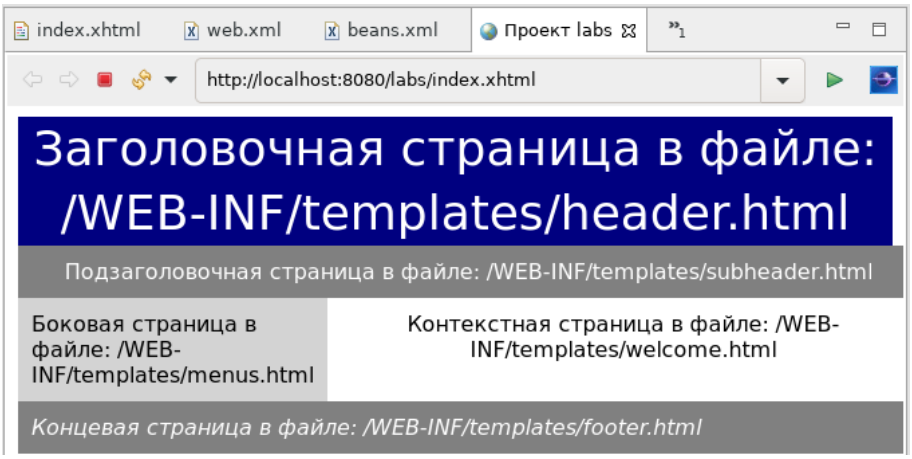

Рисунок 4.16 — Первоначальная реализация шаблона интерфейса проекта labs, состоящего из пяти проектных HTML-страниц [44]

Шаблон интерфейса, показанный на рисунке 4.16 и заданный набором HTML-страниц, ссылается на конкретные имена файлов и адреса их размещения в пределах реализуемого проекта.

Таким образом:

а) для отдельных HTML-страниц создаются компоненты-подложки, реализующие функционал некоторой части бизнес-приложения ***Model*** на различных уровнях иерархии, согласно общей схеме рисунка 4.14;

б) конкретная HTML-страница заменяется XHTML-страницей, с необходимой привязкой к нужным компонентам-подложкам.

> **Примечание** — Описанная выше технология JSF позволяет проектировщику создавать пользовательские интерфейсы на стороне сервера и последовательно реализовывать бизнес-приложения ***Model***.

Для примера, на рисунке 4.17 показан один из вариантов пользовательского интерфейса, реализованного в рамках учебного проекта ***labs*** [44].

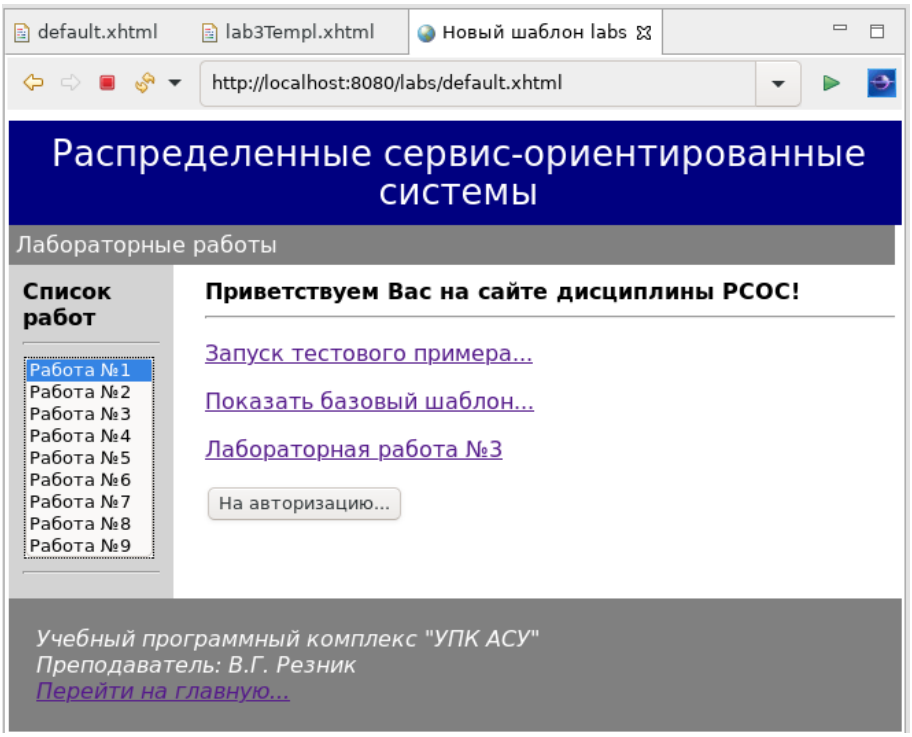

Рисунок 4.17 — Один из вариантов интерфейса проекта labs, реализуемого учебного проекта [44]

Для более подробного изучения примеров использования технологии JSF следует воспользоваться учебным пособием [44].

4.1.5 Проектирование ИС как Web-службы в стиле REST

Технология REST является современной популярной альтернативой Web-службам SOAP.

Действительно, основным недостатком модели SOA, реализуемым на основе протокола SOAP, — необходимость описания взаимодействия компонентов распределённой системы на языке WSDL, который для многих приложений является явно избыточным.

В основе языка WSDL лежит язык XML, обеспечивающий не только описание самого взаимодействия распределённых систем, но также — описание XML-схем, необходимых для синтаксического контроля таких описаний. В результате, проектные решения на основе протокола SOAP становятся сложными не только для поставщиков сервисов, но главное — это является сложным для основной массы потребителей этих сервисов.

Технология REST (*Representational State Transfer*) — технология передачи состояния представления, означающая, что REST-запрос клиента к серверу содержит всю нужную информацию о желаемом ответе сервера.

> **Передача состояния представления** (*REST*) — архитектурный стиль проектирования и реализации распределённых сервис-ориентированных систем, максимально использующих возможности Web-технологий.

Основная позитивная идея сторонников REST-технологий — *замена сложных описаний* сетевого взаимодействия на языке WSDL *на более простые описания* сетевого взаимодействия клиентов и серверов, в виде URI/URN-ресурсов.

Проектное решение использования URI/URN-ресурсов основано на публикации этих

ресурсов *в контенте уже используемых средств гипермедиа*, например, в тексте обычных страниц на языке HTML.

Таким образом, потребитель сервиса, вместо чтения описаний интерфейсов и реализации программных агентов, обеспечивающих доступ к сервисам через эти интерфейсы, просто выбирает ссылку в окне браузера и получает необходимый сервис.

> **Web-службы**, построенные с учётом ограничений REST-технологий, называются RESTful-системами.

В отличие от Web-сервисов, построенных на основе протокола SOAP, *RESTful-системы* не имеют официального стандарта на своё API, поэтому принято считать, что это — *архитектурный стиль программирования*.

> **RESTful** — это архитектурный стиль проектирования и реализации сервис-ориентированных систем.

Считается, что идейной парадигмой технологии REST стала стандартная классификация набора действий по работе с базами данных *CRUD*, которая была введена Джеймсом Мартином в 1983 году.

CRUD — это акроним, обозначающий четыре базовых действия (функций) для работы с базами данных:

а) CREATE — создание записей (INSERT);

б) READ — чтение записей (SELECT);

в) UPDATE — модификация записей;

г) DELETE — удаление записей.

Применительно к Web-службам, термин CRUD проецируется на четыре HTTP-запроса к Web-серверам:

а) POST — запрос на создание ресурса;

б) GET — запрос на получение ресурса;

в) PUT — запрос на модификацию ресурса;

г) DELETE — запрос на удаление ресурса.

Сам термин REST был введён Роем Филдингом в 2000 году и упоминается в ключе его докторской диссертации: «Архитектурные стили и дизайн сетевых программных архитектур». Им было предложено и шесть ограничений, нарушение которых не позволяет считать сервис-приложение REST-системой:

1. **Модель «Клиент-сервер»** — предполагается приведение приложения, возможно целостного по начальной реализации, к архитектуре, где выделены *поставщик сервиса* и *потребители сервиса*. Это улучшает масштабируемость приложения и позволяет отдельным частям системы развиваться независимо друг от друга.
2. **Отсутствие состояния** — обеспечение на стороне сервера *протокола взаимодействия без сохранения состояния*. При этом, состояние сессии должно сохраняться на стороне клиента (потребителя сервиса).
3. **Кэширование ответов сервера** — способность сервера или промежуточных узлов *кэшировать свои ответы*. Это позволяет повысить производительность и расширяемость системы.
4. **Единообразие интерфейсов сервисов** — фундаментальное требование дизайна REST-сервисов. Унификация интерфейсов должна соответствовать четырём дополни-

тельным условиям: *идентификации ресурсов посредством URI*; *возможности манипуляции ресурсов на основе представлений*; *использованию «самозаписываемых» сообщений* и *применению гипермедиа как средства изменения состояния приложения*.

5. **Слои взаимодействия** — использование взаимодействия клиента и сервера на основе иерархической структуры сетей.
6. **Код по требованию** (*необязательное ограничение*) — возможность расширения функциональности клиентов за счёт загрузки кода с серверов приложений в виде апплетов или сценариев.

> **По мнению Роя Филдинга**, приложения, не соответствующие приведённым условиям, не могут называться REST-приложениями.

По мнению Филдинга, приложения, которые соответствуют указанным выше шести условиям, получают следующие преимущества:

а) *надёжность* по причине отсутствия необходимости сохранять информацию о состоянии клиента, поскольку она может быть утеряна;

б) *производительность*, за счёт использования кэша, и масштабируемость, за счёт разделения поставщиков и потребителей сервисов;

в) *простота интерфейсов и портативность компонентов*, создаваемых систем;

г) *лёгкость внесения изменений* и *способность «эволюционировать»* под воздействием новых требований к системам.

> **Программная платформа Java EE** обеспечивает полную поддержку технологии RESTful с помощью инструментальных средств проекта *JAX-RS*.

JAX-RS (*Java API for RESTful Web Services*) — это спецификация API языка программирования Java, обеспечивающая поддержку создания Web-сервисов в соответствии с архитектурным шаблоном передачи состояния представления REST.

JAX-RS не имеет своего RFC, но содержит достаточно полное описание для версии 2.0 от 22 мая 2013 года, представленное корпорацией Oracle. Полный перечень пакетов технологи JAX-RS представлен в таблице 4.1.

Таблица 4.1 — Основные пакеты JAX-RS [45]

Пакет	Описание пакета
javax.ws.rs	Высокоуровневые интерфейсы и аннотации, используемые для создания Web-служб с передачей состояния представления.
javax.ws.rs.client	Классы и интерфейсы клиентского API JAX-RS.
javax.ws.rs.container	API JAX-RS контейнера.
javax.ws.rs.core	Низкоуровневые интерфейсы и аннотации, используемые для создания Web-ресурсов с передачей состояния представления.
javax.ws.rs.ext	API, предоставляющие расширения для типов, поддерживаемых в JAX-RS API.

Инструментальные средства разработки Eclipse EE позволяют подключать сервер приложений Apache TomEE и, в рамках проекта «Dynamic Web Project», реализовать серверную часть приложений ИС, как Web-службу (сервисную службу) в стиле REST.

RESTful-сервис — специальный сервлет проекта «*Dynamic Web Project*», импортирующий пакеты JAX-RS и являющийся JAVA-классом, с базовой аннотацией *@Path(...)*, и одной из аннотаций *@Stateless* или *@Singleton*.

@Stateless — аннотация определяющая, аннотируемый класс не сохраняет состояния.

@Singleton — аннотация определяющая, аннотируемый класс может продуцировать только один объект.

Наиболее важные аннотации пакета JAX-RS представлены в таблице 4.2.

Таблица 4.2 — Основные аннотации пакета JAX-RS

Аннотация	Описание
@Path(...)	Указывает относительный путь для класса или метода запрашиваемого ресурса.
@GET, @PUT, @POST, @DELETE и @HEAD	Указывают тип HTTP-запроса, разделяя ПО сервиса на соответствующие части.
@Produces(...)	Указывает MIME-тип ответа, генерируемый методом сервлета.
@Consumes(...)	Указывает принимаемый MIME-тип запроса, принимаемый методом сервлета от клиента.

Главной аннотацией, по которой контейнер определяет тип RESTful-сервлета, является аннотация @Path(...), имеющая общий формат:

@Path("/Путь1/Путь2/.../ПутьN") **(4.1)**

где ***ПутьK*** — слово, задающее часть пути, определённое в адресе URI-запроса.

Методы RESTful-сервлета аннотируются парами: *@Path(...)* и одной из аннотаций таблицы 4.2, указывающей тип HTTP-запроса. Причём аргументы аннотации *@Path(...)* учитываются в качестве аргументов методов RESTful-сервлета.

Технология RESTful является одним из стилей проектирования сервис-ориентированных систем, а не его стандартом.

Контейнерная технология JAX-RS предлагает общий адекватный класс ответа сервера, возвращаемый методами RESTful-сервлета и определённый выражением (4.2):

javax.ws.rs.core.Response **(4.2)**

Для реализации запросов приложений клиентов к RESTful-сервлетам, технология JAX-RS предлагает соответствующий набор классов, главным из которых является класс клиента, определённый выражением (4.3):

javax.ws.rs.client.Client **(4.3)**

Для изучения конкретных примеров вариантов использования технологии RESTful следует воспользоваться учебным пособием [44].

Технология RESTful ориентирована *на максимально упрощённый удалённый доступ к хранилищам* информационных систем (ИС).

Практическое использование технологии RESTful предполагает детальное знание всех особенностей применения протокола HTTP, а также MIME-типов возвращаемых им ответов сервера. Кроме того, следует учесть, что браузеры Web-технологий обеспечивают только HTTP-запросы GET и POST.

Подводя общий итог изложенному в данном подразделе учебному материалу, проектировщику следует учесть следующие общие рекомендации:

а) *контейнерные технологии Java EE*, реализуемые серверами приложений, предлагают множество служебных сервисов, обеспечивающих реализацию распределённых приложений на достаточно высоком профессиональном уровне;

б) *наиболее адекватными для реализации ИС* являются Web-технологии серверов приложений, например Apache TomEE, предлагающие шаблоны проектирования MVC, поддерживаемые сервлетами JSP или JSF;

в) *доступ к локальным хранилищам данных ИС* следует обеспечивать на сервере приложений, с помощью реализации соответствующих EJB-компонент;

г) *доступ к удалённым хранилищам данных ИС* рекомендуется обеспечивать разработкой классов, использующих технологию RESTful, если это конечно позволяют удалённые сервера приложений.

ЛАБОРАТОРНАЯ РАБОТА №4