Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное

учреждение высшего профессионального образования

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

Кафедра автоматизации обработки информации (АОИ)

К ЗАЩИТЕ ДОПУСТИТЬ

Заведующий кафедрой АОИ

д-р техн. наук, профессор

\_\_\_\_\_\_\_\_\_\_\_\_ Ю.П.Ехлаков

**«\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_** 2014 г.

**АВТОМАТИЗИРОВАННАЯ ИНФОРМАЦИОННАЯ СИСТЕМА**

**«ПРИЕМ И ОБРАБОТКА ЗАЯВОК НА ОБСЛУЖИВАНИЕ»**

Дипломный проект

по специальности 230102.65 – Автоматизированные системы обработки информации и управления

Пояснительная записка

|  |  |
| --- | --- |
| СОГЛАСОВАНО | |
| Консультант по ТЭО  Ст. преп. кафедры АОИ  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Б.А. Рыбалов  **«\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_**2014 г. | Студент гр. з-429-б  **\_\_\_\_\_\_\_\_\_\_** О.С. Алексеев  **«\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_**2014 г. |
|  | Руководитель от предприятия,  программист  ИП Кунгуров А.Н., «Русское ТВ»  **\_\_\_\_\_\_\_\_\_\_\_** Р.Е. Верещагин  **«\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_**2014 г. |

Руководитель от ТУСУРа

доцент каф. АОИ, канд. техн. наук

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ П.В. Сенченко

**«\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_**2014 г.

Томск 2014

**РЕФЕРАТ**

Дипломная работа 97с., 24 рис., 12 табл., 52 источника, 3 прил., 4 л. графического материала.

АВТОМАТИЗИРОВАННАЯ ИНФОРМАЦИОННАЯ СИСТЕМА, ЗАЯВКИ, ОБСЛУЖИВАНИЕ, WEB, JAVA, ORACLE, ORM, INTERNET, APACHE, SPRING, MVC, REST

Объектом исследования является процесс осуществления обслуживания по входящим заявкам в сети магазинов «Русское ТВ», ИП Кунгуров А.Н.

Целью работы является разработка автоматизированной информационной системы для приема и обработки заявок на основе анализа предметной области и текущем ручном процессе приема и обработки заявок на обслуживание.

Результатом выполнения дипломной работы является система «RFS». Система внедрена в рабочий процесс call-центра сети магазинов «Русское ТВ».

Пояснительная записка содержит описание предметной области и проблемной ситуации, формулировки требований к разрабатываемой системе, обоснование выбора и описание систем и методов разработки, описание функций реализованной системы.

Дипломная работа выполнена в текстовом редакторе MS Word 2010.

**THE ABSTRACT**

Graduate qualification work, ………. (дописать)

AUTOMATED INFORMATION SYSTEM, REQUESTS, SERVICE, JAVA, ORACLE, ORM, INTERNET, APACHE, SPRING, MVC, REST.

Object of research is service process by incoming requests in “Russian TV” chain stores, individual entrepreneur – A. N. Kungurov.

Main purpose of work is development of automated information system for requests for service receiving and processing, based on current manual processing of requests.

Result of graduate work is “RFS” system embedded in work process of chain stores “Russian TV” call-center.

Explanatory note contains description of subject domain and problem situation, requirements formulations for developed system, justification of selection and description of development systems and methods, description of realized system functionality.

The diploma thesis is performed in MS Word 2010 text editor.

Федеральное агентство по образованию РФ

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ

УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

Кафедра автоматизации обработки информации (АОИ)

**УТВЕРЖДАЮ**

Заведующий кафедрой АОИ

д-р техн. наук, профессор

\_\_\_\_\_\_\_\_\_\_\_ Ю.П. Ехлаков

"\_\_\_\_"\_\_\_\_\_\_\_\_\_2014г.

**ЗАДАНИЕ**

**на дипломный проект**

студенту Алексееву Олегу Сергеевичу гр. з-429-б факультета систем управления

1. Тема работы: «АИС для приема и обработки заявок на обслуживание» утверждена приказом по ВУЗу от № \_\_\_\_\_\_\_\_
2. Срок сдачи работы на кафедру: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
3. Содержание работы (перечень подлежащих разработке вопросов):

1) проектирование системы, создание прототипа (если он необходим);

2) разработка и модульное тестирование компонентов системы;

3) тестирование системы;

4) внедрение системы на предприятие.

1. Дата выдачи задания: « » \_\_\_\_\_\_\_\_\_ 2014 г.

**Руководитель**: Программист, ИП Кунгуров А.Н., сеть магазинов «Русское ТВ»

Верещагин Роман Евгеньевич

***\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*** «\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2014 г.

(подпись руководителя)

Задание принял к исполнению:

***\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*** «\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2014 г.

(подпись студента)

**СОДЕРЖАНИЕ**

[1 ВВЕДЕНИЕ 7](#_Toc389436911)

[2 ОСНОВНАЯ ЧАСТЬ 8](#_Toc389436912)

[2.1 Общие сведения о предприятии 8](#_Toc389436913)

[2.2 Сведения об организационной структуре предприятия 8](#_Toc389436914)

[2.3 Выявление проблем и постановка задач 9](#_Toc389436915)

[2.4 Построение архитектуры системы 13](#_Toc389436916)

[2.5 Средства и методы разработки АИС 18](#_Toc389436917)

[2.5.1 Выбор языка и платформы программирования 18](#_Toc389436918)

[2.5.2 Выбор средств хранения данных 32](#_Toc389436923)

[2.5.3 Сервер приложений 36](#_Toc389436928)

[2.5.4 Инструменты и методы разработки 38](#_Toc389436929)

[2.6 Обзор аналогов системы 44](#_Toc389436940)

[2.6.1 LANDesk Service Desk 44](#_Toc389436941)

[2.6.2 Luxoft Service Desk 45](#_Toc389436942)

[2.6.3 Сравнительная таблица продуктов 46](#_Toc389436943)

[2.7 Разработка системы «RFS» 47](#_Toc389436944)

# 1 ВВЕДЕНИЕ

Торговля сложным техническим товаром влечет за собой необходимость поддерживать потребителя, предоставляя ему гарантию и платный сервис. И, если гарантийным ремонтом техники, как правило, занимаются авторизованные производителем сервисные центры, то обслуживание, не попадающее под условия гарантии, будь то установка, конфигурирование, или пост гарантийный ремонт, осуществляется сторонними производителю организациями. Такой вид деятельности является хорошим толчком для расширения бизнеса торговой сети.

Для того, чтобы производить сервисное обслуживание, необходимо организовать на предприятии инфраструктуру для этого. Помимо физического оборудования, должен быть организован слой взаимодействия между клиентом и сервис-инженерами. Таким слоем служит call-центр – диспетчеры принимают звонки от клиентов и передают заявки инженерам. Долгое время основная часть этого процесса выполнялась на предприятии вручную – диспетчер принимал звонок, фиксировал полученные от клиента сведения удобным для себя способом и обзванивал инженеров, пока не находил свободного, для передачи ему сведений о заявке. Руководству предприятия было предложено автоматизировать большую часть этого процесса, чтобы работа диспетчера заключалась только в приеме звонка и фиксации сведений. Для выполнения остальной работы была разработана система «RFS» (Request for service – запрос обслуживания).

В перспективе, рассматриваемая система может быть доработана до коробочного продукта с дальнейшей продажей.

# 2 ОСНОВНАЯ ЧАСТЬ

# 2.1 Общие сведения о предприятии

Сеть магазинов «Русское ТВ» занимается реализацией оборудования и услуг, связанных со спутниковым телевидением. Компания оказывает такие услуги, как установка оборудования, сервисное обслуживание, ремонт оборудования, помощь в настройке и т.д.

Компания «Русское ТВ» является официальным дилером и сертифицированным сервисным центром известных российских операторов спутникового телевидения, таких как «НТВ-ПЛЮС», «Триколор ТВ» и т.д.

# 2.2 Сведения об организационной структуре предприятия

Форма собственности предприятия – индивидуальный предприниматель.

Предприятие делится на несколько отделов (рисунок 2.1), каждый отдел обособлен, все отделы подчиняются администрации.



*Рисунок 2.1 – Организационная схема предприятия*

Разрабатываемая система напрямую касается call-центра, сведения о рабочих графиках сотрудников должны быть доступны вследствие интеграции с внешней системой учета персональных данных сотрудников, разрабатываемой в рамках предприятия для службы, а сервисный отдел должен получать уведомления из системы по электронной почте.

# 2.3 Выявление проблем и постановка задач

До разработки системы «RFS» заявки обрабатывались вручную, и сведения о них хранились только на бумажных носителях. Такой подход применялся достаточно долго. С расширением штата сотрудников и появлением нескольких диспетчеров call-центра, выявилась проблемная ситуация – большие объемы таких данных тяжело обработать вручную, т.к. отчеты о заявках за период и отчеты о заявках по инженеру также приходилось составлять вручную, на что требовалось длительное время и человеческие ресурсы. Руководством предприятия было принято решение ускорить этот процесс за счет автоматизации обработки заявок и хранения данных о заявках в электронном виде.

Для решения этой проблемы необходимо проанализировать процесс ручной обработки заявок, выявить этапы жизненного цикла заявки и определить, какие из этих этапов можно автоматизировать для сокращения работы диспетчера и инженера.

По итогам исследования, в которое входило общение с диспетчерами call-центра и просмотр доступных записей о заявках, было выявлено семь этапов жизненного цикла заявки:

1. Звонок клиента;
2. Сбор информации о проблеме;
3. Поиск свободного инженера;
4. Назначение инженера на заявку;
5. Уведомление инженера;
6. Уведомление диспетчера;
7. Закрытие заявки.

Для большей наглядности была составлена UML-диаграмма потока, иллюстрирующая жизненный цикл (рисунок 2.2).



*Рисунок 2.2 – Жизненный цикл заявки при ручной обработке*

Ключевой этап жизненного цикла – *сбор информации о проблеме*. При использовании бумажного носителя эти данные никак не структурировались и записывались диспетчером со слов клиента в произвольной форме. При этом, у диспетчеров не было четких инструкций о том, какие именно данные необходимы для исполнения заявки. При таком подходе был опыт передачи заявки инженеру без некоторой важной информации. Следовательно, для корректной обработки сведений о проблеме клиента, диспетчеру нужна четкая инструкция, в качестве которой может быть использована экранная форма с обязательными полями.

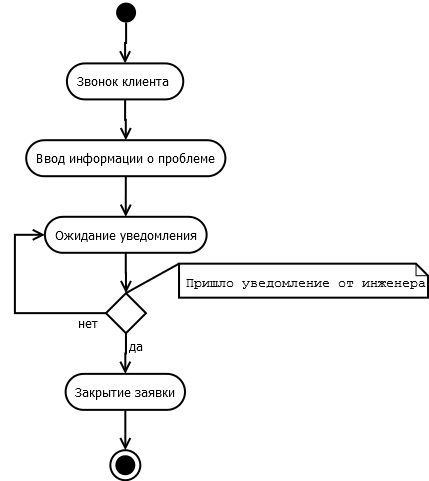
Следующий шаг – *поиск свободного инженера*. Он также осуществлялся диспетчером, с использованием доступных средств коммуникации – телефон, электронная почта и интернет-мессенджеры. Диспетчер последовательно связывался с инженерами до тех пор, пока не находил свободного на назначенную дату. В качестве решения этой проблемы было предложено разработать подсистему, предоставляющую сведения о графике работы сотрудников с возможностью выявления инженеров, свободных на выбранную дату.

Не менее важный этап – *назначение инженера на заявку*. Диспетчер, помимо уведомления инженера, отслеживает исполнение заявок. Для этого приходилось хранить на рабочем месте сведения о заявках, находящихся в работе и, ежедневно сверять заявки по каждому инженеру. Для решения этой проблемы было предложено использовать отчеты о заявках по датам и отчеты с детализацией по сотрудникам.

На следующем этапе – *уведомлении инженера* – диспетчер вручную формировал и отправлял по электронной почте сообщение, с полученными от клиентами сведениями о заявке. После исполнения заявки инженер, также вручную, посылал ответное уведомление диспетчеру. Для сокращения времени формирования и отправки уведомлений было предложено стандартизировать их текст с возможностью автоматического дополнения, и переложить работу по отправке сообщений автоматизированной системе.

На этапе закрытия заявки диспетчер помещал бумажный носитель со всеми полученными сведениями в архив. Носители передавались в службу управления персоналом и бухгалтерию для дальнейшего взаимодействия предприятия с инженером. Для ускорения и повышения удобства этого процесса было предложено предусмотреть в разрабатываемой системе возможность экспорта в другие системы посредством сетевого соединения.

Таким образом, по итогам анализа жизненного цикла заявки на обслуживания, была выявлена возможность сократить работу диспетчера (рисунок 2.3), переложив этапы, связанные с поиском и уведомлением инженера, на автоматизированную систему, при этом, предоставив диспетчеру четкую инструкцию по сбору данных от клиента.



*Рисунок 2.3 – Работа диспетчера при автоматизированной обработке*

Помимо проблемы, связанной с жизненным циклом заявки, диспетчерам необходимо поддерживать данные о сотрудниках в актуальном состоянии. Служба управления персоналам не уведомляет call-центр о приеме новых инженеров или увольнении работающих, но, сохраняет все данные о них в специально предназначенной для этого автоматизированной системе с возможностью экспорта данных через сетевое соединение. Следовательно, проблему поддержания базы данных о сотрудниках в актуальном состоянии целесообразно решить, предоставив системе приема и обработки заявок на обслуживание возможность импортировать данные об инженерах.

Исходя из того, что в разрабатываемой системе должен быть доступен функционал для приема заявок, отметки об исполнении заявки и обмена данными с внешними системами, следует, что целесообразно разграничить доступ к системе, разделив пользователей на администраторов, диспетчеров и инженеров.

Таким образом, на этапе выявления проблем были поставлены следующие задачи:

* Предоставление диспетчеру четкой инструкции по сбору сведений от клиента, посредством экранной формы;
* Разработать подсистему автоматического поиска свободных инженеров на текущую дату;
* Обеспечить возможность автоматического уведомления инженеров о новых заявках;
* Обеспечить возможность автоматического уведомления диспетчеров об исполнении инженером заявки;
* Обеспечить возможность получения отчетов о заявках по дате;
* Обеспечить возможность получения отчетов о заявках по инженеру;
* Предоставить внешним системам доступ к данным о заявках;
* Обеспечить возможность импорта данных о сотрудниках из системы службы управления персоналом;
* Обеспечить разграничение доступа между администраторами, диспетчерами и инженерами.

# 2.4 Построение архитектуры системы

Любая корпоративная информационная система должна обладать набором качеств, облегчающих ее разработку и сопровождение:

* Гибкость;
* Модульность;
* Способность к изменению.

*Гибкость* системы заключается в возможности и простоте ее конфигурирования без декомпиляции и доступа к исходному коду. Самые частые подходы к конфигурированию: сохранение параметров приложения в XML-конфигурации, либо в базе данных. При соблюдении этого подхода можно достичь хорошей переносимости системы из одного окружения в другое без перекомпиляции.

*Модульность* предполагает, что в системе должна быть возможность как простого добавления нового функционала (модуля), так и изъятия функционала без ущерба системе. Система с высокой модульностью легко тестируема - некоторые модули можно заменить специально подготовленными тестовыми модулями.

*Способность к изменению* заключается в том, что изменения, вносящиеся в исходный код программы не повлекли за собой так называемый эффект «стрельбы дробью», при котором изменения кода одного класса или метода могут повлечь за собой последствия по всей системе. При этом, изменения могут носить любой характер – это могут быть как изменения бизнес-правил (добавление роли пользователя, новый функционал, удаление или изменение существующего функционала), так и изменения в инфраструктуре (переход на другого вендора БД, добавление интеграции с внешними системами).

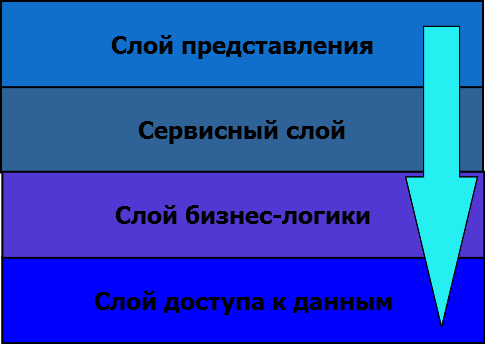
Одной из хороших практик достижения этих качеств является разбиение системы на слои:

* Слой доступа к данным;
* Слой бизнес-логики;
* Сервисный слой;
* Слой представления;

В такой архитектуре есть три важных ограничения, за счет которых достигается модульность системы:

1. Взаимодействие между слоями должно происходить «сверху вниз» от слоя представления к слою доступа к данным. При этом нижний слой не должен «знать» ничего о верхнем;
2. Взаимодействовать между собой могут только соседние слои. То есть, слой бизнес-логики взаимодействует со слоем доступа к данным, но не «видит» сервисного слоя и т.д;
3. Слой взаимодействует с нижележащим слоем через абстракции, не привязываясь к конкретной реализации. Соблюдение этого ограничение позволяет разработчику подменять реализации, либо использовать несколько для разных окружений (к примеру, приложение имеет один интерфейс для доступа к данным, реализацию которого можно написать как для реляционной БД, так и для NoSQL);

При соблюдении этих ограничений, разрабатываемая система получает еще одно немаловажное качество – тестируемость. Разработчику предоставляется возможность произвести модульное тестирование каждого слоя по отдельности, подменив слой и объекты, с которыми взаимодействует тестируемый слой, «заглушками», предоставляющими тестовые данные.



*Рисунок 2.4 – Многослойная архитектура системы*

Каждый слой в системе решает свои задачи:

* Слой доступа к данным обеспечивает взаимодействие с источником данных. В его задачи так же входит отображение полученных данных на т.н. объекты передачи данных (data transfer object), предназначенные для доставки данных слою бизнес-логики;
* Слой бизнес-логики содержит в себе объекты, отвечающие за бизнес-процесс, который автоматизирует данная система. На этом слое определяется основное поведение системы, связанное с заявками;
* Сервисный слой содержит функциональность, не связанную с бизнесом, но необходимую в системе (т.н. логика приложения), такую, как уведомление по электронной почте, интеграция с внешними системами и т.д;
* Слой представления отвечает за отображение данных на клиенте. В данном случае это web-страницы, но, при необходимости, остается возможность добавлять клиентов (например, мобильный клиент, консоль, или обычные экранные формы);

Такая архитектура системы лучше всего ложится на объектно-ориентированную парадигму программирования.

Объектно-ориентированная парадигма программирования в корпоративных приложениях предполагает разбиение предметной области на пакеты (пространства имен), которые, в свою очередь, разбиваются на классы, каждый из которых решает свою задачу. При этом, хорошей практикой разработки является соблюдение принципов SOLID. SOLID – это аббревиатура, каждая буква которой обозначает один принцип:

* S – Single responsibility (одиночная ответственность). Каждый класс должен решать только одну задачу;
* O – Open/closed (принцип открытости/закрытости). Программные сущности должны быть открыты для расширения, но закрыты для изменения;
* L – Liskov substitution (принцип Лисков). Любой объект в программе может быть подменен своим наследником без изменения свойств программы. Назван в честь польской программистки Барбары Лисков;
* I – Interface segregation (принцип разделения интерфейсов). Несколько специализированных интерфейсов лучше, чем один универсальный;
* D – Dependency Inversion (принцип инверсии зависимостей). Зависимости внутри системы строятся на основе абстракций. Модули верхнего уровня не зависят от модулей нижнего уровня. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракции;

Проектирование системы целесообразно проводить с соблюдением этих принципов.

Проектирование слоя бизнес-логики данной системы целесообразно начать с декомпозиции основного объекта системы, коим является заявка. Исходя из проанализированных данных call-центра предприятия, было выявлено, что в заявку вносятся:

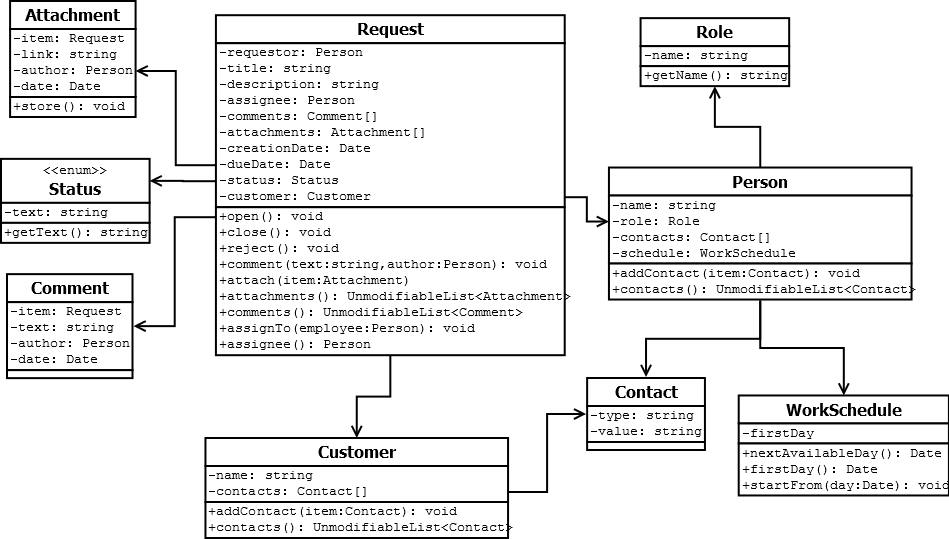
* Данные о диспетчере, принявшем заявку;
* Заголовок, идентифицирующий заявку;
* Краткое описание заявки;
* Данные о клиенте;
* Дата создания заявки;
* Дата крайнего срока исполнения заявки;
* Инженер, на которого назначена заявка;
* Статус заявки (открыта, отложена, не исполнена, отказана, закрыта);
* Комментарии диспетчера и/или инженера.

Из полей заявки целесообразно представить в виде классов: сотрудников предприятия, заказчиков, комментарии к заявкам и статусы. Статусы заявок здесь – перечислимый тип.

В ходе дальнейшей декомпозиции выявлены следующие типы объектов и построен прототип модели бизнес-слоя (рисунок 2.5):

* Роль сотрудника на предприятии (диспетчер, инженер, администратор);
* Рабочий график сотрудника;
* Контакты (адрес, телефон, e-mail и т.д.).

Здесь объект «Контакт» может относиться как к сотруднику, так и к заказчику.



*Рисунок 2.5 – Прототип модели бизнес-слоя*

# 2.5 Средства и методы разработки АИС

# 2.5.1 Выбор языка и платформы программирования

В связи с тем, что для разработки системы выбрана объектно-ориентированная парадигма программирования, целесообразно подбирать такие язык и платформу, которые максимально облегчают ООП подход и снижают затраты человеческих ресурсов на разработку. Также желательно снизить порог вхождения для новых разработчиков. На сегодняшний день самыми популярными, развивающимися и востребованными платформами являются: 1) Java и одноименный язык; 2) .NET и язык C# (си-шарп).

# Java

Java – объектно-ориентированный язык программирования, разработанный компанией Sun Microsystems (которая в дальнейшем была приобретена компанией Oracle) в 1995 году. Изначально язык назывался Oak и разрабатывался для программирования бытовых устройств. Впоследствии язык был переименован и стал использоваться для разработки клиентского и серверного программного обеспечения. Язык назван в честь знаменитой марки кофе, на эмблеме языка изображена кофейная кружка.

Программы на Java компилируются в байт-код, выполняемый виртуальной машиной Java (JVM – Java virtual machine) – программой, обрабатывающей байтовый код и передающей инструкции оборудованию как интерпретатор. Достоинством такого способа является полная независимость байт-кода от операционной системы и оборудования, что позволяет выполнять Java-приложения на любом устройстве, для которого существует соответствующая JVM. Другой важной особенностью платформы Java является гибкая система безопасности благодаря тому, что исполнение программы полностью контролируется виртуальной машиной. Любые операции, которые превышают установленные полномочия программы (например, попытка несанкционированного доступа к данным или подключения к другому компьютеру) вызывают немедленное прерывание.

Часто к недостаткам концепции виртуальной машины относят то, что исполнение байт-кода виртуальной машиной может снижать производительность программ и алгоритмов, реализованных на языке Java. В связи с этим, в платформу был внесен ряд усовершенствований, которые несколько увеличили скорость выполнения Java-приложений:

* Just-in-time compilation (JIT-компиляция) – технология трансляции байт-кода в машинный код непосредственно во время выполнения программы с возможностью поддержки версионности Java-классов;
* Широкое использование платформенно-ориентированного кода (native code) в стандартных библиотеках;
* Разработка аппаратных средств, обеспечивающих ускоренную обработку байт-кода (например, технология Jazelle, поддерживаемая некоторыми процессорами фирмы ARM).

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

# .NET

.NET Framework – программная платформа, выпущенная компанией Microsoft в 2002 году. Основой платформы является общеязыковая среда исполнения CLR (Common Language Runtime), которая подходит для разных языков программирования. Функциональные возможности CLR доступны в любых языках программирования, использующих эту среду.

Считается, что платформа .NET явилась ответом компании Microsoft на набравшую большую популярность платформу Java.

Основной идеей при разработке .NET Framework являлось обеспечение свободы разработчика за счет предоставления ему возможности создавать приложения различных типов, способные выполняться на различных устройствах и различных средах. Вторым принципом стала ориентация на системы, работающие под управлением операционных систем семейства Microsoft Windows.

Программа для .NET Framework, написанная на любом поддерживаемом языке программирования, сначала переводится компилятором в единый для .NET промежуточный байт-код Common Intermediate Language (CIL, ранее назывался Microsoft Intermediate Language - MSIL). В терминах .NET в итоге получается сборка (assembly). Затем код либо исполняется виртуальной машиной CLR, либо транслируется утилитой NGen.exe в исполняемый код для конкретного целевого процессора. Использование виртуальной машины предпочтительно, т.к. избавляет разработчиков от необходимости заботиться об особенностях аппаратной части. В случае использования виртуальной машины CLR, встроенный в нее JIT-компилятор «на лету» (just in time) преобразует промежуточный байт-код в машинные коды нужного процессора. Современная технология динамической компиляции позволяет достигнуть высокого уровня быстродействия. Виртуальная машина CLR также сама заботится о базовой безопасности, управлении памятью и системе исключений, избавляя разработчика от части работы.

Архитектура .NET описана и опубликована в спецификации Common Language Infrastructure (CLI), разработанной Microsoft.

Реализация платформы .NET Framework также вызывает некоторые нарекания:

* Менее рациональное использование ресурсов системы, по сравнению с другими технологиями;
* Управляемый CIL-байт-код, не подвергшийся обфускации, легко может быть декомпилирован;
* Поддержка стандартов, фактически ограничивающих кросс-платформенную реализацию .NET Framework, в результате чего полноценная реализация платформы присутствует лишь для Windows.

# Сравнение .NET и Java

Целесообразно, помимо особенностей платформ Java и .NET провести сравнение входящих в них языков программирования – Java и C# (C# - не единственный язык в платформе .NET, но для сравнения выбран именно он, т.к. по синтаксису и семантике больше других схож с языком программирования Java).

Языки Java и C# появились в разное время – Java был создан задолго до появления C#. Таким образом, если Java создавался, опираясь, в большей степени, на опыт языков C и Objective-C, то для C# такой опорой являлись C++ и сам Java. И, несмотря на своё название, C# оказался ближе к Java, чем к C++ (символ «#» в названии языка был взят из нотной грамоты, и означает «диез» - повышение ноты на полтона (C# в ABC-записи аккордов можно прочесть как «до-диез»), что аналогично инкременту «++» (увеличению на 1) в программировании).

С точки зрения разработчика языки Java и C# очень похожи. Оба языка являются строго типизированными, объектно-ориентированными. Оба вобрали в себя многое из синтаксиса C++, но, в отличие от C++, проще в освоении для начинающих. Оба языка используют *сборку мусора*. Оба используют фигурные скобки для выделения блоков. Оба языка сопровождаются огромным количеством библиотек. Но, в языках также есть свои особенности и различия. В C# учлись многие недостатки Java и исправились в реализации языка. Но и Java – постоянно развивающийся параллельно с C# язык.

Многие специалисты утверждают, что C# более сложный язык, чем Java. Так, Ким Рэдек – разработчик из Microsoft – сказал: «Язык Java построен таким образом, чтобы не дать разработчику возможности выстрелить себе в ногу, а C# был построен так, чтобы дать разработчику ружье, но оставить его на предохранителе».

В качестве синтаксической основы оба языка используют язык C. В частности, без изменений от него унаследованы:

* Обозначение начала и конца блока кода фигурными скобками;
* Обозначение, ассоциативность и приоритет большинства встроенных операций (присвоение, арифметические, логические, побитовые операции, инкремент/декремент, тернарная условная операция);
* Синтаксис описания и использования переменных и функций (порядок «тип имя», использование модификаторов, обязательность скобок для функций, описание формальных параметров);
* Синтаксис всех основных конструкций: условного оператора, циклов, операций множественного выбора.

Синтаксических различий в языках также достаточно (таблица 2.1).

Таблица 2.1 – Синтаксические различия языков Java и C#

| Синтаксис | Java | C# |
| --- | --- | --- |
| Импорт статических имен (import static) | Позволяет отдельно импортировать некоторые или все статические методы и переменные класса и использовать их имена без квалификации в импортирующем модуле. | Импортируется только сборка и при каждом использовании импортируемых статических имен требуется указывать класс. |
| Оператор switch | Аргумент оператора switch должен относиться либо к целочисленному, либо к перечислимому типу. Начиная с версии Java 7 в операторе switch стало возможно использование строковых литералов. | Поддерживаются как константные, так и строковые типы. В отличие от Java, прямого перехода к следующему блоку case нет, для этого нужно использовать оператор goto. |
| Оператор goto | От использования goto сознательно отказались, но существует механизм, позволяющий выйти на внешний цикл из вложенного, пометив его меткой, и используя операторы break, continue вместе с меткой. | Goto сохранился, его обычное использование – передача управления на разные метки case в операторе switch и выход из вложенного цикла. |

Продолжение таблицы 2.1

|  |  |  |
| --- | --- | --- |
| Синтаксис | Java | C# |
| Константы | Слово const в языке зарезервировано, но констант, как таковых, нет, вместо них используются статические поля, помеченные модификатором final. | Существует отдельное понятие именованной типизированной константы и ключевое слово const. |
| Точность вычислений с плавающей точкой | Содержит конструкцию strictfp, гарантирующую одинаковые результаты операций с плавающей точкой на всех платформах. Однако, в настоящее время эта функция не используется, но возможно скомпилировать программу со включенным strictfp. | Полагается на реализацию, гарантии строго одинаковых результатов вычислений нет. |
| Отключение проверок | Все динамические проверки включаются и выключаются только на уровне пакета. | Содержит конструкции checked и unchecked, позволяющие локально включать и выключать проверку арифметического переполнения. |

Оба языка реализуют одну модель работы с динамическими данными: объекты создаются динамически с помощью конструкции *new*, среда исполнения отслеживает наличие ссылок на них, а garbage collector (сборщик мусора) периодически очищает память от объектов, ссылок на которые нет. Для оптимизации сборки мусора спецификации языков и сред исполнения не содержат ограничений на время жизни объекта после удаления последней ссылки на него, сборщик работает независимо от исполнения программы, поэтому реальное уничтожение объекта может произойти в любой момент после удаления последней ссылки до завершения работы программы. В реальности сборщики мусора оптимизируют исполнение так, чтобы обеспечить приемлемый расход памяти при минимальном замедлении работы программ.

И в Java и в C# есть сильные и слабые ссылки на объекты. Оба языка поддерживают методы-финализаторы. Из-за неопределенности момента удаления объекта финализаторы не могут использоваться для освобождения системных ресурсов, занятых объектом, что вынуждает создавать дополнительные методы для «очистки» объекта и вызывать их явно.

C# содержит в стандартной библиотеке интерфейс *IDisposable*, и специальную конструкцию *using*, гарантирующую своевременный вызов метода очистки. В Java подобной конструкции не было до версии 7, в которую добавлена конструкция *try-with-resources*, обеспечивающая очистку полностью аналогично C#. Обязательное условие – классы объектов для очистки должны реализовывать интерфейс java.lang.AutoClosable.

Java позволяет зарегистрировать слушателя (listener), который будет получать сообщения, когда ссылка подвергается сборке мусора, что дает улучшение производительности.

Среда CLR позволяет отменить выполнение финализатора для данного объекта. Это бывает полезно, поскольку финализация считается относительно дорогой операцией при сборке мусора, и объект с финализатором существует дольше.

И Java и C# – объектно-ориентированные языки с синтаксисом, унаследованным от C++, но значительно переработанным. Код и данные могут быть описаны только внутри классов. Некоторые различия есть и в подходе к инкапсуляции.

В Java модификатор *protected*, помимо доступа из классов-потомков, разрешает доступ из всех классов, входящих в тот же пакет, что и класс-владелец.

В C# для объектов, которые должны быть видны только в пределах сборки (примерный аналог пакета в Java), введен отдельный модификатор *internal*, а *protected* сохраняет свой изначальный смысл, взятый из C++ – доступ только из классов-потомков. Допускается комбинация *internal* и *protected* – тогда получается область доступа, аналогичная *protected* в Java.

Оба языка позволяют определить класс внутри класса.

В Java внутренние классы используются для эмуляции замыканий. Внутренние классы Java имеют доступ к нестатическим членам родительского класса. Кроме того, внутри методов можно определять локальные классы, имеющие доступ по чтению к локальным переменным, и анонимные (безымянные) локальные классы, которые, фактически позволяют создавать объекты из интерфейсов, перекрывающие методы своего класса, непосредственно в месте их использования. На этом механизме в Java может использоваться обработка событий (событие генерирует вызов метода, являющегося в исходном классе-обработчике абстрактным; там, где нужен конкретный обработчик события, программист создает экземпляр локального анонимного класса – наследника класса обработчика – и непосредственно использует его). Таким образом, исчезает необходимость в специальном типе и синтаксической поддержке для событий, но сам код, создающий обработчики, несколько более сложен для понимания. В частности, сложнее становятся области видимости переменных.

В C# есть замыкания и лямбды (которые так же появились в Java, начиная с версии 8, вышедшей в марте 2014 г.). Подход C# более напоминает C++: внутренние классы имеют доступ только к статическим членам внешнего класса, а для доступа к нестатическим членам нужно явно указывать экземпляр внешнего класса. Локальные внутренние классы в C# не поддерживаются.

В обоих языках методы определяются через функции класса. Тело метода располагается внутри описания класса. Поддерживаются статические методы, абстрактные методы. В C# также есть явная реализация методов интерфейса, что позволяет классу реализовывать их отдельно от собственных или давать разные реализации одноименных методов, принадлежащих двум разным интерфейсам.

В Java примитивные типы (*int*, *long*, *double*, *boolean* и пр.) передаются по значению, а для остальных по значению передается ссылка на объект.

В C# по значению, в дополнение к примитивам, передаются структуры (*struct*), остальные типы передаются по ссылке. C# также поддерживает явное указание передачи параметров по ссылке (ключевые слова *ref* и *out*). При использовании *out* компилятор контролирует наличие в методе присваивания значения. Использование их стоит только при работе с неуправляемым кодом, который этого требует (например, WinAPI), т.к. это нарушает концепцию объектно-ориентированного программирования.

C# копирует концепцию виртуальных методов C++: виртуальный метод должен быть явно объявлен с ключевым словом *virtual*, прочие методы виртуальными не являются. Такое выборочное определение виртуальных методов введено потому, что объявление всех методов виртуальными может сильно замедлить исполнение. Кроме того, C# требует явного объявления о перекрытии виртуального метода в произвольном классе ключевым словом *override*. Если требуется скрыть виртуальный метод, то есть просто ввести новый метод с такой же сигнатурой, требуется указать ключевое слово *new*, в случае отсутствия которого компилятор выдает предупреждение. Запрещается перекрывать абстрактные методы. Объявление *override*-метода с ключевым словом *sealed* запрещает переопределять этот метод в классах-потомках, однако по прежнему позволяет скрыть его.

В Java, наоборот, все открытые методы, кроме статических, являются виртуальными, а переопределить метод так, чтобы механизм виртуальности не включился, невозможно. Метод всегда виртуально перекрывает метод базового класса с той же сигнатурой, если он есть. Ключевое слово *final* позволяет запретить создание метода с такой же сигнатурой в классах-потомках.

Подход Java синтаксически проще, он гарантирует, что всегда вызовется метод того класса, к которому обращается объект. С другой стороны, виртуальность, действительно, нужна не всегда, а накладные расходы на вызов виртуальных методов несколько больше.

Виртуальность всех методов потенциально небезопасна – если программист по ошибке объявит метод, который уже есть в базовом классе, не имея намерения его перекрывать, то новый метод перекроет метод базового класса, хотя это и не входит в планы разработчика. В C# такая ошибка тоже возможна, но компилятор выдаст предупреждение, что перекрывающий метод объявлен без *new* и *override*. В Java 5 появился аналогичный механизм – если метод перекрывает метод класса-предка, компилятор выдает предупреждение, которого можно избежать, пометив перекрывающий метод аннотацией *@Override*.

Оба языка поддерживают идею примитивных типов (в C# они называются *value type* – значимые типы) и оба, для трансляции примитивных типов в объектные, обеспечивают их автоматическую упаковку (*auto boxing*) и распаковку. В C# есть возможность обращения к значимым типам как к объектам. В Java разделяют примитивы и объекты, а для обращения к примитивным типам как к объектам используются классы-обертки, например, *Integer* для типа *int*.

Примитивных типов в C# больше, чем в Java, за счет наличия беззнаковых (*unsigned*) целых типов, имеющихся парно ко всем знаковым и специального типа *decimal* для высокоточных вычислений с фиксированной точкой (в Java для этого служат классы BigInteger и BigDecimal).

В Java отказались от большинства беззнаковых типов для упрощения языка. Одна из известных проблем с такими типами – сложность определения типа результата арифметических операций над двумя аргументами, один из которых является знаковым, а другой беззнаковым. Однако, этот отказ порождает свои проблемы. Поскольку значительная часть технических данных, используемых на низком уровне имеет именно беззнаковый тип, и отсутствие таких типов приводит к необходимости использования небезопасной конвертации, в ходе которой возможна потеря данных.

В синтаксисе обоих языков также получили выражение массивы и коллекции. В обоих языках массив является объектом класса Array, но в Java он не реализует какие-либо интерфейсы коллекций, хотя по массивам возможна итерация циклом *for-each*. Оба языка имеют в стандартной библиотеке классы типичных коллекций.

В Java могут быть объявлены только одномерные массивы. Многомерный массив в Java – массив массивов, которые в C# обычно называются неровными, или ступенчатыми. В двумерной терминологии многомерные массивы всегда «прямоугольны», тогда как массивы массивов могут хранить строки разной длины. Многомерные массивы ускоряют доступ к памяти, а неровные массивы работают медленнее, но экономят память, когда заполнены не все строки. Многомерные массивы требуют для своего создания только один вызов оператора *new*, а ступенчатые требуют явно выделять память для каждой строки.

В обоих языках типы могут быть параметризированными, что поддерживает парадигму обобщенного программирования (*generic*). Синтаксически определение таких типов в обоих языках достаточно близко – оно унаследовано от шаблонов (*templates*) C++, однако с некоторыми модификациями.

Обобщения типов в Java являются чисто языковой конструкцией и реализованы лишь в компиляторе. Компилятор заменяет все обобщенные типы на их верхние границы и вставляет соответствующее приведение типов в те места, где используется параметризируемый тип. В результате получается байт-код, который не содержит ссылок на обобщенные типы и их параметры. Такая техника реализации обобщенных типов называется затиранием типов. Это означает, что информация об исходных обобщенных типах во время выполнения недоступна, и обусловливает некоторые ограничения, такие как невозможность создавать новые экземпляры массивов из аргументов обобщенного типа. Среда выполнения Java (JRE - Java Runtime Environment) не знакома с системой обобщенных типов, вследствие чего новым реализациям JVM понадобились лишь минимальные обновления для работы с новыми форматами классов.

C# пошел другим путем. Поддержка обобщенности была интегрирована в саму CLR. Язык здесь стал лишь внешним интерфейсом для доступа к этим возможностям среды. Как и в Java, компилятор производит статическую проверку типов, но, в дополнение к этому, JIT производит проверку корректности во время загрузки. Информация об обобщенных типах полностью присутствует во время выполнения и позволяет полную поддержку рефлексии обобщенных типов и создание их новых реализаций.

Подход Java требует дополнительных проверок во время выполнения, не гарантирует, что клиент кода будет следовать соответствию типов и не обеспечивает рефлексии для обобщенных типов. Java не позволяет специализировать обобщенные типы примитивными (это можно сделать лишь используя классы-обертки), в то время как C# обеспечивает обобщение как для ссылочных, так и для значимых типов. Как в Java, так и в C# специализации обобщенного типа на разных ссылочных типах дают одинаковый код, но для C# CLR динамически генерирует оптимизированный код при специализации на типах-значениях, что позволяет хранить и извлекать их без операции оборачивания/разворачивания.

Внешние модули в Java и C# подключаются схожим образом. В Java используется ключевое слово *import*, в C# - *using*. Существенным отличием здесь является то, что C# использует концепцию пространств имен (namespace), напоминающую одноименный механизм в C++. В Java используется концепция пакетов (*package*). Пространства имен никак не связаны с компилированными модулями (сборками). Несколько сборок могут содержать одно и то же пространство имен, в одной сборке может объявляться несколько пространств имен, не обязательно вложенных. Модификаторы области видимости C# никак не связаны с пространствами имен. В Java же объявленные в одном пакете классы по умолчанию образуют единый компилированный модуль. Модификатор области видимости по умолчанию (без явного указания модификатора) ограничивает область видимости пределами пакета.

В Java структура файлов и каталогов исходных текстов связана со структурой пакетов – пакету соответствует каталог, входящим в него пакетам – подкаталоги этого каталога. Таким образом, дерево исходных текстов повторяет структуру пакета. В C# местонахождение файла с исходным текстом никак не связано с пространством имён.

Ни один из вариантов подключения внешних модулей не обладает значительным превосходством в мощности, просто используются разные механизмы для разрешения неоднозначностей.

В C# классы могут располагаться в файлах произвольным образом. Имя файла исходного кода никак не связано с именами определяемых в нем классов. Допускается расположить в одном файле несколько общедоступных (*public*) классов. Начиная с версии 2.0, C# позволяет разбить класс на несколько файлов, используя ключевое слово *partial*. Эта особенность сделана для разделения кода, написанного программистом, и кода, сгенерированного автоматически. Она используется, например, визуальными средствами построения интерфейса: часть класса, в которой находятся поля и методы, управляемые конструктором интерфейса, выделяются в отдельный файл.

В Java каждый файл может содержать только один общедоступный класс, причем Java требует, чтобы имя этого класса совпадало с именем файла, что исключает путаницу в именах классов и файлов. Более того, согласно рекомендуемому Oracle соглашению об оформлении кода, размер файла исходного текста не должен превышать 2000 строк кода, поскольку в файлах большого размера труднее разбираться. Большой размер файла также считается признаком плохого проектирования.

С момента появления C#, язык постоянно сравнивается с Java. Невозможно отрицать, что сам язык и CLR во многом обязаны Java и JRE. Но, тем не менее, в настоящее время становится всё более очевидным, что два языка редко конкурируют друг с другом на рынке. Java доминирует в мобильном секторе и имеет много приверженцев на рынке web-приложений. C# получил хорошее признание на рынке настольных приложений Windows и, благодаря ASP.NET, также является игроком на рынке web-приложений. Оба языка широко используются в сфере корпоративной (enterprise) разработки.

# Язык и платформа разрабатываемой системы

Руководством предприятия было предъявлено два основных требования к разрабатываемой системе:

1. Клиенты системы должны работать с web-приложением, т.к. основные корпоративные системы предприятия также являются web-приложениями;
2. Система не должна привязываться к операционной системе сервера, т.к. на предприятии установлены сервера как под управлением Windows Server, так и под управлением Debian Linux.

Учитывая эти требования и результаты сравнения платформ и языков программирования, подходящих под задачу, было принято решение использовать для разработки системы «RFS» платформу Java последней, на текущий момент, версии (8) и одноименный язык программирования.

# 2.5.2 Выбор средств хранения данных

В дата-цетре предприятия установлено и настроено несколько различных систем управления базами данных. На выбор предоставлены следующие варианты:

* Oracle Database;
* MySQL;
* Microsoft SQL Server.

Также, к базе данных разрабатываемой системы были предъявлены следующие требования:

* Поддержка транзакций;
* Поддержка триггеров и хранимых процедур;
* Возможность размещения бизнес-логики, связанной с ежемесячными отчетами непосредственно в базе данных.

Необходимо проанализировать доступные для использования СУБД, учитывая предъявленные требования.

# Oracle Database

Oracle Database – объектно-реляционная система управления базами данных, разработанная и поддерживаемая компанией Oracle.

Одной из основных характеристик СУБД Oracle является ее функционирование на большинстве существующих платформ, в том числе на больших ЭВМ, UNIX-серверах, персональных компьютерах и т.д.

Другой важной характеристикой Oracle является поддержка всех возможных вариантов архитектур, в том числе симметричных многопроцессорных систем, кластеров и т.д. Очевидна значимость этих характеристик для крупномасштабных и динамично развивающихся предприятий.

Начиная с версии Oracle 8i, в состав сервера СУБД включена JVM.

О механизме транзакций в Oracle можно сказать, что он полностью соответствует стандарту SQL-92. Также, поддерживаются транзакции «только на чтение» (*read only*) и автономные транзакции.

Если транзакция объявлена как *read only*, то гарантируется, что любой запрос на выборку, выполненный в рамках этой транзакции, выдаст данные в том состоянии, в каком они были до начала транзакции. В пределах такой транзакции запрещено любое изменение данных. При этом транзакция «только на чтение» не устанавливает никаких блокировок данных и не мешает изменять эти данные в других сессиях.

Реализованный в Oracle механизм автономных транзакций позволяет создать новую транзакцию в пределах текущей. Подтверждение или откат изменений, сделанных в рамках автономной транзакции, производится независимо от родительской транзакции. Автономные транзакции могут быть вложенными, максимальный уровень вложенности определяется настройками сервера.

В Oracle Database осуществляется поддержка триггеров и хранимых процедур. Также, в продукт встроен язык PL/SQL (Procedural Language / Structured Query Language) – процедурное расширение языка SQL. Базируется на языке программирования Ada.

PL/SQL дает возможность использовать переменные, операторы, массивы, курсоры и исключения. Начиная с версии Oracle 8i доступна и объектно-ориентированная модель.

Стандартный SQL является специализированным декларативным языком программирования. На язык наложены определенные ограничения, такие как, например, отсутствие прямой поддержки циклов. PL/SQL же, как Тьюринг-полный язык, позволяет разработчикам обрабатывать данные в реляционной базе, используя императивный стиль программирования. Операторы SQL могут быть легко вызваны непосредственно из PL/SQL-процедуры, функции или триггера (иногда с некоторыми ограничениями).

Процедуры и функции, написанные в Oracle Database, можно объединять в пакеты. Также присутствует возможность писать и компилировать Java-код непосредственно в базе данных. Таким образом, в Oracle Database можно размещать некоторые элементы бизнес-логики, требующие быстрого доступа к данным.

# MySQL

MySQL – система управления базами данных, разрабатываемая и поддерживаемая компанией Oracle, получившая права на торговую марку вместе с поглощенной Sun Microsystems.

MySQL является решением для малых и средних приложений. Гибкость MySQL обеспечивается поддержкой большого количества типов таблиц: пользователи могут выбрать как таблицы MyISAM, поддерживающие полнотекстовый поиск, так и таблицы InnoDB, поддерживающие транзакции на уровне отдельных записей. Более того, СУБД поставляется со специальным типом таблиц EXAMPLE, демонстрирующим принципы создания новых типов таблиц. Таким образом, новые типы появляются постоянно, благодаря открытой архитектуре и лицензированию MySQL.

Поддержка транзакций в MySQL осуществляется не всеми типами таблиц. Так, например, используемые большинством разработчиков таблицы MyISAM не поддерживают транзакции, в отличие, например, от InnoDB. То есть, для полной поддержки транзакции базой данных, сервер MySQL должен быть соответствующе сконфигурирован, что не всегда возможно.

По умолчанию MySQL работает в режиме *autocommit* (автоподтверждение), т.е. результаты любого оператора, изменяющего данные, будут автоматически фиксироваться в базе. Этот режим можно отключить, тогда каждую транзакцию нужно будет завершать явно, подтверждением или откатом.

В поддержке транзакций MySQL есть серьезная проблема – если в транзакции участвуют таблицы разных типов (транзакционные и нетранзакционные), то поведение операторов на изменение для этих таблиц будет различным: изменения в таблицы, не поддерживающие транзакции, будут вноситься немедленно, тогда как в поддерживающие – после явного завершения транзакции. Это грозит тем, что после отката транзакции, данные в базе станут неконсистентыми.

MySQL с версии 5.0 поддерживает триггеры и хранимые процедуры.

# Microsoft SQL Server

Microsoft SQL Server – система управления реляционными базами данных, разрабатываемая и поддерживаемая компанией Microsoft.

Основной используемый в СУБД язык запросов – Transact-SQL (T-SQL) – является расширением языка SQL. Microsoft SQL Server используется от персональных баз данных до крупных баз данных масштаба предприятия, и конкурирует с другими СУБД в этом сегменте.

Работа транзакций в Microsoft SQL Server обеспечивается с помощью журнала транзакций базы данных. Так как журнал записывает всю активность и происходящие изменения, СУБД может восстанавливать данные автоматически в момент потери питания, системной ошибки или запроса отмены.

Microsoft SQL Server гарантирует, что все завершенные транзакции отражаются в базе данных в момент и после системной ошибки.

С помощью скрытых транзакций сервер гарантирует, что если оператор, изменяющий данные, выполнен удачно, то данные будут сохранены в таблице, а если во время изменения произошла ошибка, данные откатятся.

Пользователь может установить точку сохранения или маркер внутри транзакции. Точка сохранения определяет положение, до которого транзакция может быть возвращена. То есть, по достижению этой точки, транзакция либо продолжит работу, либо будет откат.

Также, СУБД поддерживает вложенные транзакции.

На языке T-SQL можно разрабатывать процедуры и функции и объединять их в логические структуры. СУБД также поддерживает триггеры.

# Инструмент хранения данных в разрабатываемой системе

Фактически, учитывая результаты анализа предложенных вариантов систем управления базами данных и предъявленные руководством предприятия требования, выбор производился между двумя СУБД – Oracle Database и Microsoft SQL Server. Здесь выбор был сделан в пользу Oracle, т.к., приложение пишется на платформе Java, а Oracle Database хорошо поддерживает эту платформу и позволяет размещать в базе данных бизнес-логику, написанную на Java.

# 2.5.3 Сервер приложений

В связи с тем, что разрабатываемая система является web-приложением, ее необходимо развернуть на сервере, принимающем HTTP-запросы от клиента. Здесь выбора предоставлено не было, т.к. в дата-центре предприятия предустановлен сервер приложений Oracle WebLogic.

WebLogic поддерживает следующие стандарты и технологии:

* HTTP;
* J2EE;
* Веб-сервисы;
* JMS;
* JDBC;
* EJB;
* Servlet.

HTTP – Hyper Text Transfer Protocol – протокол прикладного уровня передачи данных. Основой HTTP является технология «клиент-сервер».

J2EE – Java 2 Enterprise Edition – набор спецификаций и соответствующей документации для языка Java, описывающий архитектуру серверной платформы для задач средних и крупных предприятий.

Веб-сервисы – идентифицируемые веб-адресом программные системы со стандартизированными интерфейсами. Веб-сервисы могут взаимодействовать друг с другом и со сторонними приложениями.

JMS – Java Message Service – стандарт промежуточного ПО для рассылки сообщений, позволяющий приложениям, выполненным на платформе J2EE создавать, посылать, получать и читать сообщения.

JDBC – Java Database Connectivity – платформо-независимый промышленный стандарт взаимодействия Java-приложений с различными системами управления базами данных, основанный на концепции так называемых драйверов, позволяющих получать соединение с базой данных по специально написанному URL.

EJB – Enterprise Java Beans – спецификация технологии написания и поддержки серверных компонентов, содержащих бизнес-логику. Входят в состав J2EE.

Servlet – Java-интерфейс, реализация которого расширяет функциональные возможности сервера. Взаимодействует с клиентами посредством принципа «запрос-ответ».

Помимо Oracle WebLogic, было принято решение использовать для тестирования приложения в окружении разработки web-сервер Apache Tomcat, содержащий в себе сервлет-контейнер.

# 2.5.4 Инструменты и методы разработки

Помимо выбора языка, платформы и базы данных, для разработки системы используются такие инструменты, как:

* Система управления проектом Apache Maven;
* Фреймворк уровня приложения Spring Framework;
* Фреймворк для доступа к данным Spring-JPA;
* ORM-фреймворк Hibernate;
* Фреймворк для слоя презентации Spring MVC;
* Фреймворк для разграничения прав Spring Security;
* Фреймворк для генерации JSON из Java-классов Jackson;
* API javax.mail для посылки уведомлений по электронной почте;
* Фреймворк для модульного тестирования JUnit;
* Фреймворк для подмены реализаций Mockito;
* Система контроля версий GitHub;
* Инструмент для организации окружения разработки Vagrant.

# Apache Maven

Apache Maven – фреймворк для автоматизации сборки проектов, специфированных на XML-языке POM (Project Object Model – проектно-объектная модель).

Maven обеспечивает декларативную сборку проекта. Все задачи инструмент выполняет через плагины.

Информация для программного проекта содержится в XML-файле с именем *pom.xml*. При исполнении Maven в первую очередь проверяет, содержит ли файл все необходимые данные и синтаксическую правильность составления. Листинг файла pom.xml для системы «RFS» можно найти в Приложении А.

Конфигурация включает имя проекта, его собственника и зависимости от других проектов. Возможно также конфигурировать индивидуальные фазы процесса построения проекта, реализованные плагинами.

Apache Maven требует от всех проектов одинаковой структуры каталогов, включающую в себя исходный код, модульные тесты для исходного кода и ресурсные файлы.

# Spring Framework

Spring Framework – универсальный фреймворк с открытым исходным кодом для Java-платформы. Он обеспечивает решения многих задач, с которыми сталкиваются Java-разработчики и организации, которые хотят создать информационную систему на языке Java. Фреймворк предлагает последовательную модель и делает ее применимой к большинству типов приложений, которые уже созданы на платформе Java. Считается, что Spring Framework реализует модель разработки, основанную на лучших стандартах индустрии, и делает ее доступной во многих областях Java.

Центральной частью Spring Framework является контейнер, который предоставляет средства конфигурирования и управления объектами Java с помощью механизма рефлексии. Контейнер отвечает за управление жизненным циклом объекта: создание объектов, вызов методов инициализации и конфигурирование объектов путем связывания их между собой.

Объекты, создаваемые контейнером, также называются управляемыми объектами, или бинами (*beans*). Обычно конфигурирование контейнера осуществляется путем загрузки XML-файлов, содержащих определения бинов и предоставляющих информацию, необходимую для их создания.

Объекты могут быть получены следующими способами:

* Поиск зависимости – шаблон проектирования, в котором вызывающий объект запрашивает у объекта-контейнера экземпляр объекта с определенным именем или определенного типа;
* Внедрение зависимости – шаблон проектирования, в котором контейнер передает экземпляры объектов по их имени другим объектам с помощью конструктора, свойства или фабричного метода.

# Spring-JPA

Java Persistence API – интерфейс прикладного программирования, входящий в состав платформы Java и предоставляющий возможность сохранять в удобном виде объекты Java в базе данных. Spring-JPA реализует этот интерфейс и позволяет создавать на его основе репозитории объектов, по умолчанию содержащие CRUD-методы и позволяющие дополнять интерфейсы специализированными методами поиска.

Работа Spring-JPA основывается на шаблоне проектирования «Репозиторий», описанного Мартином Фаулером. Репозиторий, фактически, является посредником между уровнями определения и хранения данных, работая как обычная коллекция объектов области определения. Объекты-клиенты создают описание запроса и направляют их репозиторию для обработки.

Таким образом, репозиторий инкапсулирует объекты, представленные в хранилище данных и операции, производимые над ними, предоставляя объектное представление реальных данных.

# Hibernate

Объектно-реляционное отображение – технология программирования, которая связывает базы данных с концепциями объектно-ориентированных языков программирования, создавая виртуальную объектную базу данных. Другими словами, объектно-реляционное отображение – представление реляционных данных как объектов реального мира и наоборот.

Hibernate – библиотека для языка программирования Java с открытым исходным кодом, предназначенная для решения задач объектно-реляционного отображения. Библиотека представляет собой простой в использовании фреймворк.

Целью Hibernate является освобождение разработчика от значительного объема сравнительно низкоуровнего программирования по обеспечению хранения объектов в базе данных. Разработчик может использовать Hibernate как в процессе проектирования системы таблиц и классов с нуля, так и для работы с уже существующей базой данных.

Hibernate предоставляет средства для автоматической генерации и обновления набора таблиц из Java-кода, поиска и обновления данных, что позволяет значительно уменьшить время, которое обычно тратится на ручное написание SQL и JDBC кода.

Важной возможность Hibernate является также обеспечение независимости приложения от конкретной СУБД. Таким образом облегчается возможность перехода с одной системы управления базами данных на другую без изменения исходного кода.

# Spring MVC

Шаблон проектирования «Модель-представление-контроллер» обеспечивает разделение модели данных, пользовательского интерфейса и механизма взаимодействия с пользователем на отдельные компоненты.

Основная цель применения такого подхода состоит в отделении бизнес-логики приложения от ее визуализации. За счет такого разделения повышается возможность повторного использования кода. Наиболее полезно применение данной концепции в тех случаях, когда пользователь должен видеть одни и те же данные в разных контекстах, или с разных устройств, и, изменение реакции на действия пользователя при таком подходе не влекут за собой изменение пользовательского интерфейса.

Фреймворк Spring MVC решает эту задачу с использованием контейнера, аналогичного Spring Framework, в котором в качестве бинов описываются Java-классы, являющиеся контроллерами взаимодействия с пользователем и компоненты, разрешающие представление в виде web-страниц.

# Spring Security

Spring Security – фреймворк, предоставляющий механизмы построения систем аутентификации и авторизации, а также другие возможности обеспечения безопасности для промышленных приложений, созданных с помощью Sprinf Framework.

Данный фреймворк конфигурируется с помощью XML-контейнера, в котором возможно определение схемы авторизации, перехват адресов, разграничение прав. Spring Security также предоставляет наборы тэгов для разграничения доступа к элементам web-страниц.

# JUnit

JUnit – библиотека для модульного тестирования программного обеспечения на языке Java. Предназначен для написания модульных тестов – классов, автоматически тестирующих функциональность программных модулей.

# Mockito

Mockito – библиотека для создания mock-объектов – фиктивных реализаций существующих в системе интерфейсов, предназначенных только для модульного тестирования. С помощью таких объектов-заглушек можно подменить, к примеру, реальный репозиторий фиктивным, чтобы протестировать функционал работы с данными локально.

# GitHub

Контроль версий – подход к хранению изменяющейся информации, при котором в любой момент можно получить любую версию этой информации. Системы контроля версий помогают хранить большое количество версий одних и тех же данных и, при необходимости, откатываться до более ранних версий, а также, определять, когда и кем были внесены те или иные изменения. Также системы контроля версий позволяют хранить несколько ветвей одной и той же информации – так, для одной и той же версии исходного кода в разных ветвях, скомпилированная система может иметь разное поведение. Использование систем контроля версий позволяет избежать потери важных данных.

GitHub – бесплатный онлайн-сервис для работы с версиями. Для каждого проекта в сервисе предоставляются системы отслеживания ошибок, редактируемые страницы описания (wiki) и возможность просмотра исходного кода онлайн с подсветкой синтаксиса для большинства языков. Помимо этого, пользователям предоставляется консольная утилита для работы с репозиторием исходного кода.

Поддержка GitHub встроена в большинство современных интегрированных сред разработки программного обеспечения (IDE).

***Vagrant***

Vagrant – инструмент для быстрого разворачивания окружения для разработки. Он работает с виртуальной машиной, в которую в фоновом режиме загружает заранее подготовленный образ операционной системы с необходимыми для разработки настройками и предустановленным программным обеспечением, будь то СУБД, сервер приложений и т.д.

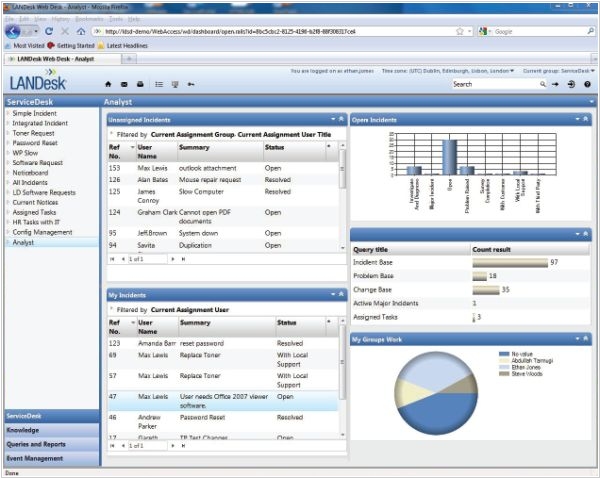
# 2.6 Обзор аналогов системы

Существует множество систем приема и обработки заявок на обслуживание. Среди них, под развернутые на предприятии окружение подошли следующие:

* LANDesk Service Desk;
* Luxoft Service Desk.

# 2.6.1 LANDesk Service Desk

LANDesk Service Desk – коммерческий программный комплекс с закрытым исходным кодом, предназначенный для автоматизации работы службы технической поддержки.



*Рисунок 2.6 – LANDesk Service Desk*

Первоначально продукт разрабатывался компанией TouchPaper Software, которая специализировалась на создании систем для работы технической поддержки. Назывался продукт HelpDesk. Он позволял регистрировать заявки на обслуживание ИТ-инфраструктуры и отслеживать сроки их исполнения. В дальнейшем продукт претерпел значительные изменения.

LANDesk Service Desk реализован на платформе .NET Framework и построен по архитектуре клиент-сервер. Серверные модули функционируют под управлением систем семейства Microsoft Windows Server. В качестве системы управления базами данных поддерживается Microsoft SQL Server, либо Oracle Database. Продукт поддерживает разные типы консолей – для доступа через браузер, через карманный компьютер и т.д.

Данный продукт имеет возможность интеграции с другими системами, базами данных, серверами электронной почты и экспорт во внешние форматы.

Для разграничения прав в LANDesk ServiceDesk может использоваться домен Microsoft Active Directory.

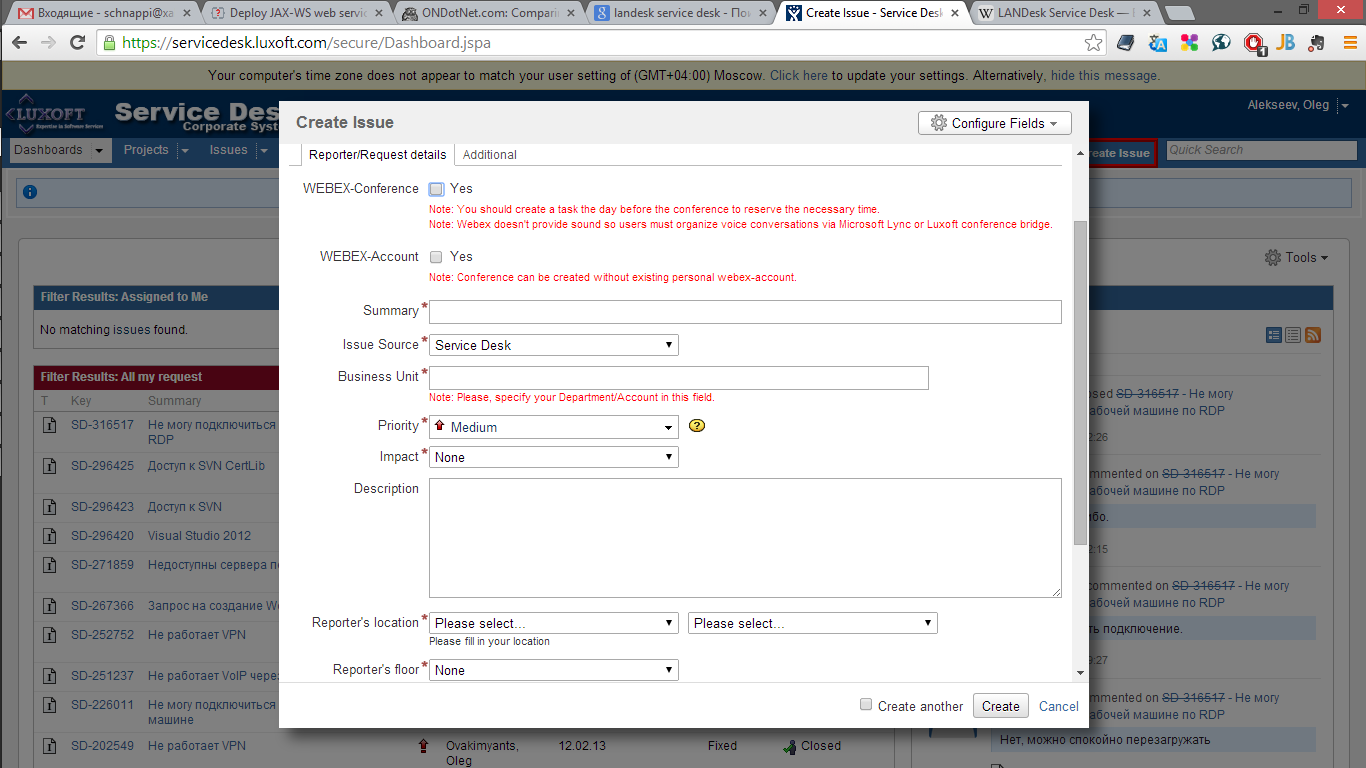
# 2.6.2 Luxoft Service Desk

Luxoft Service Desk (рисунок 2.7) – внутрикорпоративная система приема и обработки заявок на сервисное обслуживание, разработанная российской компанией Luxoft.

Данная система имеет следующие возможности:

* Создание заявки на обслуживание с полным описанием проблемы;
* Назначение инженера на заявку;
* Комментирование заявки;
* Просмотр истории заявок;
* Создание запроса на организацию телефонной конференции;
* Система уведомлений по электронной почте;
* Разграничение прав.

Разграничение прав в системе Luxoft Service Desk осуществляется с помощью Microsoft Active Directory.



*Рисунок 2.7 – Luxoft Service Desk*

# 2.6.3 Сравнительная таблица продуктов

Для наглядности сравнения описанных систем с разрабатываемой системой «RFS» была составлена сравнительная таблица (таблица 2.2), с учетом требований руководства предприятия.

Таблица 2.2 – сравнение систем обработки заявок

|  |  |  |  |
| --- | --- | --- | --- |
| Параметр | LANDesk Service Desk | Luxoft Service Desk | RFS |
| Экранная форма ввода заявки | Есть | Есть | Есть |
| Автоподбор свободных инженеров | Есть | Нет | Есть |
| Ввод рабочего графика инженеров | Есть | Нет | Есть |
| Уведомление по электронной почте | Есть | Есть | Есть |
| Разграничение доступа | Есть | Есть | Есть |

Продолжение таблицы 2.2

|  |  |  |  |
| --- | --- | --- | --- |
| Параметр | LANDesk Service Desk | Luxoft Service Desk | RFS |
| Поддержка Active Directory | Есть | Есть | Нет |
| Управление учетными записями сотрудников | Есть | Нет | Есть |
| Отчеты о заявках | Есть | Есть | Есть |
| Интеграция с HR-системой | Есть | Есть | Есть |
| Открытый исходный код | Нет | Нет | Есть |
| Коробочная версия продукта | Есть | Нет | Планируется |
| Кроссплатформенность | Нет | Нет | Есть |
| Платформа разработки | Microsoft .NET | Microsoft .NET | Oracle Java |

Из таблицы видно, что все три системы схожи по объему функционала. Сравнительная таблица была предоставлена для изучения руководству предприятия, и, ключевым фактором выбора программного продукта явилась независимость от платформы.

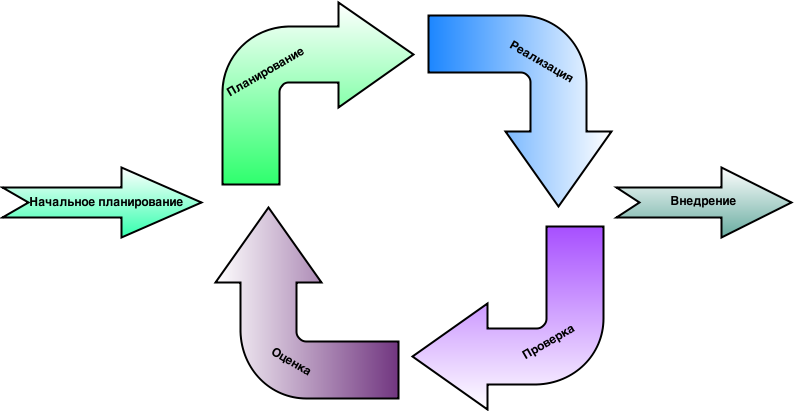
# 2.7 Разработка системы «RFS»

# 2.7.1 Методология разработки

Система «RFS» разрабатывается с использованием гибкой методологии разработки Agile. Agile – это серия подходов к разработке программного обеспечения, ориентированных на использование итеративной разработки, динамическое формирование требований и обеспечение их реализации в результате постоянного взаимодействия внутри самоорганизующихся рабочих групп, состоящих из специалистов различного профиля. Существует несколько методик, относящихся к классу гибких методологий разработки, в частности:

* Разработка, управляемая функциональностью (FDD – Feature Driven Development);
* Экстремальное программирование (XP – eXtreme Programming);
* Метод разработки динамических систем (DSDM – Dynamic Systems Development Method;
* Схватка (Scrum).

Все перечисленные методики основаны на итеративной разработке - выполнению работ параллельно с непрерывным анализом полученных результатов и корректировкой предыдущих этапов работы. При этом подходе каждая фаза развития проекта представляет собой повторяющийся цикл (рисунок 2.8): планирование – реализация – проверка – оценка (*plan-do-check-act*).



*Рисунок 2.8 – Итеративная разработка*

Главными преимуществами итеративной разработки являются:

* Снижение воздействия серьезных рисков на ранних стадиях проекта, что ведет к минимизации стоимости их устранения;
* Организация эффективной обратной связи разработчика с заказчиком и создание продукта, реально отвечающего его потребностям;
* Акцент усилий на наиболее важные и критичные направления проекта;
* Непрерывное итеративное тестирование, позволяющее оценить успешность всего проекта в целом;
* Раннее обнаружение конфликтов между требованиями, моделями и реализацией проекта;
* Эффективное использование накопленного опыта;
* Реальная оценка текущего состояния проекта;
* Затраты распределяются по всему проекту, а не группируются в его конце.

Для разработки системы «RFS» применялась методология итеративной разработки Scrum – процесс разработки строится на небольших, но строго фиксированных по времени итерациях, называемых, в рамках методологии, спринтами, по окончанию каждой из которых конечному пользователю предоставляется рабочий программный продукт с новыми возможностями, для которых определен наивысший приоритет. Считается, что чем короче спринт, тем выше гибкость проекта. По истечению каждого спринта проводится совещание с заказчиком, на котором подводятся итоги окончившейся итерации и планируется следующая.

Для разработки системы «RFS» был определен спринт в две недели. По завершению реализации был составлен список спринтов, каждая запись которого названа, исходя из основного реализованного в нем функционала. Помимо этого, было отведено четыре недели на анализ проекта, выбор инструментов и проектирование доменной модели. В итоге, помимо подготовительного этапа, называемого, в рамках методологии, резервом проекта, разработка разделилась на следующие спринты:

* Разработка справочника «Сотрудники»;
* Разработка функционала рабочих графиков сотрудников;
* Добавление формы подачи заявки;
* Добавление формы подбора свободных инженеров в заявку;
* Разработка сервиса e-mail уведомлений;
* Добавление возможности комментирования заявки;
* Разработка отчетов по заявкам;
* Разграничение прав доступа между инженерами, диспетчерами и администраторами;
* Добавление интеграции с системой HR-службы.

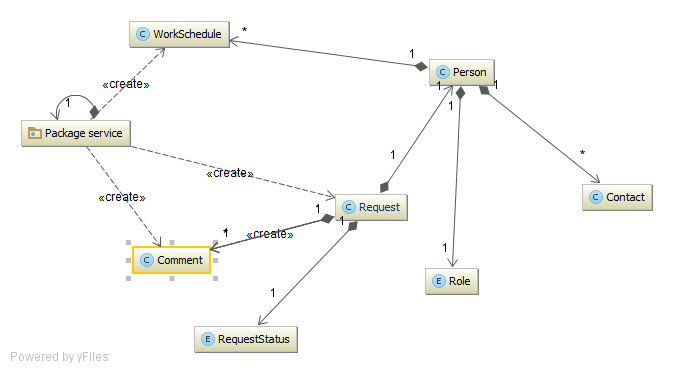
Функционал системы разделен между спринтами таким образом, чтобы после каждой итерации разработки диспетчеры могли начать работу с системой: вносить сведения о сотрудниках, их рабочем расписании, начать прием заявок и т.д.

# 2.7.2 Проектирование и разработка доменной модели

Доменная модель системы – набор классов, которые содержат основную бизнес-логику системы, ее основное состояние и поведение. За основу будущей доменной модели был взят прототип, описанный в разделе 2.4. По итогам встречи с руководителем предприятия, было решено отложить разработку дополнительных сущностей «Заказчик» (Customer) и «Вложение» (Attachment), по причине отсутствия острой необходимости. В итоге, были разработаны следующие классы:

* Request – основная бизнес-сущность модели. В этом классе содержится состояние и поведение объектов типа «Заявка», такое как номер, описание, адрес назначения, ссылка на назначенного инженера, желаемая дата, крайний срок, методы закрытия, отказа заявки;
* RequestStatus – перечислимый тип (*enum*), отвечающий за статусы заявок;
* Comment – класс, инкапсулирующий комментарии к заявкам;
* Person – сущность, инкапсулирующая в себе сведения о сотрудниках и ссылки на их рабочие графики;
* Role – перечислимый тип, отвечающий за роли пользователей системы;
* WorkSchedule – класс, инкапсулирующий состояние и поведение объекта «Рабочий график сотрудника». Содержит именованные списки рабочих дней и методы для выявления рабочего или выходного времени сотрудников;
* Contact – класс для хранения и обработки контактных данных сотрудников.

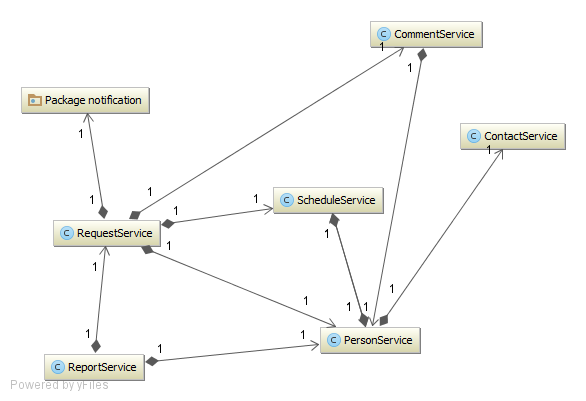
Для наглядности приведена диаграмма зависимостей между классами (рисунок 2.9), сгенерированная интегрированной средой разработки IntelliJ IDEA.



*Рисунок 2.9 – Зависимости в доменной модели*

Из приведенной диаграммы видно, что на доменные классы ссылается пакет *service*. Этот пакет содержит логику для связи доменного слоя со слоем хранения данных, в том числе обращения к так называемым DTO-объектам (Data Transfer Object – объект передачи данных) – объектам, осуществляющих доставку данных из классов-репозиториев в слой бизнес-логики. Классы, входящие в пакет *service* называются классами-сервисами и осуществляют конвертацию DTO-объектов в доменные объекты.

Таким образом, каждому доменному объекту соответствует сервисный класс. Сервисы так же могут быть связаны друг с другом, если, например, для формирования одной сущности нужно получить данные другой. Зависимости между сервисными классами показаны на рисунке 2.10.



*Рисунок 2.10 – Зависимости в пакете service*

Внутри себя *service* содержит пакет *notification*, предназначенный для обеспечения работы службы уведомлений. Данный пакет содержит в себе:

* Notifier – интерфейс, в котором определен единственный метод – *sendNotification()*. Классы, отвечающие за отправку уведомлений должны реализовывать этот интерфейс. Такой подход используется для того, чтобы была возможность разработки нескольких видов нотификации (SMS, E-Mail, и т.д.) без указания явного типа сервиса;
* EmailNotifier – реализация интерфейса Notifier, предназначенная для отправки уведомлений по электронной почте.

Таким образом, пакет *service* отвечает за доставку данных из слоя доступа к данным слою бизнес-логики и за отправку уведомлений пользователям. Помимо этого, пакет *service* является «интерфейсом» для внешних клиентов, коими в системе являются сервисный слой и слой представления.

# 2.7.3 Проектирование и разработка слоя доступа к данным

Слой доступа к данным должен быть спроектирован таким образом, чтобы:

* Минимизировать затраты доставки данных в слой бизнес-логики;
* Минимизировать время проектирования модели данных на уровне источника данных.
* Минимизировать затраты переноса бизнес-слоя на различные источники данных.

Задачу минимизации затрат доставки данных в бизнес-слой в системе решает шаблон проектирования «репозиторий», реализованный в фреймворке Spring-JPA. Данный фреймворк содержит интерфейсы репозиториев со стандартизированными CRUD-методами.

Каждый интерфейс репозитория параметризирован двумя типами: типом хранимой сущности, и типом первичного ключа этой сущности. Соответственно, методы доступа к данным репозиториев возвращают коллекции этих сущностей, либо отдельные их экземпляры.

Помимо стандартных CRUD-методов, можно дополнять интерфейсы специфическими для сущности методами поиска, например, поиск заявки по ключу назначенного инженера, поиск комментариев по ключу заявки и т.д. При этом, программисту нет необходимости писать реализации этих методов – к фреймворку можно подключить ORM-библиотеку, в которой написаны стандартные реализации. Пример интерфейса репозитория отображен в листинге 2.1.

1 **public** **interface** ScheduleRepository **extends** JpaRepository<ScheduleDTO, Integer> {

2

3 List<ScheduleDTO> getByEmployee\_Id(**int** id);

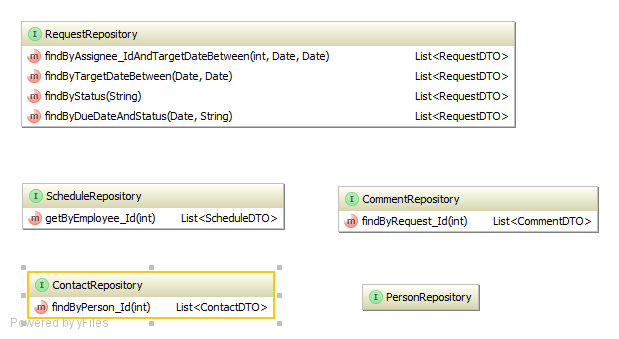
4

5 }

*Листинг 2.1 – Пример репозитория*

Из листинга видно, что интерфейс наследует JpaRepository – интерфейс Spring-JPA, описывающий CRUD-методы создания, чтения, обновления и удаления данных из базы. Также видно, что описан специализированный метод поиска сотрудника по ключу.

Интерфейсы-репозитории системы сосредоточены в пакете *ds.repo* и не имеют ссылок друг на друга (рисунок 2.11) – зона ответственности каждого из них ограничивается одной сущностью, указанной в параметре интерфейса.



*Рисунок 2.11 – Пакет ds.repo*

Вторую важную задачу проектирования слоя данных – минимизацию низкоуровневого проектирования на уровне системы управления базами данных – решает ORM-фреймворк Hibernate.

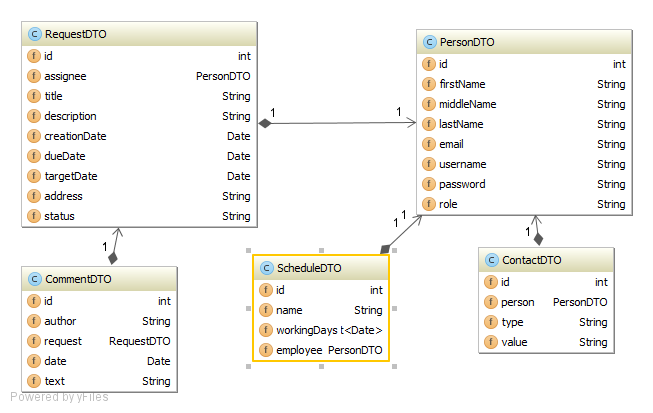
Для проектирования модели данных используется технология Code-First. Она заключается в том, что в первую очередь на языке программирования пишутся классы, содержащие поля данных и методы доступа к ним, а фреймворк своими силами генерирует из них DDL (Data Definition Language – язык доступа к данным), основываясь на подключенном драйвере базы данных, и автоматически создает в базе данных таблицы, последовательности и триггеры.

Используя такой подход, необходимо проектировать классы сущностей модели данных таким образом, чтобы максимально облегчить их отображение на доменные объекты для того, чтобы использовать эти классы как DTO. В связи с этим, целесообразно принять следующие соглашения кодирования:

* Именовать класс как имя сущности, добавляя постфикс DTO (например, RequestDTO);
* Именовать поля класса таким же образом, как они названы в доменном объекте. Так, например, полю *assignee* в классе Request соответствует поле *assignee* в классе RequestDTO;
* Использовать в качестве связи сущностей поле внешнего ключа не числового, а ссылочного типа, например, класс RequestDTO для связи с классом PersonDTO использует поле типа PersonDTO с аннотацией *@ManyToOne*, а не поле типа Integer.

С учетом этих соглашений был разработан набор классов, составляющий пакет *ds.dto* (рисунок 2.12). Связи внутри пакета формируют отношения в источнике данных.

Использование такого подхода к организации слоя данных освобождает разработчиков от проектирования на уровне источника данных и ручного создания схемы данных, которое, как правило, занимает достаточно много времени, вследствие чего растет стоимость проекта. Code-First технология позволяет программиста абстрагироваться от конкретной системы управления базами данных и, также облегчает миграцию системы в другие окружения. Таким образом, подход Code-First дает системе важное качество – переносимость – с минимальными затратами.



*Рисунок 2.12 – Пакет ds.dto*

Описание объектов передачи данных сведено в таблицу 2.3.

Таблица 2.3 – Описание DTO

|  |  |  |  |
| --- | --- | --- | --- |
| Класс | Поля | Типы данных в Java | Типы данных в Oracle |
| RequestDTO | id | int | NUMBER (PK) |
| assignee | PersonDTO | NUMBER (FK) |
| title | String | VARCHAR2 |
| description | String | VARCHAR2 |
| creationDate | Date | DATE |
| targetDate | Date | DATE |
| dueDate | Date | DATE |
| address | String | VARCHAR2 |
| status | String | VARCHAR2 |
| CommentDTO | id | int | NUMBER (PK) |
| author | String | VARCHAR2 |
| request | RequestDTO | NUMBER (FK) |
| date | Date | DATE |
| text | String | VARCHAR2 |

Продолжение таблицы 2.3

|  |  |  |  |
| --- | --- | --- | --- |
| Класс | Поля | Типы данных в Java | Типы данных в Oracle |
| PersonDTO | id | int | NUMBER (PK) |
| firstName | String | VARCHAR2 |
| middleName | String | VARCHAR2 |
| lastName | String | VARCHAR2 |
| email | String | VARCHAR2 |
| username | String | VARCHAR2 |
| password | String | VARCHAR2 |
| role | String | VARCHAR2 |
| ScheduleDTO | id | int | NUMBER (PK) |
| name | String | VARCHAR2 |
| workingDays | List<Date> | Доп. таблица |
| employee | PersonDTO | NUMBER (FK) |
| ContactDTO | id | int | NUMBER (PK) |
| person | PersonDTO | NUMBER (FK) |
| type | String | VARCHAR2 |
| value | String | VARCHAR2 |

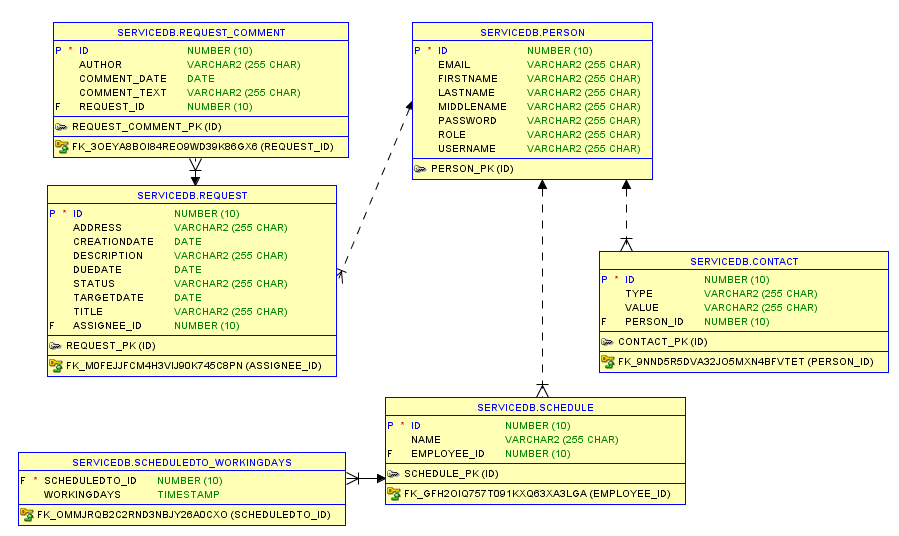
Из таблицы видно, что для поля *workingDays* класса ScheduleDTO сгенерировалась дополнительная таблица (таблица 2.4), т.к. поле содержит в себе список дат.

Таблица 2.4 – Дополнительная таблица БД для списка дат.

|  |  |
| --- | --- |
| Имя поля | Тип данных |
| SCHEDULEDTO\_ID | NUMBER (PK) (FK) |
| WORKINGDAYS | TIMESTAMP |

Сгенерированная структура базы данных свободна для расширения и изменения. То есть, при необходимости добавления новой сущности или нового параметра для уже существующей сущности, потери данных не произойдет. А при переносе приложения на новую платформу, структура данных создастся с нуля.

Диаграмма «сущность-связь» для сгенерированной в Oracle Database структуры данных изображена на рисунке 2.13.

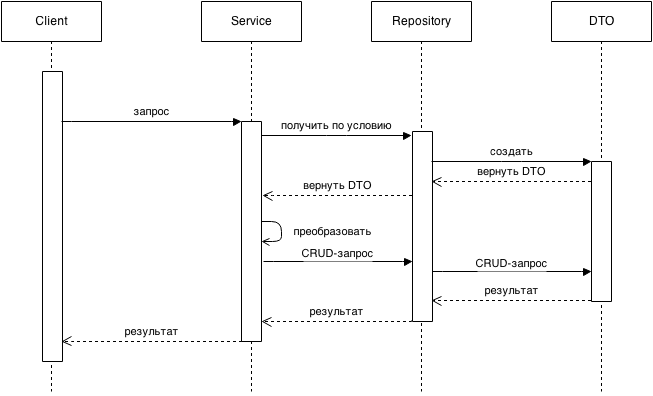


*Рисунок 2.12 – ER-диаграмма сгенерированной БД.*

# 2.7.4 Механизм конвертации DTO-объекта в бизнес-сущность

Преобразование объекта передачи данных в доменный объект производится классом-сервисом в пакете *domain.service*, получая DTO-объекты из репозитория по запросу клиента (слоя презентации).

Клиент посылает сервисному классу запрос на какое-либо доступное действие, содержащий условие, по которому необходимо получить объекты, либо произвести с ними действие. Сервис обрабатывает полученные условия и посылает сформированный из них запрос на получение объекта репозиторию. Репозиторий в ответ возвращает DTO-объект. Сервис выполняет преобразование, присваивая значения полей DTO-объекта доменному объекту, при необходимости, конвертируя, затем проверяет соответствие объекта условиям запроса и посылает репозиторию финальный запрос, выполняющий нужные клиенту действия. Эти действия производятся над DTO-объектом, их результат возвращается сервису. Сервис возвращает результат клиенту. Этот процесс можно записать в нотации диаграммы последовательности UML (рисунок 2.13).



*Рисунок 2.13 – Преобразование DTO в доменный объект*

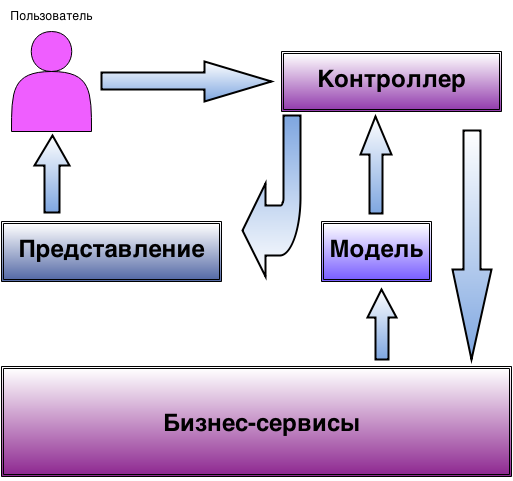
Таким образом, с помощью сервиса и репозитория поддерживается полное соответствие данных, хранящихся в базе с данными, отображаемыми пользователю. Исходный код, содержащий преобразования в классе RequestService находится в приложении Б.

# 2.7.5 Проектирование и разработка слоя представления

Главная задача слоя представления – обеспечить взаимодействие конечного пользователя системы с ее функционалом. При этом, между слоем представления и нижележащими слоями не должно быть «сцепленности» – он должен содержать в себе только логику отображения и вызов публичных методов сервисного или доменного слоёв. Помимо этого, на слое представления должна быть возможность замены одного интерфейса на другой или добавления нового интерфейса (например, мобильной версии или командной строки). Другими словами, слой представления должен служить фасадом системы по отношению к пользователю.

Это достигается при помощи шаблона проектирования «Модель-Представление-Контроллер» (MVC – Model-View-Controller). Идея шаблона состоит в том, чтобы пользовательский интерфейс только получал данные, не производя над ними никаких действий, кроме, возможно, простейших преобразований (например, строку в дату), но и для таких преобразований должны вызываться внешние утилитные классы.

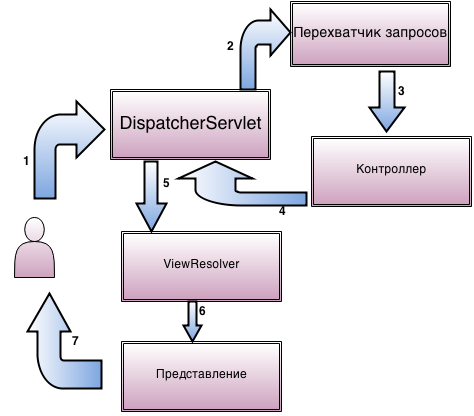
Шаблон работает следующим образом (рисунок 2.14): пользователь посылает запросы контроллеру. Контроллер обращается в соответствующий запросу бизнес-сервис и получает из него модель, содержащую данные для отображения. Затем контроллер передает ссылку на полученную модель представлению, которое уже отображает информацию пользователю.



*Рисунок 2.14 – Схема работы MVC*

При этом важно, чтобы ни контроллер, ни представление не ссылались на доменные объекты, о них должна знать только модель, а контроллер должен обращаться только к бизнес-сервисам. Это нужно для того, чтобы максимально отвязать бизнес-сущности от слоя представления и, тем самым, предотвратить нежелательные изменения представления в случае изменений доменных объектов, т.к. изменить в такой ситуации только модель, которая, по сути, имеет лишь ссылку на доменный объект, а не изменять полностью слой представления.

Слой представления сам по себе не принимает HTTP-запросы и не посылает ответы. Это зона ответственности системного объекта – сервлета. В Spring MVC за это отвечает класс DispatcherServlet. Он принимает запрос от web-браузера и обращается к другому системному объекту – перехватчику запросов. Перехватчик разбирает запрос и посылает его контроллеру, который его обрабатывает и возвращает сервлету модель и имя представления. Имя представления передается специальному системному объекту ViewResolver, сконфигурированному в XML-контейнере. ViewResolver по имени представления находит файл, в котором оно описано и возвращает в DispatcherServlet, который отображает конечную страницу пользователю (рисунок 2.15).



*Рисунок 2.15 – Схема работы Spring MVC*