МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение высшего образования
«Тольяттинский государственный университет»

Институт **математики, физики и информационных технологий**

Кафедра **«Прикладная математика и информатика»**

09.04.03 ПРИКЛАДНАЯ ИНФОРМАТИКА

ПРИКЛАДНАЯ ИНФОРМАТИКА В ОБРАЗОВАНИИ И ОБРАЗОВАТЕЛЬНЫХ ТЕХНОЛОГИЯХ

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

на тему «**МЕТОДЫ И ПРОГРАММНЫЕ СРЕДСТВА РЕАЛИЗАЦИИ КОМПАРАТОРА БАЗ ДАННЫХ В ИНФОРМАЦИОННЫХ СИСТЕМАХ»**

Студент А.В. Давыдов _____

Научный руководитель Е.А. Ерофеева _____

Руководитель программы д.пед.н., профессор, А.Н. Ярыгин _____

«_____»_____20_____г.

Допустить к защите

Заведующий кафедрой к.тех.н., доцент, А.В. Очеповский _____

«_____»_____20_____г.

Тольятти 2016

Содержание

Введение

В наш век информации на крупном предприятии не обойтись без информационной системы (сокр. ИС), которая бы обрабатывала полученную информацию.

Информационная система — система обработки информации и соответствующие организационные ресурсы (человеческие, технические, финансовые и т. д.), которые обеспечивают и распространяют информацию[4][5].

Информация может быть любых видов, будь то информация о сотрудниках, их зарплате, клиентах, товарах, абонентской плате, адресах и т.д. Чем сложнее обрабатываемый информационный процесс, тем более сложная информационная система требуется. И чем больше скапливается разнородной информации, тем более сложная требуется база данных для её хранения. В процессе сопровождения и дальнейшей разработки таких информационных систем всегда используется как минимум три экземпляра информационной системы:

- Клиентский сервер – рабочая версия ИС, используемая клиентом. Содержит рабочую информацию и именно при работе с ней клиент может встретить ошибки и сообщить о них разработчикам.
- Тестовый сервер – тестовая среда, на которой клиент может ознакомиться будущими нововведениями и исправлениями.
- Рабочий сервер – место, где происходит активная разработка приложения и первичное тестирование. К нему у клиента нет доступа. Также здесь в БД хранятся тестовые данные, нужные разработчикам.

Теперь рассмотрим ситуацию: клиент сообщает, что на таком-то аккаунте при определённых условиях ИС работает неправильно. Разработчики пытаются воспроизвести эту проблему на своём сервере со своими данными, однако всё работает корректно. В этом случае разработчику необходимо выяснить, чем данные клиента отличаются от наших, для того, чтобы воспроизвести ошибку. Для этого ему необходимо сравнить данных в базе данных клиента со своей базой данных.

База данных (сокр. БД) — представленная в объективной форме совокупность самостоятельных материалов (статей, расчётов, нормативных актов, судебных решений и иных подобных материалов), систематизированных таким образом, чтобы эти материалы могли быть найдены и обработаны с помощью электронной вычислительной машины[1][20].

Если БД состоит всего из нескольких десятков таблиц, это не займёт много времени. Но если ИС оперирует с сотнями, или даже тысячами таблиц, то поиск различия в данных может занять несколько дней или даже недель. От этого не выигрывает ни клиент, т.к. ему приходится платить за каждый день этой работы, ни разработчик, т.к. монотонное сравнение таблиц не самое приятное занятие.

Актуальность: в описанных выше обстоятельствах на помощь могло бы прийти приложение, которое бы взяло всю рутинную работу по сравнению данных на себя. Оно бы сократило расходы клиента и нервы разработчика.

Гипотеза: создание компаратора баз данных на основе исследования логической связи хранимых данных между собой приведёт к упрощению и ускорению работы по сопровождению крупных информационных систем.

Объект исследования: процесс сравнения данных в разных базах данных по определённому критерию. Или, говоря профессиональным языком, компарация баз данных по критерию.

Предмет исследования: возможность компарации двух баз данных по критерию программными средствами.

Цель работы: исследовать способы компарации баз данных и реализовать наиболее подходящий способ.

Задачи исследования:

1. Проанализировать литературу, которая может помочь в решении задачи, а также существующие компараторы баз данных, их функционал и область применения.
2. Исследовать компарируемую базу данных для установления связи между таблицами в БД, а также для исключения из процесса компа-

рации таблиц, чьё содержимое не меняется на протяжении длительного времени (например, таблицы, содержащие расшифровку кодов ошибок).

3. Разработать приложение, которое бы получало данные из всех нужных нам таблиц по определённому критерию, а также компаратор, который бы сравнивал полученные данные из разных БД между собой, и выводил разработчику подробных отчёт.

1 Анализ существующих аналогов

1.1 Область применения компараторов

Компарирование (фр. comparer - сравнивать, сличать), сравнение мер или измеряемой величины с величиной эталона. Компарирование производят при помощи приборов сравнения (компараторов): фотометрической скамьи с фотометром, электроизмерительного потенциометра, равноплечных весов, компараторов для линейных мер и т.п. Компараторы являются неотъемлемой частью большинства поверочных схем [3][36].

Компараторы обычно используются в технике [13]. Цифровой компаратор или компаратор кодов - логическое устройство с двумя входами, на которые подаются два разных двоичных слова равной в битах длины и тремя двоичными выходами, на которые выдаётся признак сравнения входных слов, — первое слово больше второго, меньше или слова равны [15].

Компараторы широко используются в вычислительной технике, измерительной технике, радио- и проводной связи, бытовых приборах. Например, цифровые часы с будильником содержат цифровой компаратор, при совпадении текущего времени с заданным подаётся звуковой сигнал [11].

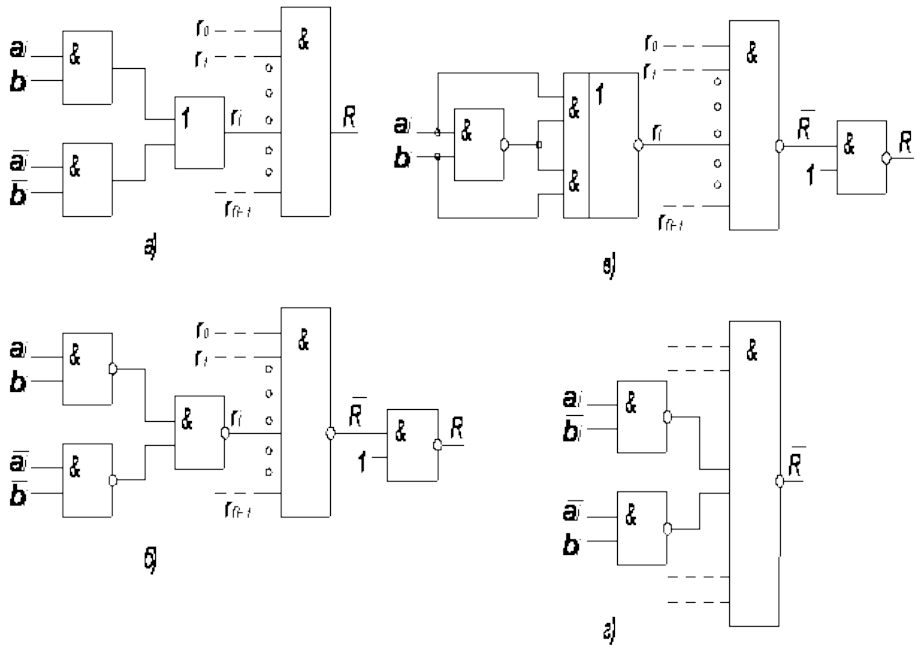

Рисунок 1.1 – Варианты схем цифровых компараторов

Аналоговым эквивалентом цифрового компаратора является компаратор напряжений или токов [25]. Некоторые микроконтроллеры имеют входные встроенные аналоговые компараторы, состояние выходов которых может быть считано программой контроллера или вызывать её прерывание подпрограммой [23].

Компаратор (аналоговых сигналов) (англ. comparator — сравнивающее устройство) — электронная схема, принимающая на свои входы два аналоговых сигнала и выдающая логическую «1», если сигнал на прямом входе («+») больше, чем на инверсном входе («−»), и логический «0», если сигнал на прямом входе меньше, чем на инверсном входе [22].

Одно напряжение сравнения двоичного компаратора делит весь диапазон входных напряжений на два поддиапазона. Двоичный логический сигнал (бит) на выходе двоичного компаратора указывает, в каком из двух поддиапазонов находится входное напряжение [2].

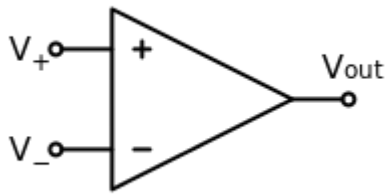

Рисунок 1.2 – Аналоговый компаратор

Простейший компаратор представляет собой дифференциальный усилитель [7].

В нашем случае, компарации будет подвергнуты данные в базах данных. В общем случае, процесс компарации выглядит следующим образом:

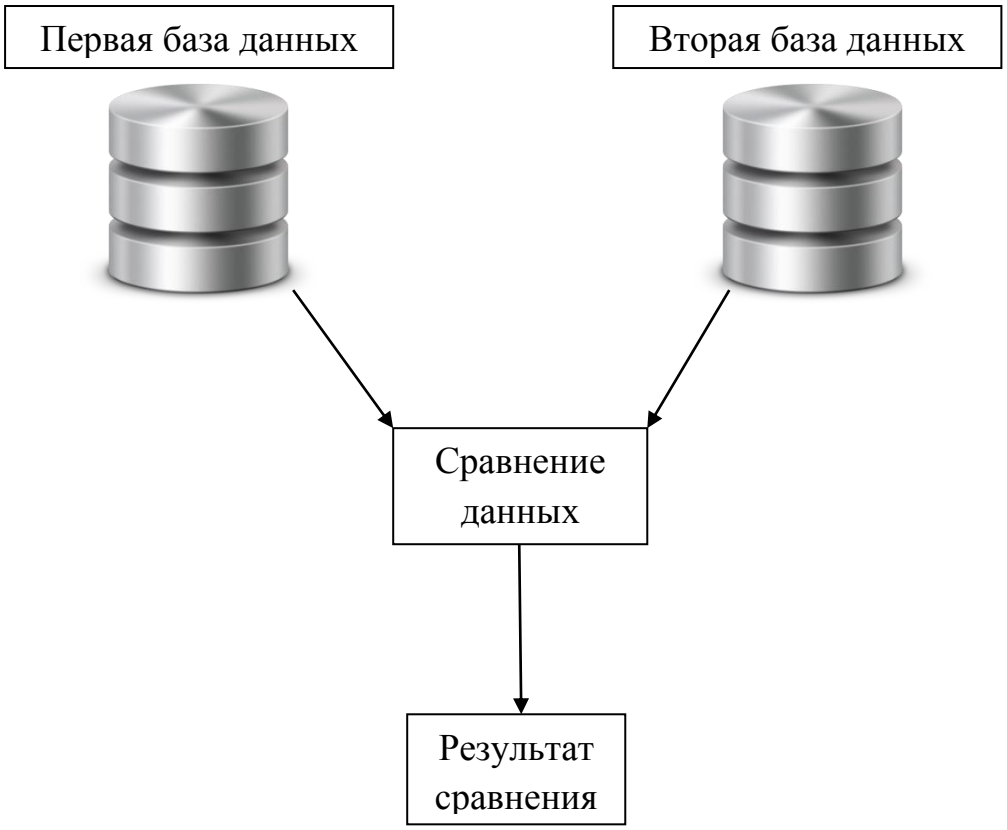

Рисунок 1.3 – Общая схема компарации баз данных

Данные из двух баз данных сравнивается между собой, и результат сравнения предоставляется пользователю для анализа. По этой теме уже существует несколько решений, о них пойдёт речь ниже.

1.2 Анализ существующих аналогов компараторов баз данных

Не смотря на распространённость различных компараторов баз данных, научная литература на русском языке на эту тему отсутствует. Поэтому ниже будут рассмотрены существующие решения:

dbForge Data Compare for SQL Server - это мощный, быстрый и простой в эксплуатации инструмент сравнения и синхронизации БД SQL, способный использовать собственные резервные копии SQL Server в качестве источника метаданных.

С его помощью можно:

- Сравнивать данные в БД SQL Server и выявлять внесённые в них изменения.
- Сравнивать "рабочие" БД с резервными копиями SQL Server.
- Анализировать различия между двумя БД.
- Синхронизировать две рассинхронизированные БД.
- Восстанавливать данные конкретной таблицы из резервной копии.
- Создавать отчёты о сравнении данных в формате Excel и HTML.
- Копировать данные поиска из разрабатываемой БД в финальную.
- Автоматизировать повседневные задачи по синхронизации данных с интерфейсом командной строки [50].

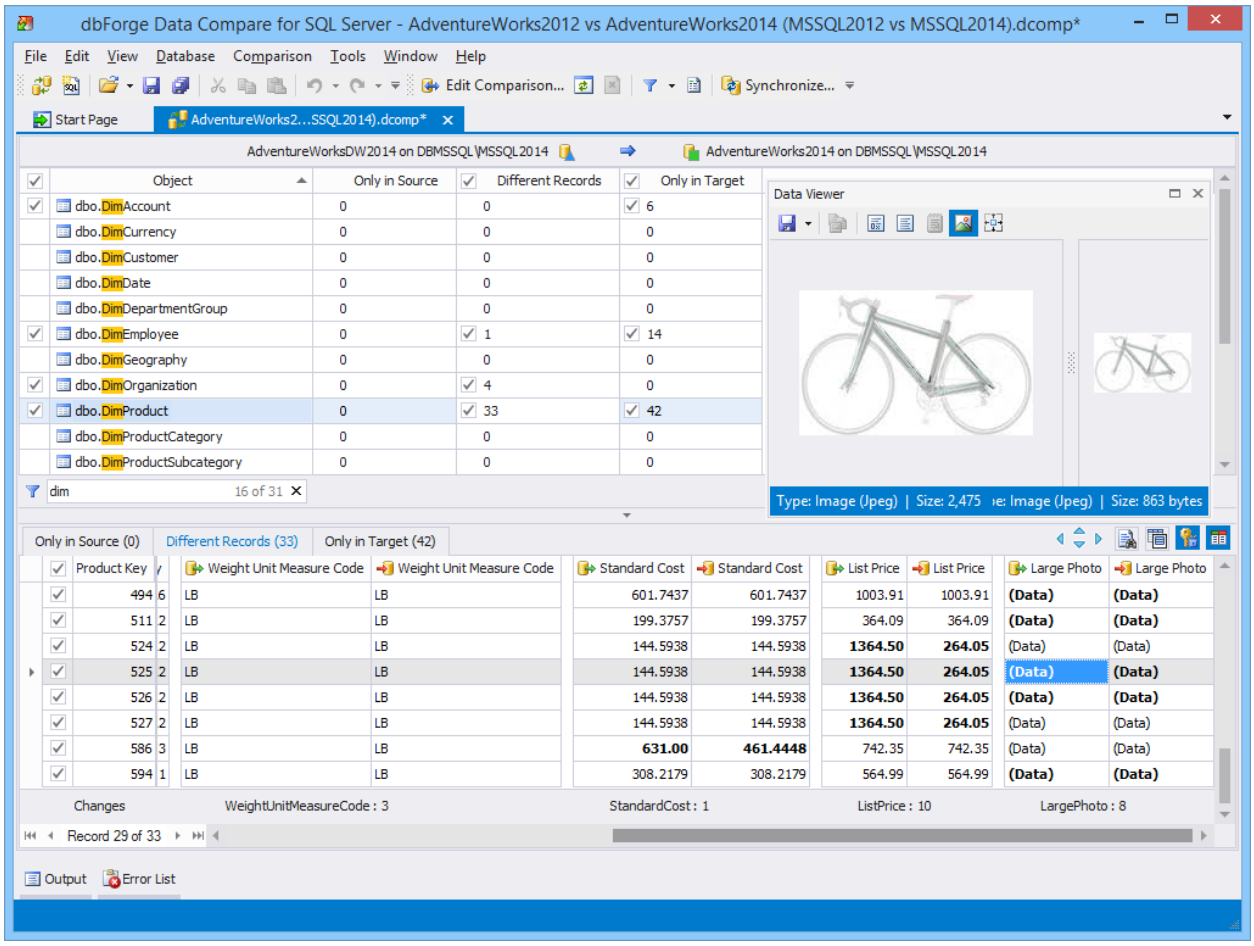

Рисунок 1.4 – dbForge Data Compare for SQL Server

ApexSQL Diff - решение для комплексного сравнения сценариев БД по множеству различных факторов, включая уникальные особенности: элементы кода, RegEx-фильтры, проектные особенности и многие другие:

- сравнение и синхронизация структуры и содержимого баз данных;
- автоматизация операций по сравнению БД с помощью мощного интерфейса командной строки;
- мастер синхронизации данных и структур;
- встраивание программы сравнения в процесс разработки;
- HTML-отчёты для структур и данных;
- XML-экспорт данных;
- встроенный редактор скриптов, позволяющий просматривать даже запущенные скрипты.
- создание снимков БД;
- поддержка всех объектов SQL 2005;
- выполнение миграции с SQL 2000 на SQL 2005;
- сравнение зашифрованных объектов баз данных SQL Server 7 и 2000 [41].

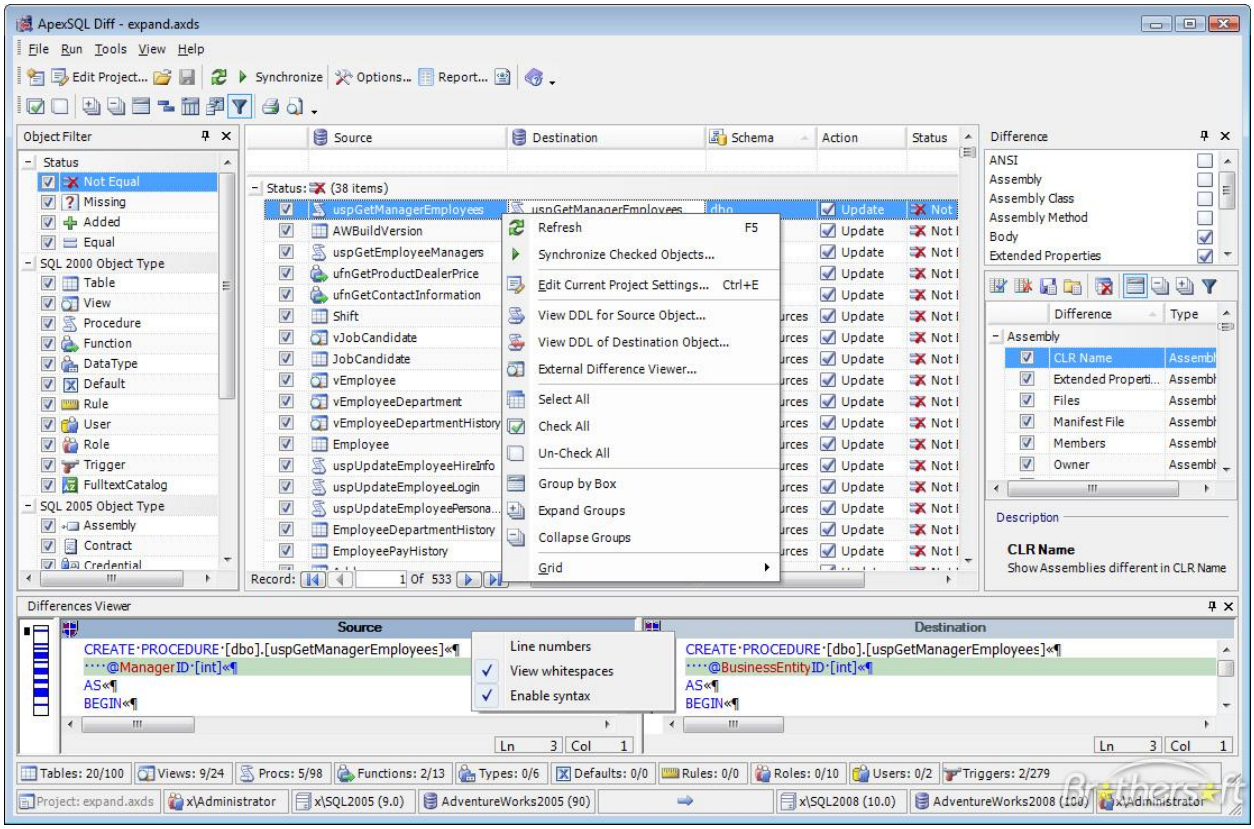

Рисунок 1.5 – ApexSQL Diff

xSQL Object - это профессиональная сервисная программа работы с SQL Server для сравнения и синхронизации SQL Server баз данных. Данная программа позволяет SQL Server разработчикам и администраторам баз данных следить за синхронизированной работой своих сред разработки, тестирования и производства с минимальными усилиями.

Основные особенности:

- Полная поддержка SQL Server 2000 и SQL Server 2005 – все объекты.
- Сравнение по версиям – сравнение баз данных SQL Server 2000 с базами данных SQL Server 2005 и генерирование сценариев с определёнными изменениями по версиям.
- Снимки схем – поддерживает историю изменения схем и предоставляет безопасный и эффективный анализ схем, если потребуется.
- Широкий спектр опций, которые позволяют частично контролировать поведение системы сравнения.
- Расширенная фильтрация объектов – сравнение и синхронизация только требуемых объектов.
- Сравнение результатов, отображаемых в простой и удобной решётке, которая позволяет пользователю не вдаваться в подробности, быстро просматривать сценарии объектов, результаты фильтрации и т.д.
- Генерирование безопасных, стандартизированных сценариев изменений и сценариев обзора перед исполнением в целевой базе данных.
- Утилита командной строки предоставляет сравнение и синхронизацию по расписанию.
- Расширенные функции создания сценариев – создавайте сценарий по любому объекту или всей базе данных, сценарию схемы или по схемам и данным одновременно [43][45].

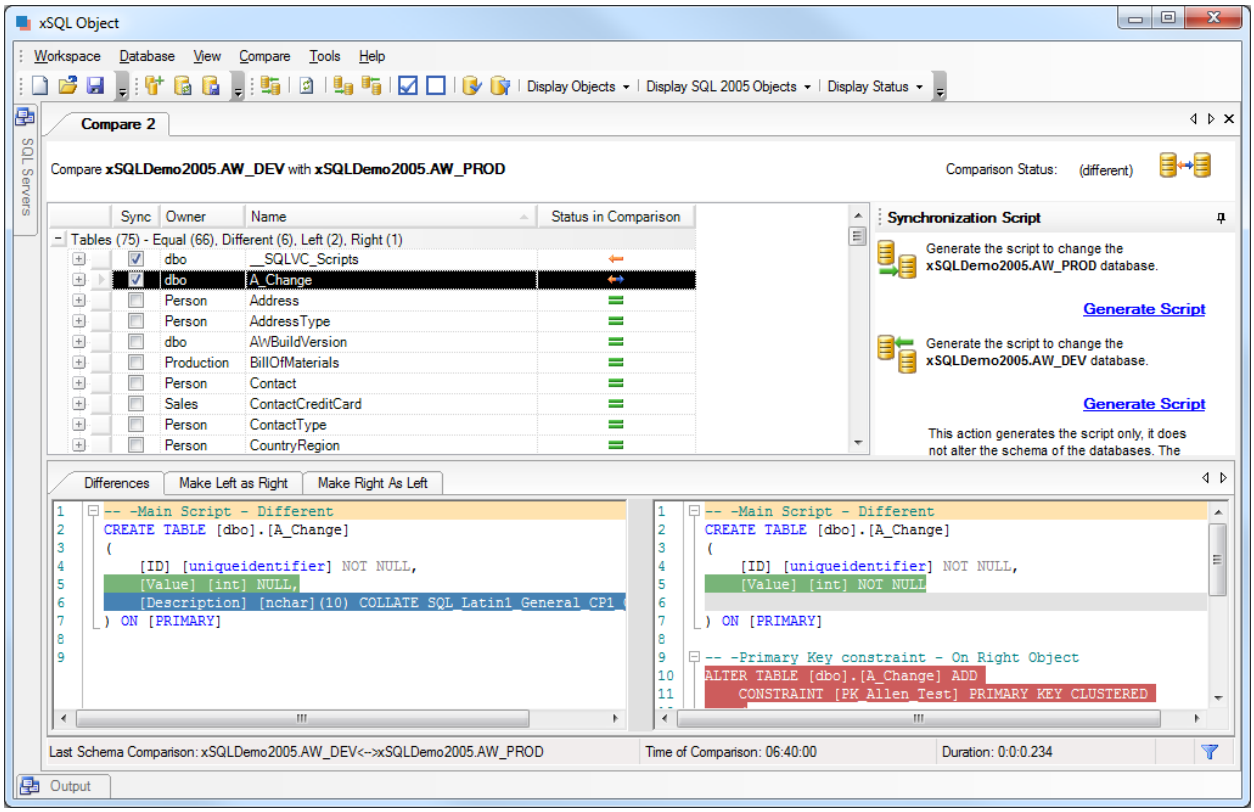

Рисунок 1.6 – xSQL Object

Теперь, когда мы перечислили основные особенности существующих компараторов баз данных, сравним их между собой по ключевым критериям. Представим сравнение в виде таблицы, в которой по строкам будут расположены ключевые особенности, а по столбцам сами компараторы.

Таблица 1.1 – Сравнение существующих компараторов

Критерий	**dbForge Data Compare for SQL Server**	**ApexSQL Diff**	**xSQL Object**
Поддержка SQL Server	Есть	Есть	Есть
Возможность сравнения с резервными копиями базы данных	Есть	Нет	Нет
Синхронизация двух баз данных между собой	Есть	Есть	Есть
Восстановление данных из резервной копии	Есть	Нет	Нет

Продолжение таблицы 1.1

Критерий	**dbForge Data Compare for SQL Server**	**ApexSQL Diff**	**xSQL Object**
Частичное восстановление данных для определённой таблицы из резервной копии	Есть	Нет	Нет
Создание отчётов результатов компарации баз данных	Есть	Есть	Нет
Встроенный редактор скриптов	Нет	Есть	Нет
Создание снимков схем	Нет	Есть	Есть
Возможность ограничить область компарации определёнными таблицами	Нет	Нет	Есть

Как можно видеть по представленной таблице у выбранных нами компараторов есть свои плюсы и минусы. Все эти компараторы в основном предназначены для синхронизации данных в разных версиях базы данных, и их восстановления в случае утери. Однако нам требуется «вытащить» из БД данные логически связанные с нужным нам критерием. Поэтому эти приложения использовать не удастся.

С другой стороны, привязывание компаратора к логике информационной системы приведёт к невозможности применения его к другим базам данных без изменения логики приложения.

Выводы по главе 1

В первой главе было проведено описание того, что такое компараторы и где они используются. Были описаны цифровые и аналоговые компараторы.

Был проведён анализ существующих аналогов компараторов баз данных. Для анализа были выбраны компараторы dbForge Data Compare for SQL Server, ApexSQL Diff и xSQL Object. Были описаны их особенности, а также проведено сравнение этих компараторов между друг другом по их возможностям, на основе которого были выделены сильные и слабые стороны этих компараторов.

2 Проектирование компаратора

2.1 Выделение основных компонентов разрабатываемого приложения

Основной задачей нашего приложения является получение данных из двух схожих по структуре баз данных, их сохранение и сравнение между собой. Поэтому условно приложение можно разбить на две части:

1. Получение данных из БД, их структуризация и сохранение в файл.
2. Чтение данных и файлов, относящихся к разным БД, их компарация и предоставление подробного отчёта разработчику.

Промежуточный этап сохранения данных в файл и его чтение был введён по той причине, что зачастую у разработчика нет прямого доступа к базе данных клиента и наоборот. По этой причине первый компонент приложение нужно будет запускать отдельно на машине клиента и отдельно на машине разработчика.

Для представления процесса компарации мы будем использовать диаграмму IDEF0.

IDEF0 — методология функционального моделирования (англ. function modeling) и графическая нотация, предназначенная для формализации и описания бизнес-процессов. Отличительной особенностью IDEF0 является её акцент на соподчинённость объектов. В IDEF0 рассматриваются логические отношения между работами, а не их временная последовательность (поток работ) [6]. Стандарт IDEF0 представляет организацию как набор модулей, при этом самый важный процесс находится в левом верхнем углу.

Контекстная диаграмма — это самая верхняя диаграмма на которой моделируемый объект представлен одним блоком с граничными стрелками. Она нумеруется как A-0. Стрелки этой диаграммы связаны с внешней средой. Диаграмма A-0 устанавливает область и границы моделирования.

Стандарт IDEF0 поддерживает процедуру декомпозиции основных процессов до необходимого уровня детализации. Уровней декомпозиции может быть несколько.

Все блоки в контактной диаграмме IDEF0 должны располагаться по диагонали сверху вниз. Блоки диаграммы, расположенные выше слева доминируют над теми, которые расположены ниже справа. Под доминированием понимается влияние, которое один блок оказывает на другой [35].

На контекстной диаграмме присутствует четыре типа стрелок: вход, выход, механизм и управление. *Входы* используются процессами чтобы создать то, что появляется на выходе этих процессов. *Управления* предоставляют необходимые условия для получения результата на выходе из процессов. *Выходы* – результат, производимый процессом. *Механизмы* указывают средства, необходимые для выполнения процесса. Суммировав вышеперечисленное, мы получаем, что нотация IDEF0 демонстрирует процесс преобразования *входа* в *выход* с помощью *механизмов*, работающих под *управлением*.

Разберём графические символы, использующиеся в нотации:

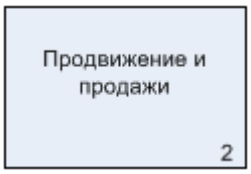

Рисунок 2.7 – Процесс

Процесс обозначается прямоугольным блоком. Внутри блока отображается его номер и имя. Имя должно быть глаголом и обозначать действие. Номер блока находится в нижнем правом углу блока.

Рисунок 2.8 – Стрелка

Стрелки обозначают входящие и выходящие данные. Роль стрелки определяется стороной, с которой она соединяется с процессами. Слева – вход, справа – выход, сверху – управление, а снизу – механизм.

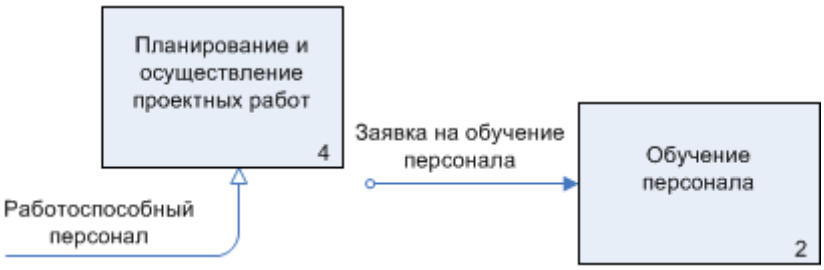

Рисунок 2.9 – Туннелированная стрелка

Туннелированные стрелки обозначают, что передаваемые ими данные не рассматриваются на родительской или дочерней диаграмме. Если такая стрелка присоединена к блоку на текущем уровне диаграммы, то это не гарантирует её наличие на следующем уровне декомпозиции. На свободном же конце туннельная стрелка означает, что она отсутствует на родительском уровне.

Рисунок 2.10 – Внешняя ссылка

Внешние ссылки обозначают субъект, находящийся за пределами моделируемой системы. Они используются в качестве приёмника или источника стрелок вне модели. Обозначаются в виде квадрата.

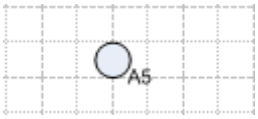

Рисунок 2.11 – Междиаграммная ссылка

Междиаграммная ссылка обозначает другую диаграмму на текущей диаграмме. Она служит для перехода стрелок на диаграмму другого процесса без отображения стрелки на диаграмме более высокого уровня.

Рисунок 2.12 – Сноска и текст

Комментарии могут быть использованы как вместе со сноской, так и без неё [39].

Теперь представим наш процесс в виде диаграммы IDEF0.

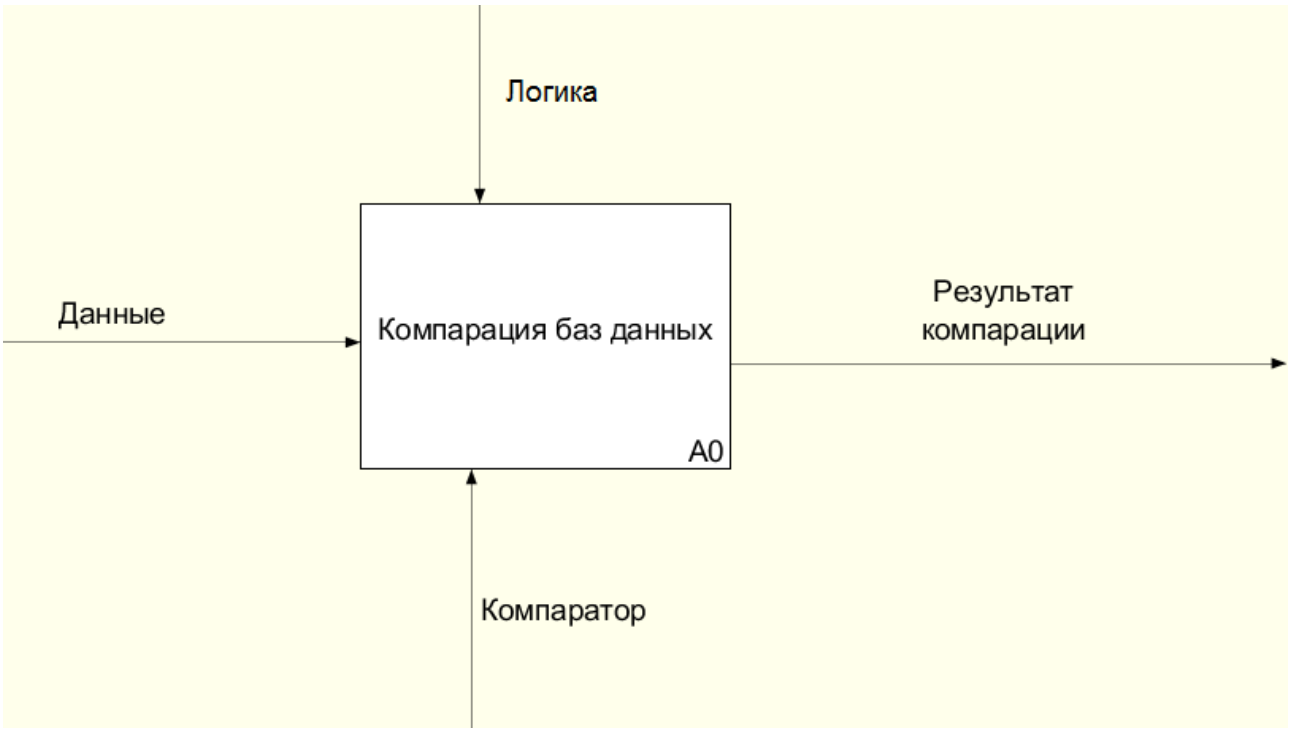

Рисунок 2.13 – Функциональная модель

На входе у нас данные из двух баз данных. На выходе результат их сравнения между собой. Процессом сравнения будет заниматься компаратор.

Проведём декомпозицию процесса компарации.

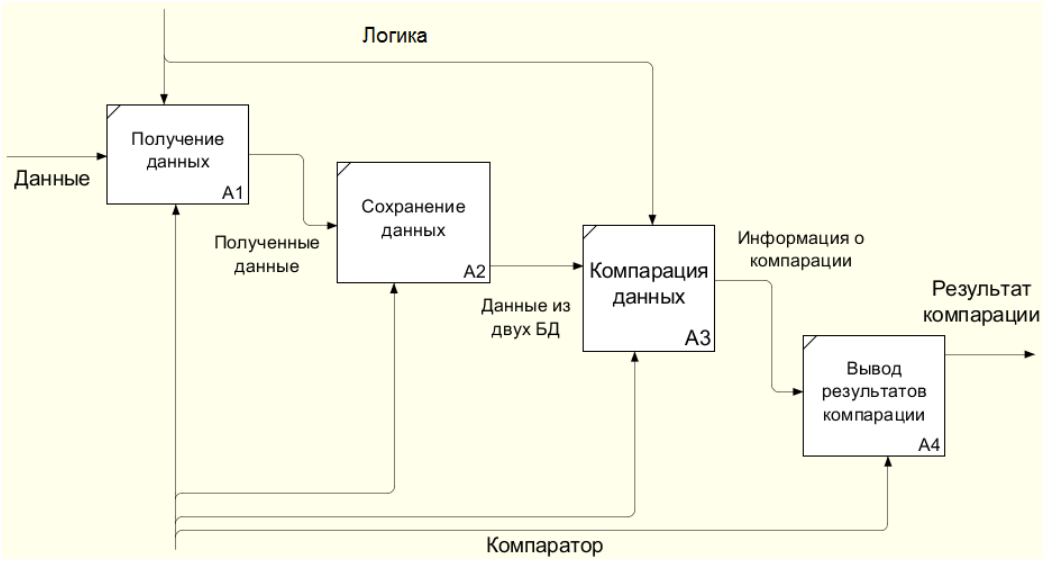

Рисунок 2.14 – Декомпозиция процесса компарации

Вначале производится получение сравниваемых данных из базы данных и их сохранение для последующей компарации. Этот процесс повторяется для второй базы данных. Затем производится компарация сохранённых данных, а также вывод её результатов конечному пользователю. Непосредственно перед процессом компарации данных, сами данные извлекаются из двух файлов, записанных на втором этапе. Все четыре процесса будут выполняться компаратором.

Рассмотрим, каким образом между этими процессами будут передаваться данные.

В первую очередь пользователь должен указать по какому критерию следует извлекать данные. Затем, после извлечения данных из двух баз данных, их следует сохранить. Это делается по двум причинам:

1. Сохранение данных позволить уменьшить число запросов к базе данных в случае эталонного сравнения. Когда несколько различных наборов данных сравниваются с одним.
2. Сохранённые данные можно передавать на другую машину, на которой будет производиться компарация. Это может пригодиться в случаях, когда к базе данных имеет доступ ограниченное число машин, доступ к которым есть только через консоль. В таком случае эти машины будут выступать в качестве посредника в извлечении данных.

Сохранённые данные считываются другой частью программы, а затем сравниваются между собой. Результат сравнения выводится на экран.

Представим всё это в виде диаграммы потоков данных.

DFD — общепринятое сокращение от англ. Data Flow Diagrams — диаграммы потоков данных. Так называется методология графического структурного анализа, описывающая внешние по отношению к системе источники и адресаты данных, логические функции, потоки данных и хранилища данных, к которым осуществляется доступ[17].

Внешние сущности порождают потоки данных, переносящие информацию к подсистемам и процессам, которые преобразуют полученную информацию и передают другим процессам, подсистемам, накопителям данных или внешним сущностям. Рассмотрим каждый из компонентов DFD подробнее.

Рисунок 2.15 – Внешняя сущность

Внешней сущностью может выступать объект или лицо являющееся источником или приёмником информации. На приведённом выше примере это заказчик. Внешней сущностью выступает объект, находящийся за пределами анализируемой информационной системы. В ходе анализа некоторые внешние сущности могут быть перенесены внутрь диаграммы, и наоборот, вынесены за её пределы. Внешняя сущность обозначается прямоугольником с острыми краями [10].

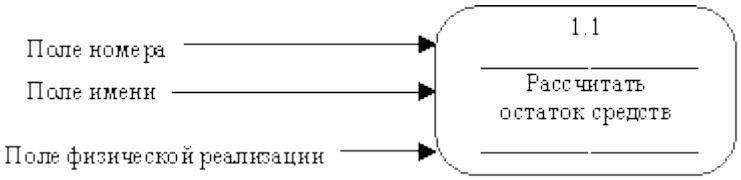

Рисунок 2.16 – Процесс

Процесс преобразует входящие потоки данных в исходящие по заданному алгоритму. Процесс может быть реализован различными способами. Например, отделом организации, обрабатывающим документы, программой и так далее. Процесс на диаграмме отображается прямоугольником со скруглёнными краями. Текст в прямоугольниках должен выражать определённое действие над данными.

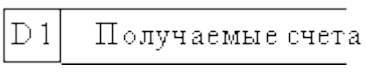

Рисунок 2.17 – Накопитель данных

Накопителем данных выступает любое устройство для хранения информации, которую можно в любой момент времени получить и извлечь из устройства [29]. Он может быть реализован в любом виде: от ящика в картотеке до таблицы в базе данных. Накопитель данных на диаграмме отображается прямоугольником с острыми углами без одной из граней с правой стороны. В общем случае накопитель данных выступает прообразом будущей базы данных.

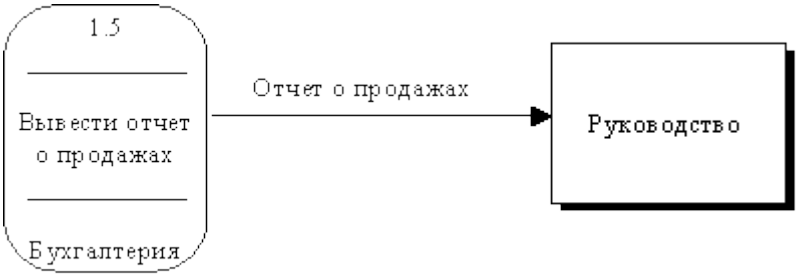

Рисунок 2.18 – Поток данных

Поток данных отображает информацию, передаваемую от источника к приёмнику. В реальности эту информация может быть передана кабелем, письмом, дискетами и прочими носителями информации. Поток данных на диаграмме отображается стрелкой с именем. Имя содержит информацию о передаваемых данных. [12]

Представим процесс компарации в виде модели DFD:

Рисунок 2.19 – DFD процесса компарации

В качестве первых двух хранилищ данных выступают те базы данных, данные из которых следует сравнить между собой. Третье хранилище отведено под извлечённые из первых двух данные.

Первоначально пользователь указывает критерии сравнения. Ему необходимо указать, какие таблицы сравнивать, а также каким образом таблицы связаны между собой. Затем компаратор на основе полученной логики извлекает данные первой из базы данных и записывает их в файл. Для второй базы данных также производится извлечение данных и запись из во второй файл. На этом этап извлечения данных заканчивается.

Начинается второй этап – компарация полученных данных. Пользователю необходимо выбрать для компарации два файла, полученные на предыдущем этапе. Компаратор загружает данные из файлов обратно в классы и сравнивает их между собой согласно логике, указанной пользователем на первом этапе. Результат сравнения данных показывается пользователю, который должен проанализировать результат компарации, и выбрать нужное различие в данных.

Сам же компаратор должен обладать следующими возможностями:

- Долговременное хранение сохранённых данных;
- Построковый вывод сравниваемых данных в два столбца;
- Сравнение длины строк;
- Проверка возможности перевода строковых данных в числа;
- Возможность ограничивать набор таблиц для сравнения для ускорения поиска различий.

2.2 Описание базы данных, под которую разрабатывается компаратор

Наш компаратор разрабатывается под конкретную базу данных образовательного приложения, поэтому необходимо остановиться на её структуре. Для её представления используется модель ADO.NET EDM.

Модель EDM — это модель для определения данных как наборов сущностей и связей, с которыми можно сопоставить типы CLR и структуры хранилища данных. Модель EDM позволяет разработчикам создавать программы с помощью концептуальной модели данных, а не использовать непосредственно схемы хранения.

Конструктор моделей EDM ADO.NET (конструктор сущностей) представляет собой визуальное средство, которое позволяет изменять модель EDM в интерактивном режиме. С помощью конструктора сущностей можно визуально создавать и изменять сущности, ассоциации, сопоставления и связи наследования. Кроме того, можно проверить модель EDM.

Конструктор сущностей совместно с мастером моделей EDM и мастером обновления моделей обеспечивает создание, изменение и обновление модели EDM [40]. Полученную с помощью конструктора модель можно превратить в реальную базу данных с таблицами. Разберём основные элементы и особенности этой модели.

При создании файл модели имеет формат edmx, открыв который, мы увидим конструктор моделей. На него можно добавлять следующие элементы:

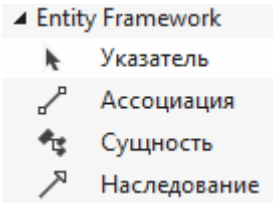

Рисунок 2.20 – Элементы Entity Framework

У нас используются лишь две из них: «Сущность» и «Ассоциация».

У «сущности» нам понадобятся 2 свойства:

- Имя – название сущности в конструкторе
- Имя набора сущностей – название таблицы, которая будет создана из этой сущности.

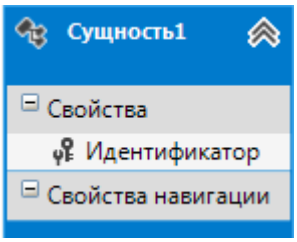

Рисунок 2.21 – Сущность

При создании сущности, ей автоматически добавляется скалярное свойство «Идентификатор». Стоит упомянуть, что именно скалярные свойства станут колонками в создаваемой таблице БД.

У скалярного свойства нам понадобятся следующие свойства:

- Допускает значения NULL
- Имя – имя свойства и колонки в создаваемой таблице
- Ключ сущности – определяет, является ли столбец ключевым.
- Тип – тип данных столбца

Создав 2 сущности с их скалярными свойствами, мы можем их связать по ключевому столбцу одной из сущностей. Для этого, в конструкторе необходимо добавить «Ассоциацию».

Здесь мы указываем, какие сущности связываем, и кратность связи.

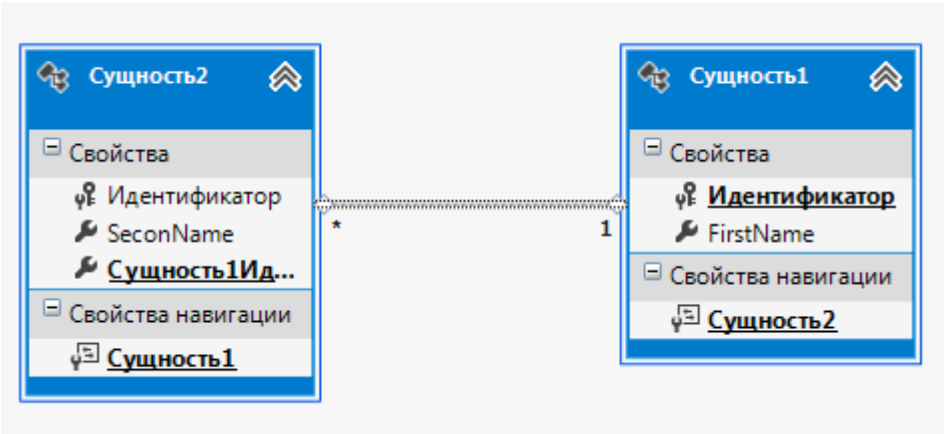

Рисунок 2.22 – Внешний вид ассоциации в конструкторе.

При ассоциации, дочерней сущности автоматически добавляется наследуемый столбец для ассоциации строк из разных таблиц.

Завершив создание модели базы данных из полученной модели можно автоматически сгенерировать базу данных, структура которой будет основана на созданной нами модели.

Наш компаратор будет разрабатываться под базу данных, созданную вышеописанным способом. Приведём её ADO.NET EDM модель:

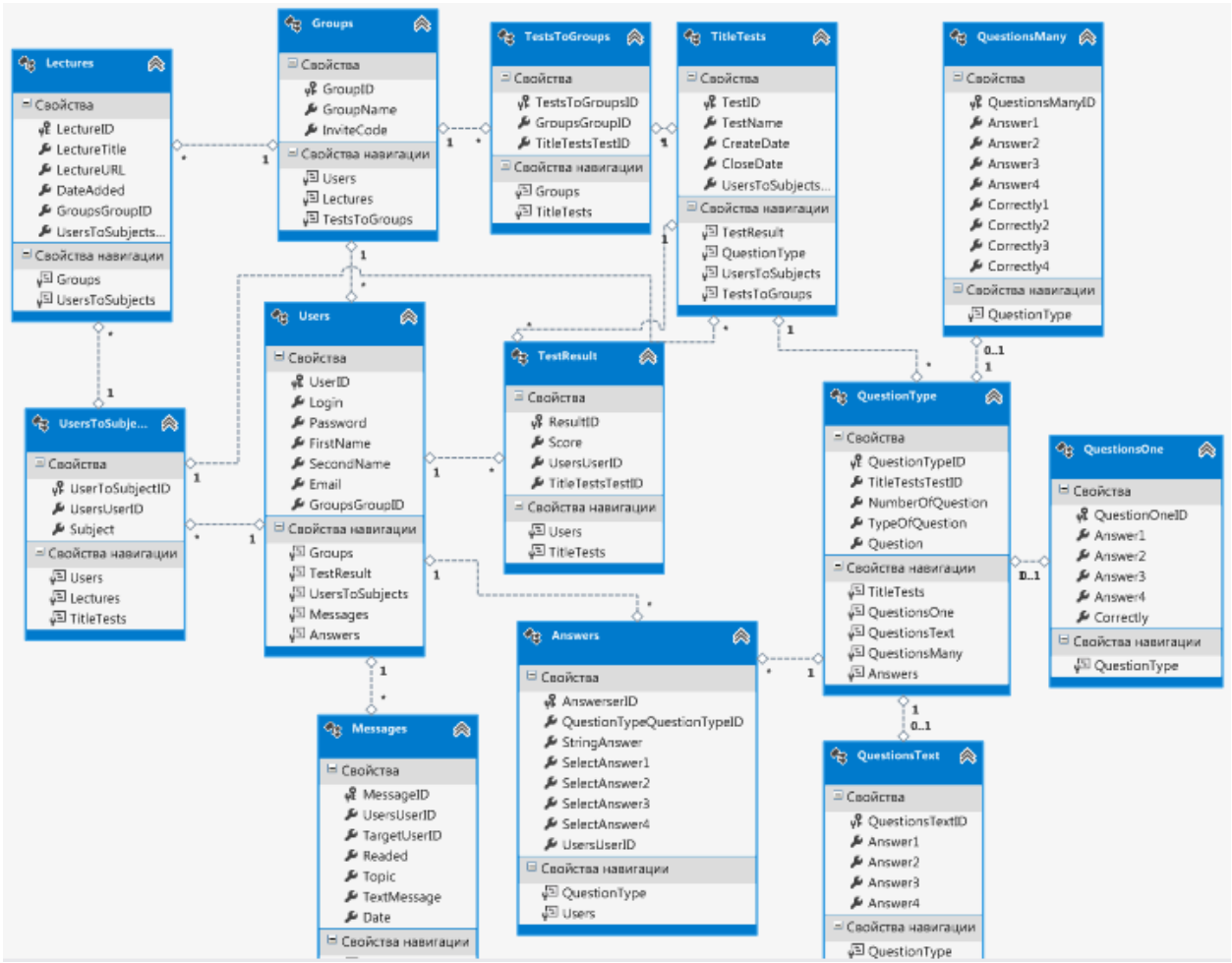

Рисунок 2.23 – ADO.NET EDM модель базы данных

Остановимся на типах данных в каждой таблице:

Таблица Groups:

- GroupID – INT.
- GroupName – NVARCHAR (MAX).

Таблица Users:

- UserID – INT.
- Login – NVARCHAR (MAX).

- Password – NVARCHAR (MAX).
- FirstName – NVARCHAR (MAX).
- SecondName – NVARCHAR (MAX).
- GroupsGroupID – INT.

Таблица TitleTests:

- TestID – INT.
- TestName – NVARCHAR (MAX).
- UsersUserID – INT.
- UsersToSubjectsUserToSubjectID – INT.

Таблица TestsToGroups:

- TestsToGroupsID – INT.
- GroupsGroupID – INT.
- TitleTestsTestID – INT.

Таблица QuestionType:

- NumberOfQuestion – INT.
- Question – NVARCHAR (MAX).
- QuestionTypeID – INT.
- TypeOfQuestion – NVARCHAR (MAX).
- TitleTestsTestID – INT.

Таблица QuestionsText:

- QuestionsTextID – INT.
- Answer1 – NVARCHAR (MAX).
- Answer2 – NVARCHAR (MAX).
- Answer3 – NVARCHAR (MAX).
- Answer4 – NVARCHAR (MAX).

Таблица QuestionsOne:

- Answer1 – NVARCHAR (MAX).
- Answer2 – NVARCHAR (MAX).

- Answer3 – NVARCHAR (MAX).
- Answer4 – NVARCHAR (MAX).
- Correctly – NVARCHAR (MAX).
- QuestionsOne ID – INT.

Таблица QuestionsMany:

- Question – NVARCHAR (MAX).
- Answer1 – NVARCHAR (MAX).
- Answer2 – NVARCHAR (MAX).
- Answer3 – NVARCHAR (MAX).
- Answer4 – NVARCHAR (MAX).
- Correctly1 – Boolean.
- Correctly2 – Boolean.
- Correctly3 – Boolean.
- Correctly4 – Boolean.
- QuestionsManyID – INT.

Таблица TestResult:

- ResultID – INT.
- Score – INT.
- UsersUserID – INT.
- TitleTestsTestID – INT.

Таблица Messages:

- MessageID – INT.
- Readed – Boolean.
- UsersUserID – INT.
- TargetUserID – INT.
- Topic – NVARCHAR (MAX).
- TextMessage – NVARCHAR (MAX).
- Date – DateTime.

Таблица Lectures:

- LectureID – INT.
- LectureTitle – NVARCHAR (MAX).
- LectureURL – NVARCHAR (MAX).
- DateAdded – DateTime.
- GroupsGroupID – INT.
- UsersToSubjectsUserToSubjectID – INT.

Таблица UserToSubjectID:

- UserID – INT.
- UsersUserID – INT.
- Subject – NVARCHAR (MAX).

В основном используется 2 типа данных: INT и NVARCHAR.

Точкой входа, местом от которого мы начнём получать логически связанные данные, может быть любая таблица. Связь идёт следующим образом:

- Таблица Groups связана с таблицей Users через колонку GroupsGroupID.
- Таблица Users связана с таблицей UserToSubjectID через колонку UsersUserID.
- Таблица UserToSubjectID связана с таблицей Lectures через колонку UsersToSubjectsUserToSubjectID.
- Таблица Lectures связана с таблицей Groups через колонку GroupsGroupID.
- Таблица Users связана с таблицей Messages через колонки UsersUserID и TargetUserID. Двойная связь используется потому что сообщения, для хранения которых используется таблица Messages, отравляются от пользователя к другому пользователю.
- Таблица Groups связана с таблицей TestsToGroups через колонку GroupsGroupID.

- Таблица TestsToGroups связана с таблицей TitleTests через колонку TitleTestsTestID.
- Таблица TitleTests связана с таблицей UserToSubjectID через колонку UsersToSubjectsUserToSubjectID.
- Таблица TitleTests связана с таблицей TestResult через колонку TitleTestsTestID.
- Таблица TestResult связана с таблицей Users через колонку UsersUserID.
- Таблица TitleTests связана с таблицей QuestionType через колонку TitleTestsTestID.
- Таблица QuestionType связана с таблицей QuestionsMany через колонку QuestionsManyID.
- Таблица QuestionType связана с таблицей QuestionsOne через колонку QuestionsOneID.
- Таблица QuestionType связана с таблицей QuestionsText через колонку QuestionsTextID.
- Таблица QuestionType связана с таблицей Answers через колонку QuestionTypeQuestionTypeID.
- Таблица Answers связана с таблицей Users через колонку UsersUserID.

Также упомянем, для чего используется каждая из таблиц:

- Groups – группы студентов электронного университета и отдельная группы преподавателей;
- Users – пользователи и их контактная информация;
- UserToSubjectID – связующая таблица между преподавателями и их предметами;
- Lectures – здесь хранятся ссылки на скачивание лекций и учебных материалов

- TitleTests – таблица для хранения названий тестов, созданных преподавателями;
- Messages – личные сообщения пользователей;
- TestsToGroups – связующая таблица между созданными тестами и группами студентов;
- QuestionType – таблица, определяющая тип вопросов в тесте;
- QuestionsMany – здесь хранятся вопросы с множественным выбором;
- QuestionsOne – здесь хранятся вопросы с единственно верным ответом;
- QuestionsText – здесь хранятся вопросы, требующие написания ответа вручную;
- TestResult – таблица, содержащая результаты прохождения тестов студентами.

Данная база данных используется для прототипа электронного университета, позволяющего распространять лекции между студентами и проводить тестирование. В самой базе данных хранится информация о пользователях, пользовательских группах, лекциях, тестах и результатах прохождения тестирования каждым пользователем.

Выводы по главе 2

Во второй главе были построены модели нотаций IDEF0 и DFD для разрабатываемого компаратора. Были выделены основные процессы и компоненты компаратора.

Также была описана база данных, под которую будет разрабатываться компаратор и выделены логические связи между таблицами базы данных.

3 Разработка компаратора

3.1 Выбор языка программирования

В качестве вероятных языков программирования рассмотрим C# и Java. Java это объектно-ориентированный язык программирования, разработанный компанией Sun Microsystems (которая в последствии была приобретена компанией Oracle) [48]. Приложения Java обычно транслируются в особый байт-код, что позволяет им работать на любой виртуальной Java-машине вне зависимости от архитектуры компьютера [14]. Дата официального выпуска — 23 мая 1995 года [37].

Программы на Java транслируются в байт-код, выполняемый виртуальной машиной Java (JVM) — программой, обрабатывающей байтовый код и передающей инструкции оборудованию как интерпретатор [9].

Достоинством подобного способа выполнения программ является абсолютная независимость байт-кода от операционной системы или оборудования, позволяющая выполнять Java-приложения на любой машине, для которой существует соответствующая виртуальная машина [8]. Другой немаловажной особенностью Java является гибкая система безопасности, в рамках которой исполнение программы полностью контролируется виртуальной машиной [19]. Все операции, превышающие установленные полномочия программы (к примеру, попытка несанкционированного соединения с другим компьютером или несанкционированный доступ к данным) вызывают немедленное прерывание [33].

Теперь рассмотрим C#.

C# (C Sharp) это объектно-ориентированный язык программирования, разработанный 1998 году компанией Майкрософт под руководством инженера Андерса Хейлсберга. Он позиционировался для использования на платформе NET Framework [16].

C# имеет Си-подобный синтаксис, близкий к C++ b Java. Обладает статической типизацией, поддерживает исключения, итераторы, свойства, события, атрибуты, делегаты, перегрузку операторов и полиморфизм. Тем не менее, C#, пусть и являясь наследником языков C++, Smalltalk, Модула, Pacal и Java, всё

же исключает использование некоторых проблемных модулей. Например, он не поддерживает множественное наследование классов. Впрочем, множественное наследование интерфейсов присутствует [24].

Сравним оба языка между собой. C# и Java – строго типизированы и объектно-ориентированы. Оба опираются на сборщик мусора. Однако существуют и различия. Рассмотрим некоторые из них.

Синтаксис.

- В Java отказались от оператора goto, тогда как в C# он сохранился.
- В Java вместо констант используется модификатор final, в C# константы обозначаются через const.
- В Java существует особый конструктор strictfp, гарантирующий одинаковые вычисления значений с плавающей точкой на всех платформах, тогда как C# полагается на реализацию и гарантий одинаковых вычислений на разных платформах нет.
- В Java любые динамические проверки могут быть включены или выключены только на уровне пакетов. В C# существуют специальные конструкции, позволяющие локально управлять динамической проверкой арифметического переполнения.

Объектные средства

- В C# запрещается давать методам имена, совпадающие с их классами, тогда как в Java таким образом можно определить конструктор, который будет являться методом [47].
- В Java все открытые методы, не являющиеся статическими, являются виртуальными, в C#, чтобы метод стал виртуальным, это необходимо специально указать ключевым словом virtual.

Типы данных

- Примитивных типов данных в Java меньше, чем в C# за счёт того, что в Java отказались от большинства беззнаковых типов данных,

что может привести к необходимости использовать небезопасные операции конвертации данных.

- Перечислимые типы в C# происходят от целочисленных примитивных типов, в Java перечислимые типы это классы, а их значения – объекты.
- В Java существуют только одномерные массивы. Многомерные массивы, по сути, являются массивами массивов. В C# имеются как настоящие многомерные массивы, так и массивы массивов.

C# поддерживает перегрузку операторов с некоторыми ограничениями. Java перегрузку операторов не поддерживает ради поддержания простоты языка.

В Java один файл должен содержать один public класс. При этом название файла должно совпадать с именем класса. Также Sun рекомендует, чтобы в одном файле содержалось 2000 строк кода, иначе класс считается трудночитаемым. В C# один файл может содержать несколько классов. Имя файла может быть любым. Кроме того, начиная с версии C# 2.0, один класс может быть разбит на несколько файлов.

C# и Java поддерживают механизм обработки исключений. Однако Java, в отличие от C#, поддерживает проверяемые исключения, которые явно задаются для каждого метода.

Интерпретаторы Java для использования могут быть скопированы в любую версию Windows, начиная с 2000 [18]. Фреймворк C# требует установки от имени администратора соответствующей версии на соответствующую ему версию Windows. Кроме того, C# тесно привязывает разработчиков к Windows платформе, в отличие от Java, даже не смотря на использование Mono, написанная на Windows C# программа, может некорректно работать на других платформах.

В плане мобильных платформ имеется довольно чёткое разделение власти. Java используется для разработки под Android платформами, а C# для разработки под Windows платформами.

Представим возможные поддерживаемые парадигмы программирования для Java и C# в виде таблицы.

Таблица 3.2 – Поддерживаемые парадигмы программирования

Парадигма	**C#**	**Java**
Императивная	Поддерживается	Поддерживается
Объектно-ориентированная	Поддерживается	Поддерживается
Функциональная	Поддерживается по большей части	Частично поддерживается
Рефлексивная	Частично поддерживается	Частично поддерживается
Обобщённое программирование	Поддерживается	Поддерживается
Логическая	Не поддерживается	Не поддерживается
Декларативная	Частично поддерживается использованием Language Integrated Query	Не поддерживается
Распределённая	Частично поддерживается за счёт распределённых модификация языка. Например Parallel C#.	Поддерживается

В плане поддерживаемых парадигм программирования, C# и Java идут примерно вровень.

Теперь рассмотрим возможности типизации языков.

Таблица 3.3 – Типизация

Типизация	**C#**	**Java**
Статическая	Поддерживается	Поддерживается
Динамическая	Поддерживается начиная с версии 4.0 за счёт специального псевдо-типа dynamic.	Не поддерживается

Продолжение таблицы 3.3

Типизация	**C#**	**Java**
Явная	Поддерживается	Поддерживается
Неявная	Частично поддерживается (var)	Не поддерживается
Неявное приведение типов без потери данных	Поддерживается	Поддерживается
Неявное приведение типов в неоднозначных ситуациях	Поддерживается	Не поддерживается
Алиасы типов	Поддерживается	Не поддерживается
Вывод типов переменных из инициализатора	Поддерживается	Не поддерживается
Параметрический полиморфизм с ковариантностью	Поддерживается начиная с версии C# 4.0 для интерфейсов и делегатов.	Не поддерживается
Информация о типах-параметрах в runtime	Частично поддерживается	Не поддерживается

Как можно видеть из таблицы выше, в плане возможностей типизации C# немного превосходит Java.

Перейдём к возможностям компиляторов.

Таблица 3.4 – Возможности компилятора

Возможности компилятора	**C#**	**Java**
Open-source компилятор	Поддерживается	Поддерживается
Возможность компиляции	Поддерживается	Поддерживается

Продолжение таблицы 3.4

Возможности компилятора	**C#**	**Java**
Bootstrapping	Поддерживается	Поддерживается начиная с версии 6.0
Многопоточная компиляция	Не поддерживается	Поддерживается
Интерпретатор командной строки	Поддерживается начиная с версии 5.0	Не поддерживается
Условная компиляция	Поддерживается	Частично поддерживается

В плане возможностей компиляции C# и Java обладают примерно одинаковыми возможностями.

Рассмотрим управление памятью в этих языках.

Таблица 3.5 – Управление памятью

Критерий	**C#**	**Java**
Создание объектов на стеке	Поддерживается	Не поддерживается
Неуправляемые указатели	Поддерживается	Частично поддерживается через FFI
Ручное управление памятью	Поддерживается посредством unsafe и System.Runtime.InteropServices	Частично поддерживается через FFI
Сборка мусора	Поддерживается	Поддерживается

Как видно из таблицы, в плане управления памятью C# превосходит Java по многим параметрам.

Рассмотрим возможности управления потоком вычислений.

Таблица 3.6 – Управление потоком вычислений

Критерий	**C#**	**Java**
Инструкция goto	Поддерживается	Не поддерживается
Инструкции break без метки	Поддерживается	Поддерживается
Инструкция break с меткой	Не поддерживается	Поддерживается
Поддержка try/catch	Поддерживается	Поддерживается
Блок finally	Поддерживается	Поддерживается
Блок else (исключения)	Поддерживается	Поддерживается за счёт нескольких последовательных блоков catch
Перезапуски	Поддерживается	Поддерживается
Ленивые вычисления	Частично поддерживается	Не поддерживается
Легковесные процессы (Coroutines)	Не поддерживается	Поддерживались вплоть до Java 1.1
Continuations	Не поддерживается	Не поддерживается

Различия между возможностями управления потоками вычислений зачастую вызваны развитием языка. И если от какой-то особенности в процессе развития отказались, это не значит, что язык стал чем-то хуже. Java и C# в этом плане мало отличаются и оба претерпели изменения в процессе развития.

Перейдём к сравнению используемых типов и структур данных.

Таблица 3.7 – Управление потоком вычислений

Типы данных	**C#**	**Java**
Кортежи	Поддерживается в FCL 4.0.	Не поддерживается
Алгебраические типы данных	Не поддерживается	Не поддерживается

Продолжение таблицы 3.7

Типы данных	**C#**	**Java**
Многомерные массивы	Поддерживается	Поддерживается в виде массивов массивов
Динамические массивы	Поддерживается	Поддерживается
Ассоциативные массивы	Поддерживается	Поддерживается
Контроль границ массивов	Поддерживается	Поддерживается
Цикл foreach	Поддерживается	Поддерживается
Списковые включения	Частично поддерживается	Не поддерживается
Целые числа произвольной длины	Поддерживается (System.Numerics.BigInteger)	Поддерживается (BigInteger и BigDecimal)
Целые числа с контролем границ	Не поддерживается	Не поддерживается

В плане управления потоком вычислений между C# и Java так же мало различий.

Рассмотрим объектно-ориентированные возможности.

Таблица 3.8 – Объектно-ориентированные возможности

Критерий	**C#**	**Java**
Интерфейсы	Поддерживается	Поддерживается
Мультиметоды	Частично поддерживается за счёт эмуляции dynamic	Частично поддерживается за счёт реализации сторонними библиотеками
Mixins	Не поддерживается	Поддерживается

Здесь так же мало различий между C# и Java.

Перейдём к сравнению функциональных возможностей.

Таблица 3.9 – Функциональные возможности

Критерий	**C#**	**Java**
First class functions	Поддерживается	Не поддерживается
Анонимные функции	Поддерживается начиная с версии 3.0	Поддерживается
Лексические замыкания	Поддерживается	Поддерживается через анонимные классы
Каррирование	Поддерживается	Не поддерживается

По функциональным возможностям C# несколько превосходит Java.

Теперь рассмотрим те особенности, которые не вошли перечисленные выше категории.

Таблица 3.10 – Прочие возможности языков

Критерий	**C#**	**Java**
Макросы	Частично поддерживается	Не поддерживается
Шаблоны	Поддерживается	Поддерживается
Поддержка Unicode в идентификаторах	Поддерживается	Поддерживается
Перегрузка функций	Поддерживается	Поддерживается
Именованные параметры	Поддерживается начиная с версии C# 4.0	Не поддерживается
Значения параметров по умолчанию	Поддерживается начиная с версии C# 4.0	Не поддерживается
Локальные функции	Частично поддерживается	Частично поддерживается

Продолжение таблицы 3.10

Критерий	C#	Java
Сопоставление с образцом	Частично поддерживается	Не поддерживается
Наличие библиотек для работы с графикой и мультимедиа	Поддерживается (DirectX через Net и OpenGL через стороннюю библиотеку OpenTK)	Поддерживается

Теперь, рассмотрев основные различия, можно решить, на каком языке остановиться.

Мы решили остановиться на языке программирования Java. Выбор основан простотой использования этого языка на разных платформах. Таким образом, нам не придётся волноваться о том, какая операционная система стоит у нас и у клиента. Кроме того, тот функционал, который присутствует в C#, но отсутствует в Java, для нас не требуется. Java прекрасно подходит для реализации компаратора, который может быть применён для любой платформы.

3.2 Выбор графического пользовательского интерфейса

Пользовательский интерфейс на Java прошёл весьма тернистый путь становления и развития. Его долгое время обвиняли в жадности к ресурсам системы, медленной работе и ограниченной функциональности. Появление .NET с более быстрыми графическими компонентами ещё больше пошатнуло позиции Java. Но подобная конкуренция лишь подстёгивала разработчиков Java к развитию и улучшению графических библиотек. И ниже мы рассмотрим, что из этого получилось.

Abstract Window Toolkit

Abstract Window Toolkit (сокращённо AWT) впервые была выпущена в 1995 году компанией Sun Microsystems. Это была первая попытка создать графический интерфейс для Java. AWT выступал в качестве прослойки, вызываю-

щей методы из библиотек, написанных на C. А эти методы, в свою очередь, использовали графические компоненты операционной системы [31]. С одной стороны, программа, построенная таким образом, внешне была похожа на все остальные программы в используемой операционной системе, но с другой, одна и та же программа может выглядеть совершенно по-разному на разных операционных системах, что осложняло разработку. К тому же, ради мультиплатформенности пришлось унифицировать интерфейсы вызовов компонентов, что привело к несколько урезанной функциональности. Набор компонентов также довольно скромный. Например, отсутствуют таблицы, а в кнопки нельзя поместить иконки. AWT старается автоматически освобождать использованные ресурсы. Это влияет на производительность и усложняет архитектуру. AWT прост для освоения, но написание чего-то сложного вызывает затруднения. Сейчас AWT используется в основном для аплетов. Oracle в данный момент поощряет переход разработчиков на Swing, как более безопасный.

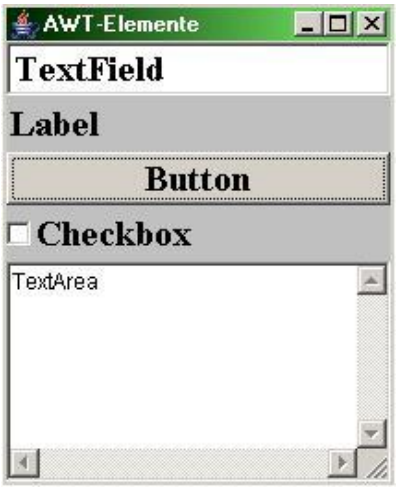

Рисунок 3.24 – Образец программы, написанной с использованием AWT в среде Windows

Swing

После AWT, в 1998 году, Sun выпустила Swing. Он полностью написан на Java и для отрисовки использует 2D. В Swing гораздо больше разнообразных компонентов, чем в AWT. Сами компоненты стало гораздо проще создавать, наследуя их от существующих [27]. Также была введена возможность исполь-

зования различных стилей и скинов. Однако, скорость работы ранних версий Swing была довольно низкой, а ошибки в написании программы могли и вовсе привести к зависанию операционной системы.

Однако, благодаря лёгкому освоению и наличию большого количества документации, Swing стал самым популярным графическим интерфейсом в Java. На его основе появилось множество расширения, например SwingX и JGoodies, которые ещё больше упрощают создание визуально сложных приложений. Все современные среды программирования на Java включают в себя графические редакторы Swing. Даже несмотря на то, что сейчас существуют более современные фреймворки, Swing остаётся самым популярным.

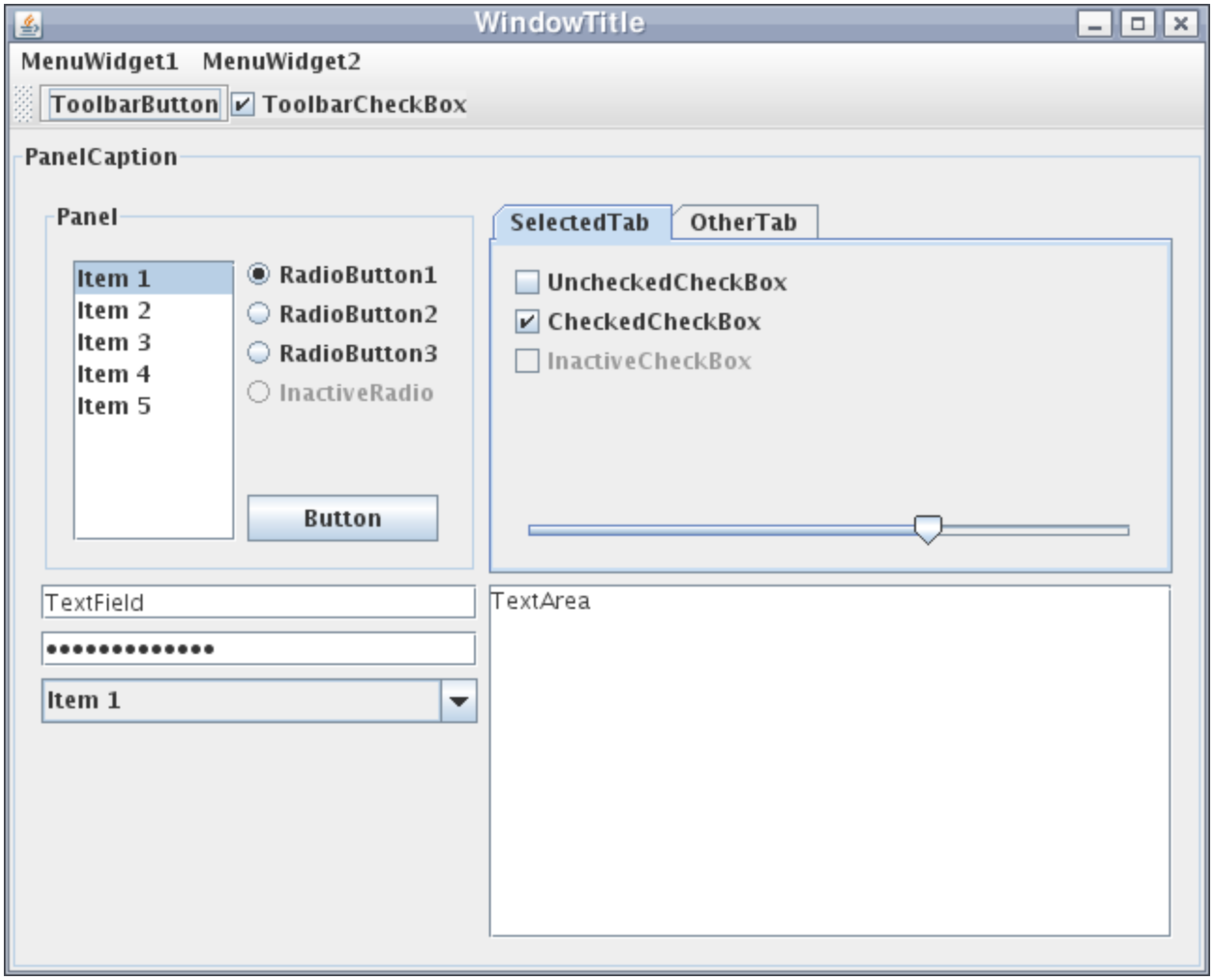

Рисунок 3.25 – Образец программы, написанной с использованием Swing

Standard Widget Toolkit

SWT был выпущен компанией IBM во времена, когда Swing был ещё медленным, и в основном для продвижения среды программирования Eclipse.

Как и AWT, SWT использует компоненты ОС, но для различных платформ используются различные интерфейсы взаимодействия [49]. Таким образом для каждой операционной системы необходимо поставлять отдельную JAR-библиотеку. Это позволяет более полно использовать функции, соответствующие различным операционным системам. А недостающие компоненты были реализованы с помощью 2D. Тем не менее, SWT получилась более сложной для освоения, чем Swing. Кроме того, программист должен сам реализовывать освобождение ресурсов приложением.

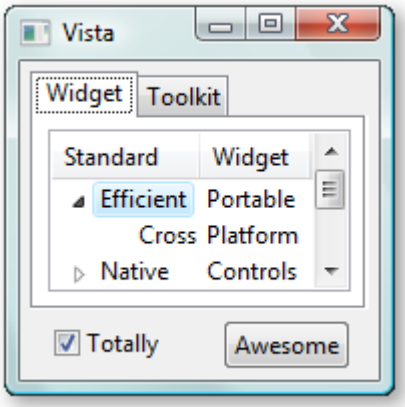

Рисунок 3.26 – Образец программы, написанной с использованием Swing

JavaFX

JavaFX была выпущена в 2008 году компанией Oracle. Она позиционируется как платформа для создания насыщенного интернет-приложения. Для отрисовки используется графический конвейер, что значительно ускоряет работу приложения. Имеется большой набор встроенных компонентов. Также имеются отдельные компоненты для построения графиков. Реализована поддержка мультимедийного контента, анимации и даже множественное касание. Внешний вид компонентов настраивается при помощи CSS-стилей [46].

Кроме того, в набор утилит JavaFX входит возможность сделать родной инсталлятор для самых популярных платформ: exe или msi для Windows, deb или rpm для Linux, dmg для Mac. На сайте Oracle имеется подробная документация и большое число готовых примеров.

Таким образом, описав основным особенности и недостатки вышеперечисленных графических пользовательских интерфейсов, мы можем решить, для каких задач они лучше подходят. Abstract Window Toolkit больше подойдёт для создания аплетов. Новичку можно порекомендовать Swing в виду того, что для него можно найти огромное количество документации в интернете, в том числе и на русском языке. Для создания насыщенных интернет-приложений отлично подойдёт JavaFX.

В нашем же случае, для нас требуется лишь наличие простых и удобных в использовании таблиц, то нам вполне подойдёт Swing. Он обладает необходимыми нам компонентами и достаточно удобен в использовании, а в случае необходимости можно будет легко добыть документацию по интересующим моментам.

3.3 Извлечение данных для компарации

Основной задачей нашего приложения является получение данных из двух схожих по структуре баз данных, их сохранение и сравнение между собой. Поэтому условно приложение можно разбить на две части:

1. Получение данных из БД, их структуризация и сохранение в файл.
2. Чтение данных и файлов, относящихся к разным БД, их компарация и предоставление подробного отчёта разработчику.

Промежуточный этап сохранения данных в файл и его чтение был введён по той причине, что зачастую у разработчика нет прямого доступа к базе данных клиента и наоборот.

Для работы с базой данных используется Java DataBase Connectivity (JDBC), входящий в состав Java SE. JDBC основан на концепции драйверов, позволяющих подключаться к базе данных по определённым образом написанной ссылке. Загрузка самих драйверов может быть динамической. После загрузки драйвер сам регистрируется и автоматически вызывается в моменты использования в ссылках протоколов, за которые отвечает драйвер [44].

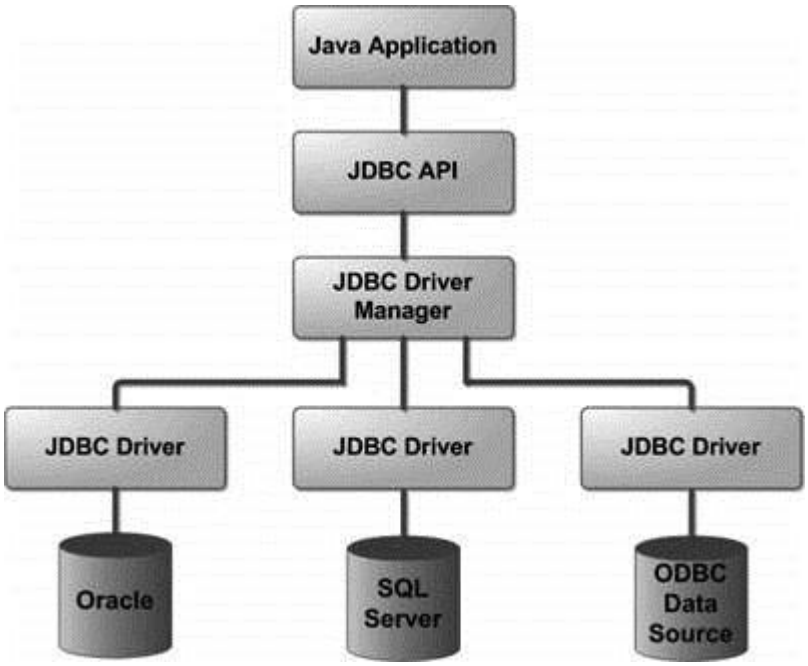

Рисунок 3.27 – Java DataBase Connectivity

Для соединения с базой данных используется класс, реализующий интерфейс java.sql.Connection, а для исполнения запросов к базе данных - java.sql.Statement.

Пример:

```
Statement statement =
core.db.Connection.Connection.getInstance().getStatement();
ResultSet resultSet = statement.executeQuery("select * from
WIZ_ACC_TRX_HISTORY where ACCOUNT_NUMBER=" + num);
WizardList list = new WizardList();
while (resultSet.next()) {
AccTrxHistory order = new AccTrxHistory();
order.setOPERATOR_ID(resultSet.getLong("OPERATOR_ID"));
order.setACCOUNT_NUMBER(
        resultSet.getLong("ACCOUNT_NUMBER"));
order.setSERIAL_NUMBER(resultSet.getString("SERIAL_NUMBER"));
order.setMANUFACTURER(resultSet.getString("MANUFACTURER"));
order.setCONVERTER_TYPE(
```

```
            resultSet.getString("CONVERTER_TYPE"));
        order.setHEADEND(resultSet.getLong("HEADEND"));
        order.setREQUESTED_ACTION(
            resultSet.getString("REQUESTED_ACTION"));
        order.setRESPONSE_STATUS(
            resultSet.getLong("RESPONSE_STATUS"));
        order.setRESPONSE_TIME(
            resultSet.getDate("RESPONSE_TIME"));
        order.setSEQUENCE_NUMBER(
            resultSet.getLong("SEQUENCE_NUMBER"));
        order.setDETAILLINE1(resultSet.getString("DETAILLINE1"));
        order.setDETAILLINE2(resultSet.getString("DETAILLINE2"));
        order.setDETAILLINE3(resultSet.getString("DETAILLINE3"));
        order.setDETAILLINE4(resultSet.getString("DETAILLINE4"));
        order.setDATE_CREATED(resultSet.getDate("DATE_CREATED"));
        list.add(order);
        resultSet.close();
        statement.close();
        return list;
```

Здесь statement содержит в себе оба компонента: java.sql.Connection и java.sql.Statement, которые затем вызываются для извлечения данных.

3.4 Сохранение данных

Для сохранения полученных данных используется сериализация. Сериализация объекта это способность объекта сохранять полную копию его и любых других объектов, на которые он ссылается, используя поток вывода (например, во внешний файл). Таким образом, объект может быть воссоздан из сериализованной(сохранённой) копии немного позже, когда это потребуется [26].

Сериализация используется для передачи объектов по сети и сохранения их в файл. Такой подход используется для создания распределённых приложений со сложной структурой. Для тех типов данных, которые необходимо передать, пишется код, который осуществляет сериализацию и десериализацию. Сначала производится заполнение объекта данными, а затем производится его сериализация, и на выходе мы получаем файл с этим обьектом. Этот файл может быть передан по сети или электронной почте на другую машину, где будет произведена десериализация. В результате мы получаем такой же объект, что и сериализовали, но на другой машине [34].

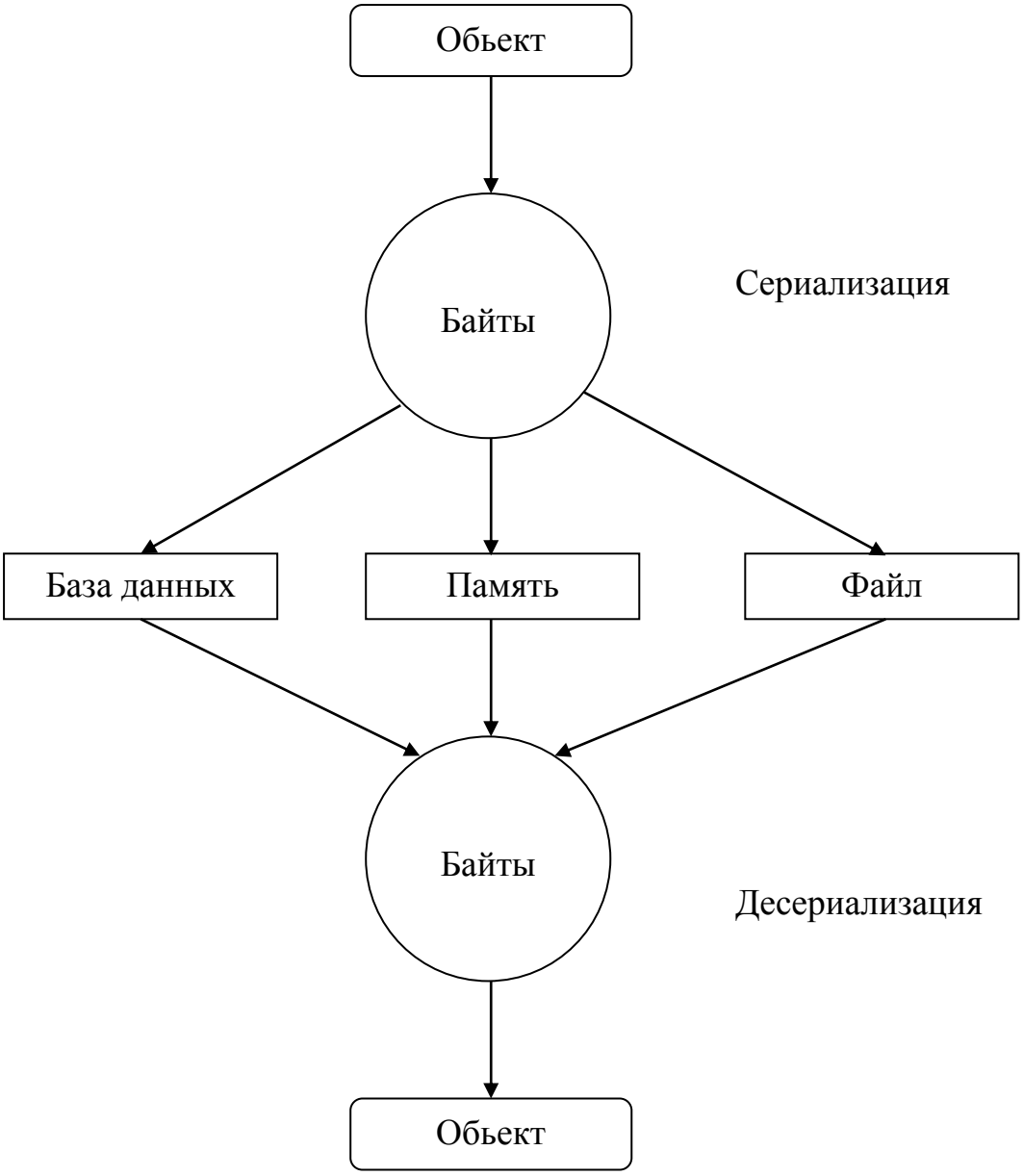

Рисунок 3.28 – Процесс сериализации и десериализации

Любой схеме сериализации присуще последовательное кодирование данных. Поэтому, для того, чтобы извлечь часть из сериализованных данных, объект необходимо полностью десериализовать от начала и до конца. Подобная линейность полезна во многих приложениях, поскольку позволяет применять простые интерфейсы ввода-вывода для передачи и сохранения состояния объекта. Однако, в приложениях, где требуется высокая производительность, может потребоваться применение более сложной организации хранения данных [30].

Сериализация может предоставлять следующие полезные возможности:

- реализация сохраняемости объектов, которая может быть более удобна, чем запись свойств объектов в текстовый файл и их последующее считывание;
- возможность удалённого вызова процедур;
- распространение объектов (особенно применимо в компонентно-ориентированном программировании);
- обнаружение изменений в данных, которые могут меняться со временем.

Для полноценной реализации приведённых выше возможностей необходимо соблюдать независимость от архитектуры. Должна быть возможность восстановить сериализованный объект вне зависимости от порядка данных, использующихся в текущей архитектуре. Это означает, что самая быстрая и простая функция прямого копирования участков памяти, содержащих нужные структуры данных, может быть использована не для всех архитектур. Сериализация объектов в формат, независимый от архитектуры, подразумевает, что проблем из-за различий в механизмах распределения памяти, порядка следования байтов и представления данных в разных языках программирования быть не должно.

Сериализация объектов, как новая возможность, введённая в JDK 1.1, предоставляет функцию для преобразования групп или отдельных объектов, в поток битов или массив байтов, для хранения или передаче по сети [28]. И как

было сказано, данный поток битов или массив байтов, можно преобразовать обратно в объекты Java. Главным образом это происходит автоматически благодаря классам ObjectInputStream и ObjectOutputStream. Программист может решить реализовать эту функцию, путём реализации интерфейса Serializable при создании класса [21].

Класс, реализующий вышеупомянутый интерфейс, выглядит следующим образом:

```
import java.io.Serializable;
public class SomeClass implements Serializable {}
```

Для начала необходимо импортировать библиотеку java.io.Serializable, а затем имплементировать в сериализуемый нами класс интерфейс Serializable. Этот интерфейс помечает наш SomeClass как сериализуемый. Все подклассы, находящиеся внутри SomeClass, также будут сериализуемыми.

При попытке сериализовать объект, содержащий несериализуемые атрибуты, будет возвращена ошибка NotSerializableException. Этого можно избежать, помечая те объекты, которые сериализовать не нужно, ключевым словом transient. В этом случае выбранные программистом атрибуты не будут сериализованы и если значения этих атрибутов потребуются в дальнейшем, то программисту необходимо будет сохранить их иным способом.

Теперь рассмотрим сам процесс сериализации объекта:

```
Manager m = new Manager();
ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(filename));
 oos.writeObject(m);
```

В данном примере мы сериализовали объект класса Manager. Вначале мы создали сам объект, затем создали выходной поток OutputStream, которым в нашем случае стал FileOutputStream. После этого вызвали writeObject() для указанного нами потока. writeObject использует сериализацию для преобразования объекта в поток. В нашем примере мы сериализовали объект в файл, но для любого другого типа сериализации технология остаётся неизменной.

Теперь разберём, как десериализовать сериализованный нами объект обратно в класс.

ObjectInputStream ois = new ObjectInputStream(new FileInputStream(filename));

Manager m = (Manager)ois.readObject();

В данном примере мы сначала создали ObjectInputStream на основе FileInputStream для того, чтобы указать нужный нам файл для сериализации. Затем прочитали его с помощью метода readObject и выполнили явное преобразование полученных данных в класс Manager.

Отдельно следует упомянуть serialVersionUID. Рассмотрим ситуацию: мы сериализовали объект, сохранили полученный файл, поменяли в коде нашей программы структуры класса объекта, который был ранее сериализован. После этого, если мы попытаемся десериализовать старый сохранённый объект в новый изменённый класс, то у нас могут возникнуть непредвиденные проблемы. И в этом случае на помощь может прийти свойство serialVersionUID.

Значение свойства serialVersionUID вычисляется автоматически на основе атрибутов класса, его положения на локальном кластере и имени.. При удалении или добавлении атрибутов в сериализуемый класс, значение serialVersionUID будет перерасчитано. И при конфликте значений будет возвращена ошибка InvalidClassException. Если этого необходимо избежать, то можно явно объявить значение serialVersionUID:

private static final long serialVersionUID = 3060516;

Изменять значение этого свойства необходимо при внесении в класс несовместимых изменений. Например, при удалении или добавлении атрибутов или методов [38].

3.5 Компарация данных

Компарация может состоять из разных типов сравнения:

1. Компарация строк по значению.
2. Компарация строк по длине строки.

3. Компарация строк по возможности конвертации строки в число.
4. Компарация строк по возможности конвертации строки в дату.

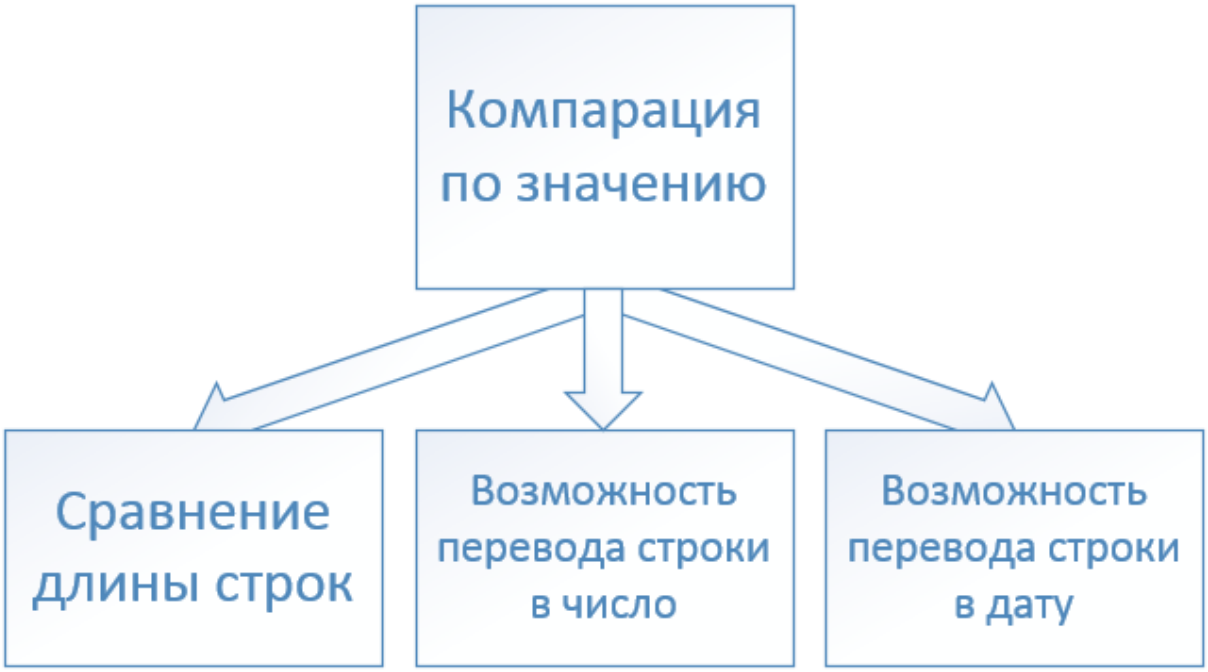

Рисунок 3.29 – Способы компарации

2-4 этапы компарации выполняются только если сравниваемые строки не равны между собой. При сравнении один из наборов данных выбирается как эталон, а второй сравнивается с ним.

Компаратор разбит на 2 основных окна: количественный компаратор и построковый компаратор.

Количественный компаратор отображает все, полученные из обеих баз данных, таблицы, объединённые в единый список.

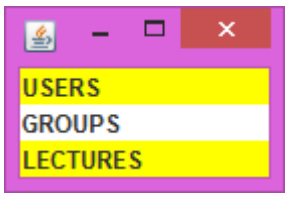

Рисунок 3.30 – количественный компаратор

На этапе количественного сравнения предоставляется первичная информация о наличии сравниваемых строк в двух базах данных. Выделение названий таблиц красным цветом, говорит о том, что все строки этой таблицы, содержащиеся в первой базе данных, отсутствуют во второй. Жёлтый цвет гово-

рит о том, что есть различия между количеством строк в обеих базах данных для этой таблицы. И зелёный, в противовес красному, сообщает о том, что во второй базе данных в указанной таблице появились записи, хотя в первой базе данных в этой таблице нет ни одной такой строки.

В приведённом выше примере видно, что есть различия в количестве строк в таблицах USERS и LECTURES, тогда как количество строк в таблице GROUPS совпадает в обеих базах данных.

Окно построкового компаратора состоит из двух основных областей, в которых построково расположены компарируемые данные. Совпадающие данные выделены белым цветом, новые – зелёным, отсутствующие – красным, для остальных типов сравнения так же отведён свой цвет.

Образец:

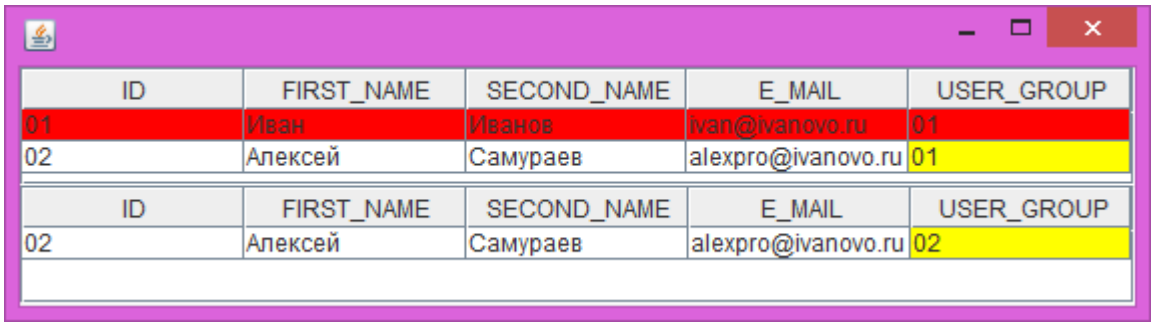

Рисунок 3.31 – построковый компаратор

В данном примере видно, что красным цветом в верхней таблице выделена строка, отсутствующая в таблице ниже. Также жёлтым цветом указаны различия в значениях ячеек колонки USER_GROUP для строки с ID = 02.

Подобное выделение цветом позволяет быстрее находить различия в данных и устранять ошибки в данных.

Выводы по главе 3

В третьей главе были проанализированы языки программирования C# и Java. Было проведено сравнение этих языков между друг другом и выделены и сильные и слабые стороны. Для разработки компаратора был выбран язык программирования Java.

Для выбранного языка программирования был проведён выбор графического пользовательского интерфейса. Анализировались платформы Abstract Window Toolkit, Swing, Standard Widget Toolkit и JavaFX. Были выделены особенности этих платформ и выбрана наиболее подходящая для реализации компаратора.

Был описан способ, которым будет производится извлечение данных. Данные будут извлекаться из базы данных с использованием Java DataBase Connectivity.

Сохранение данных в файл и их последующее извлечение будет реализовано с помощью сериализация.

Также был описан способ сравнения данных и различные критерии сравнения. Были показаны основные окна разработанного компаратора.

4 Тестирование компаратора

Тестирование компаратора проводилось во время прохождения научно-исследовательской практики в компании Неткрэкер.

4.1 Описание места внедрения компаратора

Компания Неткрэкер является мировым лидером в области создания и внедрения решений по управлению телекоммуникационными операциями. Основу успешных трансформационных решений Неткрэкер составляет обширное продуктовое и сервисное портфолио, которое включает управление взаимоотношениями с клиентами, управление продуктами, доходами и устройствами, предоставление и обеспечение качества услуг, IT-платформы, управление ресурсами и сетями. Неткрэкер является дочерней компанией корпорации NEC.

Компания Неткрэкер была основана в 1993 году и на протяжении 15 лет развивалась и достигла внушительных успехов как самостоятельная компания.

Для дальнейшего расширения сферы деятельности и совершенствования решений и услуг компании в 2008 году было принято решение об объединении с NEC Corporation. Решение корпорации NEC о консолидации программного обеспечения и сервисов по управлению телекоммуникационными операциями в руках Неткрэкер позволило существенно расширить портфолио Неткрэкер рядом сервисных платформ, инновационными приложениями и продуктами, а также профессиональными сервисами.

Объединение Неткрэкер с NEC стало уникальным событием в индустрии. Оно способствовало образованию крупнейшей компании-поставщика решений по управлению телекоммуникационными операциями с обширной базой клиентов по всему миру, богатым опытом внедрения инновационных решений и успешной историей развития.

Основные вехи:

1993 – Создание коробочного продукта NetCracker Professional.

1999 – Создание решения по управлению ресурсами сети NetCracker Inventory System.

2001 – Создание первого OSS-решения по учёту ресурсов на основе технологий Web и J2EE.

2005 – Выпуск новой версии продукта с расширенными возможностями по предоставлению инфраструктурных и OSS-решений для мобильных операторов.

2007 год – Получение компанией сертификата зрелости CMMI третьего уровня, что свидетельствует о хорошей организации внутренних процессов компании и возможности обеспечить стабильно высокое качество разработок.

2008 – Создание продуктового портфолио NetCracker Product Pyramid.

2009 – Расширение продуктового портфолио за счет нового продукта «Управление трудовыми ресурсами» (Workforce Management).

2010 – Начало экспансии в сферу управления телекоммуникационными операциями и создание портфолио Telecom Operations and Management Solutions (TOMS). Сюда входят функциональные возможности из области управления взаимоотношениями с клиентами, предоставления и обеспечения качества услуг, биллинга, онлайн-тарификации и формирования начислений, управления ресурсами, сетями и IT-платформами.

В 2011 году компания Неткрэкер приобрела бизнес Subex, специализирующийся на разработке решений по активации услуг. Благодаря этому компания Неткрэкер получила надёжную, масштабируемую технологию и добавила к своим комплексным решениям на клиентском и сервисном уровнях богатые функциональные возможности по активации сетевых ресурсов. Такие возможности быстрой активации и предоставления услуг становятся решающим фактором успеха для конвергентных провайдеров уровня Tier 1, внедряющих сети и услуги нового поколения.

В мае 2011 года компания Неткрэкер блестяще прошла сертификацию TM Forum на соответствие отраслевым стандартам. Портфолио компании Неткрэкер «Решения и сервисы по управлению телекоммуникационными операциями» (Telecom Operations and Management Solutions) получило максимально высокие баллы как полностью согласующееся с эталонными моделями индустрии.

Сертификат TM Forum подтверждает соответствие портфолио Неткрэкер «Решения и сервисы по управлению телекоммуникационными операциями» (версия 8.2) структурной модели бизнес-процессов eTOM (версия 8.0) и информационной модели SID (версия 9.0).

Программные продукты, отвечающие требованиям индустриальных стандартов, гарантируют операторам связи своевременное и успешное внедрение решений, способствуют снижению рисков и затрат, а также помогают эффективно решать актуальные бизнес-задачи. Стандарты упрощают переход на новые бизнес-модели, интеграцию бизнес-процессов разных предприятий, внедрение новых технологий и предоставление услуг нового поколения.

В июне 2011 года компания Неткрэкер была удостоена сертификата зрелости CMMI пятого уровня. Высший – пятый – уровень зрелости модели CMMI присваивается Университетом Carnegie Mellon компаниям, которые используют передовые методы управления организацией, проектами и операционными процессами, а также имеют обширный опыт в области разработки и интеграции программных продуктов. Компания Неткрэкер стала единственным подразделением корпорации NEC, удостоенным такой высокой оценки CMMI.

Сертификат зрелости CMMI пятого уровня свидетельствует о приверженности компании Неткрэкер признанным методологиям управления процессами и реализации проектов в рамках запланированных сроков и бюджета. Стандартизация процессов позволяет Неткрэкер оптимально использовать ресурсы, минимизировать количество неисправностей и необходимых доработок. Усовершенствование процессов способствует сокращению затрат на разработку, внедрение и поддержку продуктов, а также расширению функциональных возможностей продуктов, что в конечном счёте ведёт к улучшению взаимоотношений с клиентами и повышению качества их обслуживания.

В 2012 году корпорация NEC приняла решение о приобретении подразделения Global Information Management компании Convergys, лидера в области BSS-решений, для его последующей интеграции в Неткрэкер. Это объединение позволило Неткрэкер расширить экспертные знания в области BSS-решений и

обогатить своё портфолио новыми функциональными возможностями и профессиональными сервисами, расширить штат специалистов и клиентскую базу.

Проведённая в предыдущие годы консолидация ресурсов NEC, Неткрэкер, Subex и Convergys, включая программное обеспечение, сервисы, центры обработки данных и инвестиции в исследования и разработки, позволила Неткрэкер создать одно из наиболее комплексных в индустрии портфолио решений и сервисов по управлению телекоммуникационными операциями. В 2013 году компания Неткрэкер продолжила гармоничное развитие портфолио в соответствии с требованиями индустрии и расширила спектр своих предложений продуктами в области управления предприятием (Enterprise Management).

Сегодня компания Неткрэкер продолжает занимать лидирующие позиции в области создания и внедрения комплексных BSS/OSS-решений для провайдеров услуг связи, крупных предприятий и государственных учреждений.

За годы своего развития компания существенно увеличила своё мировое присутствие, штат сотрудников и количество клиентов:

- офисы и представительства Неткрэкер находятся в 38 странах мира;
- в команде компании более 5 тысяч профессионалов;
- у компании более 250 клиентов по всему миру.

Неткрэкер является лидером и в области OSS-, и BSS-решений.

Так, консалтинговая компания Gartner, специализирующаяся на рынках информационных технологий, в очередной раз объявила Неткрэкер лидером в своем отчете «Магический квадрант для поставщиков OSS-систем» (Magic Quadrant for Operations Support Systems) за 2013 год. В этом отчёте мировые поставщики OSS-решений в области предоставления и обеспечения качества услуг оцениваются на основе таких критериев, как целостность стратегии развития и способности успешной реализации проектов.

Очередным подтверждением значительных успехов Неткрэкер в области BSS-решений стало признание со стороны аналитической компании Stratecast,

которая назвала Неткрэкер лидером на мировом рынке биллинговых решений в своём отчёте Global CSP Billing 2013.

Среди прочих наград Неткрэкер можно назвать следующие:

- Лидер на мировом рынке решений по предоставлению услуг – Analysys Mason;
- Поставщик года – World Vendor Awards;
- Лучшая система поддержки – World Vendor Awards;
- Лучший поставщик инновационных решений, лучшее решение в области управления качеством обслуживания клиентов – B/OSS World Magazine;
- Лидер рынка в категории «Управление контентом и партнерскими отношениями» – Frost & Sullivan;
- Лучший поставщик телеком-решений по управлению сетями – Frost & Sullivan.

Тот факт, что многие независимые исследовательские компании называют Неткрэкер лидером в области BSS/OSS-решений, свидетельствует о том, что Неткрэкер успешно реализует свою стратегию по предоставлению комплексных решений и сервисов по управлению телекоммуникационными операциями благодаря успешной политике слияний и поглощений, крупным инвестициям в исследования и разработки, активному наращиванию клиентской базы и - самое главное - динамичному развитию портфолио.

Штаб-квартира компании Неткрэкер находится в городе Уолтем, штат Массачусетс, США.

Рисунок 4.32 – Местоположение штаб-квартиры компании Неткрэкер

Офисы компании расположены в Москве, Одессе, Сумах, Воронеже, Киеве, Самаре, Тольятти, Саратове. Штат компании составляет около 2300 сотрудников, инженерный персонал - около 60% человек.

Рисунок 4.33 – Территориальное расположение офисов компании в России, Украине и Беларуси

Структура компании Неткрэкер разбита на следующие отделы:

Engineering:

- **RNDSE:** Разработка программных систем Неткрэкер, адаптация возможностей этих систем для конкретных проектов, а также связанные со всем этим исследовательские задачи.
- **SD:** Департамент внедрения проектных решений (Solution Delivery Business Practice или сокращённо SD) занимается настройкой, установкой, наладкой и запуском решений Неткрэкер для заказчиков. Часто работа не ограничивается простой настройкой готового продукта: специалисты департамента предлагают абсолютно новые решения. Для решения этой задачи мы тесно работаем с продуктовыми командами и, конечно, с заказчиками.
- **Sales Engineering:** Подразделение Sales Engineering занимается технической поддержкой цикла продаж – сопровождение сейлз-процесса, разработка стратегии совместно с Sales и архитектурной командой, проведение презентаций, демонстраций и POC’s (Proof of Concepts).
- **IT:** Управление всем оборудованием, каналами связи и программным обеспечением, которые используются в компании, развёртывание продукта Неткрэкер
- **EDU:** Сотрудничество с высшими учебными заведениями, создание лабораторий и учебных центров Неткрэкер на базе университетов, отбор и обучение талантливых студентов, организация программ стажировки.

Customer Assurance:

- **QA:** Перед отделом контроля качества стоят две основные задачи: обеспечение качества продукта и работа с заказчиком. Задачи по работе с продуктом включают тестирование, обнаружение и исправление дефектов, а также предотвращение ошибок и дефектов.

- **CS:** Подразделение Customer Support отвечает за техническую поддержку заказчиков – исправление ошибок в продукте, анализ проблем пользователей, предложения по их решению и улучшению продукта.
- **WFM:** Работа департамента ориентирована на работу по поиску ресурсов на проекты компании.
- Прогнозирования количества человеческих ресурсов в краткосрочной перспективе.

Solution Management Support:

- **Product Management:** Одним из основных потоков прибыли для компании NetCracker являются лицензионные отчисления за право пользоваться продуктами из портфолио Неткрэкер «Решения и сервисы по управлению телекоммуникационными операциями» (Telecom Operations and Management Solutions/TOMS). Создание структуры лицензий, названий и описаний продуктов является задачей отдела управления развитием продукта (Product Management).
- **Product Marketing:** Отдел продуктового маркетинга (Product Marketing) решает задачи технического маркетинга и предпродажной поддержки. Основные направления включают создание корпуса текстов по продуктам и решениям компании для позиционирования её на рынке в целом (веб-сайт, трейд-шоу и прочее), а также поддержку конкретных задач в области продаж и маркетинга документами и предметной экспертизой (встречи с заказчиками, поддержка различных запросов в процессе тендера (RFx), взаимодействие с аналитическими агентствами и т.д.).
- **Bid Management:** Отдел технико-коммерческих предложений (Bid Management) компании Неткрэкер занимается управлением процесса подготовки технико-коммерческих предложений, включая общение с заказчиком (стадия развития бизнеса (business

development), обсуждение более конкретного объема работ (ответы на различные запросы, такие как RFi, RFx, RFq) и обсуждение контрактных условий объёма работ конкретного технического задания (SOW).

- **Customer Documentation & Training:** Отдел Customer Documentation & Training совмещает в себе два основных направления - создание и предоставление документации по продукту заказчику (внутреннему и внешнему), а также создание и проведение учебных курсов для сотрудников заказчика.

Sales:

Департамент занимается развитием бизнеса, как с точки зрения поиска новых клиентов, так и постоянной поддержки и развития существующих заказчиков. Отвечает за участие в тендерах, ведение и подписание контрактов. А также сотрудники принимают участие во всех маркетинговых мероприятиях, в которых участвует Компания, таких как отраслевые конференции и форумы, и пр.

HR & Administration:

- **HR:** поиск и найм сотрудников, оформление на работу и последующая адаптация. Участие в корпоративных процессах, взаимодействие с линейными менеджерами.
- **CDC:** Основная задача департамента – повышение эффективности обучения сотрудников. Команда CDC обеспечивает передачу знаний внутри компании; координирует разработку внутренних тренингов; ведёт и развивает базу тренеров; организовывает тренинги и другие обучающие мероприятия во всех офисах компании.
- **ADM:** Обеспечение функционирования офиса (аренда, закупки, поставщики и т.п.).

Travel Support:

Отдел обеспечивает процесс подготовки и организации всех бизнес-поездок в Компании: покупку билетов, оформление виз, бронирование жилья, страхование, информационное сопровождение и др.

KMR:

Отдел обеспечивает поддержку и развитие корпоративного портала BASS, а также создаёт и улучшает процессы обмена знаниями между проектами Компании.

Legal and Finance:

Обеспечение финансовой и юридической политики компании, взаимодействие со всеми департаментами, контроль за использованием ресурсов Компании, контроль за соблюдением всех норм действующего законодательства и внутренних политик.

Marketing:

Постоянный анализ факторов, которые влияют на деятельность Компании и оценка состояния рынка. Стратегия продвижение продуктов, организация различных бизнес-мероприятий (выставки, семинары и т.п.), обеспечение маркетинговой информацией всех подразделений компании.

Тольяттинский офис был открыт в январе 2008 года недалеко от самарского офиса. Это открытие повлекло за собой создание первого регионального кластера Тольятти-Самара. При принятии решения об открытии офиса выбор был сделан в пользу Тольятти благодаря высокому уровню развития в городе IT-технологий и возможности подготовки в ВУЗах IT-специалистов. Офис в Тольятти – один из самых удачных проектов развития региональных офисов.

Первые тольяттинские сотрудники участвовали в проекте МТС, следующим этапом была работа по интеграции и миграции данных. Таким образом, первым подразделением тольяттинского офиса стало Implementation Engineering.

Открытие и запуск производственных процессов в кратчайшие сроки стали возможными благодаря эффективной работе слаженной команды самарского офиса и поддержке московских коллег.

За первые полгода штат тольяттинского офиса увеличился в 4 раза, были основаны подразделения IT, HR и запущены все процессы.

2009 - открыты группы RND.DSI, QA и учебный центр.

2010 - открывается направление Testing Automation.

2011 - начинают развиваться направления Customer Support, GUI, DIMI.

2012 - появляются отделы Infrastructure Management, Service Fulfillment.

2013 - отмечается самый интенсивный рост числа новых сотрудников - команда тольяттинского офиса увеличилась более чем на 140 человек.

Организационная структура

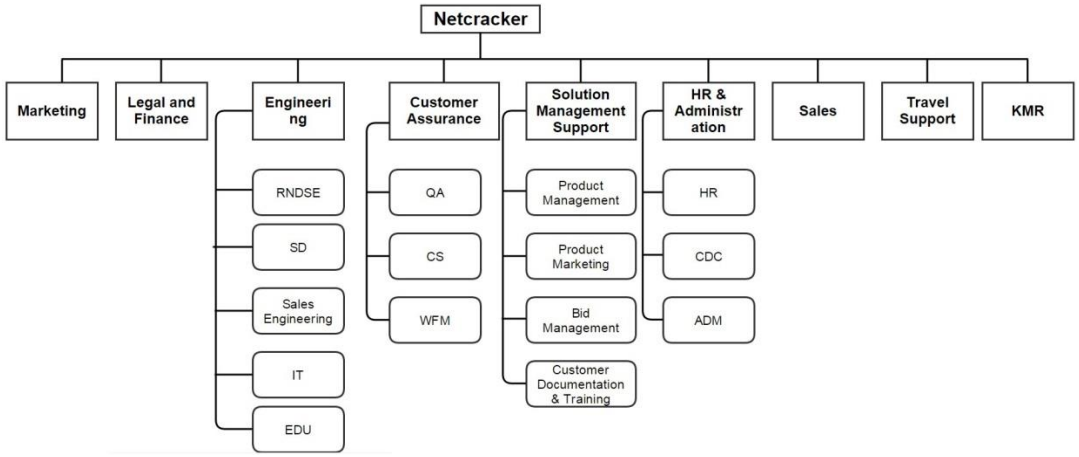

Рисунок 4.34 – Организационная структура Тольяттинского офиса

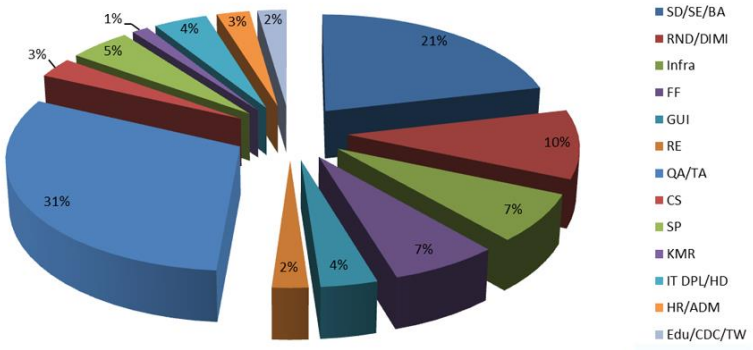

Рисунок 4.35 – Распределение сотрудников по должностям в Тольяттинском офисе

Сегодня тольяттинский офис насчитывает более 350 сотрудников. Всех сотрудников отличает энтузиазм, стремление к росту, готовность делиться своими знаниями и оказывать поддержку молодым специалистам.

В Тольятти активно развивается учебно-научный центр «Инфокоммуникационные системы», открытый в феврале 2009 г. на базе факультета «Математики и информатики» Тольяттинского государственного университета. Постепенно расширяется сотрудничество с другими вузами города, выпускающими студентов профильных специальностей. В Тольятти их семь: ТГУ, ВУиТ, ПВГУС, ТАУ, ТФ СГАУ, ТФ РГГУ, ТФ РГСУ.

На сегодняшний день обучение студентов идёт по двум направлениям: разработка программного обеспечения (DEV) и контроль качества программного обеспечения (QA).

4.2 Внедрение компаратора

Компаратор применялся для анализа данных в базе данных под управлением Oracle Database.

Oracle Database - это объектно-реляционная система, поддерживающая технологии, реализующие объектно-ориентированный подход, а именно обеспечивающая управление создания и использования баз данных [32].

Основные возможности Oracle Database:

- Real Application Cluster позволяет одному экземпляру базы данных работать на нескольких узлах, позволяя пользователю управлять нагрузкой и масштабировать систему в при необходимости.
- Automatic Storage Management обеспечивает автоматическое распределение данных между ресурсами систем хранения данных. Он повышает отказоустойчивость системы и снижает общую стоимость владения (TCO).

- Производительность. Oracle Database может автоматически управлять уровнями сервиса и тиражировать эталонные конфигурации в рамках всей сети.
- Простые средства разработки. Новый инструмент разработки приложений HTML DB позволяет простым пользователям создавать приложения для работы с базами данных в сжатые сроки.
- Самоуправление. Специальные механизмы Oracle Database позволяют пользователю самостоятельно перераспределять нагрузку на систему, оптимизировать и корректировать SQL-запросы, выявлять и прогнозировать ошибки.
- Большие базы данных. Текущий максимальный размер экземпляра базы данных Oracle может достигать до 8 экзабайт.
- Дешёвые серверные системы. Oracle Database может использовать недорогие однопроцессорные компьютеры или модульные системы из «серверов-лезвий».
- Реализована поддержка переносимых табличных пространств, система управления потоками данных Oracle Streams и модель распределённых SQL-запросов. Для переноса существующих баз данных в среду Grid в них не требуется вносить изменений, что позволяет быстро приступать к использованию всех преимуществ Oracle Database [42].

В ходе исследования базы данных, содержащей несколько тысяч таблиц, нами было выделено около трёхсот таблиц, попадающих под условия компарации.

Так как основной особенностью нашего компаратора является извлечение логически связанных данных из таблиц, то в качестве примера приведём базовую схему логических связей с основной таблицей Customer HP Life:

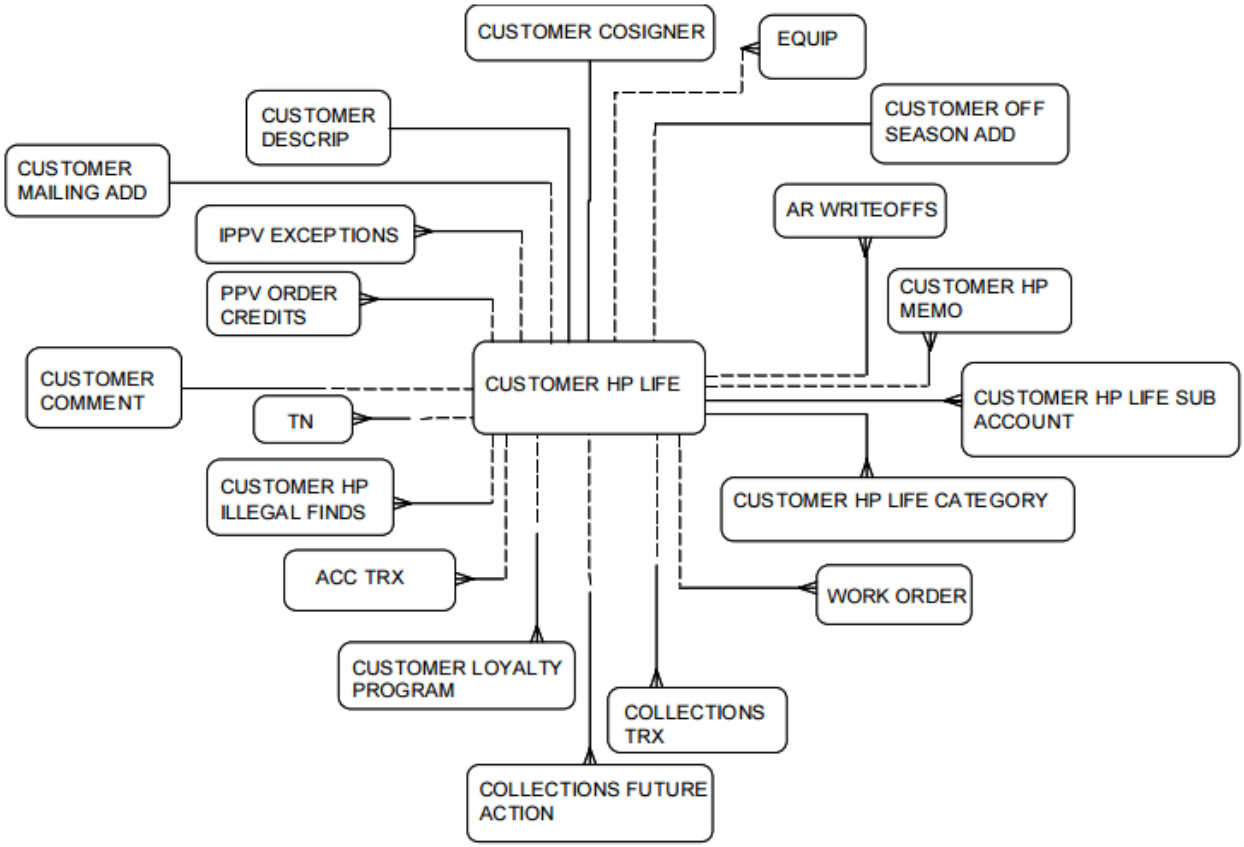

Рисунок 4.36 – Логическая связь между таблицами

Ниже приведён общий список всех выбранных для компарации таблиц: ArBatchDetail, ArBatchHeader, ArBatchProdDetail, ArTrx, ArTrxCodesHist, ArTrxComments, ArTrxCommentsArchive, ArTrxDeferralCodes, ArTrxDeferralCodesDel, ArTrxDetail, ArTrxPayment, ArTrxPaymentArchive, ArTrxPpvInfo, ArTrxProvision, ArTrxT, ArTrxTaxes, ArTrxTransferDetail, ArTrxWriteoffs, AuditSeqLog, BlkHierarchyLevel, BlockCustomer, BlockCustomerContract, CollectionsFutureAction, CollectionsTrx, CollectUsageLog, CollTransferFileInfo, CollTransferInvoiceDtl, ContractCategory, ContractCategoryHist, ContractItem, ContractItemHist, CorrectiveAdjustmentLink, CreditAllocation, CreditAllocationDtl, CustCancelInvToAgency, CustContProdRateHist, CustContractProdRate, CustDemographInfo, CustDirectoryListHist, CustHpDebitAuthHist, CustHpLifeCategoryHist, CustHpLifeHistory, CustHpOccurrenceHist, CustHpProductHist, CustHpSubAccountHist, CustHpSvcFeatAttrHist, CustHpSvcFeatureAttrib, CustHpSvcFeatureHist, CustHpUsageSummary, CustHpUsageSummaryPlan, CustLineCir-

cuitTnHist, CustLoyaltyProgDetail, CustMaxUserNum, CustomerAcctDeposits, CustomerBillTo, CustomerBmail, CustomerComment, CustomerContacts, CustomerContactsHist, CustomerContactsTmp, CustomerContracts, CustomerContractsHist, CustomerCosigner, CustomerCoupon, CustomerDescrHistory, CustomerDescrip, CustomerDescripTmp, CustomerDirectoryListing, CustomerFulFillment, CustomerHpDebitAuth, CustomerHpDms, CustomerHpExpiredPromo, CustomerHpIllegalFinds, CustomerHpLife, CustomerHpLifeCategory, CustomerHpMemo, CustomerHpMemoHistory, CustomerHpOccurrence, CustomerHpProduct, CustomerHpSubAccount, CustomerHpSvc, CustomerHpSvcFeature, CustomerHpSvcHistory, CustomerLoyaltyProgram, CustomerMailbox, CustomerMailboxTaxes, CustomerMailingAdd, CustomerOffSeasonAdd, CustomerTelephoneLine, CustomerTelLineCircuit, CustomerUsers, CustPaymentToAgency, CustTelephoneLineHist, CustTelLineCircuitHist, CustTelLineCircuitTn, CustUsersHist, CycBillProcFailedHist, DebitAllocationDetail, DebitStmtAllocation, EmailAddresses, Equip, EquipCallback, EquipEMM, EquipExtBits, EquipExtension, EquipTrx, EventDispute, ExtBlacklistInput, ExtWaiveFlagsInput, FinResult, FinResult1, FinResultf, FinResulth, FinXTemp, FinXTemp1, FinXTempf, FinXTemph, GlJnlDetail, GnvAdvancPayCommEvent, GnvCommissionEvents, HierarchyRelations, HpProductChargesHistory, InstallmentPlan, IppvExceptions, KDGBlkCustomerContract, LockboxPaymentReference, LockboxPreCheck, MvApDeptorAcctSummary, Nsf, PendingCreditPreOrder, PendingCredits, PhoneNumber, PpvBucketSummary, PpvOrders, PpvOrdersCredits, RepOiAccountHist, RepOiBlacklist, RepOiCpdAccount, RepOiCustomer, RepOiExceptions, RepOiGlInput, RepOiJournal, RepOiMailbox, RepOiSubAccountHist, SapAddress, SapAddressTmp, SapBankDetails, SapBusinessPartner, SapContractAccount, SapProcessingRequest, SapProcessingRequestTmp, SapSendTblInfoTmp, SubAccountAgbVersion, SubAccountLiabilityHist, SubAccountUsageBillSeq, TaxExemptHistory, TemporaryCustHpProduct, TempStmtReqest, TempStmtReqestSave, TerminatedAccounts, TmContactDetails, Tn, TnHistory, TnReservations, TnReservationsHist, UserRecentUsedEntity, WorkOrder, WorkOrderActivity, WorkOrderActivityDependencies, WorkOrderAc-

tivityDependenciesHist, WorkOrderActivityHist, WorkOrderAttributes, WorkOrderCloseByBatch, WorkOrderComplant, WorkOrderComplantDetail, WorkOrderComplantDetailHist, WorkOrderComplantHist, WorkOrderContract, WorkOrderContractCategory, WorkOrderContractCategoryHist, WorkOrderContractHist, WorkOrderContractItem, WorkOrderContractItemHist, WorkOrderContractProductRates, WorkOrderContractProductRatesHist, WorkOrderCoupons, WorkOrderCouponsHist, WorkOrderDirectoryListing, WorkOrderFulfillment, WorkOrderFulfillmentHist, WorkOrderHistory, WorkOrderItemMdu, WorkOrderItemMduHist, WorkOrderItemMduUnit, WorkOrderItemMduUnitHist, WorkOrderJobs, WorkOrderJobsHist, WorkOrderLineCircuitTnHist, WorkOrderNotCompletedReason, WorkOrderOccurrence, WorkOrderOccurrenceHist, WorkOrderProducts, WorkOrderProductsHist, WorkOrderQueueIrbNotAvail WorkOrderQuickCloseDetail, WorkOrderQuickCloseHeader, WorkOrderScheduleCalendar, WorkOrderServiceFeatures, WorkOrderServiceFeaturesHist, WorkOrderServices, WorkOrderServicesHistory, WorkOrderSvcFeatureAttributes, WorkOrderSvcFeatureAttributesHist, WorkOrderTasks, WorkOrderTaskSheduleHistory, WorkOrderTasksHist, WorkOrderTelephoneLine, WorkOrderTelephoneLineCircuit, WorkOrderTelephoneLineHist, WorkOrderTelLineCircuitTn, WorkOrderUser, WorkOrderUserHist, WorkOrderUsers, WorkOrderUsersHist, AccTrx, AccTrxHistory, ApDebtorAcctSummary, ApiFailures, ApiFailuresPlanDetails.

4.3 Анализ результатов внедрения

Основной гипотезой внедрения компаратора было уменьшение затрачиваемого времени на поиск различий в данных, то критерием анализа результата тестирования компаратора будет время.

На написание загрузчика данных из базы данных продукта компании Неткрэкер потребовалась рабочая неделя и усилия трёх человек. В неделе 40 рабочих часов, умножаем их на трёх человек и получаем 120 часов.

За время прохождения практики (6 недель), произошло две ситуации, в которых возникла необходимость сравнения двух баз данных. Обычно нахож-

дение нужных различий между данными занимают около часа, однако с использованием компаратора затрачиваемое время сократилось до получаса. На графике ниже представлено сравнение затраченного времени с использованием компаратора и без него.

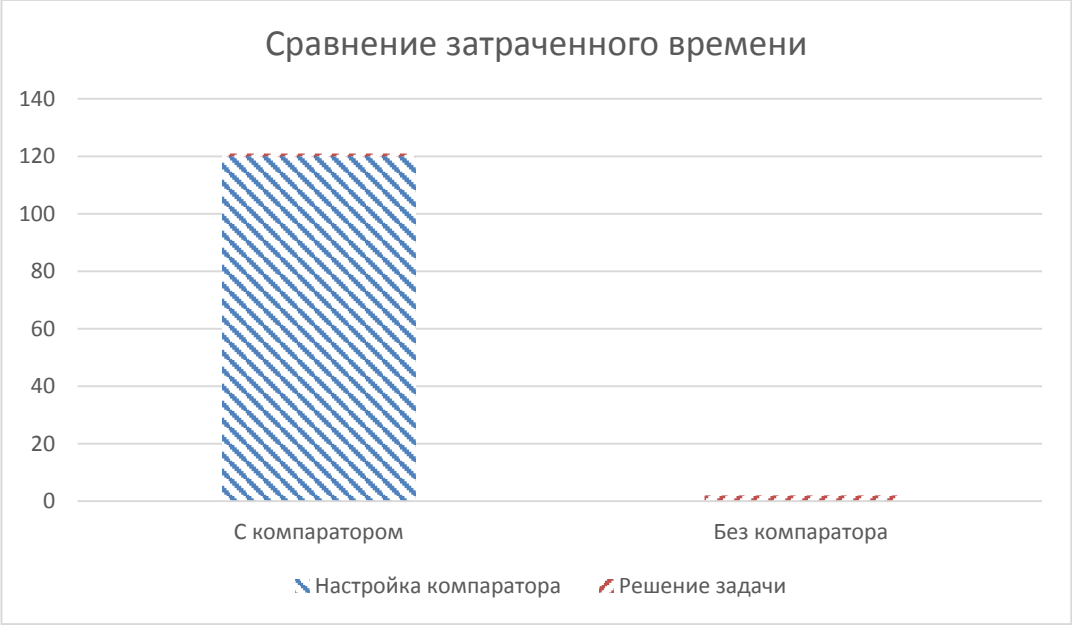

Рисунок 4.37 – сравнение затраченного времени для решения 2 задач

На решение двух задач и настройку компаратора потребовался 121 час, из которых 120 часов ушло на настройку и ещё по полчаса на решение каждой задачи. На решение задач без компаратора ушло бы всего 2 часа.

Согласно полученным данным, внедрение компаратора для всего двух задач не только не ускоряет процесс, но и значительно его замедляет.

Попробуем теоретически увеличить число задач до 10.

Рисунок 4.38 – сравнение затраченного времени для решения 10 задач

Как видно из графика, при 10 задачах применение компаратора всё ещё не оправдывает себя. Попробуем увеличить число задач до 100.

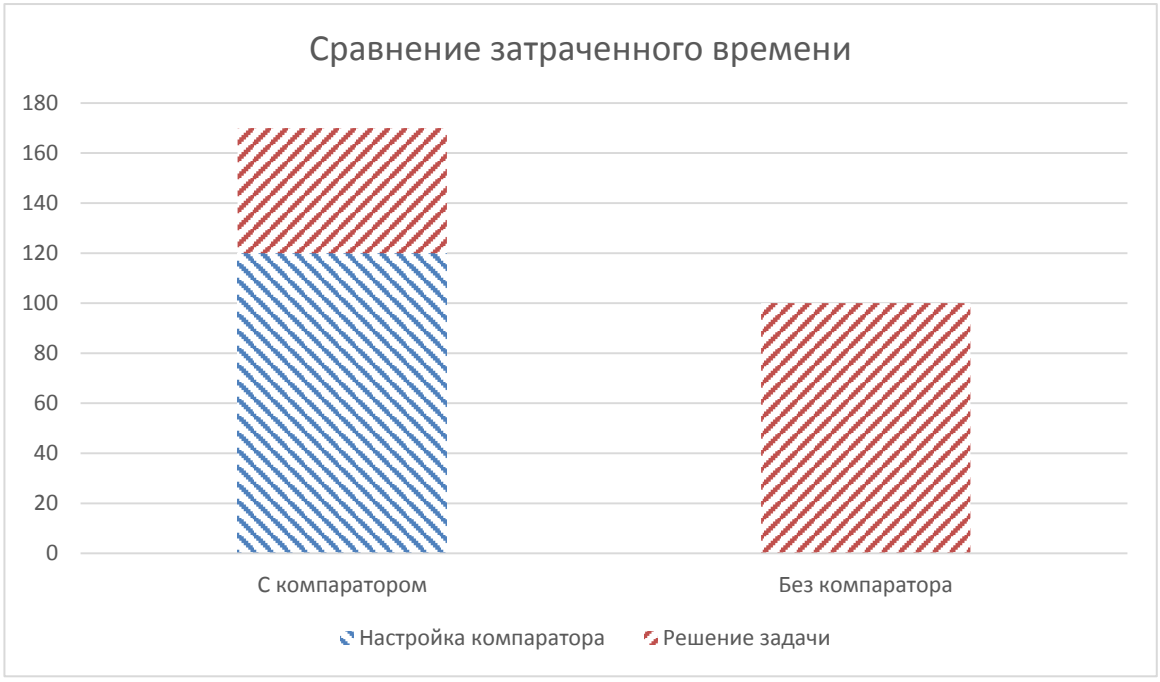

Рисунок 4.39 – сравнение затраченного времени для решения 100 задач

И даже при 100 задачах наш компаратор скорее вредит, чем приносит пользу.

Однако мы можем вычислить, сколько потребуется задач, чтобы внедрение компаратора начинало приносить пользу. Для этого применим формулу:

$$120 \text{ часов} + \frac{x}{2} < x,$$

где x- число задач.

Решив это неравенство, мы получим x > 240.

Именно столько задач на нахождение различий в данных необходимо получить, чтобы время, затраченное на внедрение компаратора, окупилось, и он начал приносить пользу. Проект с 240 проблемами в данных вряд ли можно назвать успешным.

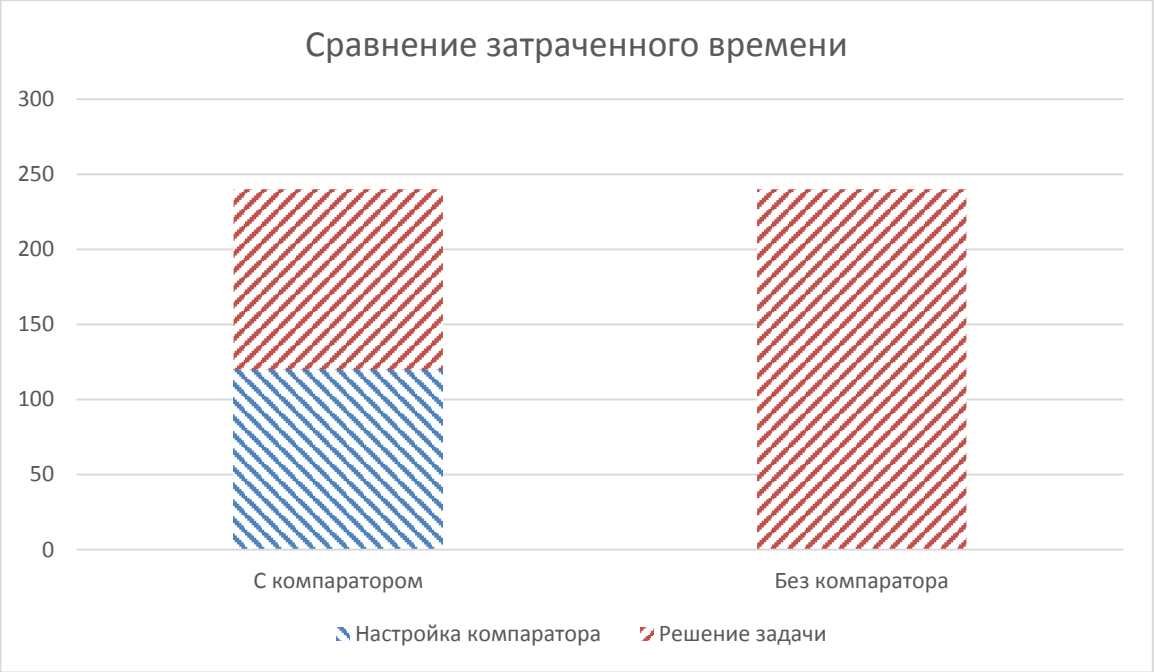

Рисунок 4.40 – сравнение затраченного времени для решения 240 задач

Таким образом, мы можем сделать вывод, что выбранный нами подход к получению данных требует пересмотра для того, чтобы от компаратора была польза.

Наш изначальный подход к извлечению данных основан на сериализации дерева классов, являющихся отображением логической структуры базы данных. Для подготовки компаратора к работе с определённой базой данных, программисту необходимо выполнить следующий порядок действий:

1. Проанализировать базу данных и выделить те таблицы, которые необходимо сравнивать.
2. Написать древовидную структуру классов, соответствующую выбранным таблицам из базы данных и иерархически повторяющую логическую связь между этими таблицами.
3. Написать загрузчик для каждой таблицы, который бы помещал данные из базы данных в созданную программистом структуру.

Второй и третий пункты, написание структуры и загрузчика, занимают очень много времени. Поэтому было принято решение изменить способ получения данных.

Можно отказаться от сохранения логической структуры данных в пользу сокращения затрачиваемого на настройку времени. Для этого внутреннюю логическую иерархию классов мы заменим на двумерный массив данных, который будет содержать в себе всю информацию о таблицах, столбцах, и самих данных.

Шаблон, используемый для хранения данных, выглядит следующим образом:

1 строка: название таблицы, название 1’ой колонки, название 2’ой колонки … название n’ой колонки

2 строка: название таблицы, данные ячейки из 1’ой колонки, данные ячейки из 2’ой колонки … данные ячейки из n’ой колонки

3 строка: название таблицы, название 1’ой колонки, название 2’ой колонки … название n’ой колонки

4 строка: название таблицы, данные ячейки из 1’ой колонки, данные ячейки из 2’ой колонки … данные ячейки из n’ой колонки

И так далее по вышеописанному шаблону записываются все извлечённые данные. Сам же загрузчик данных тоже изменится. Теперь программисту, вместо написания отдельных загрузчиков для каждой таблицы, необходимо составить список таблиц с ключевыми ячейками и связями между ними для того, чтобы компаратор сам занялся загрузкой данных в массив.

Шаблон загрузчика выглядит следующим образом:

1 строка: Главная таблица, ключевая ячейка

2 строка: Главная таблица, связующая ячейка - первая дочерняя таблица, ключевая ячейка

3 строка: первая дочерняя таблица, связующая ячейка - вторая дочерняя таблица, ключевая ячейка

4 строка: первая дочерняя таблица, связующая ячейка - третья дочерняя таблица, ключевая ячейка

И так далее по вышеописанному шаблону записывается вся логическая структура выбранных в базе данных таблиц.

Такой подход позволяет значительно сократить время, затрачиваемое на первичную настройку компаратора за счёт отсутствия необходимости писать множество классов описывающих каждую таблицу в базе данных. Также теперь не нужно писать отдельный загрузчик для каждой таблицы с указанием всех колонок, достаточно указать лишь ключевые и связующие колонки таблиц.

Данный подход позволяет сократить затрачиваемое время на настройку для каждой таблицы с 40 минут (120 часов / 300 таблиц) до 30 секунд, так как теперь от программиста не требуется писать отдельный загрузчик и класс, описывающий структуру для каждой таблицы. Теперь достаточно лишь написать пару строк, содержащих имя таблицы, ключевой столбец и связующий столбец, чтобы компаратор смог получить нужные данные.

Теперь рассчитаем затрачиваемое время на настройку компаратора для 300 таблиц и выполнение двух задач на нахождение двух задач на нахождение различий в данных.

0.5 минут × 300 таблиц = 150 минут

150 минут + (30 минут × 2 задачи) = 210 минут

Теперь представим сравнительный график.

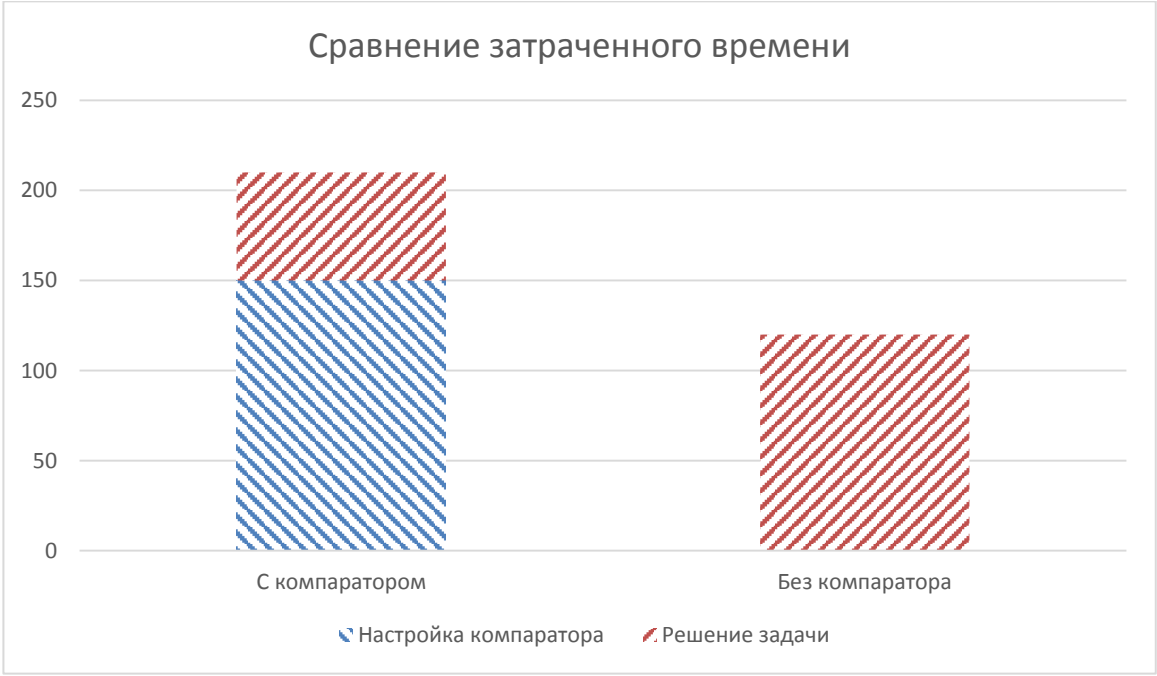

Рисунок 4.41 – сравнение затраченного времени для решения 2 задач

Согласно представленному графику для двух задач, мы видим, что применение компаратора для двух задач всё ещё неактуально и тормозит процесс выполнения задач. Однако, давайте рассчитаем, при каком количестве задач, компаратор начнёт приносить пользу:

$$150 \text{ минут} + 30x \ < \ 60x,$$

где x- число задач.

При x = 5 затраченное время с применением компаратора и без него сравнится.

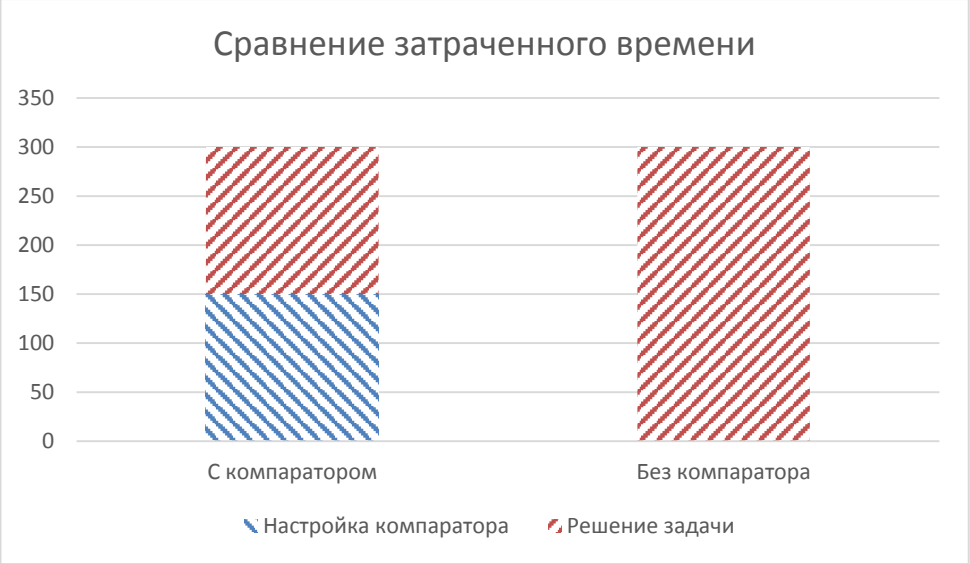

Рисунок 4.42 – сравнение затраченного времени для решения 5 задач

А при x> 5 применение компаратора начнёт оправдывать себя.

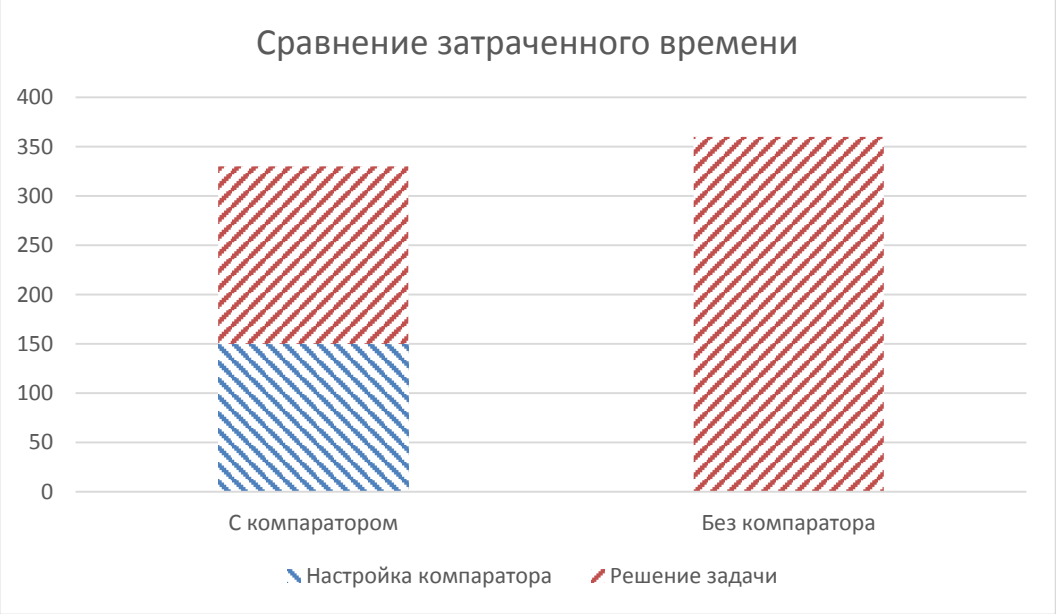

Рисунок 4.43 – сравнение затраченного времени для решения 6 задач

Далее, при 10 задачах разница становится ощутимой.

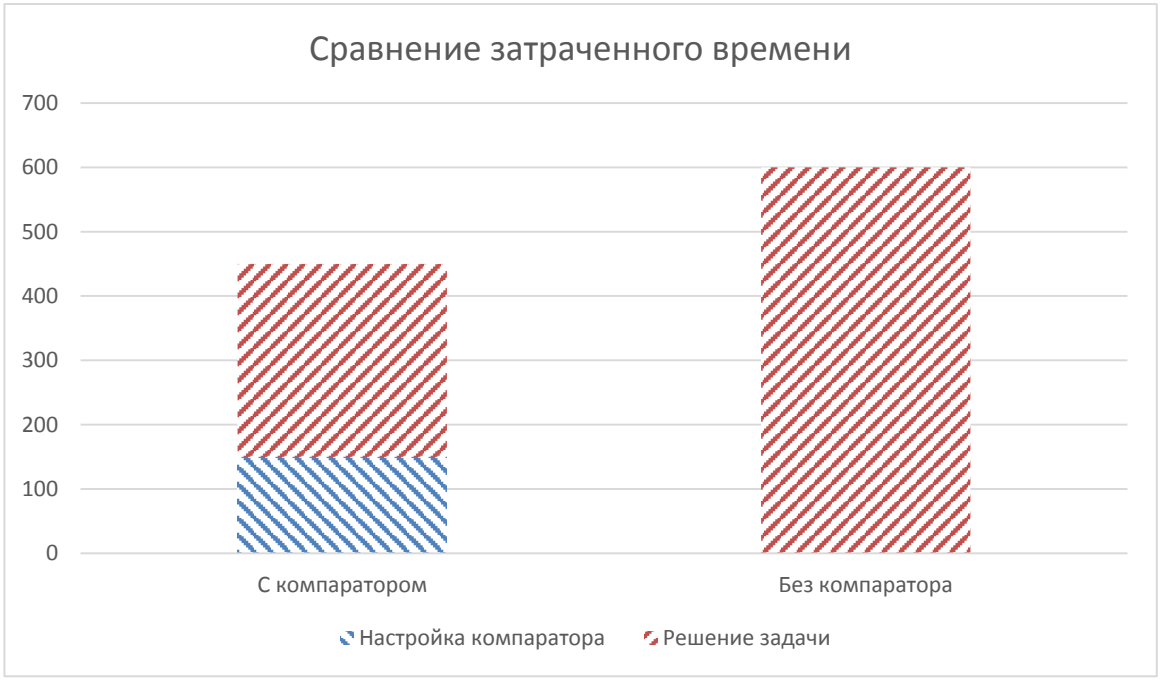

Рисунок 4.44 – сравнение затраченного времени для решения 10 задач

Теперь же, сравним затраченное время для >240 задач, выделенных нами для предыдущей версии компаратора.

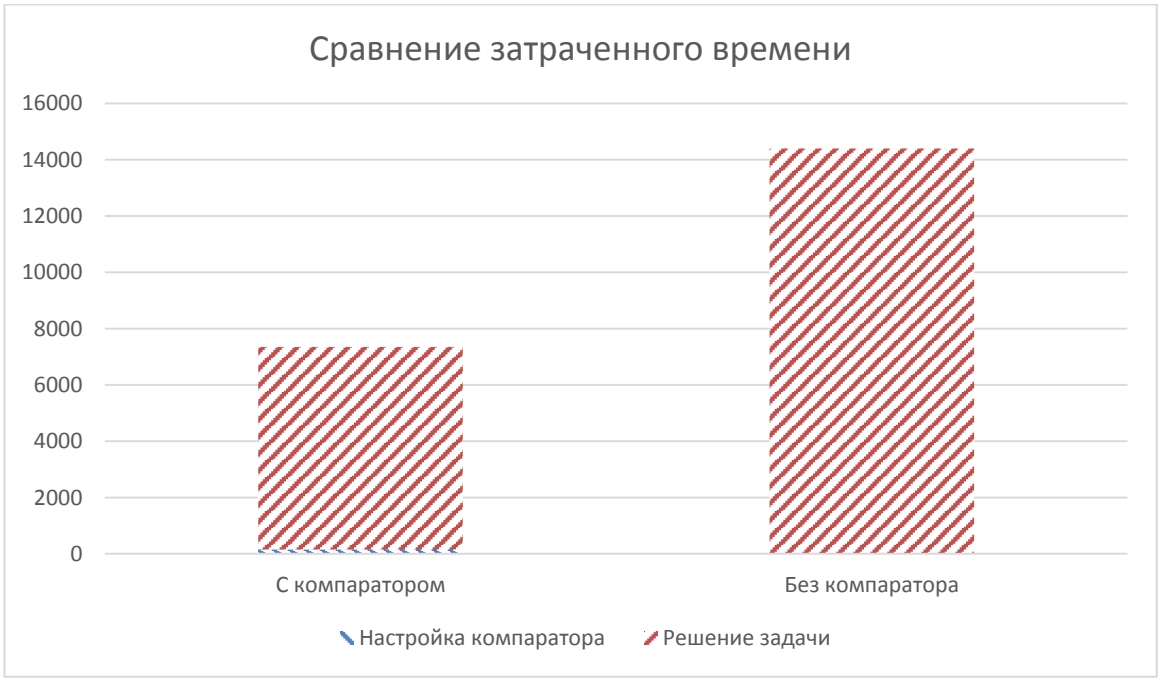

Рисунок 4.45 – сравнение затраченного времени для решения 240 задач

Как видим, за счёт сокращения времени настройки компаратора, нам удалось добиться значительного сокращения затрачиваемого времени на саму компарацию.

Для наглядности, сравним оба метода между собой. Измерение будет проводиться в часах настройки.

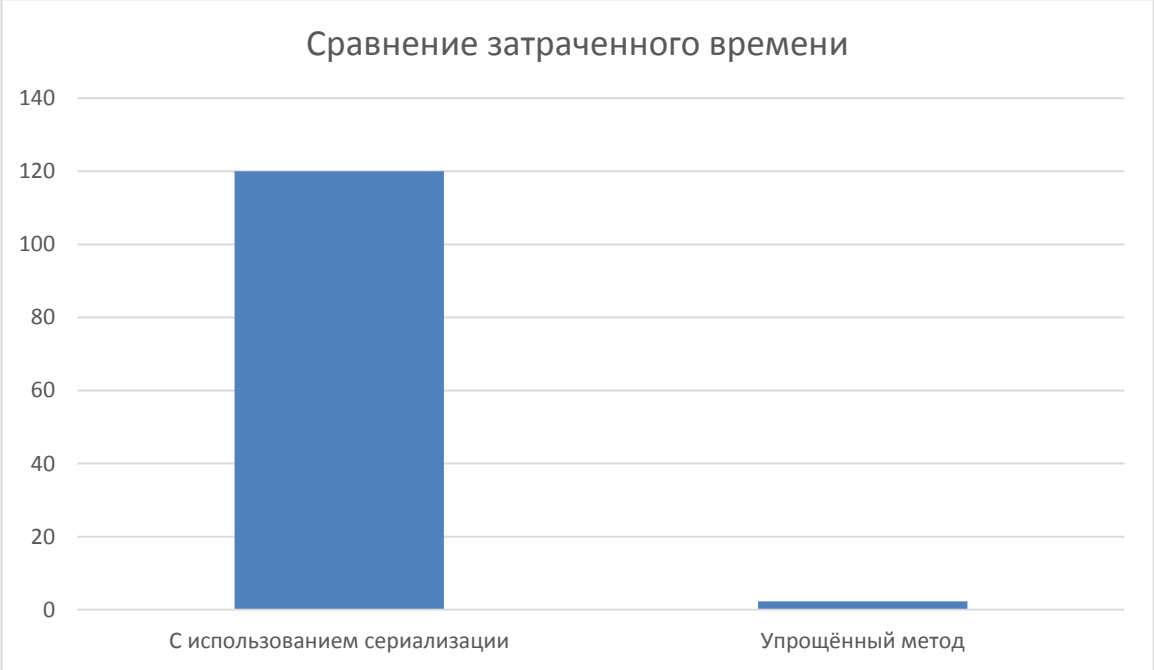

Рисунок 4.46 – сравнение затраченного времени для решения 240 задач

При использовании сериализации на настройку у нас ушло 120 часов. С упрощённым методом 2,5 часа. Таким образом, за счёт упрощения метода хранения и загрузки данных, удалось сократить время настройки в 48 раз.

Из чего мы можем сделать вывод, что при подобной реализации компаратора за счёт упрощения процедуры получения и хранения данных компаратор баз данных может быть применён для ускорения процедуры сравнения данных в двух базах данных, а следовательно, ускорения поиска ошибок в данных в случаях, когда есть исправный образец данных.

Выводы по главе 4

В четвёртой главе была описана компания, в которой производилось тестирование компаратора. Было произведено поверхностное описание базы данных, на которой компаратор был протестирован.

Результаты тестирования были проанализированы, и был сделан вывод, что подход к получению данных для компаратора был выбран неверно. После этого, подход был изменён на упрощённый, что позволило компаратору ускорить выполнение заданий по сравнению данных, при наличии более пяти задач по сравнению.

Заключение

Не смотря на широкую распространённость компараторов баз данных для синхронизации и восстановления данных, компараторов, основанных на логическом анализе компарируемых данных, в широком распространении нет. Исследование в этой области может позволить сэкономить время разработчику требуемое на поиск решения какой-либо проблемы, связанной с несоответствием данных.

В ходе данной работы были выделены основные компоненты разрабатываемого приложения и связь между ними. Также был определён минимальный набор функций приложения. И выбран язык программирования, на котором приложение было написано.

Были разработаны основные модули комаратора баз данных. Был разработан модуль извлечения данных, описана база данных, под которую разрабатывается компаратор, описана последовательность компарации данных и приведён образец компарации данных.

Также было проведено тестирование компаратора, результаты которого говорят о том, что подход к извлечению данных был выбран неверно. В результате чего на настройку компаратора затрачивается слишком много времени.

Затем, подход к получению данных был упрощён, что позволило сделать его применения полезным при наличии >5 задач по сравнению данных.

Таким образом были достигнуты цели исследования и подтверждена гипотеза о пользе компаратора.

Список используемой литературы

Нормативно-правовые акты

1. Гражданский Кодекс Российской Федерации. Часть 1 [Электронный ресурс] : федер. закон от 30 нояб. 1994 г. № 51-ФЗ : (в ред. от 30 дек. 2012 г.; с изм. и доп. от 2 янв. 2013 г.)

Учебники и учебные пособия

2. Беломестных В. А. и др. Многоточечная измерительная система с коммутатором на полевых транзисторах //Автометрия. – 1970. – Т. 2. – С. 37-47.
3. Большая советская энциклопедия : в 30 т. / гл. ред. А. М. Прохоров. — 3-е изд. — М.: Советская энциклопедия, 1969—1978.
4. Бородакий Ю. В., Лободинский Ю. Г. Эволюция информационных систем (современное состояние и перспективы). — М.: Горячая линия - Телеком, 2011. — 368 с. — 1000 экз. — ISBN 978-5-9912-0199-5.
5. Бройдо В. Л. Вычислительные системы, сети и телекоммуникации: [учебное пособие по специальностям "Прикладная информатика", "Информационные системы в экономике"]. – Издательский дом «Питер», 2011.
6. Верников Г. Основные методологии обследования организаций. Стандарт IDEF0 – 2000.
7. Волгин Л. И. Синтез и схемотехника аналоговых электронных средств в элементном базисе усилителей и повторителей тока / Л. И. Волгин, А. И. Зарукин ; Федер. агентство по образованию, Ульянов. гос. техн. ун-т. - Ульяновск : УлГТУ, 2005. - 200 с. : ил.; 20 см. - Библиогр.: с. 180-195. - 100 экз. - ISBN 5-89146-749-6
8. Гиззатуллина А. Р. Языки программирования C# И Java. Сравнение. – 2014. – С. 137.

9. Гладун А. Я., Несен М. В., Штонда В. Н. Интеллектуальные агентно-ориентированные услуги, базирующиеся на платформах интеллектуальных сетей. – 2004.
10. Жданов С., Соболева М., Алфимова А. Информационные системы. – Litres, 2016.
11. Жежеленко И. В. и др. Электромагнитная совместимость потребителей. – 2012.
12. Калянов Г. Н. Консалтинг при автоматизации предприятий: подходы, методы, средства //М.: СИНТЕГ. – 1997. – С. 203.
13. Камкин А. С., Чупилко М. М. Механизмы поддержки функционального тестирования моделей аппаратуры на разных уровнях абстракции //Труды Института системного программирования РАН. – 2011. – Т. 20.
14. Комаров П. С., Градусов А. Б. База данных для диспетчера станции техобслуживания автомобилей.
15. Корнев Е. А. Схемотехника цифровых, аналого-цифровых и цифро-аналоговых устройств: Учебное пособие //Оренбург: ГОУ ОГУ. – 2005.
16. Котов О. М. Язык C#: краткое описание и введение в технологии программирования: учебное пособие. – 2014.
17. Маклаков С. В. Моделирование бизнес-процессов //М.: Диалог. – 2002.
18. Молоков К. А. Основы информатики и программирование под Windows. Учебное пособие. – Издательство Проспект, 2015.
19. Монахов В. В. Язык программирования Java и среда NetBeans. 3-е изд. – БХВ-Петербург, 2012.
20. Мухин В. Управление интеллектуальной собственностью: учебник для вузов. – Litres, 2013.

21. Наумов А. С., Шалыто А. А. Объектно-ориентированное программирование с явным выделением состояний //СПб.: СПбГУ ИТМО. – 2004.
22. Норенков И. П., Трудоношин В. А. Телекоммуникационные технологии и сети. – 2000.
23. Операционные усилители и компараторы : Справочник. - М. : Додэка-XXI, 2001. - 560 с. : ил. - (Интегральные микросхемы). - ISBN 5-94120-004-8
24. Пугачёв С. В. Разработка приложений для Windows 8 на языке C#. – БХВ-Петербург, 2013.
25. Рогаткин Ю. Б. Опыт разработки и методология проектирования смешанных МЭС на примере быстродействующего 10-разрядного АЦП //В настоящем сборнике. – 2005.
26. Рыбина Г. В. Обучающие интегрированные экспертные системы: некоторые итоги и перспективы //Искусственный интеллект и принятие решений. – 2008. – Т. 1. – С. 22-46.
27. Рыженко А. В. Объектно-ориентированное программирование: Учебно-методический комплекс по дисциплине для специальности 010501 – "Прикладная математика и информатика". – 2007.
28. Салмре И. Программирование мобильных устройств на платформе. Net Compact Framework:[разработка программного обеспечения для мобильных устройств]. – Издательский дом Вильямс, 2006.
29. Соловьев Н., Чернопрудова Е. Системы автоматизации разработки программного обеспечения. – Litres, 2016.
30. Усталов Д. Распределённые объектные технологии. – 2015.
31. Хабибуллин И. Ш. Java 7 (4-е изд.). – БХВ-Петербург, 2012.
32. Хабибуллин И. Ш. Создание распределенных приложений на Java 2. – БХВ-Петербург, 2003.

Периодические издания

33. Алексеев В. Приложения пользователя в GSM/GPRS модулях ведущих мировых производителей //Компоненты и технологии. – 2005. – №. 2.
34. Брескаленко А. А. Сериализация как инструмент сохранения и передачи данных //Международный научно-исследовательский журнал. – 2012. – №. 7-1 (7).
35. Вахрушева М. Ю., Ершов Д. В. Финансовый анализ в менеджменте: прикладной аспект //Современные тенденции в экономике и управлении: новый взгляд. – 2015. – №. 34.

Электронные ресурсы

36. Большая энциклопедия Кирилла и Мефодия 2000 [Электронный ресурс]. – Москва: Кирилл и Мефодий : Рос. энцикл., 2000.
37. Вязовик Н., Жилин Е. Программирование на Java //М.: Интуит. ру. – 2003.
38. Дж. Стивен Перри. Введение в Java-программирование: Часть 2. Конструкции реальных приложений [Электронный ресурс] – 2011. - Режим доступа: http://www.ibm.com/developerworks/ru/edu/j-introtojava2
39. Документация Business Studio [Электронный ресурс] – 2016. – Режим доступа: http://www.businessstudio.ru/wiki
40. Microsoft Developer Network [Электронный ресурс] – 2014. - Режим доступа: http://msdn.microsoft.com

Литература на иностранных языках

41. Ahmadi R., Cami B. R., Hassanpour H. Article: Automatic Data Migration between Two Databases with Different Structure} //International Journal of Applied. – Т. 3. – С. 13-28.
42. Andersen L., Lead S. JDBC™ 4.2 Specification. – 2014.

43. Buchheit M. et al. Subsumption between queries to object-oriented databases. – Springer Berlin Heidelberg, 1994. – С. 15-22.
44. Carpenter B. et al. Object serialization for marshalling data in a Java interface to MPI //Proceedings of the ACM 1999 conference on Java Grande. – ACM, 1999. – С. 66-71.
45. Chang Y., Raschid L., Dorr B. Transforming queries from a relational schema to an object schema: A prototype based on F-logic. – Springer Berlin Heidelberg, 1994. – С. 154-163.
46. Clarke J., Connors J., Bruno E. J. JavaFX: Developing Rich Internet Applications. – Pearson Education, 2009.
47. Gruntz, Dominik C# and Java: The Smart Distinctions // Journal of Object Technology. — 2002. — Fasc. November-December. — No. vol. 1, no. 5. — P. 163-176.
48. Loney K. Oracle database 10g: the complete reference. – McGraw-Hill/Osborne, 2004.
49. Northover S., Wilson M. Swt: the standard widget toolkit, volume 1. – Addison-Wesley Professional, 2004.
50. Target T. C., Date L. Comparison of data modeling tools //JSON 195 XML 204 Web Ontology Language 216 Resource Description Framework 226. – С. 100.

Приложение

Код класса Main

```
import javax.swing.*;
import javax.swing.table.DefaultTableCellRenderer;
import javax.swing.table.TableCellRenderer;
import java.awt.*;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.util.Arrays;

public class Main {
   public static void main(String args[ ])
   {
      JFrame f = new JFrame();
      f.setDefaultCloseOperation (JFrame.DISPOSE_ON_CLOSE);
      JList list = createJList();
      f.add(list);
      f.pack();
      f.setVisible(true);
   }

   static private JList createJList ()
   {
      JList list = new JList();
      DefaultListModel listModel = new DefaultListModel();
      Tables tables = new Tables();
      int itr = 0;
      TableComparator comparator = new TableComparator();
      TableCompareResult tablesRes = comparator.compareTables(tables);
      while (!(tablesRes.getValue(itr) == null))
      {
         TableCompareResultLine line = tablesRes.getValue(itr);
         if(!(line.tableName== null)) {
            listModel.addElement(line);
```

```
            }
            itr++;
        }

        list.setModel(listModel);
        //отрисовка различных цветов
        list.setCellRenderer(new DefaultListCellRenderer() {

            @Override
            public Component getListCellRendererComponent(JList list, Object value, int index,
                                                          boolean isSelected, boolean cellHasFocus) {
                Component c = super.getListCellRendererComponent(list, value, index, isSelected,
cellHasFocus);
                if (value instanceof TableCompareResultLine) {
                    TableCompareResultLine val = (TableCompareResultLine) value;
                    setText(val.tableName);
                    setBackground(val.color);
                    if (isSelected) {
                        setBackground(getBackground().darker());
                    }

                } else {
                    setText("whodat?");
                }
                /*if (value instanceof User) {
                    User nextUser = (User) value;
                    setText(nextUser.name);
                    if (nextUser.loggedIn) {
                        setBackground(Color.GREEN);
                    } else {
                        setBackground(Color.RED);
                    }
                    if (isSelected) {
                        setBackground(getBackground().darker());
                    }
```

```
            } else {
                setText("whodat?");
            }
            */
            return c;
        }

    });
    //обработка двойного нажатия
    list.addMouseListener(new MouseAdapter() {
        public void mouseClicked(MouseEvent evt) {
            JList list = (JList)evt.getSource();
            if (evt.getClickCount() == 2) {

                // Double-click detected
                int index = list.locationToIndex(evt.getPoint());
                System.out.println("index2: "+tablesRes.getValue(index).tableName);
                showRowsFrame(tables,tablesRes.getValue(index));
            } else if (evt.getClickCount() == 3) {

                // Triple-click detected
                int index = list.locationToIndex(evt.getPoint());
                System.out.println("index3: "+tablesRes.getValue(index).tableName);
            }
        }
    });
    return list;
}

public static void showRowsFrame(Tables tables, TableCompareResultLine line)
{
    JFrame f = new JFrame();
    f.setDefaultCloseOperation (JFrame.DISPOSE_ON_CLOSE);
    SingleTable data1 = new SingleTable(tables.firstBaseNames, tables.firstBaseValues);
    SingleTable data2 = new SingleTable(tables.secondBaseNames, tables.secondBaseValues);
```

```
    JTable table1 = createJTable(data1, data2, line, false);
    JScrollPane scrollPane1 = new JScrollPane(table1);
    JPanel panel1 = new JPanel();
    panel1.setLayout(new GridLayout(2,1));
    panel1.add(scrollPane1);

    JTable table2 = createJTable(data2, data1, line, true);
    JScrollPane scrollPane2 = new JScrollPane(table2);
    panel1.add(scrollPane2);
    f.add(panel1);
    f.pack();
    f.setSize(1000,600);
    f.setVisible(true);
  }

  static private JTable createJTable (SingleTable firstTables, SingleTable secondTables, Table-
CompareResultLine line, boolean secondFlag)
  {
    int columns = 0;
    int firstRows = 0;
    int secondRows = 0;

    //получение числа строк и столбцов для первой таблицы
    int itrColumnNames = 0;
    while (!(firstTables.BaseNames[itrColumnNames][0] == null) && !(firstTa-
bles.BaseNames[itrColumnNames][0] == ""))
    {
      if(firstTables.BaseNames[itrColumnNames][0] == line.tableName)
      {
        // получение числа строк
        if(columns == 0) {
          int itr = 1;
          while (!(firstTables.BaseNames[itrColumnNames][itr] == null)) {
            columns++;
            itr++;
```

```
            }
        }
        firstRows++;
        //foundColumnNameFlag = true;
        //columnNames = tables.firstBaseNames[itrColumnNames];
      }
      itrColumnNames++;
    }

    //получение числа строк для второй таблицы
    itrColumnNames = 0;
    while (!(secondTables.BaseNames[itrColumnNames][0] == null) &&
!(secondTables.BaseNames[itrColumnNames][0] == ""))
    {
      if(secondTables.BaseNames[itrColumnNames][0] == line.tableName)
      {
        secondRows++;
      }
      itrColumnNames++;
    }

    //создание таблиц для таблиц
    String columnNames[] = new String[columns];
    String firstColumnValues[][] = new String[firstRows][columns];
    String secondColumnValues[][] = new String[secondRows][columns];
    Color cellsColors[][] = new Color[firstRows][columns];

    //заполнение таблицы имён
    itrColumnNames = 0;
    boolean foundColumnNameFlag = false;
    while (!(firstTables.BaseNames[itrColumnNames][0] == null) && !foundColumnNameFlag)
    {
      if(firstTables.BaseNames[itrColumnNames][0] == line.tableName)
      {
        for (int i = 0; i < columns; i++) {
```

```
                columnNames[i] = firstTables.BaseNames[itrColumnNames][i+1];
            }
            foundColumnNameFlag = true;
        }
        itrColumnNames++;
    }

    //заполнение первой таблицы значений
    int itrFinalColumnValues = 0;
    int itrColumnValues = 0;
    while (!(firstTables.BaseValues[itrColumnValues][0] == null))
    {
        if(firstTables.BaseValues[itrColumnValues][0] == line.tableName)
        {
            for (int i = 0; i < columns; i++) {
                firstColumnValues[itrFinalColumnValues][i] = firstTa-
bles.BaseValues[itrColumnValues][i+1];
            }
            itrFinalColumnValues++;
        }
        itrColumnValues++;
    }

    //заполнение второй таблицы значений
    int itrSecondColumnValues = 0;
    itrColumnValues = 0;
    while (!(secondTables.BaseValues[itrColumnValues][0] == null))
    {
        if(secondTables.BaseValues[itrColumnValues][0] == line.tableName)
        {
            for (int i = 0; i < columns; i++) {
                secondColumnValues[itrSecondColumnValues][i] =
secondTables.BaseValues[itrColumnValues][i+1];
            }
            itrSecondColumnValues++;
```

```
    }
    itrColumnValues++;
}

//заполнение таблицы цветов
for (int i = 0; i < firstRows; i++) {
    boolean foundRowFlag = false;
    for (int j = 0; j < secondRows && !foundRowFlag; j++) {
        if(firstColumnValues[i][0] == secondColumnValues[j][0])
        {
            foundRowFlag = true;
            cellsColors[i][0] = Color.WHITE;
            for (int x = 1; x < columns; x++) {
                if(firstColumnValues[i][x] == secondColumnValues[j][x])
                {
                    cellsColors[i][x] = Color.WHITE;
                }
                else
                {
                    if(firstColumnValues[i][x] != "" && secondColumnValues[j][x] == "")
                    {
                        cellsColors[i][x] = Color.RED;
                    }
                    else {
                        cellsColors[i][x] = Color.YELLOW;
                    }
                }

            }

        }
        if(!foundRowFlag)
        {

            for (int x = 0; x < columns; x++) {
```

```
                if(!secondFlag) {
                    cellsColors[i][x] = Color.RED;
                }
                else
                {
                    cellsColors[i][x] = Color.GREEN;
                }
              }
            }
          }
        }

        JTable table = new JTable(firstColumnValues, columnNames) {
          public TableCellRenderer getCellRenderer(int row, int column) {
            DefaultTableCellRenderer tcr = (DefaultTableCellRenderer) super
                .getCellRenderer(row, column);
            tcr.setBackground(cellsColors[row][column]);
            return tcr;
          }
        };
        table.setFillsViewportHeight(true);
        return table;
    }
}
```

Код класса MyTableRender

```
import javax.swing.*;
import javax.swing.table.DefaultTableCellRenderer;
import java.awt.*;

public class MyTableRender extends DefaultTableCellRenderer {
    public Component getTableCellRendererComponent(JTable table, Object value, boolean is-
Selected,
                                                   boolean hasFocus, int row, int column) {
        super.getTableCellRendererComponent(table, value, isSelected, hasFocus, row, column);
```

```
        setHorizontalAlignment(SwingConstants.CENTER);
        if((row == 1) && (column == 1)) {
            setBackground(Color.RED);
        }
        return this;
    }
}
```

Код класса SingleTable

```
public class SingleTable {
    public String[][] BaseNames = new String[1000][100];
    public String[][] BaseValues = new String[1000][100];

    public SingleTable(String[][] baseNames, String[][] baseValues) {
        BaseNames = baseNames;
        BaseValues = baseValues;
    }
}
```

Код класса TableComparator

```
import java.awt.*;

public class TableComparator {
    public TableComparator() {
    }

    public TableCompareResult compareTables(Tables tables)
    {
        String[] tableName = new String[1000];
        boolean[] founds = new boolean[1000];
        String[] finaltableName = new String[1000];
        int[] finalfounds = new int[1000];
        Color color[] = new Color[1000];

        TableCompareResult result = new TableCompareResult();
```

```
//построение таблиц tableName и founds, содержащих флаги tables.firstBaseNames
int itr1 = 0;
while (!(tables.firstBaseNames[itr1][0] == null))
{
   boolean foundFlag = false;
   int itr2 = 0;
   while (!(tables.secondBaseNames[itr2][0] == null))
   {
      if(tables.firstBaseNames[itr1][0] == tables.secondBaseNames[itr2][0])
      {
         if(tables.firstBaseValues[itr1][1] == tables.secondBaseValues[itr2][1])
         {
            foundFlag = true;
         }
      }
      itr2++;
   }

   tableName[itr1] = tables.firstBaseNames[itr1][0];
   founds[itr1] = foundFlag;
   itr1++;
}

//построение финальных таблиц
int recount1 = 0;
int finalIterator = 0;
while (!(tableName[recount1] == null))
{
   int nameCount = 1;
   int foundCount = 0;
   if(founds[recount1])
   {
      foundCount++;
   }
```

```
int iter3 = 0;
boolean existFlag = false;
while (!(finaltableName[iter3] == null)) {
   if (finaltableName[iter3] == tableName[recount1]) {
      existFlag = true;
   }
   iter3++;
}

if(!existFlag) {
   finaltableName[finalIterator] = tableName[recount1]; //поместить имя таблицы в финальный массив

   int recount2 = recount1 + 1;
   while (!(tableName[recount2] == null)) {
      if (tableName[recount1] == tableName[recount2]) {
         nameCount++;
         if (founds[recount2]) {
            foundCount++;
         }
      }
      recount2++;
   }
   if(foundCount == 0)
   {
      color[finalIterator] = Color.RED;
   }
   else {
      if (nameCount == foundCount) {
         color[finalIterator] = Color.WHITE;
      }
      if (nameCount > foundCount) {
         color[finalIterator] = Color.YELLOW;
      }
```

```
            }
            finalIterator++;
          }

          recount1++;
        }
        result.setValues(finaltableName, color);
        return  result;
    }
}
```

Код класса TableCompareResult

```
import java.awt.*;

public class TableCompareResult {
    public String tableNames[];
    public Color colors[];

    public TableCompareResult() {
    }

    public void setValues(String tableNameValues[], Color colorValues[])
    {
        this.tableNames = tableNameValues;
        this.colors = colorValues;
    }

    public TableCompareResultLine getValue (int index)
    {
        if(index < this.tableNames.length) {
            TableCompareResultLine resultLine = new TableCompareResult-
Line(this.tableNames[index], colors[index]);
            return resultLine;
        }
        else
```

```
        {
            return null;
        }
    }
}
```

Код класса TableCompareResultLine

```
import java.awt.*;

public class TableCompareResultLine {
    public String tableName;
    public Color color;

    public TableCompareResultLine(String tableName, Color color) {
        this.tableName = tableName;
        this.color = color;
    }
}
```

Код класса TableCompareResultLine

```
import javax.swing.*;
import java.awt.event.ComponentAdapter;
import java.awt.event.ContainerAdapter;
import java.awt.event.ContainerEvent;
public class TablesList{
    private JButton button1;
    private JList list1;
    public TablesList() {
        this.
        list1.addContainerListener(new ContainerAdapter() {
            @Override
            public void componentAdded(ContainerEvent e) {
                super.componentAdded(e);
                DefaultListModel listModel = new DefaultListModel();
                listModel.addElement("123");
```

```
                list1.setModel(listModel);
            }
        });
    }
}
```