УДК 004.652, 004.42

АВТОМАТИЗАЦИЯ ЧТЕНИЯ СВЯЗАННЫХ ДАННЫХ ИЗ РЕЛЯЦИОННЫХ И НЕРЕЛЯЦИОННЫХ БАЗ ДАННЫХ В КОНТЕКСТЕ ИСПОЛЬЗОВАНИЯ СТАНДАРТА JPA

А. С. Савинчева[1] [0009-0005-0756-6001], А. А. Ференец[2] [0000-0002-7859-9901]

[1, 2]*Институт информационных технологий и интеллектуальных систем Казанского федерального университета, ул. Кремлевская, 35, г. Казань, 420008*

[1]asanvlit@gmail.com, [2]ist.kazan@gmail.com

Аннотация

Описан процесс автоматизации управления операцией чтения связанных данных из реляционных и нереляционных баз данных.

Разработанный программный инструмент основан на использовании стандарта JPA (Java Persistence API), который определяет возможности контроля жизненного цикла сущностей в Java-приложениях. Спроектирована архитектура встраивания в событийные процессы, позволяющая интегрировать решение в проекты вне зависимости от используемой реализации JPA. Реализована поддержка различных стратегий загрузки данных, типов и параметров отношений. Осуществлена оценка производительности инструмента.

Ключевые слова*: JPA, ORM, Java, базы данных, реляционные базы данных, нереляционные базы данных.*

ВВЕДЕНИЕ

В настоящее время одним из ключевых этапов разработки программного обеспечения является взаимодействие с базами данных. Применение разнообразных типов баз данных, таких как реляционные и нереляционные, позволяет использовать уникальные особенности и преимущества каждого типа для реализации различных задач, однако требует специализированных подходов к работе. На практике разработчики могут часто сталкиваться с необходимостью взаимодействия с реляционными и нереляционными базами данных одновременно в рамках одного проекта. Это требует реализации соответствующей программной логики для контроля отношений между данными из различных хранилищ.

В Java-проектах для управления отображением объектной модели программы на записи в реляционной базе данных введён свой стандарт – JPA [1]. Существующие реализации JPA не всегда обеспечивают поддержку работы с отношениями между сущностями из различных типов баз данных. Так, Hibernate, который является одним из самых популярных JPA-фреймворков [2], позволяет управлять связями между сущностями в рамках одной базы данных, но не предоставляет возможностей обработки отношений между сущностями, хранящимися в различных типах баз данных [3].

В данной работе описан процесс разработки программного инструмента на основе стандарта JPA, автоматизирующего процесс управления связанными сущностями из реляционных и нереляционных баз данных в Java-приложениях. Ценность следования стандарту заключается в обеспечении независимости от реализации JPA, применяющейся в пользовательском проекте, что позволяет использовать уникальные преимущества, предоставляемые поставщиком, и не осуществлять интеграцию для перехода на другой инструмент. Работа охватывает исследование возможностей JPA для управления жизненным циклом сущностей, проектирование стратегии встраивания инструмента в событийные процессы, разработку механизмов контроля отношений и оценку производительности программного решения.

1. ОБЪЕКТНО-РЕЛЯЦИОННОЕ ОТОБРАЖЕНИЕ: JPA И ЕГО РЕАЛИЗАЦИИ

Объектно-реляционное отображение (ORM) – методология управления данными, предназначенная для связи объектно-ориентированных моделей с реляционными базами данных, обеспечивающая при этом автоматическое отображение информации из одной формы в другую [4]. Согласно концепции, лежащей в основе ORM, классы сущностей программы сопоставляются с таблицами базы данных, объекты – с записями, а атрибуты классов – со столбцами.

JPA – это стандарт Java для ORM. Он не является реализацией ORM, а представляет собой набор интерфейсов и правил, которые должны быть реализованы в ORM-фреймворках [1].

JPA определяет правила управления жизненным циклом сущностей, вводя понятия контекста постоянства (Persistence Context) и менеджера сущностей

(Entity Manager). Контекст постоянства представляет собой набор управляемых объектов, где отслеживаются изменения, вносимые в сущности, а менеджер сущностей обеспечивает функциональность по выполнению операций над объектами контекста и синхронизации с базой данных [1].

Hibernate является одним из наиболее известных и широко используемых ORM-фреймворков в Java, его доля на рынке составляет около 72% [5]. Он реализует базовые возможности JPA и предоставляет продвинутые механизмы, расширяющие функционал, определяемый стандартом. Например, для выполнения запросов Hibernate вводит собственный объектно-ориентированный язык HQL, который поддерживает функции, специфичные для фреймворка [6].

Hibernate обеспечивает работу с различными реляционными базами данных, включая PostgreSQL, MySQL, Oracle, Microsoft SQL Server. Для работы с нереляционными базами данных существует механизм Hibernate Object/Grid Mapper (OGM), преобразующий концепцию управления объектами из Hibernate ORM для работы с NoSQL базами данных, такими как MongoDB, Couchbase и другими [6].

В JPA введено также понятие единицы постоянства (Persistence Unit), которая представляет собой логический контейнер, содержащий информацию о подключении к базе данных и описании сущностей, используемых в приложении [1]. Hibernate и Hibernate OGM предоставляют продвинутые инструменты для работы с данными внутри одного типа баз данных, однако в них не реализована возможность построения и контроля связей между сущностями из различных типов, то есть сущностей, хранящихся в различных единицах постоянства [3]. Это приводит к необходимости самостоятельной реализации дополнительной логики для контроля целостности таких связей, что ведёт к расходу дополнительных ресурсов на разработку и поддержку.

EclipseLink – еще один ORM-фреймворк с открытым исходным кодом, реализующий JPA, его доля на рынке составляет около 13% [5]. Уникальными возможностями EclipseLink являются поддержка различных типов кэширования с настраиваемыми параметрами [7], возможность вызова собственных SQL-функций в JPQL-запросах [8], связывание Java-классов с XML и JSON-форматами [9].

EclipseLink поддерживает множество реляционных баз данных. Также он совместим с NoSQL-хранилищами, такими как MongoDB, Oracle NoSQL, XML-

файлы, JMS, Oracle AQ и JCA [10, 11]. Одной из ключевых особенностей EclipseLink является поддержка работы со связанными сущностями из различных типов баз данных. Это достигается благодаря механизму Composite Persistence Units, позволяющему объединять несколько источников данных в одну логическую композитную единицу, которая может использоваться для выполнения операций с различными хранилищами [12]. Однако создание такой единицы в EclipseLink является многоступенчатым процессом, его настройка начинается с определения XML-конфигурации, включающей в себя узлы для каждого из имеющихся источников данных с описанием схем, классов сущностей, настроек соединения, транзакций и кэш-стратегий. Далее осуществляется сборка JAR-файлов каждого из проектов, отвечающих за свою единицу постоянства, а затем источники объединяются в один итоговый проект, соответствующий композитной единице [12].

2. ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО РЕШЕНИЯ

В предыдущем разделе были проанализированы две ключевые реализации JPA: Hibernate и EclipseLink. Отмечено, что в EclipseLink реализован механизм Composite Persistence Units, обеспечивающий управление связями между сущностями, хранящимися в различных типах баз данных. Однако, как было показано, настройка этого механизма является многоступенчатой, что может усложнить разработку и поддержку приложения.

Помимо многоступенчатой конфигурации, переход на EclipseLink с другого JPA-поставщика может быть затруднен наличием следующих факторов:

- использование уникального функционала JPA-фреймворков, расширяющего базовые возможности стандарта, может быть тесно связано со специфическими требованиями проекта; переход на другую реализацию может потребовать не только доработки кода, инициированной в результате поиска альтернативных подходов в других инструментах, но и пересмотра текущих архитектурных решений;
- сообщество разработчиков, применяющих данный фреймворк [5], является достаточно узким, что ограничивает круг доступной экспертизы для решения возникающих проблем;

- объём документации может оказаться недостаточным ввиду небольшого количества практических примеров настройки.

На основании сказанного возникает одно из главных требований к разрабатываемому инструменту: он должен быть реализован на основе стандарта JPA, то есть опираться на контракт, который соблюдает каждый поставщик, что позволит обеспечить его независимость от конкретного фреймворка. Таким образом, не возникает необходимости осуществлять миграцию проекта, так как программное решение будет способно интегрироваться с любым из поставщиков, расширяя его функциональность наличием контроля отношений между связанными сущностями, хранящимися в различных типах баз данных.

Основополагающей концепцией является единообразие взаимодействия со внешними и внутренними отношениями. Внешними будем называть связи между сущностями, хранящимися в различных типах баз данных, внутренними – внутри одной базы данных. На рис. 1 представлена концепция работы библиотеки.

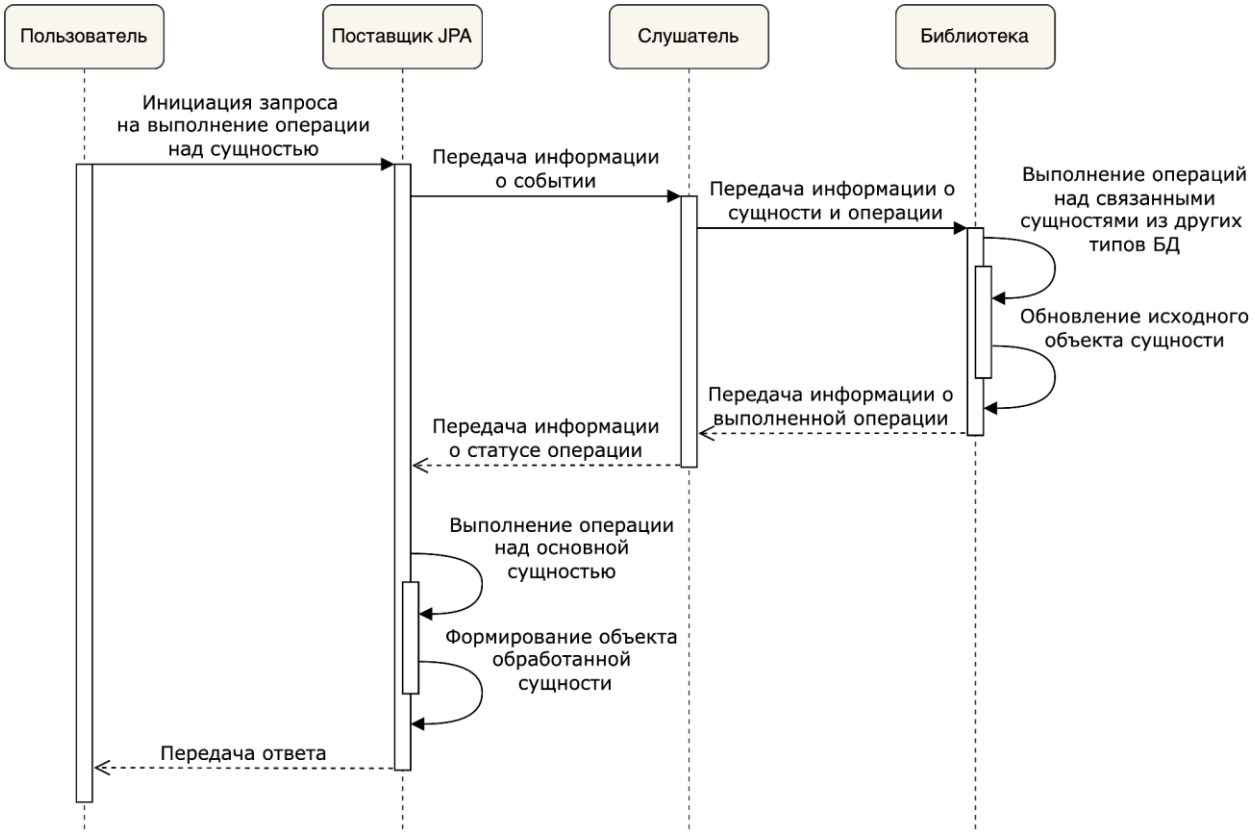

Рис. 1. Концепция работы библиотеки

Архитектура решения состоит из нескольких компонентов, осуществляющих контроль внешних отношений между сущностями. На рис. 2 изображён общий вид архитектуры библиотеки.

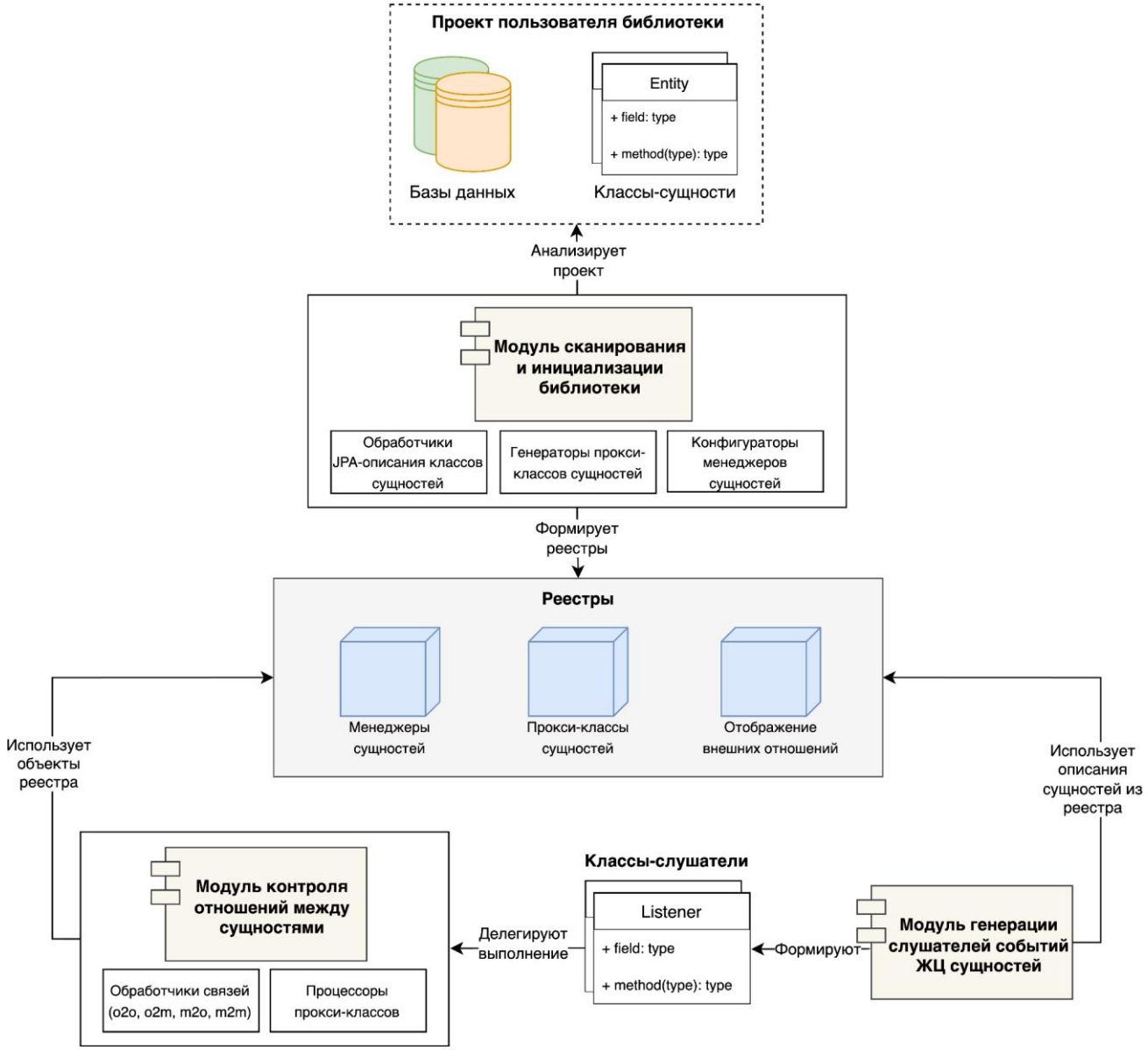

Рис. 2. Архитектура библиотеки

Модуль сканирования и инициализации анализирует структуру пользовательского проекта, определяет внешние связи между сущностями и устанавливает методы их контроля. Генераторы прокси-классов играют одну из ключевых ролей, создавая специальные классы-обёртки для сущностей из различных источников данных. Эти обёртки обеспечивают унифицированную структуру для ра-

боты с различными хранилищами данных и контроля изменений в сущностях. Результатом работы модуля являются заполненные реестры менеджеров сущностей, прокси-классов и отображений внешних связей, к которым происходит обращение в процессе жизненного цикла сущностей. На следующем этапе на основе данных из реестров генерируются классы-слушатели событий. Слушатели перехватывают вызов операций над сущностями, формируют информацию о событиях и передают её модулю контроля отношений, который является центральным узлом управления всеми внешними связями, учитывающим настройки пользовательской конфигурации. Он обеспечивает согласованность доступа к данным и выполнение операций над связанными сущностями.

3. МОДУЛЬ СКАНИРОВАНИЯ И ИНИЦИАЛИЗАЦИИ

Стандарт JPA предоставляет набор требований, определяющих объектно-ориентированное отображение сущностей на структуру в базе данных. Он вводит аннотации, такие как Entity, Table, Id и другие, использующиеся для обозначения JPA-сущностей и их настроек [1]. Для определения классов сущностей, хранящихся в нереляционных базах данных, в библиотеке введена аннотация NoSqlEntity.

В библиотеке реализованы сканеры, которые анализируют код проекта на предмет классов, соответствующих базовым критериям сущностей. На данном слое функционируют два таких компонента. Первый находит в проекте классы, соответствующие реляционным сущностям, второй – нереляционным, которые дополнительно помечены аннотацией NoSqlEntity, вводимой библиотекой.

Результат сканирования передаётся обработчикам соответствий. На этом этапе заполняются базовые свойства каждой сущности, такие как название таблицы, тип базы данных, уникальный ключ источника.

На последнем слое функционируют обработчики отношений, соответствующие каждому из четырех возможных типов связей. Их задача заключается в анализе отношений между сущностями и определении их конфигураций. Например, они сохраняют информацию о полях отношений, настройках чтения данных и определяют, какая из сущностей является владельцем отношения.

На рис. 3 показана схема данного процесса сканирования.

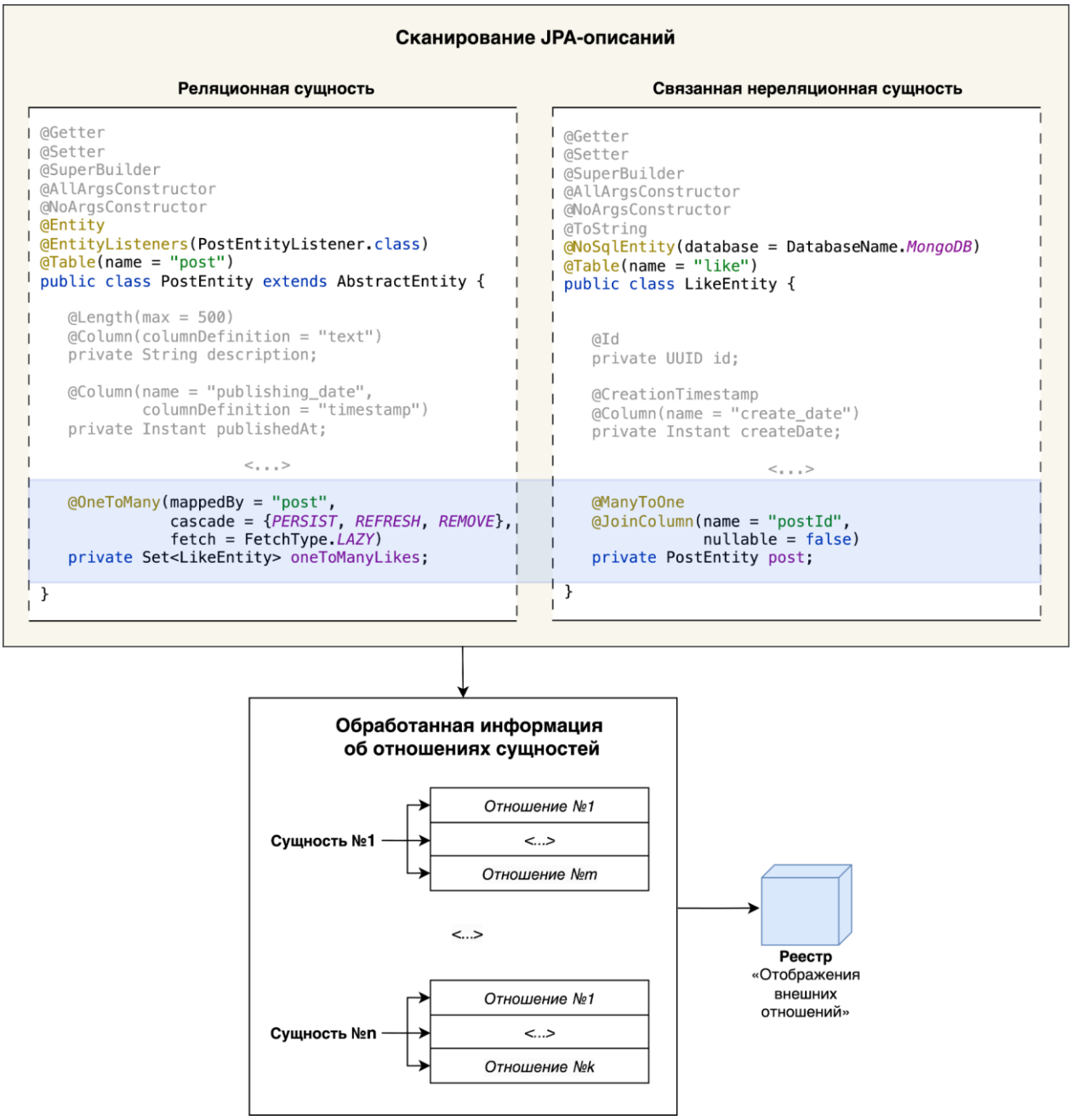

Рис. 3. Процесс сканирования JPA-описаний

На основе собранной информации формируется реестр, который служит централизованным хранилищем метаданных о всех обнаруженных сущностях, их полях, связях и конфигурации. Внутреннее устройство реестра содержит карту формата «ключ-значение», сложность операции извлечения из которой является константной. Благодаря данному первоначальному сканированию во время ра-

боты приложения происходит лишь обращение к реестру, что исключает необходимость повторного сканирования структуры этих сущностей при каждом обращении к ним.

Однако атрибуты исходных классов сущностей не соответствуют напрямую их хранимой структуре. Например, такое представление не содержит атрибутов, предназначенных для работы со внешними ключами отношения, вместо них в модели хранится объект целевого класса. Для решения этой проблемы в библиотеке реализован механизм генерации прокси-классов, которые наследуют исходную структуру сущностей, но дополнительно включают атрибуты, представляющие собой внешние ключи. Прокси является объектом, который выступает в качестве посредника между клиентским кодом и реальным объектом, тем самым обеспечивая дополнительный уровень контроля над доступом [13]. Расширенная структура прокси-класса позволяет менеджеру сущностей автоматически заполнять атрибуты внешних ключей при чтении данных, тем самым сокращая количество запросов к базе данных. Для создания прокси использовалась библиотека ByteBuddy.

На следующем этапе работы модуля происходит конфигурация менеджеров сущностей, которые представляют собой разработанный в рамках библиотеки слой абстракции, осуществляющий управление состоянием и жизненным циклом сущностей внутри контекста постоянства. Они инкапсулируют в себе уровень взаимодействия с базами данных, обеспечивая унифицированный интерфейс для работы с сущностями, независимо от типа используемой СУБД. Например, в случае MongoDB менеджер сущностей работает непосредственно с объектами MongoCollection, определяемыми драйвером MongoDB в Java, и реализует операции чтения, поиска и фильтрации данных посредством вызова соответствующих методов MongoCollection.

Созданные объекты менеджеров также регистрируются в специальном реестре для дальнейшего использования. Операции, которые требуется выполнить над связанными сущностями в течение их жизненного цикла, осуществляются посредством обращения к соответствующему менеджеру сущности из реестра.

На рис. 4 представлена схема описанного процесса создания менеджеров сущностей.

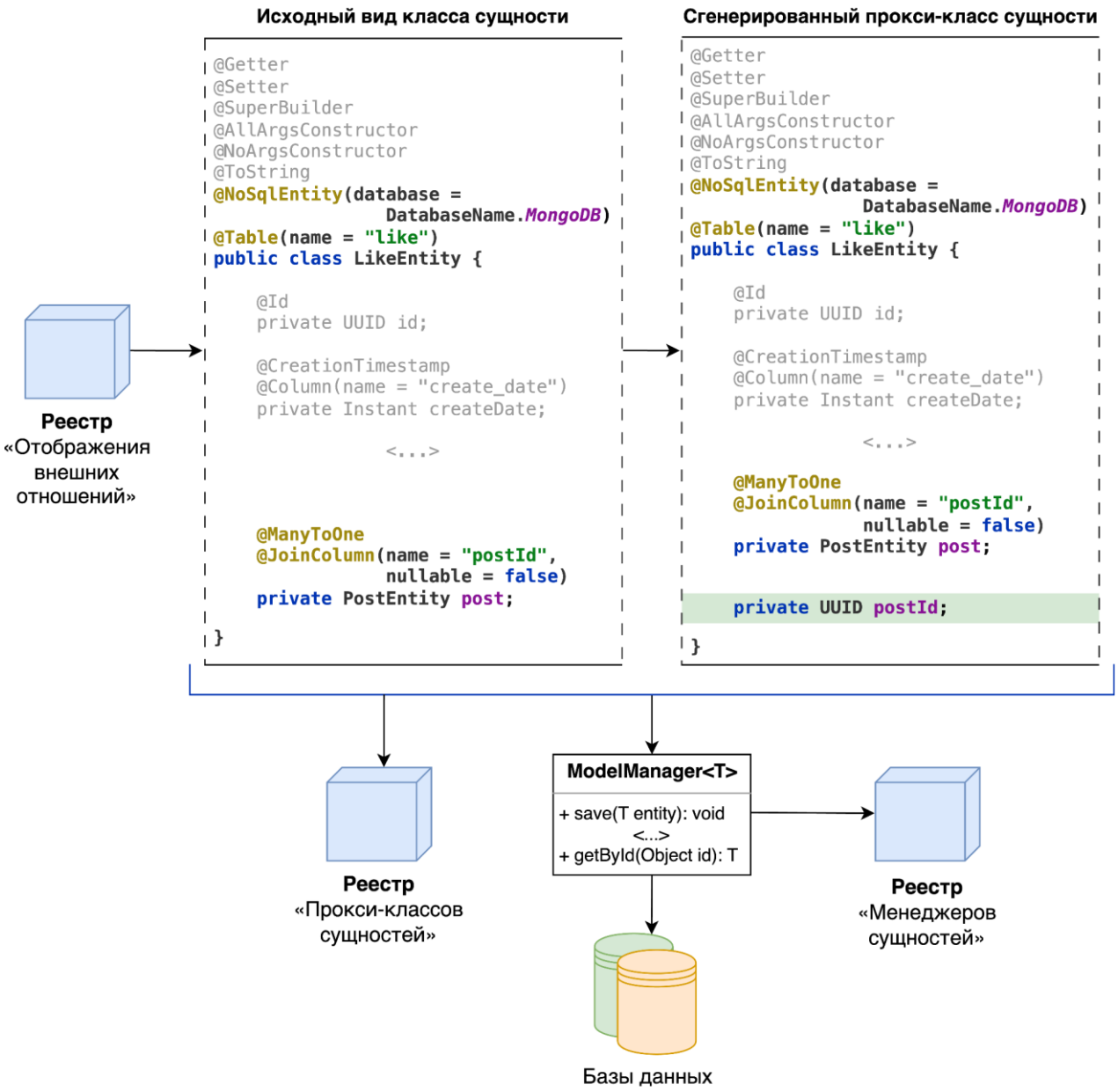

Рис. 4. Схема создания менеджеров сущностей

4. МОДУЛЬ ГЕНЕРАЦИИ СЛУШАТЕЛЕЙ СОБЫТИЙ

Слушатели – это программные компоненты, которые реагируют на наступление конкретных событий в системе и выполняют предопределенные действия в ответ на них.

JPA также определяет средства для работы со слушателями событий жизненного цикла сущностей. Контроль этих событий осуществляется с помощью аннотаций и методов обратного вызова (callback-методов), привязанных к стадиям

жизненного цикла сущности [1]. В случае операции чтения данных используется аннотация PostLoad, позволяющая определить действие, которое будет выполняться после загрузки сущности из базы данных.

В листинге 1 приведен пример JPA-слушателя событий, содержащего метод для логирования информации о загруженной сущности.

Листинг 1. JPA-слушатель событий

```
public class EmployeeListener {

    private static Log log = LogFactory.getLog(EmployeeListener.class);

    @PostLoad
    private void afterLoad(Employee employee) {
        log.info(
      "Employee with id loaded from database: " + employee.getId()
  );
    }
}
```

В рамках решаемой задачи логика методов обратного вызова может содержать обращения к другим слоям абстракции, которые реализуют механизмы контроля внешних связей. Таким образом, встраивание в событийную модель с помощью слушателей, концепция которых определена стандартом, обеспечивает независимость от используемого в проекте поставщика JPA.

На следующем шаге работы модуля происходит непосредственная интеграция описанной модели в пользовательский проект. Существуют два подхода внедрения компонентов: на этапе компиляции и во время выполнения программы. Второй подход неприменим в данном случае, так как JPA не вводит механизмы регистрации обработчиков событий непосредственно во время работы программы. Подобные возможности могут быть предоставлены конкретными поставщиками JPA, однако их использование становится невозможным ввиду принципа независимости библиотеки от реализации стандарта. Таким образом, был выбран подход, основанный на предварительной генерации кода слушателей.

Для задачи кодогенерации было принято решение использовать Freemarker – библиотеку в Java, которая применяется для генерации текстовой информации на основе шаблонов и подставляемых значений параметров [14]. Основными преимуществами работы с шаблонизаторами являются:

- специализированный язык описания шаблона, ограничения в синтаксисе которого не позволяют выйти за пределы предназначения шаблонизатора, что снижает вероятность ошибок на уровне языка программирования;
- ограниченность контекста: в рамках шаблона разработчик имеет доступ только к тем данным, которые предназначены для работы шаблона, исключая риск несанкционированного доступа к частям системы;
- расширенная функциональность с набором встроенных операций, упрощёнными записями конструкций и выражений.

Freemarker также поддерживает работу с макросами – блоками кода, которые можно определить и затем переиспользовать в нескольких местах шаблона [15]. Для структурирования генерации кода слушателя были созданы следующие макросы:

- структура класса, который определяет общий вид класса слушателя, включая объявление класса и необходимые аннотации;
- метод инициализации, в котором осуществляется контроль добавления необходимых компонентов;
- метод обратного вызова для контроля операции чтения данных.

В качестве значений переменных в блоки шаблона подставляются значения параметров из реестра. Эти параметры предварительно обрабатываются в программе и затем передаются генератору слушателей.

В листинге 2 показан фрагмент макроса для генерации структуры слушателя событий.

Листинг 2. Фрагмент макроса структуры класса

```
<#ftl encoding="utf-8">
<#import 'initMethod.ftlh' as initMethod/>
<#import 'postLoad.ftlh' as postLoadMethod/>
<#assign entityClassName = entityClass.getSimpleName()>
<#assign entityObjectName = entityClassName?uncap_first>
package ${packageName};
<...>
import ${entityClass.getName()};
<#if managers??>
<#list managers as managerClass>
import ${managerClass.getName()};
</#list>
</#if>
```

```
import java.util.*;

public class ${className} {
<@initMethod.initTemplate managers=managers entityClass=entityClass/>
<@postLoadMethod.postLoadTemplate listenerGeneratorDto=listenerGeneratorDto
entityClassName=entityClassName entityObjectName=entityObjectName/>
}
```

Результат генерации метода контроля чтения данных зависит от настройки типа загрузки (Fetch type) для каждого из внешних отношений сущности. JPA вводит два режима загрузки: «жадный» и «ленивый» [1]. При установленной «жадной» стратегии связанные сущности загружаются немедленно при чтении исходной, а при «ленивой» – только при явном обращении к ним в приложении. Это позволяет оптимизировать количество запросов к базе данных и выполнять их только при необходимости обработки отношения.

В зависимости от выбранной стратегии загрузки код слушателя будет содержать обращение к модулю контроля внешних отношений для немедленной загрузки или для отложенной, с помощью прокси-обёрток.

Сгенерированные слушатели помещаются в структуру пользовательского проекта.

5. МОДУЛЬ КОНТРОЛЯ ОТНОШЕНИЙ МЕЖДУ СУЩНОСТЯМИ

На предыдущем этапе работы библиотеки были сформированы реестры и сгенерированы слушатели событий. Эти слушатели подписываются на события жизненного цикла сущностей, имеющих внешние связи с объектами из других источников данных. Задачей слушателей является формирование и перенаправление информации о событии в модуль контроля отношений между сущностями. На рис. 5 изображена схема данного процесса.

На первом шаге обработки модуль обращается к реестрам для получения информации о настройках отношений и соответствующих прокси-классах связанных сущностей. В зависимости от типа связи этими операциями управляет соответствующий процессор отношений. Каждый процессор реализован для работы с определённым типом связи и умеет обращаться к необходимому методу менеджера сущностей, который вернет один элемент или целевую коллекцию.

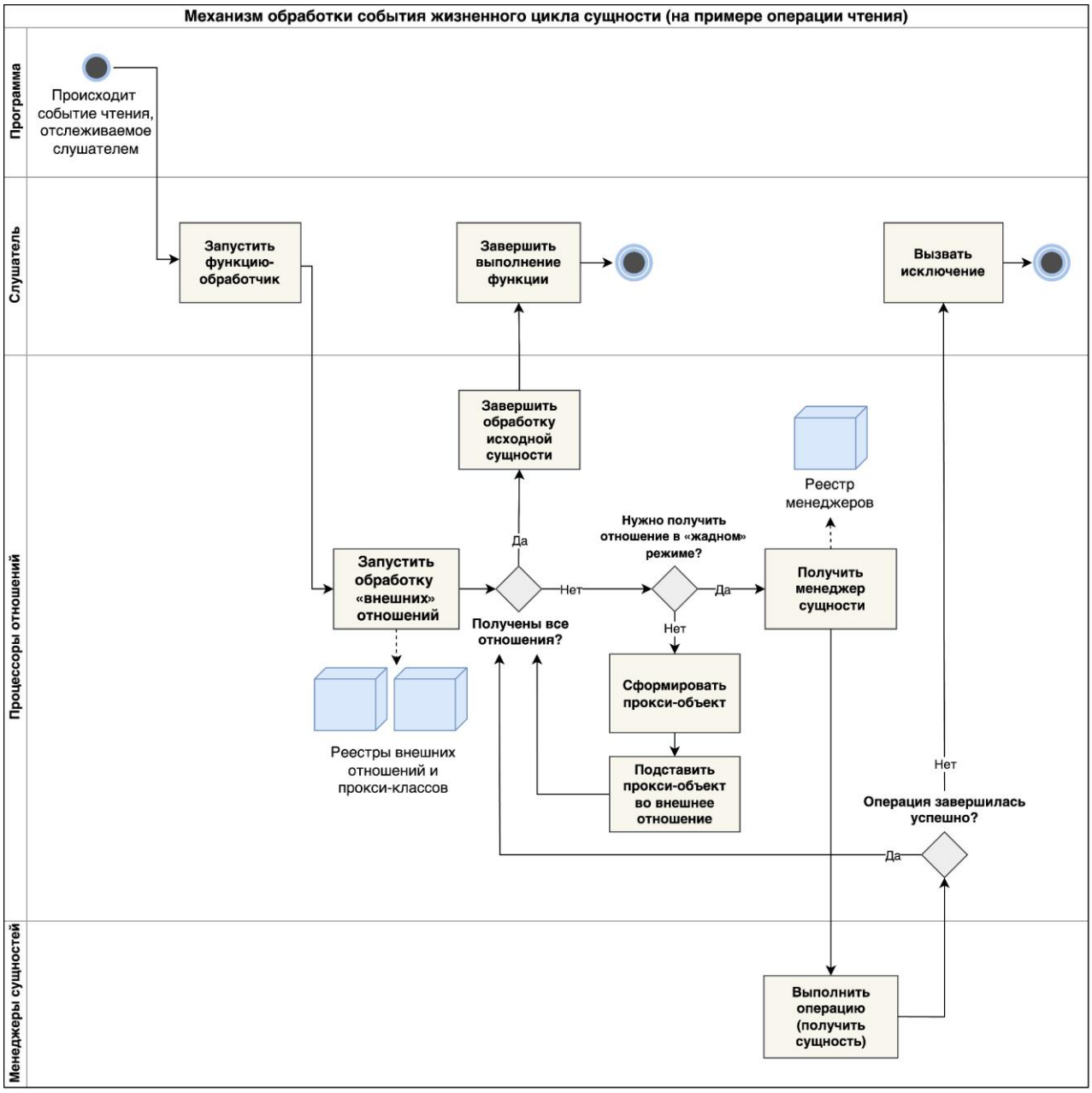

Рис 5. Механизм обработки внешних отношений

Одним из важнейших параметров конфигурации отношения является стратегия его загрузки.

При «жадной» загрузке процессор немедленно инициирует чтение связанных объектов из базы данных через менеджер сущностей.

В случае «ленивой» стратегии извлечение данных откладывается до момента фактического доступа к свойствам связанной сущности в коде приложения.

Для реализации подобной отложенной загрузки данных и идентификации осуществлённого обращения к атрибуту могут быть применены подходы, которые основаны на изменении логики вызываемых методов или подставляемых объектов. Одним из вариантов является модификация get-метода для получения данных об атрибуте отношения, но это противоречит концепции независимости библиотеки от реализации JPA, так как этот метод уже должен быть реализован в исходном классе сущности, поставщик JPA опирается на логику класса, определённую пользователем. Другой подход предполагает изменение алгоритма инициализации объекта целевого класса с использованием шаблона проектирования прокси. В Java для этой цели используется интерфейс InvocationHandler, который позволяет перехватывать вызовы методов целевого объекта [16]. Созданный прокси-объект связывается с экземпляром класса, реализующего интерфейс InvocationHandler, и делегирует выполнение методов данной логике. В контексте задачи определения момента обращения к атрибуту отношения метод invoke, принадлежащий данному интерфейсу, может использоваться для проверки состояния инициализации объекта и выполнения запроса к базе данных при первом обращении.

На рис. 6 показана схема данного взаимодействия описанных компонентов.

Также может возникнуть ситуация, когда у модели из внешнего отношения запрашиваемой исходной сущности есть другие внешние отношения, которые могут быть привязаны к иным источникам данных. Важным аспектом здесь является определение односторонних и двусторонних отношений. Одностороннее отношение характеризуется тем, что связь хранится только в одной из сущностей, в то время как двустороннее предполагает наличие ссылок обеих сущностей друг на друга. Для обеспечения корректного управления ссылками и предотвращения появления бесконечных циклов при обходе связей в двусторонних отношениях разработана соответствующая логика заполнения и проверок в процессорах отношений, учитывающая настройки и стратегию загрузки таких отношений.

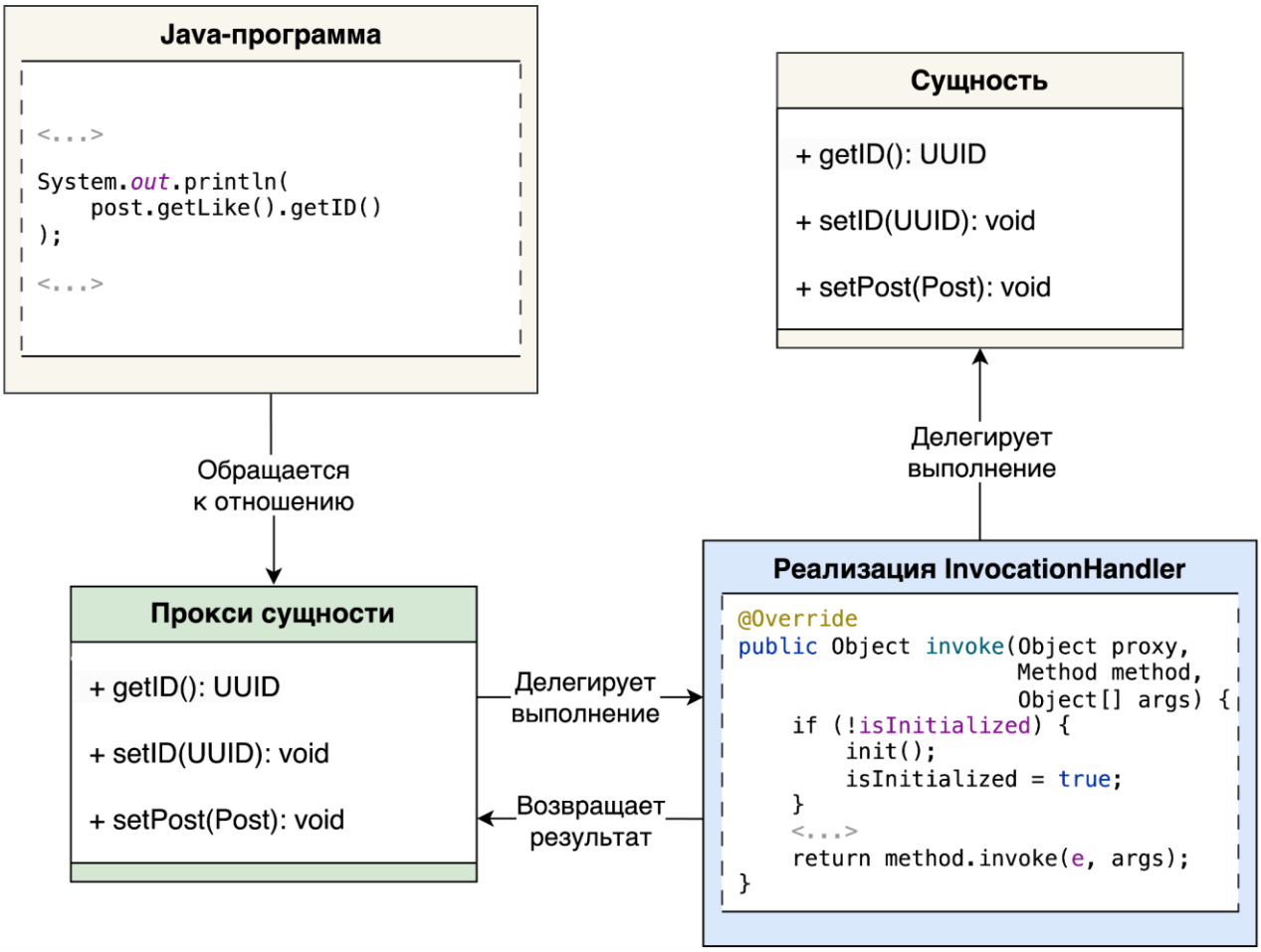

Рис. 6. Механизм взаимодействия с прокси в отношениях

6. ОЦЕНКА ПРОИЗВОДИТЕЛЬНОСТИ ИНСТРУМЕНТА

Оценка производительности проводилась путём сравнения времени выполнения операции чтения сущностей при двух подходах:

- осуществление контроля внешних отношений вручную: в данном случае разработчик самостоятельно реализует всю логику загрузки связанных сущностей;
- автоматизированный контроль с помощью разработанной библиотеки: управление связанными сущностями происходит с помощью предоставляемого функционала.

Отличие между этими подходами состоит в том, что при использовании библиотеки применяется рефлексия. Её рассмотренные механизмы позволяют исследовать и изменять внутреннее устройство объектов, а также модифициро-

вать их поведение во время работы программы, но требуют динамического разрешения типов, загрузки классов и информации, что влечёт дополнительные затраты во время выполнения [17].

Однако вся информация о сущностях, которая может быть получена с помощью рефлексии один раз, извлекается и сохраняется в реестры на этапе работы модуля сканирования и инициализации библиотеки. Это снижает влияние использования названных механизмов на производительность, что подтверждается проведёнными измерениями времени получения определённого количества сущностей.

Результат такого сравнения представлен на рис. 7.

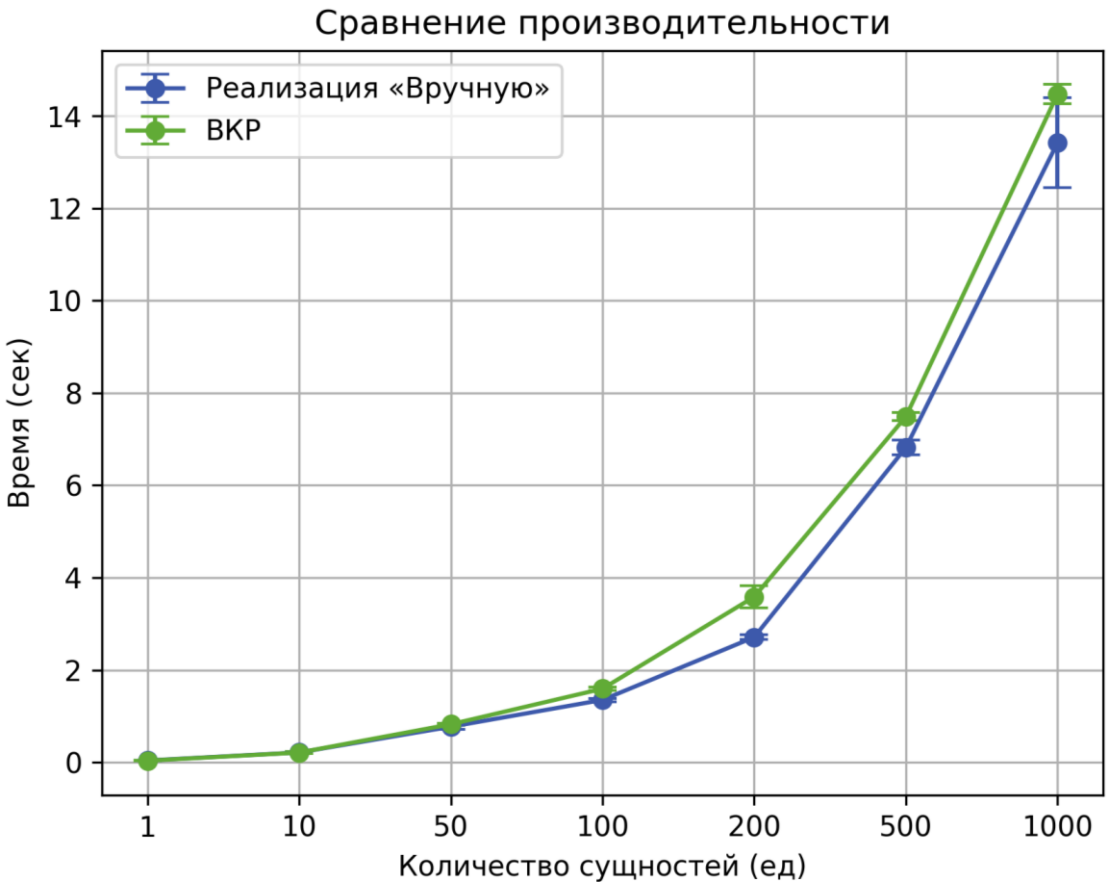

Рис. 7. Результаты измерения времени получения сущностей

Для тестирования были созданы две сущности: Entity1 (PostgreSQL) и MongoEntity (MongoDB). Сущности могли быть связаны между собой всеми типами отношений: «один-к-одному», «один-ко-многим», «многие-к-одному» и

«многие-ко-многим». Также эти структуры не содержали внутренних связей и не имели других внешних отношений, помимо перечисленных.

Генерация тестовых данных осуществлялась с помощью скрипта, который создавал экземпляры сущностей и в случайном порядке устанавливал между ними внешние отношения. Экземпляр мог не иметь внешних связей либо быть связан с другими сущностями посредством одного или нескольких (максимум четырёх) типов отношений. В случае имеющихся отношений «один-ко-многим» и «многие-ко-многим» скрипт генерировал от одной до десяти связей для каждого объекта.

Чтобы исключить влияние потенциальных выбросов, на графике, представленном на рис. 7, построены доверительные интервалы, которые были рассчитаны на основе 5 запусков с уровнем доверия 95% для каждой точки, соответствующей определённому количеству считываемых сущностей.

Согласно полученному результату, при небольшом количестве сущностей (1, 10, 50) время выполнения операций чтения сопоставимо для данных подходов, а с увеличением количества сущностей наблюдается рост времени выполнения в обоих случаях. Так, при обработке 1000 сущностей среднее время чтения для ручной реализации составило 13,42 секунды и 14,47 секунд для библиотеки. Однако при работе с сотнями и тысячами сущностей зачастую не требуется полная информация о каждой сущности. В таких случаях применяются другие подходы: пагинация, оптимизированные запросы получением исключительно основных свойств, «ленивая» загрузка. Таким образом, разработанная библиотека обеспечила сопоставимую с ручной реализацией производительность при работе с небольшим количеством сущностей, а в случае увеличения объемов данных демонстрирует приемлемую производительность, учитывая специфику практического применения.

ЗАКЛЮЧЕНИЕ

Описаны процесс и особенности разработки программного инструмента на основе стандарта JPA, автоматизирующего процесс управления связанными сущ-

ностями из реляционных и нереляционных баз данных в Java-приложениях, обеспечивающего при этом независимость от реализации JPA, используемой в проекте.

Исследованы определённые стандартом JPA возможности контроля жизненного цикла сущностей для обеспечения независимости от конкретной реализации, спроектирована стратегия встраивания программного решения в событийные процессы жизненного цикла сущностей, разработаны механизмы, автоматизирующие процесс управления связанными сущностями из реляционных и нереляционных баз данных, осуществлена оценка производительности разработанного решения. Инструмент обеспечивает работу с реляционными базами данных, перечень которых определяется JPA-фреймворком, используемым в проекте. В контексте нереляционных баз данных реализована поддержка MongoDB.

СПИСОК ЛИТЕРАТУРЫ

1. Jakarta Persistence. URL: https://jakarta.ee/specifications/persistence/3.0/jakarta-persistence-spec-3.0.html/.
2. Hibernate. URL: https://hibernate.org/.
3. Hibernate OGM. URL: https://hibernate.org/ogm/faq/.
4. What is Object-Relational Mapping (ORM) in DBMS? URL: https://www.geeksforgeeks.org/what-is-object-relational-mapping-orm-in-dbms/.
5. JPA providers market share in 2016. URL: https://vladmihalcea.com/jpa-providers-market-share-in-2016/.
6. Hibernate ORM User Guide. URL: http://docs.jboss.org/hibernate/orm/6.5/userguide/html_single/Hibernate_User_Guide.html/.
7. EclipseLink/Examples/JPA/Caching. URL: https://wiki.eclipse.org/EclipseLink/Examples/JPA/Caching.
8. JPQL. URL: https://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Querying/JPQL.
9. Developing JAXB Applications Using EclipseLink MOXy. URL: https://eclipse.dev/eclipselink/documentation/2.4/moxy/overview001.htm/.

10. Platform Specific Configurations.
URL: https://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Advanced_JPA_Development/Platform_Specific_Configurations/.
11. EclipseLink/FAQ/NoSQL.
URL: https://wiki.eclipse.org/EclipseLink/FAQ/NoSQL/.
12. Composite Persistence Units.
URL: https://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Advanced_JPA_Development/Composite_Persistence_Units/.
13. The Proxy Pattern in Java.
URL: https://www.baeldung.com/java-proxy-pattern/.
14. What is Apache FreeMarker™? URL: https://freemarker.apache.org/.
15. What are Macros?
URL: https://academy.flowmailer.com/hc/en-gb/articles/9404831036306-What-are-Macros
16. Dynamic Proxy Classes.
URL: https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/proxy.html/.
17. The performance implications of Java reflection.
URL: https://blogs.oracle.com/javamagazine/post/java-reflection-performance/.

AUTOMATION OF READING RELATED DATA FROM RELATIONAL AND NON-RELATIONAL DATABASES IN THE CONTEXT OF USING THE JPA STANDARD

A. S. Savincheva[1] [0009-0005-0756-6001], A. A. Ferenets[2] [0000-0002-7859-9901]

[1, 2]*Institute of Information Technologies and Intelligent Systems, Kazan (Volga Region) Federal University, ul. Kremlyovskaya, 35, Kazan, 420008*

[1]asanvlit@gmail.com, [2]ist.kazan@gmail.com

Abstract

The process of automating the management of the reading operation of related data from relational and non-relational databases is described.

The developed software tool is based on the use of the JPA (Java Persistence API) standard, which defines the capabilities of managing the lifecycle of entities in Java applications. An architecture for embedding in event processes has been designed, allowing the solution to be integrated into projects regardless of which JPA implementation is used. Support for various data loading strategies, types, and relationship parameters has been implemented. The performance of the tool has been evaluated.

Keywords*: JPA, ORM, Java, databases, relational databases, non-relational databases.*

REFERENCES

1. Jakarta Persistence. URL: https://jakarta.ee/specifications/persistence/3.0/jakarta-persistence-spec-3.0.html/.
2. Hibernate. URL: https://hibernate.org/.
3. Hibernate OGM. URL: https://hibernate.org/ogm/faq/.
4. What is Object-Relational Mapping (ORM) in DBMS? URL: https://www.geeksforgeeks.org/what-is-object-relational-mapping-orm-in-dbms/.
5. JPA providers market share in 2016. URL: https://vladmihalcea.com/jpa-providers-market-share-in-2016/.
6. Hibernate ORM User Guide.

URL: http://docs.jboss.org/hibernate/orm/6.5/userguide/html_single/Hibernate_User_Guide.html/.

7. EclipseLink/Examples/JPA/Caching.

URL: https://wiki.eclipse.org/EclipseLink/Examples/JPA/Caching.

8. JPQL.

URL: https://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Basic_JPA_Development/Querying/JPQL.

9. Developing JAXB Applications Using EclipseLink MOXy.

URL: https://eclipse.dev/eclipselink/documentation/2.4/moxy/overview001.htm/.

10. Platform Specific Configurations.

URL: https://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Advanced_JPA_Development/Platform_Specific_Configurations/.

11. EclipseLink/FAQ/NoSQL.

URL: https://wiki.eclipse.org/EclipseLink/FAQ/NoSQL/.

12. Composite Persistence Units.

URL: https://wiki.eclipse.org/EclipseLink/UserGuide/JPA/Advanced_JPA_Development/Composite_Persistence_Units/.

13. The Proxy Pattern in Java.

URL: https://www.baeldung.com/java-proxy-pattern/.

14. What is Apache FreeMarker™? URL: https://freemarker.apache.org/.

15. What are Macros?

URL: https://academy.flowmailer.com/hc/en-gb/articles/9404831036306-What-are-Macros

16. Dynamic Proxy Classes.

URL: https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/proxy.html/

17. The performance implications of Java reflection.

URL: https://blogs.oracle.com/javamagazine/post/java-reflection-performance/

СВЕДЕНИЯ ОБ АВТОРАХ

САВИНЧЕВА Ангелина Сергеевна – выпускник бакалавриата Института информационных технологий и интеллектуальных систем Казанского (Приволжского) федерального университета, г. Казань.

Angelina SAVINCHEVA – graduate student of the Institute of Information Technologies and Intelligent Systems, Kazan (Volga region) Federal University, Kazan.

email: asanvlit@gmail.com

ORCID: 0009-0005-0756-6001

ФЕРЕНЕЦ Александр Андреевич – старший преподаватель кафедры программной инженерии Института информационных технологий и интеллектуальных систем Казанского (Приволжского) федерального университета, г. Казань.

Alexander FERENETS – senior lecturer of Software Engineering of Institute of Information Technologies and Intelligent Systems KFU***.***

email: ist.kazan@gmail.com

ORCID: 0000-0002-7859-9901

Материал поступил в редакцию 7 июля 2024 года