**Министерство образования Республики Беларусь**

**УО «Мозырский государственный педагогический университет имени И.П. Шамякина»**

ФИЗИКО-ИНЖЕНЕРНЫЙ ФАКУЛЬТЕТ

Кафедра теоретической физики и прикладной информатики

Дипломная работа

**Использование шаблона проектирования MVC при разработке учебных информационных систем**

Выполнила:  
студентка 4 курса 2 группы  
физико-инженерного факультета  
Пиляк Ксения Юрьевна

Научный руководитель:  
кандидат физико-математических наук,

доцент  
Сергиевич Н.В.

Оценка научного руководителя: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
 *(оценка, дата сдачи, подпись)*

Итоговая оценка: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Мозырь 2017

# ВВЕДЕНИЕ

Компьютерные технологии стали неотъемлемой частью нашей жизни. Они используются во всех сферах человеческой деятельности, в том числе в промышленности, экономике, сфере услуг. Возросла потребность в специалистах, владеющих персональным компьютером, а также в разработчиках программного обеспечения. Поэтому очень важно обучить людей основам информационных технологий. Для этих целей в учреждениях образования появились компьютеры, а учащихся стали обучать компьютерной грамотности.

Довольно давно информатика стала обязательным предметом в школе. На уроках детей учат пользоваться программами, которые могут быть полезными и необходимыми в их будущей деятельности. Это такие программы как Word, Excel, PowerPoint и другие. Однако кроме обучения на пользовательском уровне, учащихся необходимо научить основам алгоритмизации и программирования, чтобы они понимали, как создаются используемые ими программы и что скрывается за их привычным интерфейсом. Для этого в школьный курс информатики включен раздел «Основы алгоритмизации и программирования», при изучении которого учащиеся знакомятся с основными видами алгоритмов, учатся писать простые программы, развивают свое мышление и логику.

В настоящее время учителя средней школы сталкиваются с определенными трудностями при преподавании раздела «Основы алгоритмизации и программирования». Как правило, основная проблема заключается в том, что задачи, предлагаемые учащимся авторами учебников, отнимают очень много времени на проверку учителем правильности их решения. Тестирующие системы есть далеко не у каждой школы, а для их написания также требуется немало времени. Поэтому целью данной работы является написание учебной информационной системы для средних школ, в которой можно осуществлять проверку задач по алгоритмизации и программированию, читать соответствующий теоретический материал, а также автоматически получать отметки в зависимости от количества решенных задач.

Поскольку главной целью этой работы является доступность для всех школ страны, то лучшей формой информационной системы является сайт. Доступность из Интернета позволяет не только организовывать обучение в школе, но и осуществлять самообучение учащихся дома, а также родительский контроль над учебной деятельностью их детей.

# ГЛАВА 1 ТЕОРЕТИЧЕСКИЕ ОСНОВЫ ПРОЕКТИРОВАНИЯ ИНФОРМАЦИОННЫХ СИСТЕМ

## 1.1 Шаблоны проектирования

В задачах проектирования информационных систем, классов, их составляющих, при распределении обязанностей и способов взаимодействия объектов этих классов перед программистом возникает серьезная проблема. Неоптимальный выбор может сделать системы и их отдельные компоненты непригодными для поддержки, восприятия, повторного использования и расширения. Систематизация приемов программирования и принципов организации классов получила название шаблона (паттерна) [1].

Существует четыре типа шаблонов проектирования: производящие, поведенческие, структурные, системные.

***Производящие шаблоны*** предназначены для обеспечения выполнения одной из самых распространенных задач в ООП – создания объектов в системе. В ходе работы большинства объектно-ориентированных систем, независимо от уровня их сложности, создается множество экземпляров объектов. Производящие шаблоны облегчают процесс создания объектов, а некоторые из них позволяют при создании объектов налагать ограничения на их тип или количество в соответствии с установленными требованиями системы. Примеры производящих шаблонов:

* *Abstract Factory* (обеспечивает создание семейств взаимосвязанных или зависящих друг от друга объектов без указания их конкретных классов);
* *Factory Method* (определяет стандартный метод создания объекта, не связанный с вызовом конструктора, оставляя решение о том, какой именно объект создавать за подклассами);
* *Singleton* (обеспечивает наличие в системе только одного экземпляра заданного класса, позволяя другим классам получать доступ к этому экземпляру).

***Поведенческие шаблоны*** применяются для передачи управления в системе. Существуют методы организации управления, применение которых позволяет добиться значительного повышения как эффективности системы, так и удобства ее эксплуатации. Примеры поведенческих шаблонов:

* *Command* (обеспечивает обработку команды в виде объекта, что позволяет сохранять ее, передавать в качестве параметра методам, а также возвращать ее в виде результата, как и любой другой объект);
* *Iterator* (предоставляет единый метод последовательного доступа к элементам коллекции и никак с ней не связанный);
* *Observer* (предоставляет компоненту возможность гибкой рассылки сообщений интересующим его получателям);
* *Strategy* (предназначен для определения группы классов, которые представляют собой набор возможных вариантов поведения программы)*.*

***Структурные шаблоны*** с одинаковой эффективностью применяются как для разделения, так и для объединения элементов приложения. Способы воздействия структурных шаблонов на приложение могут быть самые разные. Например, шаблон *Adapter* может обеспечить возможность двум несовместимым системам обмениваться информацией, тогда как шаблон *Facade* позволяет отобразить упрощенный пользовательский интерфейс, не удаляя ненужных конкретному пользователю элементов управления. Также известны следующие шаблоны:

* *Bridge* (разделение сложного компонента на две независимые, но взаимосвязанные иерархические структуры: функциональную абстракцию и внутреннюю реализацию).
* *Decorator* (предоставление механизма для добавления или удаления функциональности компонентов без изменения их внешнего представления или функций).

***Системные шаблоны*** представляют приложение на архитектурном уровне – уровне наивысшей абстракции. Системные шаблоны могут применяться в приложении для осуществления большинства процессов и даже для поддержки взаимодействия разных приложений. Примеры структурных шаблонов:

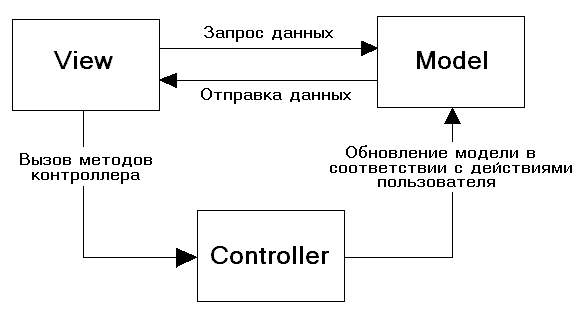
* *Model-View-Controller* (разделение компонента или подсистемы на три логические части: модель, представление и контроллер с целью облегчения модификации или настройки каждой части в отдельности);
* *Router* (отделение источников информации от ее получателей);
* *Callback* (обеспечение клиенту возможности регистрации на сервере для выполнения расширенных операций, это позволяет серверу извещать клиента о завершении операции).

## 1.2 Шаблон проектирования MVC

Шаблон проектирования Model-View-Controller («Модель-Представление-Контроллер», MVC) – схема разделения компонента или подсистемы на три логические части (модель, представление и контроллер) с целью облегчения модификации или настройки каждой части в отдельности [1]. Согласно этому шаблону, сложный элемент определяется в терминах трех логических составляющих:

* Модель (model) хранит все данные, информацию о состоянии и логику приложения. Она не знает о существовании представления и контроллера, хотя предоставляет интерфейс для получения/изменения состояния.
* Контроллер (controller) получает данные, вводимые пользователем, и определяет их смысл для модели.
* Представление (view) получает состояние и данные для отображения непосредственно от модели [3].

На уровне компонента модель управляет состоянием этого компонента, представление обеспечивает пользовательский интерфейс с ним, а контроллер выполняет обработку событий или привязку операций. На архитектурном уровне эти функции можно транслировать в подсистемы: модель представляет всю бизнес-модель, представление – отображение модели, а контроллер определяет бизнес-процессы [2]. Схема шаблона MVC представлена на рисунке 1.1.

  
**Рисунок 1.1 – Схема шаблона проектирования MVC**

Суть программной реализации шаблона MVC состоит в следующем:

* **Model**. Этот компонент содержит один или более классов и интерфейсов, отвечающих за обслуживание модели данных. Состояние модели сохраняется в атрибутах и реализации методов. Для того чтобы иметь возможность уведомлять компоненты-представления о любых изменениях, модель хранит ссылку на каждое зарегистрированное представление (одновременно может быть несколько представлений). Когда происходит изменение, каждый зарегистрированный компонент-представление извещается об этом изменении.
* **View**. Классы и интерфейсы представления обеспечивают презентацию данных, хранящихся в компоненте-модели. Представление может (но не обязано) состоять из компонентов визуального графического интерфейса. Для того, чтобы представление могло получать извещения об изменении данных от модели, оно должно быть зарегистрировано в модели. Получая извещение об изменениях, компонент-представление принимает решение о том, нужно ли их отображать, и если нужно, то как именно это сделать. Компонент-представление также сохраняет ссылку на модель для получения данных от модели, но при этом представление может лишь получать данные без возможности их изменения. Представление может также использоваться для сокрытия контроллера, но при этом все запросы на изменение всегда пересылаются компоненту-контроллеру. В этой связи представление должно сохранять ссылку или несколько контроллеров.
* **Controller**. Этот компонент управляет изменениями модели. Он хранит ссылку на компонент-модель, который отвечает за осуществление изменений, и сам в свою очередь отвечает за вызов одного или нескольких методов обновления. Запрос на изменения может поступать от компонента-представления.

## 1.3 Сервлеты

***Сервлет*** – это приложение Java, запускаемое и выполняемое в контейнере сервера приложений. Клиент общается с таким приложением посредством веб-браузера. Никаких дополнительных приложений на стороне клиента устанавливать не требуется. Сервлеты поддерживаются виртуальной машиной JVM, что предотвращает утечки памяти и обеспечивает функционирование garbage collection. Клиент посылает приложению HTTP-запрос, сервлет генерирует ответ и возвращает его клиенту в виде html-документа.

О сервлете можно сказать следующее:

* компонент приложений Java Enterprise Edition;
* загружается веб-сервером в контейнер;
* выполняется на стороне сервера;
* обрабатывает клиентские запросы;
* динамически генерирует ответы на запросы;
* находится в состоянии ожидания, если запросы отсутствуют;
* принимает запросы от других сервлетов;
* поддерживает соединения с ресурсами.

Наибольшее распространение получили сервлеты, обрабатывающие клиентские запросы по протоколу HTTP. Технология сервлетов является оболочкой протокола HTTP и поддерживает его как транспорт передачи данных от клиента серверу и обратно. Контейнер сервлетов поддерживает также протокол HTTPS для защищенных запросов.

Сервлеты в промышленном программировании используются для приема входящих данных от клиента, взаимодействия с бизнес-логикой системы, динамической генерации ответа клиенту.

Все сервлеты реализуют общий интерфейс Servlet из пакета javax.servlet. Для обработки HTTP-запросов в web можно воспользоваться в качестве базового класса абстрактным классом HttpServlet из пакета javax.servlet.http.

Жизненный цикл сервлета начинается с его инициализации и загрузки в память контейнером сервлетов при старте контейнера либо в ответ на первый клиентский запрос. Сервлет готов к обслуживанию любого числа запросов. Завершение существования происходит при выгрузке его из контейнера.

Первым вызывается метод init(). Он дает сервлету возможность инициализировать данные и подготовиться для обработки запросов. Чаще всего в этом методе размещается код, кэширующий данные фазы инициализации.

После этого сервлет можно считать запущенным, он находится в ожидании запросов от клиентов. Появившийся запрос обслуживается методом service(HttpServletRequest request, HttpServletResponse response) сервлета, вызываемый контейнером, а все параметры запроса упаковываются в экземпляр request интерфейса HttpServletRequest, передаваемый в сервлет. Еще одним параметром этого метода является экземпляр response интерфейса HttpServletResponse, в который загружается информация для передачи клиенту. Для каждого нового клиента при обращении к сервлету создается независимый поток, в котором производится вызов метода service(). Метод service() предназначен для одновременной обработки множества запросов.

Метод service() класса HttpServlet служит диспетчером для других методов, каждый из которых обрабатывает методы доступа к соответствующим ресурсам. В спецификации HTTP определены и наиболее часто употребляются методы GET и POST, передающие на сервер запросы, а также параметры для их выполнения.

Метод GET используется для запроса содержимого указанного ресурса, изображения или гипертекстового документа. Вместе с запросом могут передаваться дополнительные параметры как часть URI, значения могут выбираться из полей формы или передаваться непосредственно через URL. При этом запросы кэшируются и имеют ограничения на размер. Этот метод является основным методом взаимодействия браузера и веб-сервера.

Метод POST используется для передачи пользовательских данных в содержимом HTTP-запроса на сервер. Пользовательские данные упакованы в тело запроса согласно полю заголовка Content-Type и/или включены в URI запроса. При использовании метода POST под URI подразумевается ресурс, который будет обрабатывать запрос.

Также определены методы HEAD, PUT, DELETE, OPTIONS, TRACE. В задачу метода service() входит анализ полученного через запрос метода доступа к ресурсам и вызов метода со сходным именем: doGet(), doPost(), doPut(), doDelete(), doOptions(), doTrace(). Разработчик должен переопределить нужный метод, разместив в нем функциональную логику.

При выгрузке приложения из контейнера, то есть по окончании жизненного цикла сервлета, вызывается метод destroy(), в теле которого следует помещать код освобождения занятых сервлетом ресурсов.

Сервлеты поддерживаются серверами приложений WebSphere, Weblogic, JBoss, Oracle OC4J Server, Glassfish, Geronimo, Apache Tomcat, а также контейнерами сервлетов tomcat, jetty, grizzly и являются частью платформы JEE [1].

## 1.4 JAVA SERVER PAGES

В то время как сервлеты наилучшим образом подходят для выполнения контролирующей функции приложения в виде обработки запросов и определения содержания и вида ответа, страницы Java Server Pages (JSP) выполняют функцию отображения результатов работы приложения в виде текстовых документов типа HTML, XML, WML и некоторых других. JSP поддерживает как JavaScript, так и HTML-теги.

Принято разделять динамическое и статическое содержимое JSP.

Динамические ресурсы. Результаты их деятельности изменяются во время выполнения приложения. Обычно представлены в виде выражений Expression Language, библиотек тегов и тегов разработчика.

Статические ресурсы. Не изменяются сами в процессе работы (HTML, JavaScript, изображения и т.д.).

Смысл разделения динамического и статического содержания в том, что статические ресурсы могут находиться под управлением HTTP-сервера, в то время как динамические нуждаются в движке (JSP-engine) и в большинстве случаев в доступе к уровню данных.

Преимущества использования JSP-технологии над другими методами создания динамического содержания страниц:

* разделение динамического и статического содержания. Возможность разделить логику приложения и дизайн веб-страницы снижает сложность разработки веб-приложений и упрощает их поддержку;
* независимость от платформы. Технологии Java не зависят от платформы, следовательно, JSP могут выполняться практически на любом веб-сервере;
* многократное использование компонентов. Использование JavaBeans и Enterprise JavaBeans (EJB) позволяет многократно использовать компоненты, что ускоряет создание веб-приложений;
* теги. Спецификация JSP содержит библиотеку стандартных тегов JSTL, позволяет разработчику создавать собственные теги, кроме того, теги обеспечивают возможность использования JavaBean и обращение к классам бизнес-логики.

Содержимое Java Server Pages (теги HTML, теги JSP и скрипты) переводится в сервлет код-сервером. Этот процесс ответственен за трансляцию как динамических, так и статических элементов, объявленных внутри файла JSP.

Процессы, выполняемы с файлом JSP при первом вызове:

* браузер делает запрос к странице JSP;
* JSP-engine анализирует содержание файла JSP и создает сервлет с кодом, основанным на исходном тексте файла JSP, при этом engine транслирует статическое содержимое в методы вывода и помещает его в метод \_jspService(). Полученный сервлет будет ответственен за генерацию статических элементов, определенных во время разработки. Динамические элементы транслируются в java-код;
* код сервлета компилируется в файл \*.class и загружается в контейнер. В итоге сервлет на основе JSP установлен и готов к работе.

В JSP API введено понятие Expression Language (EL). EL используется для упрощения доступа к данным (атрибутам, параметрам и т.п.), хранящимся в различных областях видимости: page, request, session, application и вычисления простых выражений.

EL вызывается при помощи конструкции ${имя\_атрибута}.

Данные, как правило, состоят из объектов, соответствующих спецификации JavaBeans, или представляют собой коллекции: List, Map, Set, массивы и др. EL предоставляет доступ к этим объектам при помощи операторов «.» и «[]». Способ применения этих операторов зависит от типа объекта. К элементу массива или списка доступ производится обычной индексацией с оператором «[]». К элементам типа Map: либо указанием ключа после оператора «.», либо обращением к ключу также по имени, но в операторе «[]».

Обращение к экземпляру класса производится с неявным вызовом метода toString() в виде: ${object}.

Обращение к значению поля экземпляра класса производится с неявным вызовом метода getИмя() в виде: ${object.id}, то есть вызван будет метод getId().

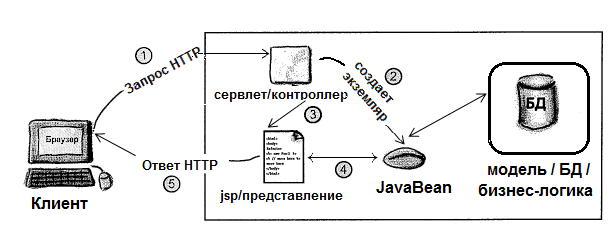
В JSP страницах используются стандартные элементы action. Он позволяют обойти явное внедрение java-кода в JSP. Примерами таких тегов являются: <jsp:include> (включение статических и динамических ресурсов в контекст генерируемой страницы), <jsp:useBean> (объявление экземпляра компонента JavaBean или класса, обладающего public-конструктором по умолчанию), <jsp:forward> (передача запроса другой странице или сервлету) и др.

JSP-страницы, включающие элементы action и пользовательские теги, не могут быть технологичными без использования JSTL (JSP Standard Tag Library). Библиотека тегов JSTL состоит из пяти групп тегов:

1. core: if/then выражения и конструкции множественного выбора; вывод; создание и удаление контекстных переменных; управление свойствами JavaBeans; перехват исключений; forEach для итерированных коллекций; создание URL и импортирование их содержимого. Примеры: <c:out/>, <c:set/>, <c:if/>, <c:forEach/> и др.;
2. formatting: установка локализации; локализация текста и структуры сообщений; форматирование и анализ чисел, процентов, денежных единиц и дат. Примеры: <fmt:setLocale/>, <fmt:requestEncoding/>, <fmt:timeZone/>, <fmt:parseNumber/> и др.;
3. sql: доступ к БД; описание источника данных; выполнение запросов, обновление данных и транзакций; обработка результатов запроса. Примеры: <sql:query/>, <sql:update/>, <sql:transaction> и др.;
4. xml: преобразование XML; доступ и преобразование XML через XPath и XSTL. Примеры: <x:parse/>, <x:set/>, <x:if/> и т.д.;
5. fn: обработка объектов типа String и определение размера коллекций. Примеры: ${fn:length(аргумент)}, ${fn:toUpperCase(String str)}, ${fn:slpit(String str, String delim)} и т.д [1].

## 1.5 Применение шаблона проектирования MVC в web-разработке

После широкого распространения Интернета разработчики начали адаптировать шаблон «Модель-Представление-Контроллер» к модели «браузер/сервер». Основная адаптация использует комбинацию сервлетов и технологии JSP для достижений той же степени логической изоляции модели, представления и контроллера, как в традиционных графических интерфейсах.

  
**Рисунок 1.2 – Схема работы шаблона MVC в web-разработке**

Шаблон MVC в web-разработке работает следующим образом (рис. 1.2):

1. Запрос HTTP принимается сервлетом. Пользователь в браузере выдает запрос HTTP. Обычно при этом передаются данные формы (например, имя и пароль). Сервлет принимает данные формы и разбирает их.
2. Сервер выступает в роли контроллера и обрабатывает запрос – вероятно, с выдачей запросов к модели (обычно к БД). Результат обработки запроса обычно упаковывается в форме JavaBean.
3. Контроллер передает запрос представлению. Представлению соответствует код JSP, единственная задача которого – сгенерировать страницу с представлением модели (4) и всеми элементами, необходимых для дальнейших действий.

(5) Представление возвращает страницу браузеру через HTTP. Пользователь создает дальнейшие запросы, которые обрабатываются по той же схеме [3].

## 1.6 Доступ к базе данных

API JDBC (Java DataBase Connectivity) – стандартный прикладной интерфейс языка Java для организации взаимодействия между приложением и СУБД. Взаимодействие осуществляется с помощью драйверов JDBC, обеспечивающих реализацию общих интерфейсов для конкретных СУБД и конкретных протоколов. С помощью JDBC отсылаются SQL-запросы только к реляционным БД, для которых существуют драйверы, знающие способ общения с реальным сервером базы данных. С JDBC совместимы такие СУБД как MySQL, Oracle, PostgreSQL, SQL Server и другие.

Связь с базой данных является важной частью любой системы, поэтому всегда выделяется часть кода, ответственная за передачу запросов к БД и обработку полученных от нее ответов. Такая часть кода реализуется с помощью шаблона Data Access Object (DAO) – прослойка между приложением и СУБД. DAO абстрагирует бизнес-сущности системы и отражает их на записи в БД. DAO определяет общие способы использования соединения с БД, моменты его открытия и закрытия или извлечения и возвращения в пул.

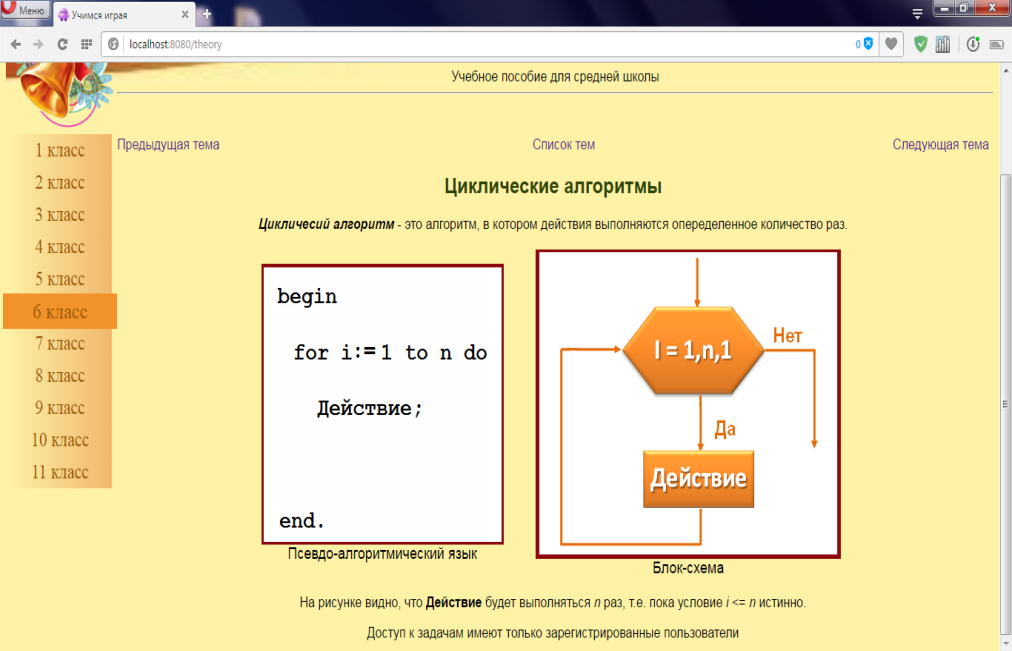
Вершина иерархии DAO представляет собой класс или интерфейс с описанием общих методов, которые будут использоваться при взаимодействии с таблицей или группой таблиц. Как правило, это методы выбора, поиска сущности по признаку, добавление, удаление и замена информации.

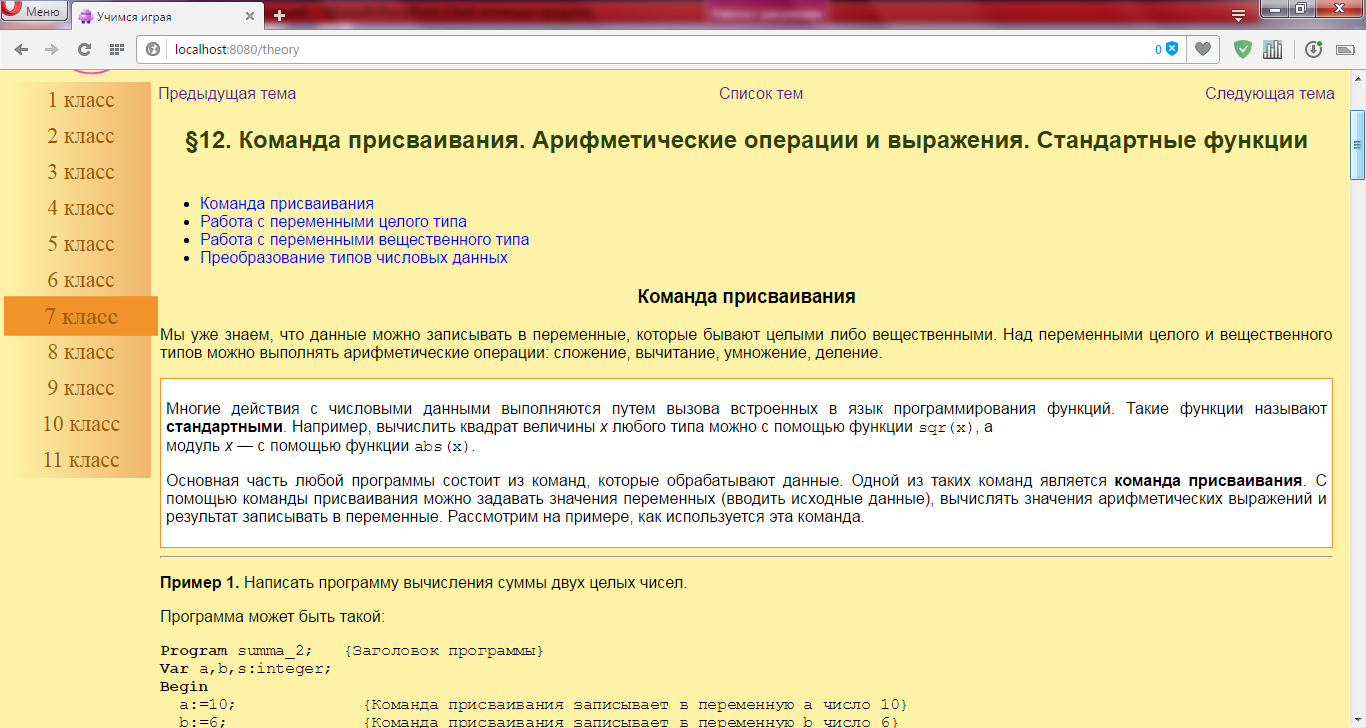
Объектно-реляционное предобразование (ORM) – простое сохранение Java-объектов в реляционную базу данных. Среди технологий, поддерживающих ORM: Entity Beans 2.x, TopLink, iBatis, EclipseLink, Hibernate, JDO, JDBC с DAO и собственными аннотациями. На основе принципов организации этих технологий разработан Java Persistence API (JPA). JPA предоставляет POJO persistence модель объектно-реляционного преобразования (т.е. такая модель, в которой некоторому классу по определенным правилам ставятся в соответствие некоторые таблица или набор столбцов в таблице БД). JPA-аннотации используются для определения объектов и отношений. Аннотации можно разбить на две категории: логические (позволяют описать объектную модель, связи между классами) и физические (описывающие физическую схему, таблицы, столбцы, индексы и т.д.). Примерами логических аннотаций служат @Entity (объявление сущности), @OneToOne (связь «один к одному»), @Basic (сохраняемое свойство объекта) и т.д. Примерами физических аннотаций являются @Table (таблица в БД), @Column (столбец таблицы), @Id (первичный ключ в таблице) и др.

Hibernate – один из первых инструментов объектно-реляционного отображения (ORM) данных для Java-окружения. Целью Hibernate является освобождение разработчика от большинства общих работ, связанных с задачами получения, сохранения данных в СУБД. Эта технология помогает удалить или инкапсулировать зависящий от поставщика SQL-код, а также решает стандартную задачу преобразования типов Java-данных в типы данных SQL и наборов данных из табличного представления в объекты Java-классов [1].

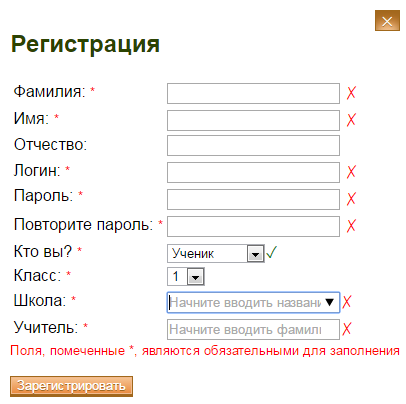
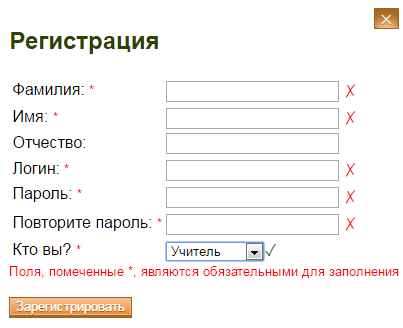
# ГЛАВА 2 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

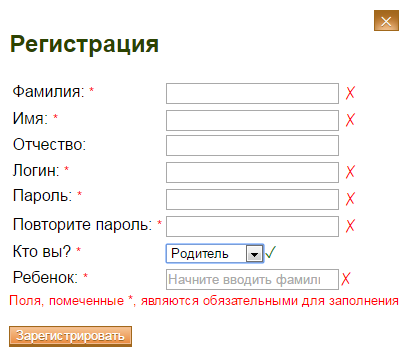
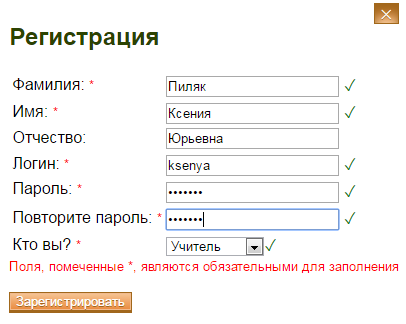
Сайт «Основы алгоритмизации и программирования» – это образовательный ресурс для обучения учащихся соответствующему разделу школьного курса информатики. По каждой из тем, которые сгруппированы по классам, представлен теоретический материал (рис. 2.1), большая часть которого взята из школьных учебников информатики (см. [4], [5]).



  
**Рисунок 2.1 – Теоретический материал**

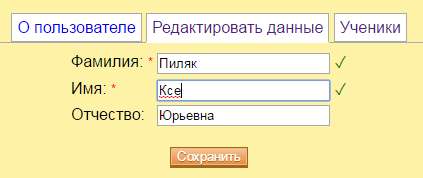
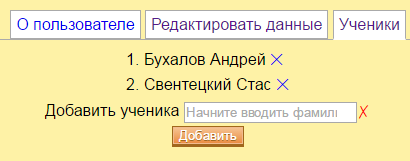
На данном сайте можно регистрироваться ученикам, учителям и родителям. Форма для регистрации учителя показана на рисунке 2.2, *а*. Обязательные для заполнения поля помечены «\*». Ученикам нужно дополнительно указывать класс, школу и учителя (рис. 2.2, *б*), а родителям – своего ребенка (рис. 2.2, *в*).

  
***а) Форма регистрации для учителя б) Форма регистрации для ученика***

**  
*в) Форма регистрации для родителя г) Пример правильно заполненной формы***  
**Рисунок 2.2 – Формы регистрации**

Неверное заполнение данных в форме не позволяет пройти регистрацию. На правильность заполнения указывает наличие зеленых «галочек» напротив каждого поля формы, обязательного для заполнения (рис. 2.2, *г*). Примеры неправильно заполненных полей: пустое поле «Имя», русские буквы в поле «Логин», количество символов в поле «Пароль» меньше четырех, указано несуществующее в базе ФИО учителя в поле «Учитель» и т.д. Соответствующие подсказки отображаются при подведении курсора мыши на соответствующее поле.

После регистрации пользователь может авторизоваться. Теперь ему доступны расширенные возможности сайта, о которых речь пойдет ниже. В так называемом личном кабинете пользователь может редактировать свои данные. Кроме этого, ученики могут выбирать своих учителей, учителя – учеников, родители – детей (рис. 2.3).

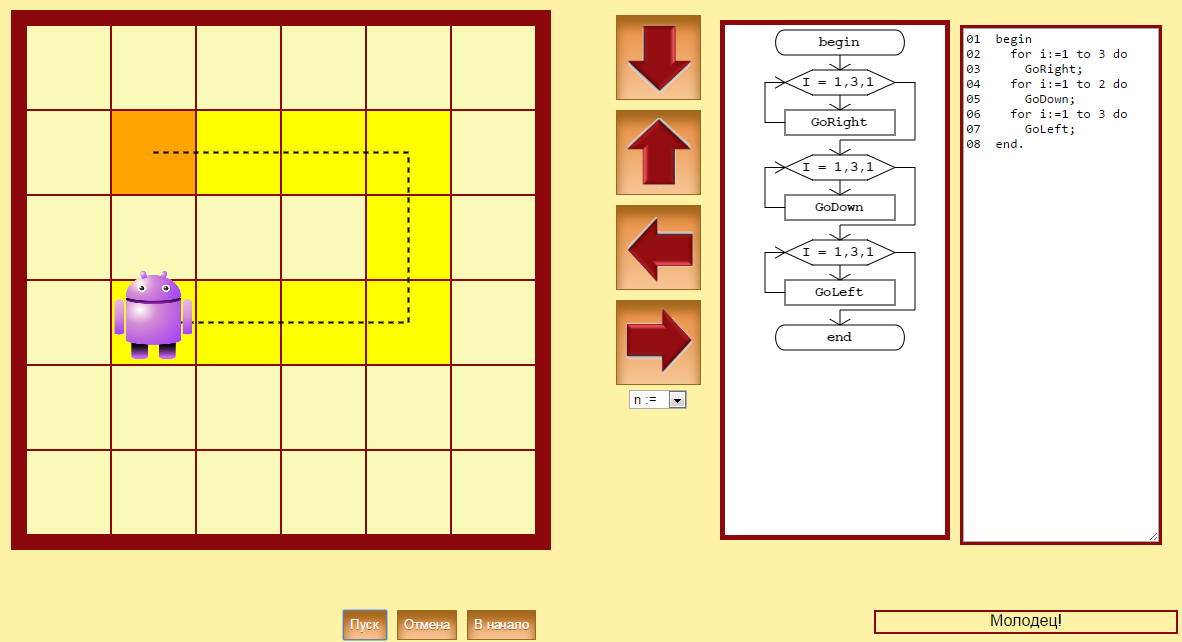
   
**Рисунок 2.3 – Кабинет пользователя**

Неавторизованные пользователи могут читать теорию по каждой из тем Зарегистрированные пользователи могут также решать задачи. Учителям и родителям задачи доступны без ограничений, ученикам нужно подписываться на задания по каждой теме. Это необходимо для формирования таблицы результатов: неподписанные ученики, а, значит, не решавшие задачи по теме, просто не отображаются в таблице. Учителям и родителям также можно решать задачи, однако в таблице результатов их не будет. Во вкладке «Задачи» имеется список задач по теме (рис. 2.4). Уже решенные задачи помечены «галочками».

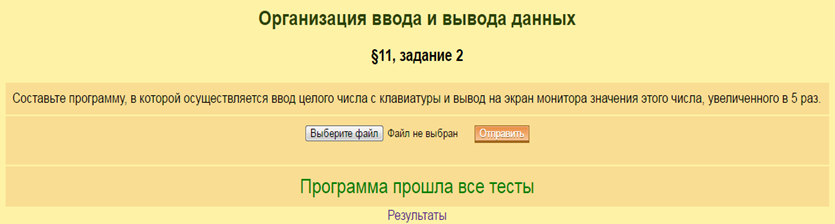
  
**Рисунок 2.4 – Список заданий по теме**

Задачи по алгоритмизации предлагается решать в среде алгоритмического исполнителя «Робот». Задачи по программированию можно отправлять на тестирование на сервер. Цель задач по алгоритмизации – написать алгоритм, с помощью которого Робот пройдет все желтые клетки. При нажатии на кнопки управления Роботом автоматически будет формироваться блок-схема алгоритма и код программы на языке, максимально приближенном к языку программирования Pascal (рис. 2.5). Если задание выполнено, то будет выведено слово «Молодец».

Цель задач по программированию – написать программу на языке программирования Pascal таким образом, чтобы она прошла все тесты на сервере. Для проверки задачи пользователь должен отправить файл с решением на сервер. После проверки в браузере будет написан один из вердиктов тестирования: «Программа прошла все тесты», «Неверный ответ на тесте №Х», «Ошибка времени выполнения (логическая ошибка)», «Превышен лимит времени (программа выполнялась слишком долго)», «Ошибка компиляции (в файле содержится синтаксическая ошибка)» (рис. 2.6).

  
**Рисунок 2.5 – Алгоритмический исполнитель «Робот»**



**Рисунок 2.6 – Задания по программированию**

По каждой теме можно просмотреть свои и чужие результаты решения задач с соответствующей отметкой. Результаты можно сортировать по столбцам таблицы, а также отбирать по классу и по школе. Кроме этого, учителя могут отсеивать результаты для просмотра только своих учащихся, а родители – своих детей (рис. 2.7).



  
**Рисунок 2.7 – Результаты выполнения заданий по теме**

# ГЛАВА 3 РАЗРАБОТКА ИНФОРМАЦИОННОЙ СИСТЕМЫ «ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ»

## 3.1 Проектирование базы данных

Наиболее важным этапом в разработке веб-приложений является проектирование базы данных. База данных хранит всю информацию, необходимую для функционирования системы. Для хранения и управления БД используется реляционная СУБД MySQL.

Поскольку на сайте будет возможность регистрации, то в первую очередь в БД нужно хранить информацию о зарегистрированных пользователях: фамилию, имя, отчество, логин, пароль, роль (учитель, ученик, родитель). Для учеников также нужна информация о классе и школе.

Учащиеся не должны вводить название школы самостоятельно, поскольку одна и та же школа может быть написана по-разному. Например, «СШ №16» и «Средняя школа №16». Это затруднит обработку данных, например, при выборке учащихся по школам. Поэтому название школ нужно хранить в отдельной таблице, а в таблице пользователей хранить ссылку на запись в таблице школ. Такая связь двух таблиц носит название «один ко многим», поскольку каждой школе ставится в соответствие несколько пользователей и нескольким пользователям ставится в соответствие одна школа.

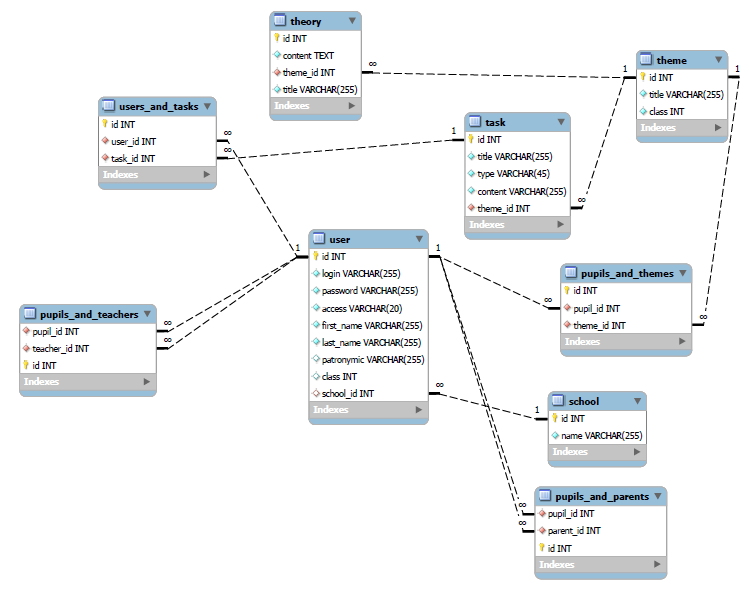
Чтобы учителя могли осуществлять контроль над своими учениками, а родители – над своими детьми, необходимо хранить в базе информацию об отношениях «ученик – учитель» и «ребенок – родитель». У каждого учителя может быть несколько учеников и у каждого ученика может быть несколько учителей (например, ученик занимается в школе и с репетитором). Аналогично отношение «ребенок – родитель»: у родителя может быть несколько детей, а у ребенка могут быть мама, папа, бабушка, дедушка и т.д. Такая связь называется «многие ко многим». Для реализации этой связи в БД используется связующая таблица, в которой хранятся ссылки на записи из таблиц, которые надо связать.

Суть веб-приложения состоит в том, чтобы дать возможность пользователям изучать теоретический материал и закреплять его на практике, т.е. выполнять задания. Логично весь материал разбить по темам, а темы в свою очередь – по классам. Для этого необходима таблица тем, в которой хранятся название темы и класс, в котором она изучается. Теорию необходимо хранить в отдельной таблице, в которой будут название теоретической статьи, ее содержимое (html-разметка) и ссылка на тему (связь «один ко многим», т.к. одной теме может соответствовать несколько статей). Задания также хранятся в отдельной таблице, содержащей название, тип, содержимое задания (условие и дополнительная информация для проверки, хранится в файлах) и ссылка на тему.

Пользователь должен видеть задачи, которые он уже решил. Пользователь может решить несколько задач, а какая-то задача может быть решена многими пользователями – связь «многие ко многим». В этом случае необходима связующая таблица, которая имеет ссылки на записи в таблице пользователя и ссылки на записи в таблице заданий.

При формировании таблицы результатов будет неудобно, если в таблице отобразятся все зарегистрированные пользователи. Поэтому возникла идея не отображать учителей и родителей в таблице результатов, а ученикам ввести подписку на задания. Для этого достаточно будет в БД указать тему, на задания которой ученик подписан. Это также связь «многие ко многим», следовательно, подписки хранятся в связующей таблице.

ER-диаграмма (entity-relationship) базы данных приведена на рисунке 3.1. Здесь показаны связи между таблицами, типы данных полей, первичные ключи. Диаграмма выполнена в инструменте визуального проектирования баз данных MySQL WorkBench.

  
**Рисунок 3.1 – ER-диаграмма**

Листинг создания таблиц БД приведен в приложении А.

## 3.2 Сущности

В объектно-ориентированных языках программирования, каким является язык Java, для преобразования таблиц баз данных в классы и наоборот используется технология программирования ORM (объектно-реляционное отображение). Кроме того, ORM предоставляет интерфейс для CRUD-операций над данными. Это избавляет разработчика от написания громоздкого SQL- и JDBC-кода. Для Java ORM-решением является фреймворк Hibernate.

Каждой таблице в БД ставится в соответствие java-класс. Такие классы называются сущностями. Они представлены POJO-классами (Plain Old Java Object), т.е. классами, имеющими конструктор без параметров, поля, set- и get-методы для полей и переопределенные методы equals() и hashCode(). Реализация последних двух методов необходима для сравнения и выявления одинаковых объектов, в противном случае производится сравнение по ссылкам на объекты, которые различны для двух объектов. Также класс должен быть сериализуемым, т.е. реализовывать интерфейс Serializable. Это нужно для обеспечения возможности данным класса быть записанными в БД. Кроме того, в классах сущностей присутствуют JPA-аннотации, речь о которых шла в пункте 1.6.

Прежде, чем приступить к написанию сущностей, нужно создать сущность AbstractEntity, от которой унаследуются все остальные сущности.

**package** model.entities;  
**public abstract class** AbstractEntity {  
}

Создание этого класса необходимо для гибкого использования интерфейса. Достоинство такого подхода будет видно при написании DAO.

Таблице school ставится в соответствие класс School:

**package** model.entities;  
  
**import** javax.persistence.\*;  
**import** java.io.Serializable;  
**import** java.util.HashSet;  
**import** java.util.Set;  
  
@Entity  
@Table(name = **"school"**, schema = **"programming\_tutorial"**, catalog = **""**)  
**public class** School **extends** AbstractEntity **implements** Serializable {  
 **private int** id;  
 **private** String name;  
 **private** Set<Pupil> pupils = **new** HashSet<>();  
  
 **public** School() {  
 }  
  
 **public** School(String name) {  
 **this**.name = name;  
 }  
  
 @Id  
 @GeneratedValue(strategy = GenerationType.IDENTITY)  
 @Column(name = **"id"**, nullable = **false**)  
 **public int** getId() {  
 **return** id;  
 }  
 **public void** setId(**int** id) {  
 **this**.id = id;  
 }  
  
 @Basic  
 @Column(name = **"name"**, nullable = **false**, length = 255)  
 **public** String getName() {  
 **return** name;  
 }  
  
 **public void** setName(String name) {  
 **this**.name = name;  
 }  
  
 @OneToMany(mappedBy = **"school"**)  
 **public** Set<Pupil> getPupils() {  
 **return** pupils;  
 }  
  
 **public void** setPupils(Set<Pupil> pupils) {  
 **this**.pupils = pupils;  
 }  
  
 @Override  
 **public boolean** equals(Object o) {  
 **if** (**this** == o) **return true**;  
 **if** (o == **null** || getClass() != o.getClass()) **return false**;  
  
 School that = (School) o;  
  
 **if** (id != that.id) **return false**;  
 **if** (name != **null** ? !name.equals(that.name) : that.name != **null**)  
 **return false**;  
 **return** pupils != **null** ? pupils.equals(that.pupils) :   
 that.pupils == **null**;  
 }  
  
 @Override  
 **public int** hashCode() {  
 **int** result = id;  
 result = 31 \* result + (name != **null** ? name.hashCode() : 0);  
 result = 31 \* result + (pupils != **null** ? pupils.hashCode() : 0);  
 **return** result;  
 }  
}

Здесь @Entity указывает на то, что класс является сущностью. @Table(name = "school", schema = "programming\_tutorial", catalog = "") содержит имя таблицы, которой сопоставляется сущность, и имя БД, в которой эта таблица находится. @GeneratedValue(strategy = GenerationType.IDENTITY) означает, что значение будет создано с помощью базы данных, в данном случае – автоинкремент поля id. @Column содержит имя и свойства поля таблицы, которому соответствует указанная переменная. @Basic – сохраняемое свойство. @OneToMany(mappedBy = "school") – связь «один ко многим»: одной школе соответствует несколько учеников. mappedBy = "school"указывает на связь с полемschoolв классе Pupil.

Таблица user отображается на класс User. Однако пользователи делятся на учеников, учителей и родителей, на что указывает поле access. Поэтому логично сделать класс User родительским и от него унаследовать классы Pupil (ученик), Teacher (учитель) и Parent (родитель).

@Entity  
@Table(name = **"user"**, schema = **"programming\_tutorial"**, catalog = **""**)  
@Inheritance(strategy = InheritanceType.***SINGLE\_TABLE***)  
@DiscriminatorColumn(name = **"access"**)  
**public class** User **extends** AbstractEntity **implements** Serializable {  
 //-------------------------  
 @Column(name = **"access"**, nullable = **false**, insertable=**false**, updatable=**false**)  
 @Enumerated(EnumType.***STRING***)  
 **public** UserType getAccess() {  
 **return access**;  
 }  
 //-------------------------  
}

@Entity  
@DiscriminatorValue(**"PUPIL"**)  
**public class** Pupil **extends** User {}

@Entity  
@DiscriminatorValue(**"TEACHER"**)  
**public class** Teacher **extends** User{}

@Entity  
@DiscriminatorValue(**"PARENT"**)  
**public class** Parent **extends** User {}

В классе User поле access типа UserType. Это перечисление, благодаря которому в зависимости от значения поля access происходит определение роли пользователя и создание соответствующего объекта типа Pupil, Teacher или Parent (указывается в @DiscriminatorValue). При записи в базу данных перечисление представляется в виде строки (@Enumerated(EnumType.***STRING***)).

**public enum** UserType {  
 ***PUPIL*** (**"Ученик"**),  
 ***TEACHER*** (**"Учитель"**),  
 ***PARENT*** (**"Родитель"**);  
  
 **private** String **value**;  
  
 UserType(String value) {  
 **this**.**value** = value;  
 }  
  
 **public** String getValue() {  
 **return value**;  
 }  
  
}

Связь между сущностями «многие ко многим» реализована с помощью аннотации @ManyToMany. В скобках указываются связующая таблица, поля таблицы, которые содержат идентификаторы связующих записей. Например, отношение «ученики и учителя»:

@ManyToMany()  
@JoinTable(name = **"pupils\_and\_teachers"**,  
 joinColumns = {@JoinColumn(name = **"teacher\_id"**, referencedColumnName = **"id"**)},  
 inverseJoinColumns = {@JoinColumn(name = **"pupil\_id"**, referencedColumnName = **"id"**)})  
**public** Set<Pupil> getPupils() {  
 **return pupils**;  
}  
  
**public void** setPupils(Set<Pupil> pupils) {  
 **this**.**pupils** = pupils;  
}

Аналогично реализовываются остальные классы-сущности.

Кроме того, создается конфигурационный файл hibernate.cfg.xml, в котором прописывается доступ к БД, а также указываются классы-сущности.

*<?***xml version='1.0' encoding='utf-8'***?>***<!DOCTYPE hibernate-configuration PUBLIC  
 "-//Hibernate/Hibernate Configuration DTD//EN"  
 "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd"*>***<**hibernate-configuration**>  
 <**session-factory**>  
 <**property name="hibernate.dialect"**>org.hibernate.dialect.MySQLDialect</**property**>  
 <**property name="connection.url"**>jdbc:mysql://localhost:3306/programming\_tutorial </**property**>  
 <**property name="connection.driver\_class"**>com.mysql.jdbc.Driver</**property**>  
 <**property name="connection.username"**>root</**property**>  
 <**property name="connection.password"**></**property**>  
  
  
<**mapping class="model.entities.Theme"**/><**mapping class="model.entities.Theory"**/><**mapping class="model.entities.Task"**/>  
 <**mapping class="model.entities.Pupil"**/>  
 <**mapping class="model.entities.Parent"**/>  
 <**mapping class="model.entities.Teacher"**/>  
 <**mapping class="model.entities.School"**/>  
 <**mapping class="model.entities.User"**/>  
 </**session-factory**>  
</**hibernate-configuration**>

## 3.3 DATA ACCESS OBJECTS

Для обеспечения доступа к базе данных используются классы Data Access Object (DAO). Такие классы предоставляют интерфейс для добавления, удаления, редактирования записей в таблицах БД, а также для выполнения запросов к БД: получение всех записей из таблицы, получение записи по id и др.

Как правило, DAO реализуется с помощью наследования классов, предоставляющих доступ для каждой сущности, от общего класса или интерфейса. Поскольку все сущности унаследованы от класса AbstractEntity, то можно обойтись без абстрактного класса или интерфейса, а сделать общий класс DAO обычным классом, реализующий все основные операции обработки данных: вставка, обновление, удаление. Для реализации запросов необходимо знать имя сущности, к таблице которой он осуществляется. Однако классу AbstractEntity не соответствует некая абстрактная таблица. Одним из решений является объявление перечисления Entity, в котором каждая переменная содержит имя сущности. Тогда таблицей для AbstractEntity будет Entity.getTableName(), значение которой зависит от сущности.

**package** model.daos;  
**public enum** Entity {  
 ***THEMES***(**"Theme"**),  
 ***THEORY***(**"Theory"**),  
 ***TASKS***(**"Task"**),  
 ***PARENTS***(**"Parent"**),  
 ***PUPILS***(**"Pupil"**),  
 ***SCHOOLS***(**"School"**),  
 ***USERS***(**"User"**),  
 ***TEACHERS***(**"Teacher"**);  
  
 **private** String **tableName**;  
  
 Entity(String tableName) {  
 **this**.**tableName** = tableName;  
 }  
  
 **public** String getTableName() {  
 **return tableName**;  
 }  
}

Реализация общего класса DAO выглядит следующим образом:

**package** model.daos;  
  
**import** model.entities.AbstractEntity;  
**import** org.hibernate.HibernateException;  
**import** org.hibernate.Session;  
**import** org.hibernate.Transaction;  
**import** org.hibernate.Query;  
**import** java.util.List;  
**public class** DaoImpl {  
 **private** Session **session**;  
 **private** Entity **entity**;  
 **protected** String **tableName**;  
  
 **public** DaoImpl(Session session, Entity entity) {  
 **this**.**session** = session;  
 **this**.**entity** = entity;  
 **tableName** = getEntity().getTableName();  
 }  
  
  
 **public void** create(AbstractEntity entity) {  
 Transaction transaction = **null**;  
 **try** {  
 transaction = **session**.beginTransaction();  
 **session**.save(entity);  
 transaction.commit();  
 }  
 **catch** (HibernateException e) {  
 e.printStackTrace();  
 transaction.rollback();  
 }  
 }  
  
 **public void** update(AbstractEntity entity) {  
 Transaction transaction = **null**;  
 **try** {  
 transaction = **session**.beginTransaction();  
 **session**.update(entity);  
 transaction.commit();  
 }  
 **catch** (HibernateException e) {  
 e.printStackTrace();  
 transaction.rollback();  
 }  
 }  
  
 **public void** delete(AbstractEntity entity) {  
 Transaction transaction = **null**;  
 **try** {  
 transaction = **session**.beginTransaction();  
 **session**.delete(entity);  
 transaction.commit();  
 }  
 **catch** (HibernateException e) {  
 e.printStackTrace();  
 transaction.rollback();  
 }  
 }  
  
 **public** List<? **extends** AbstractEntity> getAllEntities() {  
 **session**.createQuery(**"FROM "** + **entity**.getTableName());  
 **return session**.createQuery(**"FROM "** + **entity**.getTableName()).list();  
 }  
  
 **public** AbstractEntity getEntityById(**int** id) {  
 String tableName = getEntity().getTableName();  
  
 Query query = getSession().createQuery  
 (**"FROM "** + tableName + **" AS t "** +  
 **"WHERE t.id = "** + id);  
 **return** (AbstractEntity) query.uniqueResult();  
 }  
  
 **public** Session getSession() {  
 **return session**;  
 }  
  
 **public** Entity getEntity() {  
 **return entity**;  
 }  
}

Отображение таблиц БД на объекты обычно осуществляется один раз в течение работы приложения. Интерфейс Session используется для сохранения в базу данных и восстановления из нее объектов классов. Для создания объекта типа Session используется метод openSession() интерфейса SessionFactory. Объект типа SessionFactory создается следующим образом:

**package** model.utils;  
  
**import** org.hibernate.SessionFactory;  
**import** org.hibernate.boot.MetadataSources;  
**import** org.hibernate.boot.registry.StandardServiceRegistry;  
**import** org.hibernate.boot.registry.StandardServiceRegistryBuilder;  
  
**public class** HibernateSessionFactory {  
  
 **private static** SessionFactory *sessionFactory* = *buildSessionFactory*();  
  
 **protected static** SessionFactory buildSessionFactory() {**final** StandardServiceRegistry registry = **new** StandardServiceRegistryBuilder()  
 .configure() *// configures settings from hibernate.cfg.xml* .build();  
 **try** {  
 *sessionFactory* = **new** MetadataSources( registry   
 ).buildMetadata().buildSessionFactory();  
 }  
 **catch** (Exception e) {  
 e.printStackTrace();  
 }  
 **return** *sessionFactory*;  
 }  
  
 **public static** SessionFactory getSessionFactory() {  
 **return** *sessionFactory*;  
 }  
  
 **public static void** shutdown() {  
 *// Close caches and connection pools  
 getSessionFactory*().close();  
 }  
}

Для каждой сущности определяются свои DAO. Они наследуются от DaoImpl и при необходимости реализуют свои дополнительные методы. Названия сущностей берутся из переменной Entity entity и динамически подставляются в методы класса DaoImpl. В этом приеме проявляется действие полиморфизма в языке Java. Например, класс ThemesDaoImpl:

**package** model.daos;  
  
**import** model.entities.Theme;  
**import** org.hibernate.Query;  
**import** org.hibernate.Session;  
**import** java.util.List;  
  
**public class** ThemesDaoImpl **extends** DaoImpl {  
 **public** ThemesDaoImpl(Session session, Entity entity) {  
 **super**(session, entity);  
 }  
  
 **public** List<Theme> getThemesByClass(**int** clazz) {  
 Query query = getSession().createQuery(**"FROM "** + **tableName** + **" AS t"** + **" WHERE t.clazz="** + clazz);  
 **return** (List<Theme>)query.list();  
 }  
}

Здесь для написания запроса используется язык запросов HQL (Hibernate Query Language). Он очень похож на SQL. Основное отличие заключается в том, что SQL работает с таблицами в БД, а HQL – с объектами сущностей и их полями.

Таким образом, все DAO имеют свои методы и методы, реализованные в классе DaoImpl.

Сущности и DAO осуществляют работу с базой данных, поэтому они размещаются в ***модели*** приложения.

## 3.4 Сервлеты

Сервлеты играют роль ***контроллеров*** в веб-приложении. Они принимают данные от пользователя и определяют их смысл для модели. У каждого сервлета (класса-наследника абстрактного класса HttpServlet) есть метод service(), который определяет, каким методом был передан запрос: POST, GET или другим, после чего вызывается одноименный метод (doPost, doGet и т.д.). Основная задача разработчика – переопределить эти методы в сервлетах.

Класс Servlet пакета controller является основным в приложении и создан для обработки запросов главной страницы приложения от авторизованного или неавторизованного пользователя. Для неавторизованного пользователя это могут быть регистрация и вход. Для авторизованного – редактирование личных данных; добавление и удаление учителей, родителей, учеников в зависимости от роли пользователя; выход из системы. Реализация методов doPost и doGet следующая:

**protected void** doPost(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {  
 request.setCharacterEncoding(**"UTF-8"**);  
 String action = request.getParameter(**"action"**);  
 **if** (action.indexOf(**'#'**) > -1)  
 action = action.substring(0, action.indexOf(**'#'**));  
 **switch** (action) {  
 **case "login"**:  
 login(request, response);  
 **break**;  
 **case "register"**:  
 register(request, response);  
 **break**;  
 **case "edit"**:  
 edit(request, response);  
 **break**;  
 **case "add\_teacher"**:  
 addTeacher(request, response);  
 **break**;  
 **case "add\_pupil"**:  
 addPupil(request, response);  
 **break**;  
 **case "add\_child"**:  
 addChild(request, response);  
 **break**;  
 **case "exit"**:  
 request.getSession(**false**).invalidate();  
 *redirectToIndexJSP*(request, response);  
 **break**;  
 }  
}

**protected void** doGet(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {  
 String action = request.getParameter(**"action"**);  
  
 **if** (action != **null**) {  
 **if** (action.indexOf(**'#'**) > -1)  
 action = action.substring(0, action.indexOf(**'#'**));  
 **switch** (action) {  
 **case "delete\_teacher"**:  
 **int** teacherId = Integer.*parseInt*(request.getParameter(**"teacher\_id"**));  
 Pupil pupil = (Pupil) request.getSession(**false**).getAttribute(**"user"**);  
 ModificationsForUser.*deleteTeacher*(pupil, teacherId);  
 *redirectToUserJSP*(request, response);  
 **break**;  
 **case "delete\_pupil"**:  
 **int** pupilId = Integer.*parseInt*(request.getParameter(**"pupil\_id"**));  
 Teacher teacher = (Teacher) request.getSession(**false**).getAttribute(**"user"**);  
 ModificationsForUser.*deletePupil*(teacher, pupilId);  
 *redirectToUserJSP*(request, response);  
 **break**;  
 **case "delete\_child"**:  
 **int** childId = Integer.*parseInt*(request.getParameter(**"child\_id"**));  
 Parent parent = (Parent) request.getSession(**false**).getAttribute(**"user"**);  
 ModificationsForUser.*deleteChild*(parent, childId);  
 *redirectToUserJSP*(request, response);  
 **break**;  
 **default**:  
 HttpSession session = request.getSession(**false**);  
 **if** (session != **null**)  
 session.setAttribute(**"content"**, **"main"**);  
 *redirectToIndexJSP*(request, response);  
 }  
 }  
 **else** {  
 HttpSession session = request.getSession(**false**);  
 **if** (session != **null**) {  
 session.removeAttribute(**"clazz"**);  
 session.setAttribute(**"content"**, **"main"**);  
 }  
 *redirectToIndexJSP*(request, response);  
 }  
}

Из реализации видно, что обрабатывается запрос по атрибуту (для POST) или параметру (для GET) action. В зависимости от значения action вызывается соответствующий метод. Реализация этих методов приведена в приложении . После обработки запросов происходит перенаправление на JSP страницу с помощью методов redirectToIndexJSP (для страницы неавторизованного пользователя) или redirectToUserJSP (для страницы авторизованного пользователя).

В основном запросы влекут за собой изменения в БД. Сами методы изменения нужно размещать в ***модели*** приложения. В этих методах модификация данных происходит с помощью DAO. Для класса Servlet эти методы находятся в классе ModificationsForUser (приложение ).

Аналогичные сервлеты создаются для обработки запросов от страниц, содержащих темы, теорию и задания (ServletForThemes, ServletForTheory, ServletForTask). В сервлете ServletForThemes реализовано получение списка тем по классам, в ServletForTheory – получение теоретического материала по теме и список задач, а также подписка учащихся на задания, в ServletForTask – получение задания по id и проверка задач.

Для корректной работы и запуска сервлетов необходимо зарегистрировать их в конфигурационном файле web.xml следующим образом:

<**servlet**>  
 <**servlet-name**>Servlet</**servlet-name**>  
 <**servlet-class**>controller.Servlet</**servlet-class**>  
</**servlet**>  
  
<**servlet-mapping**>  
 <**servlet-name**>Servlet</**servlet-name**>  
 <**url-pattern**>/index</**url-pattern**>  
</**servlet-mapping**>

Тег <url-pattern> указывает URL для сервлета.

Сервлеты получают от JSP-страниц данные (содержатся в переменной request). Поэтому JSP используются в паре с сервлетами.

## 3.5 JSP

Третий компонент шаблона MVC – ***вид*** представлен в веб-приложении JSP-страницами. Они содержат как статическое содержимое (HTML, JavaScript), так и динамическое (теги управления данными, которые были отправлены из сервера).

Благодаря динамическому содержимому JSP нет необходимости писать многочисленное количество веб-страниц для однотипных данных. Достаточно сохранить данные в некоторой переменной в виде java-объекта и переслать клиенту. Такие объекты называются JavaBean. С помощью Expression Language и тегов JSTL данные из этих объектов отображаются на странице.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Блинов, И. Н. Java. Методы программирования: уч.-мет. пособие / И. Н. Блинов, В. С. Романчик. – Мн. : издательство «Четыре четверти», 2013. – 896 с.
2. Стелтинг, С. Применение шаблонов Java. Библиотека профессионала / С. Стелтинг, О. Маасен. : Пер. с англ. – М. : Издательский дом «Вильямс», 2002. – 576 с.
3. Паттерны проектирования / Э. Фримен [и др.]. – СПб. : Питер, 2011. – 656 с.
4. Информатика: учеб. пособие для 7-го кл. общеобразоват. учреждений с рус. яз. обучения / Г. А. Заборовский [и др.]; под ред. Г. А. Заборовского. – Минск : Нар. асвета, 2009. – 159 с.
5. Заборовский, Г. А. Информатика : учеб. пособие для 9-го кл. общеобразоват. учреждений с рус. яз. обучения / Г. А. Заборовский, А. И. Лапо, А. Е. Пупцев, – Минск : Нар. асвета, 2009. – 191 с.

# ПРИЛОЖЕНИЯ

## ПРИЛОЖЕНИЕ А. SQL-код для создания базы данных

SET @OLD\_UNIQUE\_CHECKS=@@UNIQUE\_CHECKS, UNIQUE\_CHECKS=0;

SET @OLD\_FOREIGN\_KEY\_CHECKS=@@FOREIGN\_KEY\_CHECKS, FOREIGN\_KEY\_CHECKS=0;

SET @OLD\_SQL\_MODE=@@SQL\_MODE, SQL\_MODE='TRADITIONAL,ALLOW\_INVALID\_DATES';

CREATE SCHEMA IF NOT EXISTS `programming\_tutorial` DEFAULT CHARACTER SET utf8 COLLATE utf8\_general\_ci ;

USE `programming\_tutorial`;

CREATE TABLE IF NOT EXISTS `programming\_tutorial`.`school` ( `id` INT(11) NOT NULL AUTO\_INCREMENT , `name` VARCHAR(255) NOT NULL , PRIMARY KEY (`id`) , UNIQUE INDEX `id\_UNIQUE` (`id` ASC) , UNIQUE INDEX `name\_UNIQUE` (`name` ASC) )ENGINE = InnoDBDEFAULT CHARACTER SET = utf8COLLATE = utf8\_general\_ci;

CREATE TABLE IF NOT EXISTS `programming\_tutorial`.`user` ( `id` INT(11) NOT NULL AUTO\_INCREMENT , `login` VARCHAR(255) NOT NULL , `password` VARCHAR(255) NOT NULL , `access` VARCHAR(20) NOT NULL , `first\_name` VARCHAR(255) NOT NULL , `last\_name` VARCHAR(255) NOT NULL , `patronymic` VARCHAR(255) NULL DEFAULT NULL , `class` INT(11) NULL DEFAULT NULL , `school\_id` INT(11) NULL DEFAULT NULL , PRIMARY KEY (`id`) , UNIQUE INDEX `id\_UNIQUE` (`id` ASC) , INDEX `userSchoolId\_idx` (`school\_id` ASC) , UNIQUE INDEX `login\_UNIQUE` (`login` ASC) , CONSTRAINT `userSchoolId` FOREIGN KEY (`school\_id` ) REFERENCES `programming\_tutorial`.`school` (`id` ) ON DELETE NO ACTION ON UPDATE NO ACTION)ENGINE = InnoDBDEFAULT CHARACTER SET = utf8COLLATE = utf8\_general\_ci;

CREATE TABLE IF NOT EXISTS `programming\_tutorial`.`pupils\_and\_parents` ( `pupil\_id` INT(11) NOT NULL , `parent\_id` INT(11) NOT NULL , `id` INT(11) NOT NULL AUTO\_INCREMENT , PRIMARY KEY (`id`) , UNIQUE INDEX `id\_UNIQUE` (`id` ASC) , INDEX `pupilsAndParentsPupilId\_idx` (`pupil\_id` ASC) , INDEX `pupilsAndParentsParentId\_idx` (`parent\_id` ASC) , CONSTRAINT `pupilsAndParentsPupilId` FOREIGN KEY (`pupil\_id` ) REFERENCES `programming\_tutorial`.`user` (`id` ) ON DELETE CASCADE ON UPDATE NO ACTION, CONSTRAINT `pupilsAndParentsParentId` FOREIGN KEY (`parent\_id` ) REFERENCES `programming\_tutorial`.`user` (`id` ) ON DELETE CASCADE ON UPDATE NO ACTION)ENGINE = InnoDBDEFAULT CHARACTER SET = utf8COLLATE = utf8\_general\_ci;

CREATE TABLE IF NOT EXISTS `programming\_tutorial`.`pupils\_and\_teachers` ( `pupil\_id` INT(11) NOT NULL , `teacher\_id` INT(11) NOT NULL , `id` INT(11) NOT NULL AUTO\_INCREMENT , PRIMARY KEY (`id`) , INDEX `pupilsAndTeachersPupilId\_idx` (`pupil\_id` ASC) , INDEX `pupilsAndTeachersTeacherId\_idx` (`teacher\_id` ASC) , CONSTRAINT `pupilsAndTeachersPupilId` FOREIGN KEY (`pupil\_id` ) REFERENCES `programming\_tutorial`.`user` (`id` ) ON DELETE CASCADE ON UPDATE NO ACTION, CONSTRAINT `pupilsAndTeachersTeacherId` FOREIGN KEY (`teacher\_id` ) REFERENCES `programming\_tutorial`.`user` (`id` ) ON DELETE CASCADE ON UPDATE NO ACTION)ENGINE = InnoDBDEFAULT CHARACTER SET = utf8COLLATE = utf8\_general\_ci;

CREATE TABLE IF NOT EXISTS `programming\_tutorial`.`theme` ( `id` INT(11) NOT NULL AUTO\_INCREMENT , `title` VARCHAR(255) NOT NULL , `class` INT(11) NOT NULL , PRIMARY KEY (`id`) , UNIQUE INDEX `id\_UNIQUE` (`id` ASC) )ENGINE = InnoDBDEFAULT CHARACTER SET = utf8COLLATE = utf8\_general\_ci;

CREATE TABLE IF NOT EXISTS `programming\_tutorial`.`theory` ( `id` INT(11) NOT NULL AUTO\_INCREMENT , `content` TEXT NOT NULL , `theme\_id` INT(11) NOT NULL , `user\_id` INT(11) NULL DEFAULT NULL , `title` VARCHAR(255) NOT NULL , PRIMARY KEY (`id`) , UNIQUE INDEX `id\_UNIQUE` (`id` ASC) , INDEX `theoryUserId\_idx` (`user\_id` ASC) , INDEX `theoryThemeId\_idx` (`theme\_id` ASC) , CONSTRAINT `theoryUserId` FOREIGN KEY (`user\_id` ) REFERENCES `programming\_tutorial`.`user` (`id` ) ON DELETE NO ACTION ON UPDATE NO ACTION, CONSTRAINT `theoryThemeId` FOREIGN KEY (`theme\_id` ) REFERENCES `programming\_tutorial`.`theme` (`id` ) ON DELETE CASCADE ON UPDATE NO ACTION)ENGINE = InnoDBDEFAULT CHARACTER SET = utf8COLLATE = utf8\_general\_ci;

CREATE TABLE IF NOT EXISTS `programming\_tutorial`.`task` ( `id` INT(11) NOT NULL AUTO\_INCREMENT , `title` VARCHAR(255) NOT NULL , `type` VARCHAR(45) NOT NULL , `content` VARCHAR(255) NOT NULL , `user\_id` INT(11) NULL DEFAULT NULL , `theme\_id` INT(11) NOT NULL , PRIMARY KEY (`id`) , UNIQUE INDEX `id\_UNIQUE` (`id` ASC) , INDEX `taskThemeId\_idx` (`theme\_id` ASC) , INDEX `taskUserId\_idx` (`user\_id` ASC) , CONSTRAINT `taskThemeId` FOREIGN KEY (`theme\_id` ) REFERENCES `programming\_tutorial`.`theme` (`id` ) ON DELETE CASCADE ON UPDATE NO ACTION, CONSTRAINT `taskUserId` FOREIGN KEY (`user\_id` ) REFERENCES `programming\_tutorial`.`user` (`id` ) ON DELETE NO ACTION ON UPDATE NO ACTION)ENGINE = InnoDBDEFAULT CHARACTER SET = utf8COLLATE = utf8\_general\_ci;

CREATE TABLE IF NOT EXISTS `programming\_tutorial`.`users\_and\_tasks` ( `id` INT(11) NOT NULL AUTO\_INCREMENT , `user\_id` INT(11) NOT NULL , `task\_id` INT(11) NOT NULL , `status` INT(11) NULL DEFAULT NULL , `date` DATETIME NULL DEFAULT NULL , PRIMARY KEY (`id`) , INDEX `pupilsAndTasksPupilId\_idx` (`user\_id` ASC) , INDEX `pupilsAndTasksTaskId\_idx` (`task\_id` ASC) , CONSTRAINT `pupilsAndTasksPupilId` FOREIGN KEY (`user\_id` ) REFERENCES `programming\_tutorial`.`user` (`id` ) ON DELETE CASCADE ON UPDATE NO ACTION, CONSTRAINT `pupilsAndTasksTaskId` FOREIGN KEY (`task\_id` ) REFERENCES `programming\_tutorial`.`task` (`id` ) ON DELETE CASCADE ON UPDATE NO ACTION)ENGINE = InnoDBDEFAULT CHARACTER SET = utf8COLLATE = utf8\_general\_ci;CREATE TABLE IF NOT EXISTS `programming\_tutorial`.`pupils\_and\_themes` ( `id` INT(11) NOT NULL AUTO\_INCREMENT , `pupil\_id` INT(11) NOT NULL , `theme\_id` INT(11) NOT NULL , PRIMARY KEY (`id`) , UNIQUE INDEX `id\_UNIQUE` (`id` ASC) , INDEX `pupilsAndThemesPupilId\_idx` (`pupil\_id` ASC) , INDEX `pupilsAndThemesThemeId\_idx` (`theme\_id` ASC) , CONSTRAINT `pupilsAndThemesPupilId` FOREIGN KEY (`pupil\_id` ) REFERENCES `programming\_tutorial`.`user` (`id` ) ON DELETE CASCADE ON UPDATE CASCADE, CONSTRAINT `pupilsAndThemesThemeId` FOREIGN KEY (`theme\_id` ) REFERENCES `programming\_tutorial`.`theme` (`id` ) ON DELETE CASCADE ON UPDATE CASCADE)ENGINE = InnoDBDEFAULT CHARACTER SET = utf8COLLATE = utf8\_general\_ci;SET SQL\_MODE=@OLD\_SQL\_MODE;SET FOREIGN\_KEY\_CHECKS=@OLD\_FOREIGN\_KEY\_CHECKS;SET UNIQUE\_CHECKS=@OLD\_UNIQUE\_CHECKS;

## ПРИЛОЖЕНИЕ Б. Реализация класса Servlet

**package** controller;  
  
**import** model.daos.\*;  
**import** model.entities.\*;  
**import** model.service.ModificationsForUser;  
**import** model.utils.HibernateSessionFactory;  
**import** org.apache.commons.codec.digest.DigestUtils;  
**import** org.hibernate.Session;  
  
**import** javax.servlet.ServletException;  
**import** javax.servlet.annotation.WebServlet;  
**import** javax.servlet.http.HttpServlet;  
**import** javax.servlet.http.HttpServletRequest;  
**import** javax.servlet.http.HttpServletResponse;  
**import** javax.servlet.http.HttpSession;  
**import** java.io.IOException;  
**import** java.util.List;  
  
@WebServlet(**"/index"**)  
**public class** Servlet **extends** HttpServlet {  
 **public static final** Session ***SESSION*** = HibernateSessionFactory.*getSessionFactory*().openSession();  
  
  
 **protected void** doPost(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {  
 request.setCharacterEncoding(**"UTF-8"**);  
 String action = request.getParameter(**"action"**);  
 **if** (action.indexOf(**'#'**) > -1)  
 action = action.substring(0, action.indexOf(**'#'**));  
 **switch** (action) {  
 **case "login"**:  
 login(request, response);  
 **break**;  
 **case "register"**:  
 register(request, response);  
 **break**;  
 **case "edit"**:  
 edit(request, response);  
 **break**;  
 **case "add\_teacher"**:  
 addTeacher(request, response);  
 **break**;  
 **case "add\_pupil"**:  
 addPupil(request, response);  
 **break**;  
 **case "add\_child"**:  
 addChild(request, response);  
 **break**;  
 **case "exit"**:  
 request.getSession(**false**).invalidate();  
 *redirectToIndexJSP*(request, response);  
 **break**;  
 }  
 }  
  
 **private void** addChild(HttpServletRequest request, HttpServletResponse response) **throws** IOException, ServletException {  
 **int** childId = Integer.*parseInt*(request.getParameter(**"child-hidden"**).trim());  
 Parent parent = (Parent) request.getSession(**false**).getAttribute(**"user"**);  
 ModificationsForUser.*addChild*(parent, childId);  
 *redirectToUserJSP*(request, response);  
 }  
  
 **private void** addPupil(HttpServletRequest request, HttpServletResponse response) **throws** IOException, ServletException {  
 **int** pupilId = Integer.*parseInt*(request.getParameter(**"child-hidden"**).trim());  
 Teacher teacher = (Teacher) request.getSession(**false**).getAttribute(**"user"**);  
 ModificationsForUser.*addPupil*(teacher, pupilId);  
 *redirectToUserJSP*(request, response);  
 }  
  
 **private void** addTeacher(HttpServletRequest request, HttpServletResponse response) **throws** IOException, ServletException {  
 **int** teacherId = Integer.*parseInt*(request.getParameter(**"teacher-hidden"**).trim());  
 Pupil pupil = (Pupil) request.getSession(**false**).getAttribute(**"user"**);  
 ModificationsForUser.*addTeacher*(pupil, teacherId);  
 *redirectToUserJSP*(request, response);  
 }  
 **private void** edit(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {  
 String lastName = request.getParameter(**"lastName"**).trim();  
 String firstName = request.getParameter(**"firstName"**).trim();  
 String patronymic = request.getParameter(**"patronymic"**).trim();  
 String clazz = request.getParameter(**"class"**);  
 String schoolId = request.getParameter(**"school-hidden"**);  
 User user = (User)request.getSession(**false**).getAttribute(**"user"**);  
 ModificationsForUser.*edit*(user, lastName, firstName, patronymic, clazz, schoolId);  
 *redirectToUserJSP*(request, response);  
 }  
  
 **private void** register(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {  
 String lastName = request.getParameter(**"lastName"**).trim();  
 String firstName = request.getParameter(**"firstName"**).trim();  
 String patronymic = request.getParameter(**"patronymic"**).trim();  
 String login = request.getParameter(**"login"**).trim().toLowerCase();  
 String password = DigestUtils.*md5Hex*(request.getParameter(**"password"**)).trim();  
 String accessStr = request.getParameter(**"access"**);  
 UserType access = UserType.*valueOf*(accessStr);  
 User newUser = ModificationsForUser.*register*(login, password, access, firstName, lastName, patronymic);  
 **if** (ModificationsForUser.*userIsExist*(login) == **null**) {  
 createRole(newUser, access, request, response);  
 login(request, response);  
 }  
 **else** {  
 request.setAttribute(**"result"**, **"Пользователь с таким логином уже существует или данные некорректны"**);  
 request.getRequestDispatcher(**"/index.jsp"**).forward(request, response);  
 }  
 }  
  
 **private void** createRole(User user, UserType userType, HttpServletRequest request, HttpServletResponse response) {  
 **switch** (userType) {  
 **case *PUPIL***:  
 **int** clazz = Integer.*parseInt*(request.getParameter(**"clazz"**).trim());  
 **int** schoolId = Integer.*parseInt*(request.getParameter(**"school-hidden"**).trim());  
 **int** teacherId = Integer.*parseInt*(request.getParameter(**"teacher-hidden"**).trim());  
 ModificationsForUser.*createRole*(user, userType, clazz, schoolId, teacherId);  
 **break**;  
 **case *TEACHER***:  
 ModificationsForUser.*createRole*(user, userType);  
 **break**;  
 **case *PARENT***:  
 **int** pupilId = Integer.*parseInt*(request.getParameter(**"child-hidden"**));  
 ModificationsForUser.*createRole*(user, userType, pupilId);  
 **break**;  
 }  
 }  
  
 **private void** login(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {  
 String login = request.getParameter(**"login"**).toLowerCase();  
 String password = request.getParameter(**"password"**);  
 User user = ModificationsForUser.*login*(login, password);  
  
 **if** (user != **null**) {  
 request.getSession(**false**).invalidate();  
 HttpSession newSession = request.getSession(**true**);  
 newSession.setAttribute(**"user"**, user);  
  
 newSession.setAttribute(**"schools"**, ModificationsForUser.*getEntities*(Entity.***SCHOOLS***));  
 newSession.setAttribute(**"teachers"**, ModificationsForUser.*getEntities*(Entity.***TEACHERS***));  
 newSession.setAttribute(**"pupils"**, ModificationsForUser.*getEntities*(Entity.***PUPILS***));  
 newSession.setAttribute(**"content"**, **"main"**);  
 *redirectToUserJSP*(request, response);  
 }  
 **else** {  
 request.setAttribute(**"result"**, **"Данные некорректны"**);  
 request.getRequestDispatcher(**"/index.jsp"**).forward(request, response);  
 }  
 }  
  
 **protected void** doGet(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {  
 String action = request.getParameter(**"action"**);  
 **if** (action != **null**) {  
 **if** (action.indexOf(**'#'**) > -1)  
 action = action.substring(0, action.indexOf(**'#'**));  
 **switch** (action) {  
 **case "delete\_teacher"**:  
 **int** teacherId = Integer.*parseInt*(request.getParameter(**"teacher\_id"**));  
 Pupil pupil = (Pupil) request.getSession(**false**).getAttribute(**"user"**);  
 ModificationsForUser.*deleteTeacher*(pupil, teacherId);  
 *redirectToUserJSP*(request, response);  
 **break**;  
 **case "delete\_pupil"**:  
 **int** pupilId = Integer.*parseInt*(request.getParameter(**"pupil\_id"**));  
 Teacher teacher = (Teacher) request.getSession(**false**).getAttribute(**"user"**);  
 ModificationsForUser.*deletePupil*(teacher, pupilId);  
 *redirectToUserJSP*(request, response);  
 **break**;  
 **case "delete\_child"**:  
 **int** childId = Integer.*parseInt*(request.getParameter(**"child\_id"**));  
 Parent parent = (Parent) request.getSession(**false**).getAttribute(**"user"**);  
 ModificationsForUser.*deleteChild*(parent, childId);  
 *redirectToUserJSP*(request, response);  
 **break**;  
 **default**:  
 HttpSession session = request.getSession(**false**);  
 **if** (session != **null**)  
 session.setAttribute(**"content"**, **"main"**);  
 *redirectToIndexJSP*(request, response);  
 }  
 }  
 **else** {  
 HttpSession session = request.getSession(**false**);  
 **if** (session != **null**) {  
 session.removeAttribute(**"clazz"**);  
 session.setAttribute(**"content"**, **"main"**);  
 }  
 *redirectToIndexJSP*(request, response);  
 }  
 }  
  
 **protected static void** redirectToIndexJSP(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {  
 HttpSession session = request.getSession(**false**);  
 **if** (session == **null** || session.getAttribute(**"user"**) == **null**) {  
 session = request.getSession(**true**);  
 session.setAttribute(**"schools"**, ModificationsForUser.*getEntities*(Entity.***SCHOOLS***));  
 session.setAttribute(**"teachers"**, ModificationsForUser.*getEntities*(Entity.***TEACHERS***));  
 session.setAttribute(**"pupils"**, ModificationsForUser.*getEntities*(Entity.***PUPILS***));  
 request.getRequestDispatcher(**"/index.jsp"**).forward(request, response);  
 }  
 **else** {  
 *//session.setAttribute("content", "main");* **if** (!session.getAttribute(**"content"**).equals(**"task"**)) {  
 session.removeAttribute(**"result"**);  
 session.removeAttribute(**"input"**);  
 session.removeAttribute(**"expected"**);  
 }  
 *redirectToUserJSP*(request, response);  
 }  
 }  
  
 **protected static void** redirectToUserJSP(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {  
  
 request.getRequestDispatcher(**"/user.jsp"**).forward(request, response);  
 }  
  
 @Override  
 **public void** destroy() {  
 ***SESSION***.close();  
 HibernateSessionFactory.*shutdown*();  
 **super**.destroy();  
 }  
  
}