МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования

«Гомельский государственный технический университет

имени П. О. Сухого»

Факультет автоматизированных и информационных систем

Кафедра «Информационные технологии»

направление специальности 1-40 01 02-01 «Информационные системы и

технологии в проектировании и производстве»

**РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

к курсовому проекту

по дисциплине «Базы знаний и поддержка принятия решений в САПР»

на тему: «Разработка базы знаний для интерактивной программы помощника (*chat bot*) для выбора услуг города Гомеля»

|  |  |  |  |
| --- | --- | --- | --- |
|  | Исполнитель: студентка группы ИТ-52 | | |
|  |  | Гуща А.В. | |
|  | Руководитель: профессор | | |
|  |  | | Мурашко И. А. |
|  |
|  | Дата проверки \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | | |
|  | Дата допуска к защите \_\_\_\_\_\_\_\_\_\_\_\_\_ | | |
|  | Дата защиты \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | | |
|  |  | | |
|  | Оценка работы: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | | |
|  |  | | |
| Подписи членов комиссии  По защите курсового проекта: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | | | |

Гомель 2016

**СОДЕРЖАНИЕ**

ВВЕДЕНИЕ 4

1 АНАЛИЗ БАЗ ЗНАНИЙ И СИСТЕМ ПОДДЕРЖКИ ПРИНЯТИЯ

РЕШЕНИЙ 5

1.1 Базы знаний 5

1.2 Система принятия решений 7

1.3 Существующие технологии разработки программных средств 9

1.4 Кроссплатформенность *Java*-приложений 11

2 АЛГОРИТМИЧЕСКИЙ АНАЛИЗ ЗАДАЧИ 13

2.1 Постановка задачи 13

2.2 Информационная модель 13

3 ОПИСАНИЕ РАЗРАБОТАННОГО ПРИЛОЖЕНИЯ 16

3.1 Особенности программного комплекса 16

3.2 Описание основной структуры приложения 18

3.3 Описание интерфейса пользователя 22

ЗАКЛЮЧЕНИЕ 24

Список использованных источников 25

ПРИЛОЖЕНИЕ А. Листинг классов 32

**ВВЕДЕНИЕ**

Интерфейсы взаимодействия человека и компьютера постоянно эволюционировали. Начиная с удобных компьютеру интерфейсов взаимодействия (перфокарт и магнитных лент) постепенно развились упрощая коммуникацию человека и компьютера (монитор, клавиатура и мышь). Современные мобильные устройства еще сильнее упростили это взаимодействие использую сенсорные экраны. Судя по тенденции дальнейшая эволюция будет двигать в сторону удобных для человека интерфейсов таких как голосовой ввод.

Наиболее удобным средством коммуникации для человека является нативный разговорный язык. Поэтому один из будущих интерфейсов будет основан на распознании текста свободной формы и возврат результата в том же виде.

**1 АНАЛИЗ БАЗ ЗНАНИЙ И СИСТЕМ ПОДДЕРЖКИ ПРИНЯТИЯ РЕШЕНИЙ**

**1.1 Базы знаний**

При изучении интеллектуальных систем традиционно возникает вопрос – что же такое знания и чем они отличаются от обычных данных, десятилетиями обрабатываемых ЭВМ. Можно предложить несколько рабочих определений, в рамках которых это становится очевидным.

Данные – это отдельные факты, характеризующие объекты, процессы и явления предметной области, а также их свойства.

Знания основаны на данных, полученных эмпирическим путем. Они представляют собой результат мыслительной деятельности человека, направленной на обобщение его опыта, полученного в результате практической деятельности.

Знания – это закономерности предметной области (принципы, связи, законы), полученные в результате практической деятельности и профессионального опыта, позволяющие специалистам ставить и решать задачи в этой области. Знания – это хорошо структурированные данные, или данные о данных, или метаданные.

База данных – организованная в соответствии с определёнными правилами и поддерживаемая в памяти компьютера совокупность данных, характеризующая актуальное состояние некоторой предметной области и используемая для удовлетворения информационных потребностей пользователей.

Между базами знаний и базами данных можно привести ряд отличий, наиболее значимые представлены ниже.

Базы данных:

* могут работать с однородными данными;
* представляет собой жестко структурированную модель;
* данные представлены в виде набора записей.

Базы знаний:

* могут содержать разнородные и разнотипные данные;
* представляют собой открытую модель;
* знания представлены в виде семантической сети.

База знаний (БЗ) – это особого рода база данных, разработанная для управления знаниями (метаданными), то есть сбором, хранением, поиском и выдачей знаний. Раздел искусственного интеллекта, изучающий базы знаний и методы работы со знаниями, называется инженерией знаний.

Под базами знаний понимает совокупность фактов и правил вывода, допускающих логический вывод и осмысленную обработку информации. В языке Пролог базы знаний описываются в форме конкретных фактов и правил логического вывода над базами данных и процедурами обработки информации, представляющих сведения и знания о людях, предметах, фактах событиях и процессах в логической форме.

В зависимости от уровня сложности систем, в которых применяются базы знаний, различают:

* БЗ всемирного масштаба – например, Интернет или Википедия;
* БЗ национальные – например, Википедия;
* БЗ отраслевые – например, Автомобильная энциклопедия;
* БЗ организаций;
* БЗ экспертных систем;
* БЗ специалистов.

Простые базы знаний могут использоваться для создания экспертных систем и хранения данных об организации: документации, руководств, статей технического обеспечения. Главная цель создания таких баз – помочь менее опытным людям найти существующее описание способа решения какой-либо проблемы предметной области.

В настоящее время известно значительное количество программных систем, используемых для решения таких сложных задач, как оценка ситуации и выбор решения при управлении сложными процессами; оценка и выбор оптимальных проектных решений; техническая и медицинская диагностика; оценка кредитных и инвестиционных рисков; прогнозирование финансово-экономических параметров: доходности предприятия, курсов валют, биржевых цен и т.п. В силу интеллектуального характера решаемых задач, а также того, что самим системам присущи способность достигать высокого качества формируемых решений, а также обучаться и объяснять свои решения, такие системы называются интеллектуальными и объединяют достаточно широкий круг программных продуктов. К ним относятся и экспертные системы, системы для численного обоснования принятия решения, системы для распознавания образов (текстов, изображения, речи) и некоторые другие.

В программировании в целом наметилась быстро развивающаяся тенденция к интеллектуализации. Так, сейчас практически все популярные программные продукты в большей или меньшей степени приобрели черты интеллектуальности. Запоминание действий пользователя, подсказки и рекомендации по оптимальному использованию соответствующих режимов самих программ, сложные ассоциативные поиски, оптимизация кода при программировании, сложные механизмы обеспечения корректности данных при вводе и хранении – вот неполный перечень признаков «поумнения» современных программных средств. А это, в свою очередь, является немаловажной предпосылкой дальнейшего прогресса в разработке интеллектуальных систем (ИС).

Одной из основных проблем при создании ИС является выбор модели представления знаний. Именно модель представления знаний определяет архитектуру, возможности и свойства системы, а также методы приобретения знаний ИС. В настоящее время разработано множество моделей представления знаний. Имея обобщенное название, они различаются по идеям, лежащим в их основе, с точки зрения математической обоснованности.

Наибольшую известность получили следующие модели:

* продукционные модели – модель основанная на правилах, позволяет представить знание в виде предложений типа: «ЕСЛИ условие, ТО действие». Продукционная модель обладает тем недостатком, что при накоплении достаточно большого числа (порядка нескольких сотен) продукций они начинают противоречить друг другу;
* сетевые модели (или семантические сети) – в инженерии знаний под ней подразумевается граф, отображающий смысл целостного образа. Узлы графа соответствуют понятиям и объектам, а дуги – отношениям между объектами. Обладает тем недостатком, что однозначного определения семантической сети в настоящее время отсутствует;
* фреймовая модель – основывается на таком понятии как фрейм. Фрейм – структура данных для представления некоторого концептуального объекта. Информация, относящаяся к фрейму, содержится в составляющих его слотах. Слоты могут быть терминальными либо являться сами фреймами, т.о. образуя целую иерархическую сеть [1].

**1.2 Система принятия решений**

Системы поддержки принятия решений или СППР – это компьютерная система, которая путем сбора и анализа большого количества информации может влиять на процесс принятия решений организационного плана. Интерактивные системы позволяют руководителям получить полезную информацию из первоисточников, проанализировать ее, а также выявить существующие модели для решения определенных задач.

Система поддержки решений СППР решает две основные задачи:

* выбор наилучшего решения из множества возможных (оптимизация);
* упорядочение возможных решений по предпочтительности (ранжирование).

В обеих задачах первым и наиболее принципиальным моментом является выбор совокупности критериев, на основе которых в дальнейшем будут оцениваться, и сопоставляться возможные решения (альтернативы). Система СППР помогает пользователю сделать такой выбор.

Для анализа и выработок предложений в СППР используются разные методы. Это могут быть: информационный поиск, интеллектуальный анализ данных, поиск знаний в базах данных, рассуждение на основе прецедентов, имитационное моделирование, эволюционные вычисления и генетические алгоритмы, нейронные сети, ситуационный анализ, когнитивное моделирование и др. Некоторые из этих методов были разработаны в рамках искусственного интеллекта. Если в основе работы СППР лежат методы искусственного интеллекта, то говорят об интеллектуальной СППР, или ИСППР.

Близкие к СППР классы систем – это экспертные системы и автоматизированные системы управления. Модель управления и управления данными действуют, в основном, незаметно и варьируются от простой модели до сложной комплексной модели планирования, основанной на математическом программировании.

Наиболее широкой сферой практического применения СППР являются планирование и прогнозирование для различных видов управленческой деятельности.

Системы поддержки принятия решений:

* предполагают гибкость пользователей, адаптируемость и быструю реакцию;
* допускают, чтобы пользователи управляли входом и выходом;
* оперируют с небольшой помощью профессиональных программистов или без нее;
* обеспечивают поддержку для решений и проблем, которые не могут быть определены заранее;
* используют сложный анализ и инструментальные средства моделирования.

Процесс принятия решений в СППР, включает четыре стадии:

1) распознавание или осмысление – состоит из идентификации и понимания проблем, встречающихся в организации: почему проблемы возникают, где и с каким результатом. Традиционные управляющие информационные системы (УИС), которые поставляют широкое многообразие детальной информации, могут помогать опознавать проблемы, особенно если системы сообщают об исключениях;

2) проект или продумывание – в течение принятия решений лицо, принимающее решение, продумывает возможные варианты решения проблем. Малые системы СППР идеальны в этой стадии принятия решений, потому что они оперируют на простых моделях, могут быть быстро развиты и работать с ограниченными данными;

3) выбор – заключается в подборе решений среди альтернатив. Здесь изготовитель решений мог бы нуждаться в большой системе СППР, чтобы использовать более обширные данные относительно ряда альтернатив и комплексные аналитические модели, чтобы объяснить все затраты, следствия и возможности;

4) реализация – в течение выполнения решения менеджеры могут использовать систему сообщения, которая поставляет обычные доклады относительно прогресса определенного решения. Системы поддержки выполнения могут быть от полномасштабной управляющей информационной системы до меньших систем, таких, как программное обеспечение планирования проекта, использующего микрокомпьютеры.

Рассмотрим основные классификации СППР.

По взаимодействию с пользователем выделяют три вида СППР:

* пассивные помогают в процессе принятия решений, но не могут выдвинуть конкретного предложения;
* активные непосредственно участвуют в разработке правильного решения;
* кооперативные предполагают взаимодействие СППР с пользователем.

Выдвинутое системой предложение пользователь может доработать, усовершенствовать, а затем отправить обратно в систему для проверки. После этого предложение вновь представляется пользователю, и так до тех пор, пока он не одобрит решение.

По способу поддержки различают:

* модельно-ориентированные СППР, используют в работе доступ к статистическим, финансовым или иным моделям;
* СППР, основанные на коммуникациях, поддерживают работу двух и более пользователей, занимающихся общей задачей;
* СППР, ориентированные на данные, имеют доступ к временным рядам организации. Они используют в работе не только внутренние, но и внешние данные;
* СППР, ориентированные на документы, манипулируют неструктурированной информацией, заключенной в различных электронных форматах;
* СППР, ориентированные на знания, предоставляют специализированные решения проблем, основанные на фактах.

По сфере использования выделяют общесистемные и настольные СППР. Общесистемные работают с большими системами хранения данных и применяются многими пользователями. Настольные являются небольшими системами и подходят для управления с персонального компьютера одного пользователя [2].

**1.3 Существующие технологии разработки программных средств**

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

Достоинством подобного способа выполнения программ является полная независимость байт-кода от операционной системы и оборудования, что позволяет выполнять *Java-*приложения на любом устройстве, для которого существует соответствующая виртуальная машина. Другой важной особенностью технологии *Java* является гибкая система безопасности благодаря тому, что исполнение программы полностью контролируется виртуальной машиной. Любые операции, которые превышают установленные полномочия программы (например, попытка несанкционированного доступа к данным или соединения с другим компьютером) вызывают немедленное прерывание [3].

Специальный процесс сборки мусора – это одна из интереснейших особенностей языка программирования *Java* и среды выполнения приложений *Java*, предназначенная для удаления ненужных объектов из памяти. Эта система избавляет программиста от необходимости внимательно следить за использованием памяти, освобождая ненужные более области явным образом.

Разработку приложений можно вести в среде *Eclipse*, используя при этом плагин – *Android Development Tools (ADT)* или в *IntelliJ* *IDEA*. Версия *JDK* при этом должна быть 5.0 или выше.

Язык *Java* обладает большой библиотекой программ для передачи данных на основе протоколов *TCP/IP* или *FTP*.

Приложения, написанные на языке *Java*, могут открывать объекты и получать к ним доступ через сеть с помощью *URL*-адресов так же просто, как и в локальной сети.

Язык *Java* предоставляет мощные и удобные средства для работы в сети. Каждый, кто когда-либо пытался писать программы для работы в сети интернет на других языках программирования, удивлен тем, как легко решаются на языке *Java* самые трудные задачи, к примеру, открытие сетевых соединений.

Отлаженный механизм, состоящий из так называемых сервлетов (*servlets*), дает возможность работать с сервером очень просто и эффективно.

Язык *Java* в первую очередь предназначен для создания программ, которые должны надежно работать на любых платформах и под любой нагрузкой. Основное внимание в языке *Java* было уделено раннему обнаружению возможных ошибок, динамической проверке (во время выполнения программы), а также исключению ситуаций, которые могут привести к ошибкам.

Программы, написанные на *Java*, имеют репутацию более медленных и занимающих больше оперативной памяти, чем написанные на языке *C*. Тем не менее, скорость выполнения программ, написанных на языке *Java*, была существенно улучшена с выпуском в 1997–1998 годах так называемого JIT-компилятора в версии 1.1 в дополнение к другим особенностям языка для поддержки лучшего анализа кода (такие, как внутренние классы, класс *StringBuffer*, упрощенные логические вычисления и т.д.). Кроме того, была произведена оптимизация виртуальной машины *Java* – с 2000 года для этого используется виртуальная машина *HotSpot*. По состоянию на февраль 2012 года, код *Java* 7 приблизительно лишь в 1.8 раза медленнее кода, написанного на языке *С*.

Некоторые платформы предлагают аппаратную поддержку выполнения для *Java*. К примеру, микроконтроллеры, выполняющие код *Java* на аппаратном обеспечении вместо программной *JVM*, а также основанные на *ARM* процессоры, которые поддерживают выполнение байткода *Java* через опцию *Jazelle*.

Основные возможности:

* автоматическое управление памятью;
* расширенные возможности обработки исключительных ситуаций;
* богатый набор средств фильтрации ввода-вывода;
* набор стандартных коллекций: массив, список, стек и т. п.;
* наличие простых средств создания сетевых приложений;
* наличие классов, позволяющих выполнять *HTTP*-запросы и обрабатывать ответы;
* встроенные в язык средства создания многопоточных приложений, которые потом были портированы на многие языки (например, *python*);
* унифицированный доступ к базам данных;
* поддержка обобщений;
* поддержка лямбд, замыканий, встроенные возможности функционального программирования;
* параллельное выполнение программ [4].

*Java* – интерпретируемый, многопоточный и динамический язык. Интерпретируемая природа позволяет сделать фазу линкования простой, инкрементальной и, следовательно, быстрой. Это резко сокращает цикл разработки и тестирования программных фрагментов.

Многопоточность позволяет выполнять в рамках одного приложения несколько задач одновременно. Это становится особенно актуально в современных распределенных приложениях, когда процессы сетевого обмена могут идти одновременно и асинхронно. При этом программа продолжает реагировать на ввод информации пользователем без неприятных задержек.

Многопоточность поддерживается на уровне языка – часть примитивов синхронизации встроена в систему реального времени, а библиотека содержит базовый класс *Thread*. К тому же системные библиотеки написаны *thread-safe*, т.е. все они могут быть использованы в многопоточных приложениях [5].

Система обеспечивает динамическую сборку программы. Классы подгружаются по мере необходимости, причем загружены они могут быть с любой точки сети, что позволяет сделать внесение изменений в приложения прозрачным для пользователя. Пользователь может быть уверен, что всегда работает со свежей версией приложения.

**1.4 Кроссплатформенность *Java*-приложений**

Создание приложений, действительно работающих на разных платформах – непростая задача. К сожалению, дело не ограничивается необходимостью перекомпиляции исходного текста программы для работы в другой среде. Много проблем возникает с несовместимостью программных интерфейсов различных операционных систем и графических оболочек, реализующих пользовательский интерфейс.

Программа на языке *Java* компилируется в двоичный модуль, состоящий из команд виртуального процессора *Java*. Такой модуль содержит байт-код, предназначенный для выполнения *Java*-интерпретатором. На настоящий момент уже созданы первые модели физического процессора, способного выполнять этот байт-код, однако интерпретаторы *Java* имеются на всех основных компьютерных платформах. Разумеется, на каждой платформе используется свой интерпретатор, или, точнее говоря, свой виртуальный процессор *Java*.

Если ваше приложение *Java* (или апплет) должно работать на нескольких платформах, нет необходимости компилировать его исходные тексты несколько раз. Вы можете откомпилировать и отладить приложение *Java* на одной, наиболее удобной для вас платформе. В результате вы получите кроссплатформенное приложение, способное работать в любой среде, поддерживающей процессор *Java*.

Внутренняя реализация библиотек классов, зависит от платформы. Однако все загрузочные модули, реализующие возможности этих библиотек, поставляются в готовом виде вместе с виртуальной машиной *Java*, поэтому программисту не нужно об этом заботиться. Для операционной системы *Windows*, например, поставляются библиотеки динамической загрузки *DLL*, внутри которых запрятана вся функциональность стандартных классов *Java*.

Абстрагируясь от аппаратуры на уровне библиотек классов, программисты могут больше не заботиться о различиях в реализации программного интерфейса конкретных операционных систем. Это позволяет создавать по-настоящему мобильные приложения, не требующие при переносе на различные платформы перетрансляции и изменения исходного текста [5].

**2 АЛГОРИТМИЧЕСКИЙ АНАЛИЗ ЗАДАЧИ**

**2.1 Постановка задачи**

Необходимо разработать интерактивную программу помощник для выбора услуг города Гомеля.

Программное средство должно выполнять следующие функции:

* работать как сетевой ресурс;
* содержать базу данных услуг и средства подддержания ее в актуальном состоянии;
* вести диалог с пользователем на языке близком к естественному;
* иметь интерфейс для отправки и получения текстовых сообщений.
  1. **Информационная модель**

Общий ресурс, или общий сетевой ресурс – в информатике, это устройство или часть информации, к которой может быть осуществлён удалённый доступ с другого компьютера, обычно через локальную компьютерную сеть или посредством корпоративного интернета, как если бы ресурс находился на локальной машине. Примерами такого могут служить общий доступ к файлам (также известный как общий доступ к диску и общий доступ к папкам), общий доступ к принтеру (совместный доступ к принтеру), сканеру и т. п.

Веб-приложения создаются в различных вариантах архитектуры клиент-сервер. В таком случае клиентом выступает браузер, а сервером – веб-сервер.

Логика веб-приложения распределена между сервером и клиентом, хранение и обработка данных осуществляется, преимущественно, на сервере, её представление в удобном для пользователя виде – в браузере. Обмен информацией происходит по сети. Общая структура веб-приложения приведена на рисунке 2.1.

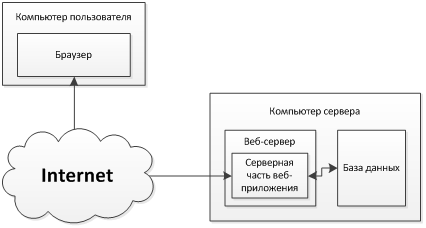


Рисунок 2.1 – Общая структура веб-приложения

Таким образом, чаще всего веб-приложения состоят как минимум из трёх основных компонентов.

Серверная часть веб-приложения – это программа или скрипт на сервере, обрабатывающая запросы пользователя (точнее, запросы браузера). Для программирования серверной части веб-приложения выбран язык Java. При каждом переходе пользователя по ссылке браузер отправляет запрос серверу. Сервер обрабатывает данный запрос, запуская некоторый скрипт, который формирует веб-страницу, описанную на языке HTML, и отсылает клиенту по сети. Браузер тут же отображает полученный в виде очередной веб-страницы результат [6].

Серверная часть веб-приложения обращается к базе, извлекая данные, которые необходимы для формирования страницы, запрошенной пользователем. В данном приложении база знаний представляет собой файлы формата *XML*.

XML-документ представляет собой обычный текстовый файл, в котором при помощи специальных маркеров создаются элементы данных, последовательность и вложенность которых определяет структуру документа и его содержание. Основным достоинством XML документов является то, что при относительно простом способе создания и обработки (обычный текст может редактироваться любым тестовым процессором и обрабатываться стандартными XML анализаторами), они позволяют создавать структурированную информацию, которую хорошо «понимают» компьютеры.

Пример формата исходного файла можно увидеть на рисунке 2.2.

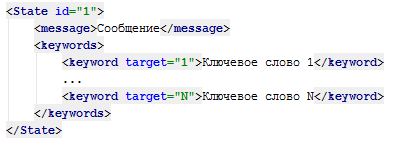


Рисунок 2.2 – Структура хранимых данных

Клиентская часть веб-приложения – это отображаемый в браузере графический интерфейс. Это то, что пользователь видит на странице. Через данный интерфейс пользователь взаимодействует с приложением, вводя информацию, нажимая на ссылки и кнопки.

Основной язык, которым описывается графический интерфейс веб-приложения – это HTML. Данный язык описывает структуру веб-страницы, размещение на ней компонентов. Оформление веб-страниц, их стиль и цветовая схема описываются в таблицах стилей – CSS. Для динамичности графического интерфейса, используются скрипты JavaScript. Программа, написанная на JavaScript и выполняющаяся на веб-странице, может управлять встроенными в страницу компонентами, тем самым реализуя пользовательский интерфейс с богатыми возможностями.

Отсутствие необходимости полностью перезагружать страницу после каждого получения данных от сервера может существенно ускорить работу веб-приложения. Такая концепция имеет название Asynchronous JavaScript and XML (асинхронный JavaScript и XML, Ajax). При использовании данного подхода динамические запросы к серверу происходят без видимой перезагрузки веб-страницы: пользователь не замечает, когда его браузер запрашивает данные.

# 3 ОПИСАНИЕ РАЗРАБОТАННОГО ПРИЛОЖЕНИЯ

## **3.1 Особенности программного комплекса**

Программный продукт представляет собой веб-приложение, разработанное на языке *Java* в среде разработки *IntelliJ IDEA*. Данная среда разработки позволяет быстро, эффективно и просто создавать полноценное и многофункциональное приложение.

В качестве архитектурного фреймворка был выбран *Spring*. Этот фреймворк, используется для обеспечения лучшей масштабируемости приложения, возможности более простого тестирования и более простой интеграции с другими фреймворками. Благодаря этому писать большие приложения становится проще - разработчики просто избегают ряда проблем, связанных с созданием больших приложений, вместо того, чтобы их решать.

*Spring* – достаточно крупный фреймворк. Потому что создатели этого его ухитрились охватить практически все аспекты программирования промышленных *Java*-приложений. Соответственно, и составных частей у *Spring Framework* немало. На иллюстрации 3.1 отображена схема фреймворка.

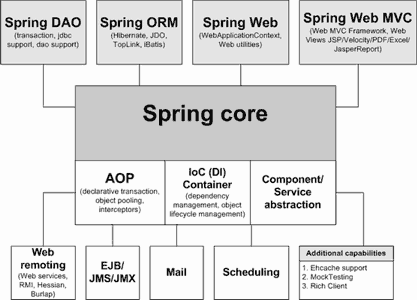


Рисунок 3.1 – Структура *Spring Framework*

*Inversion of Control* – на русский язык это название часто переводят как «инверсия управления». Довольно часто, говоря о *Spring Framework*, используют не этот термин, а «*Dependency Injection*».

Суть инверсии управления можно рассмотреть на небольшом примере. Например, у нас есть классы A и B. Причём класс A зависит от класса B (то есть, вызывает какие-то его методы). Для того, чтобы разорвать эту зависимость, вводят третье звено в этой цепочке - интерфейс C. Он должен содержать те методы B, которые нужны классу A, и быть реализованным с помощью класса B. Это и называется инверсией управления. А *IoC*-контейнер – означает что все объекты хранятся в одном контейнере. Когда нам нужен экземпляр какого-то класса, то запрашивается контейнер, который извлекает или создаёт необходимый нам экземпляр и все его зависимости. Собственно говоря, ослабление связей между компонентами приложения и есть основная задача *Spring Framework*. Именно это обеспечивает хорошую масштабируемость приложений, сравнительную простоту их тестирования и другие особенности приложений на основе *Spring*.

*Spring MVC* основан на запросах. Разработчики работают со специальными «стратегическими» интерфейсами, однако у них есть доступ и к *Servlet API*, если вдруг они захотят спуститься на более низкий уровень абстракции. Далее перечисляются наиболее важные из этих интерфейсов. *HandlerMapping* ответственен за выбор обработчиков внешних запросов к приложению, *HandlerAdapter* – за вызов и исполнение этих обработчиков, *Controller* – за управление запросами и перенаправление на нужные ответы, *View* - за отсылку нужных ответов клиенту, *ViewResolver* – за выбор нужного в соответствии с ситуацией *View*, *HandlerInterceptor* – за перехват и фильтрацию приходящих запросов, *LocaleResolver* – за определение и сохранение локали пользователя, *MultipartResolver* – за содействие загрузке файлов.

Сам шаблон *MVC*, подразумевает взаимодействие трех компонентов: контроллера (*controller*), модели (*model*) и представления (*view*).

Контроллер (*controller*) представляет класс, с которого собственно и начинается работа приложения. Этот класс обеспечивает связь между моделью и представлением. Получая вводимые пользователем данные, контроллер исходя из внутренней логики при необходимости обращается к модели и генерирует соответствующее представление.

Представление (*view*) – это собственно визуальная часть или пользовательский интерфейс приложения - например, html-страница, через которую пользователь, зашедший на сайт, взаимодействует с веб-приложением.

Модель (*model*) представляет набор классов, описывающих логику используемых данных. [7].

Общую схему взаимодействия упрощенно представлена на рисунке 3.2:

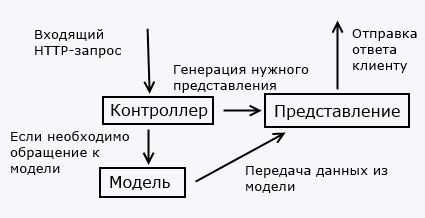


Рисунок 3.2 – Структура паттерна *MVC*

## **3.2 Описание основной структуры приложения**

Приложение можно условно разделить на серверную и клиентскую части. Серверная часть представляет собой набор *java* классов. Структура этихклассов представлена на рисунке 3.3.

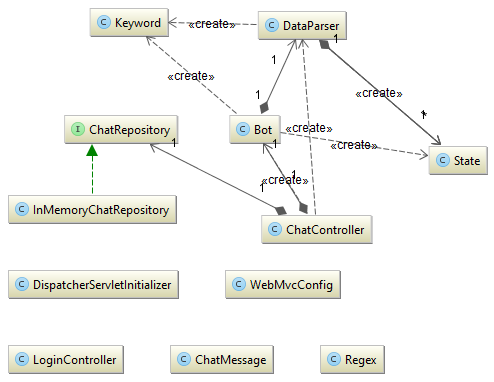


Рисунок 3.3 – *Java* классы приложения

Отображенные на рисунке выше классы, отвечают за работу приложения на сервере. Подробная структура основных классов изображена на рисунках 3.4 – 3.8. *Java* код классов находится в приложении А.

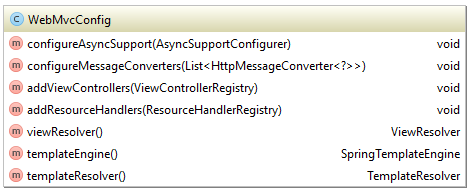


Рисунок 3.4 – Методы класса *WebMvcConfig*

Класс *WebMvcConfig* является единой точкой всех конфигураций.Отвечает за конфигурирование маппингов приложения и определение *spring* конфигов.

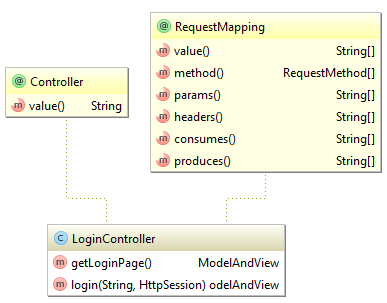


Рисунок 3.5 – Зависимости и методы класса *LoginController*

*LoginController* – это класс, который первым начинает работу с пользователем на стороне сервера. Отвечает за отображение логин страницы и аутентификацию, отправившего запрос, пользователя.

Аутентификация – проверка подлинности предъявленного пользователем идентификатора. Аутентификация требуется при доступе к таким интернет-сервисам как: электронная почта, форум, социальные сети и т.п. Положительным результатом аутентификации (кроме установления доверительных отношений и выработки сессионного ключа) является авторизация пользователя, то есть предоставление ему прав доступа к ресурсам, определенным для выполнения его задач.

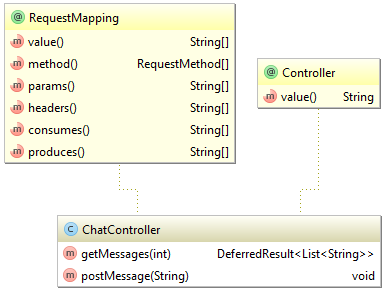


Рисунок 3.6 – Зависимости и методы класса *ChatController*

За основную работу с пользователем отвечает класс *ChatController.* Он представляет собой спринговый сервлет. И предоставляет методы получения запросов от пользователя и отправки ему ответов.

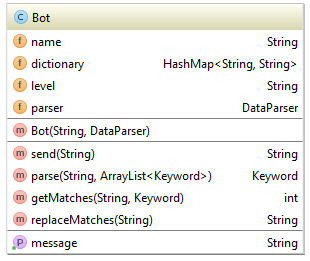


Рисунок 3.7 – Методы класса *Bot*

Класс Bot отвечает за внутреннюю работу с сообщениями. Содержит методы для определения ответа на текущее сообщение пользователя.

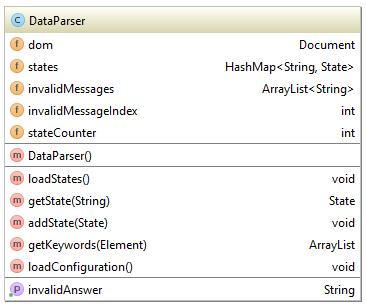


Рисунок 3.8 – Методы класса *Parser*

Класс *Parser* используется для работы с исходными файлами данных приложения. Содержит методы парсинга используемого формата файлов.

Клиентская часть приложения представлена файлами форматов: *jsp*, *html*, *css*, *js*. Содержимое основных файлов представлено в приложении А.

Технология проектирования *Java Server Pages (JSP)* – это одна из технологий *J2EE*, которая представляет собой расширение технологии сервлетов для упрощения работы с *Web*-содержимым. Страницы *JSP* позволяет легко разделить *Web*-содержимое на статическую и динамическую часть, допускающую многократное использование ранее определенных компонентов. Разработчики *Java Server Pages* могут использовать компоненты *JavaBeans* и создавать собственные библиотеки нестандартных тегов, которые инкапсулируют сложные динамические функциональные средства.

Спецификация *Java Server Pages* наследует и расширяет спецификацию сервлетов. Как и сервлеты, компоненты *JSP* относятся к компонентам *Web* и располагаются в *Web*-контейнере. Страницы *JSP* не зависят от конкретной реализации *Web*-контейнера, что обеспечивает возможность их повторного использования.

Файл *login.jsp* содержит форму для логирования пользователя в системе и предоставляет ему возможность присоединиться к чату.

Файл *chat.html* предназначен для отображения пользователю основной страницы веб-приложения. Содержит форму отображения сообщений, поле ввода нового сообщения пользователя и кнопку отправки.

Файл *chat.js* содержит *javascript* код выполняющий всю работу по взаимодействию пользователя и серверной части приложения.

**3.2 Описание интерфейса пользователя**

Для начала работы с приложением необходимо ввести *url*-адрес сайта в браузере. После чего откроется страница с предложением ввести свое имя и присоединиться к чату. Пример можно увидеть на рисунке 3.9.



Рисунок 3.9 – Стартовая страница приложения

После нажатия кнопки «*Join Chat*»откроется страница с полем сообщений и формой ввода нового сообщения.

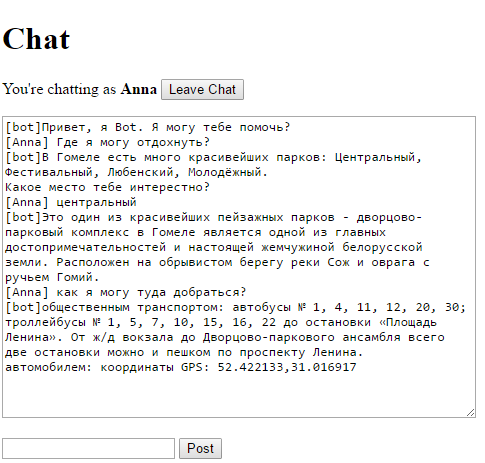


Рисунок 3.10 – Окно диалога

Для отправки сообщения необходимо ввести текст в пустое поле и нажать кнопку «*Post*». После чего отправленное сообщение и ответ на него появятся в поле чата.

По нажатию кнопки «*Leave Chat*» снова откроется стартовая страница. Если войти в чат под предыдущим именем, то история диалога будет сохранена и отображена на странице чата.

Для окончания работы с приложение достаточно закрыть окно браузера.

# ЗАКЛЮЧЕНИЕ

В рамках выполнения курсовой работы была спроектирована веб-система обмена сообщениями с чат-ботом.

Разработанное приложение работает как сетевой ресурс, что позволяет использовать его на любых гаджетах с доступом в интернет. Ведет диалог с пользователем на языке близком к естественному. Имеет простой и понятный интерфейс для отправки и получения текстовых сообщений, позволяющий пользователю быстро освоиться в системе и начать работу в нем.

Разработка программ для общения с поьзователем на языке близком к естественному является шагом на пути к разработке искусственного ителекта.

Судя по тенденци в развитии вычислительных машин и всеобщей миниатюризации, работа с компьютерами потребует новых удобных интерфейсов ввода данных. Одной из главных частей такой системы может служит написанное приложение по распознаванию писменной человеческой речи. Удобная арихитектура и возможность расширения позвоют использовать это приложение совместно с программи распознования и синтеза речи.

Выполнение данной курсовой работы позволило изучить разные варианты построения логики ботов для общения и выбрать наиболее перспективный вариант.

**Список использованных источников**

1. Гаврилова, Т.А. Базы знаний интеллектуальных систем / Т.А. Гаврилова, В.Ф. Хорошевский. – СПб.: Питер, 2000. - 209 с.
2. Кулик, С.Д. Элементы теории принятия решений (критерии и задачи) / С.Д. Кулик. - М., НИЯУ МИФИ, 2010. - 188 с.
3. Блинов, И.Н. *Java* промышленное программирование: практ. Пособие / И. Н. Блинов, В.С. Романчик – Минск: УниверсалПресс, 2007. – 704с.
4. Брюс Э. Философия *Java*. Библиотека программиста / Э. Брюс – СПб.: Питер, 2009. – 640с.
5. Хабибулин, И.Ш. Самоучитель *Java* / И.Ш. Хабибулин – Спб.: БХВ-Петербург, 2001. – 464с
6. Обзор средств создания интерфейсов веб-приложений на языке *Java* [Электронный ресурс]. – 2016. – Режим доступа: http://masters.donntu.org/2013/fknt/riabinin/library/article1.htm. – Дата доступа: 25.11.2016.
7. Паттерны проектирования. Паттерн *MVС* [Электронный ресурс]. – Режим доступа: http://metanit.com/sharp/mvc5/1.1.php. – Дата доступа: 16.10.2016.

**ПРИЛОЖЕНИЕ А**

(Обязательное)

**Листинг классов**

**package** by.gstu.chatbot.config;  
  
**import** org.springframework.context.annotation.Bean;  
**import** org.springframework.context.annotation.ComponentScan;  
**import** org.springframework.context.annotation.Configuration;  
**import** org.springframework.http.converter.HttpMessageConverter;  
**import** org.springframework.http.converter.json.MappingJackson2HttpMessageConverter;  
**import** org.springframework.web.servlet.ViewResolver;  
**import** org.springframework.web.servlet.config.annotation.AsyncSupportConfigurer;  
**import** org.springframework.web.servlet.config.annotation.ResourceHandlerRegistry;  
**import** org.springframework.web.servlet.config.annotation.ViewControllerRegistry;  
**import** org.springframework.web.servlet.config.annotation.WebMvcConfigurationSupport;  
**import** org.thymeleaf.spring3.SpringTemplateEngine;  
**import** org.thymeleaf.spring3.view.ThymeleafViewResolver;  
**import** org.thymeleaf.templateresolver.ServletContextTemplateResolver;  
**import** org.thymeleaf.templateresolver.TemplateResolver;  
  
**import** java.util.List;  
  
@Configuration  
@ComponentScan(basePackages = {**"by.gstu.chatbot"**})  
**public class** WebMvcConfig **extends** WebMvcConfigurationSupport {  
  
 @Override  
 **public void** configureAsyncSupport(AsyncSupportConfigurer configurer) {  
 configurer.setDefaultTimeout(30 \* 1000L);  
 }  
  
 @Override  
 **protected void** configureMessageConverters(List<HttpMessageConverter<?>> converters) {  
 converters.add(**new** MappingJackson2HttpMessageConverter());  
 }  
  
 **public void** addViewControllers(ViewControllerRegistry registry) {  
 registry.addViewController(**"/"**).setViewName(**"chat"**);  
 }  
  
 @Override  
 **public void** addResourceHandlers(ResourceHandlerRegistry registry) {  
 registry.addResourceHandler(**"/resources/\*\*"**).addResourceLocations(**"resources/"**);  
 }  
  
 @Bean  
 **public** ViewResolver viewResolver() {  
 ThymeleafViewResolver resolver = **new** ThymeleafViewResolver();  
 resolver.setTemplateEngine(templateEngine());  
 **return** resolver;  
 }  
  
 @Bean  
 **public** SpringTemplateEngine templateEngine() {  
 SpringTemplateEngine engine = **new** SpringTemplateEngine();  
 engine.setTemplateResolver(templateResolver());  
 **return** engine;  
 }  
@Bean  
 **public** TemplateResolver templateResolver() {  
 ServletContextTemplateResolver resolver = **new** ServletContextTemplateResolver();  
 resolver.setPrefix(**"/WEB-INF/templates/"**);  
 resolver.setSuffix(**".html"**);  
 resolver.setTemplateMode(**"HTML5"**);  
 resolver.setCacheable(**false**);  
 **return** resolver;  
 }  
}

**package** by.gstu.chatbot.controllers;  
  
**import** by.gstu.chatbot.core.Bot;  
**import** by.gstu.chatbot.core.ChatRepository;  
**import** by.gstu.chatbot.core.DataParser;  
**import** org.springframework.beans.factory.annotation.Autowired;  
**import** org.springframework.stereotype.Controller;  
**import** org.springframework.web.bind.annotation.RequestMapping;  
**import** org.springframework.web.bind.annotation.RequestMethod;  
**import** org.springframework.web.bind.annotation.RequestParam;  
**import** org.springframework.web.bind.annotation.ResponseBody;  
**import** org.springframework.web.context.request.async.DeferredResult;  
  
**import** java.util.Collections;  
**import** java.util.List;  
**import** java.util.Map;  
**import** java.util.Map.Entry;  
**import** java.util.concurrent.ConcurrentHashMap;  
  
@Controller  
@RequestMapping(**"/chat"**)  
**public class** ChatController {  
  
 **private final** ChatRepository chatRepository;  
 **private static** Bot bot;  
 **private final** Map<DeferredResult<List<String>>, Integer> chatRequests =  
 **new** ConcurrentHashMap<DeferredResult<List<String>>, Integer>();  
  
  
 @Autowired  
 **public** ChatController(ChatRepository chatRepository) {  
 **this**.chatRepository = chatRepository;  
 DataParser dp = **new** DataParser();  
 bot = **new** Bot(**"0"**, dp);  
 **this**.chatRepository.addMessage(**"[bot]"** + bot.getMessage());  
 }  
  
 @RequestMapping(method = RequestMethod.GET)  
 @ResponseBody  
 **public** DeferredResult<List<String>> getMessages(@RequestParam **int** messageIndex) {  
  
 **final** DeferredResult<List<String>> deferredResult = **new** DeferredResult<>(**null**, Collections.emptyList());  
 **this**.**chatRequests**.put(deferredResult, messageIndex);  
  
 deferredResult.onCompletion(**new** Runnable() {  
 @Override  
 **public void** run() {  
 **chatRequests**.remove(deferredResult);  
 }  
 });  
  
 List<String> messages = **this**.**chatRepository**.getMessages(messageIndex);  
 **if** (!messages.isEmpty()) {  
 deferredResult.setResult(messages);  
 }  
  
 **return** deferredResult;  
 }  
  
 @RequestMapping(method = RequestMethod.POST)  
 @ResponseBody  
 **public void** postMessage(@RequestParam String message) {  
  
 **this**.chatRepository.addMessage(message);  
 String userMessage = message.substring(message.indexOf(**']'**) + 1);  
 String response = bot.send(userMessage);  
 **if** (response.length() != 0) {  
 **this**.chatRepository.addMessage(**"[bot]"** + response);  
 } **else** {  
 **this**.chatRepository.addMessage(**"[bot]"** + bot.getMessage());  
 }  
  
 **for** (Entry<DeferredResult<List<String>>, Integer> entry : **this**.chatRequests.entrySet()) {  
 List<String> messages = **this**.chatRepository.getMessages(entry.getValue());  
 entry.getKey().setResult(messages);  
 }  
 }  
  
} **package** by.gstu.chatbot.controllers;  
  
**import** org.springframework.stereotype.Controller;  
**import** org.springframework.web.bind.annotation.RequestMapping;  
**import** org.springframework.web.bind.annotation.RequestMethod;  
**import** org.springframework.web.bind.annotation.RequestParam;  
**import** org.springframework.web.servlet.ModelAndView;  
  
**import** javax.servlet.http.HttpSession;  
  
@Controller  
@RequestMapping(**"/login"**)  
**public class** LoginController {  
  
 @RequestMapping(method = RequestMethod.GET)  
 **public** ModelAndView getLoginPage() {  
 **return new** ModelAndView(**"login"**);  
 }  
  
 @RequestMapping(method = RequestMethod.POST)  
 **public** ModelAndView login(@RequestParam String user, HttpSession session) {  
 **if** (!session.isNew()) {  
 session.invalidate();  
 }  
 session.setAttribute(**"user"**, user);  
 **return new** ModelAndView(**"login"**);  
 }  
}

**package** by.gstu.chatbot.core;  
  
**import** java.util.ArrayList;  
**import** java.util.HashMap;  
**import** java.util.Map;  
  
**public class** Bot {  
 String **name** = **"Bot"**;  
  
 *// Store all regular expression matches* **private** HashMap<String, String> **dictionary**;  
  
 *// Default state to start the bot* String **level** = **"0"**;  
 DataParser **parser**;  
  
 *// default constructor* **public** Bot(String level, DataParser parser) {  
 **dictionary** = **new** HashMap<String, String>();  
 **this**.**level** = level;  
 **this**.**parser** = parser;  
 }  
  
 *// get current state message* **public** String getMessage() {  
 State state = **parser**.getState(**level**);  
 **return** replaceMatches(state.getMessage()).trim();  
 }  
  
 *// send user message to the bot and get the response* **public** String send(String message) {  
  
 String response = **""**;  
 State state = **parser**.getState(**level**);  
  
  
 *// end of the tree* **if** (state.getKeywords().isEmpty()) {  
 **this**.**level** = **"1"**;  
 }  
  
 *// match the keyword with given message* Keyword match = parse(message, state.getKeywords());  
  
 *// if no keyword is matched, display one of the invalid answers* **if** (match == **null**) {  
 response = **parser**.getInvalidAnswer();  
 } **else** {  
  
 *// if match classname is provided, check to get the dynamic response* **if** (match.**className**.length() > 0) {  
} **else** {  
  
 *// get the new state and return the new message* **if** (response.length() == 0) {  
  
 **this**.**level** = match.**target**;  
 state = **parser**.getState(**level**);  
  
 *// if it is end of the tree* **if** (state.getKeywords().isEmpty()) {  
 response = **this**.getMessage();  
 **this**.**level** = **"1"**;  
  
 }  
 }  
 }  
 }  
 **return** response;  
 }  
  
 *// parse the given text to find best match in the keywords* **private** Keyword parse(String text, ArrayList<Keyword> keylist) {  
  
 *// set the default match to none* **int** bestMatch = -1;  
 Keyword match = **null**;  
  
 *// loop through keywords* **for** (**int** i = 0; i < keylist.size(); i++) {  
  
 *// get number of matches of the keyword with given text* **int** matches = getMatches(text, keylist.get(i));  
  
 *// if match is better than best match, replace it* **if** (matches > -1 && matches > bestMatch) {  
 match = keylist.get(i);  
 bestMatch = matches;  
 }  
 }  
  
 *// add best answers regex variable value into the dictionary for future reference* **if** (match != **null**) {  
 **if** (match.**learn**.length() > 0) {  
  
 *// get training data keyword and description* String subject = **dictionary**.get(match.**learn**);  
 String result = match.**variableValue**;  
  
  
 *// create a new state for new trained data* ArrayList<String> messages = **new** ArrayList<String>();  
 messages.add(result);  
 State myState = **new** State(String.*valueOf*(**parser**.**stateCounter**), messages, **new** ArrayList());  
 **parser**.addState(myState);  
  
 *// add the new trained keyword* Keyword keyword = **new** Keyword(subject, myState.getId(), **""**, **""**, **""**, 1, **""**);  
 State state = **parser**.getState(**"1"**);  
 ArrayList<Keyword> keywords = state.getKeywords();  
 keywords.add(keyword);  
  
 } **else** {  
 **if** (match.**variableValue**.length() > 0) {  
 **dictionary**.put(match.**variable**, match.**variableValue**);  
 }  
 }  
 }  
 **return** match;  
 }  
  
 *// get number of matches of the given keywords in the given list* **private int** getMatches(String text, Keyword keyword) {  
  
 *// no match by default* **int** result = -1;  
  
 *// return 0 match when keyword is \** **if** (keyword.**keyword**.equals(**"\*"**)) {  
 **return** keyword.**points**;  
 }  
  
 *// if regex is expected* **if** (keyword.**variable**.length() > 0) {  
 String match = Regex.*match*(keyword.**keyword**, text);  
 **if** (match.length() > 0) {  
 keyword.**variableValue** = match;  
 **return** keyword.**points**;  
 }  
 }  
  
 String[] words = keyword.**keyword**.split(**" "**);  
  
  
 *// loop through list of the keywords* **for** (String word : words) {  
  
 *// if current keyword is in the text, add points* **if** (text.toLowerCase().indexOf(word.toLowerCase()) >= 0) {  
 result = result + keyword.**points** + 1;  
 } **else** {  
 *// return null if one of the keywords does not exists* **return** -1;  
 }  
 }  
 **return** result;  
 }  
  
 *// replace given text with variables in the dictionary* **public** String replaceMatches(String text) {  
  
 *// replace variables within dictionary in the text* **for** (Map.Entry<String, String> entry : **dictionary**.entrySet()) {  
 text = text.replaceAll(**"\\["** + entry.getKey() + **"\\]"**, entry.getValue());  
 }  
  
 *// remove empty variables tags* **return** Regex.*clear*(text);  
 }  
}

**package** by.gstu.chatbot.core;  
  
**import** org.w3c.dom.Document;  
**import** org.w3c.dom.Element;  
**import** org.w3c.dom.NodeList;  
**import** org.xml.sax.SAXException;  
  
**import** javax.xml.parsers.DocumentBuilder;  
**import** javax.xml.parsers.DocumentBuilderFactory;  
**import** javax.xml.parsers.ParserConfigurationException;  
**import** java.io.IOException;  
**import** java.util.ArrayList;  
**import** java.util.HashMap;  
  
**public class** DataParser {  
  
 **private** Document **dom**;  
 **private** HashMap<String, State> **states** = **new** HashMap<String, State>();  
 **private** ArrayList<String> **invalidMessages** = **new** ArrayList();  
 **private int invalidMessageIndex** = 0;  
 **public int stateCounter** = 1000;  
  
 *// default constructor* **public** DataParser() {  
  
 *// Load the XML file and parse it* DocumentBuilderFactory dbf = DocumentBuilderFactory.*newInstance*();  
  
 **try** {  
  
 *//Using factory get an instance of document builder* DocumentBuilder db = dbf.newDocumentBuilder();  
  
 *//parse using builder to get DOM representation of the XML file* **dom** = db.parse(**"E:\\data.xml"**);  
  
 *// Load configuration and states from the XML file* loadConfiguration();  
 loadStates();  
 } **catch** (ParserConfigurationException pce) {  
 pce.printStackTrace();  
 } **catch** (SAXException se) {  
 se.printStackTrace();  
 } **catch** (IOException ioe) {  
 ioe.printStackTrace();  
 }  
 }  
  
 *// Load states from XML file* **private void** loadStates() {  
  
 *// get document element object* Element docEle = **dom**.getDocumentElement();  
  
 *// get all State node names* NodeList nl = docEle.getElementsByTagName(**"State"**);  
  
 *// if node is not null and has children* **if** (nl != **null** && nl.getLength() > 0) {  
  
 *// loop through all children* **for** (**int** i = 0; i < nl.getLength(); i++) {  
  
 *// get state element* Element el = (Element) nl.item(i);  
  
 *// get state id* String id = el.getAttribute(**"id"**);  
  
 *// get all state messages* ArrayList messages = **new** ArrayList();  
 NodeList messagesNodeList = el.getElementsByTagName(**"message"**);  
  
 *// if messages node is not null and has children* **if** (messagesNodeList != **null** && messagesNodeList.getLength() > 0) {  
  
 *// loop through all children* **for** (**int** j = 0; j < messagesNodeList.getLength(); j++) {  
  
 *// get current message element* Element elmsg = (Element) messagesNodeList.item(j);  
  
 *// append message node value to the messages list* messages.add(elmsg.getFirstChild().getNodeValue());  
 }  
 }  
  
 *// get keywords in the current state* ArrayList keywords = getKeywords(el);  
  
 *// construct a new State object* State state = **new** State(id, messages, keywords);  
  
 **stateCounter**++;  
  
 *// add the state to the states hashmap* **states**.put(id, state);  
 }  
 }  
 }  
  
 *// get state object by id* **public** State getState(String id) {  
 **return states**.get(id);  
 }  
  
 *// create a new state* **public void** addState(State state) {  
 **states**.put(state.getId(), state);  
 **stateCounter**++;  
 }  
  
 *// get all keywords in an State tag* **public** ArrayList getKeywords(Element ele) {  
  
 *// construct keywords arraylist* ArrayList keywords = **new** ArrayList();  
  
 *// get all nodes by keyword tag name* NodeList nl = ele.getElementsByTagName(**"keyword"**);  
  
 *// if the tag is not null and has children* **if** (nl != **null** && nl.getLength() > 0) {  
  
 *// loop through all the children* **for** (**int** i = 0; i < nl.getLength(); i++) {  
  
 *//get the keyword element* Element el = (Element) nl.item(i);  
  
 *// find the keyword target, classname and argument attributes* String wordTag = el.getFirstChild().getNodeValue();  
 String target = el.getAttribute(**"target"**);  
 String className = el.getAttribute(**"className"**);  
 String arg = el.getAttribute(**"arg"**);  
 String variable = el.getAttribute(**"variable"**);  
 **int** points = 0;  
 **try** {  
 points = Integer.*valueOf*(el.getAttribute(**"points"**));  
 } **catch** (Exception e) {  
  
 }  
  
 String learn = el.getAttribute(**"learn"**);  
 *// split keyword by comma* String[] words = wordTag.split(**","**);  
  
 *// loop through all words* **for** (String word : words) {  
  
 *// trim the word to remove spaces* word = word.trim();  
  
 *// construct a new keyword* Keyword keyword = **new** Keyword(word, target, className, arg, variable, points, learn);  
  
 *// add the keyword to keywords array list* keywords.add(keyword);  
 }  
 }  
 }  
  
 *// return all the keywords in the given node* **return** keywords;  
 }  
  
  
 *// returns one of the invalid messages and move the index to the next message* **public** String getInvalidAnswer() {  
  
 *// get current answer* String answer = **invalidMessages**.get(**invalidMessageIndex**);  
  
 *// increase the index, if it is end of messages, reset the index to 0* **invalidMessageIndex**++;  
 **if** (**invalidMessageIndex** >= **invalidMessages**.size()) {  
 **invalidMessageIndex** = 0;  
 }  
 **return** answer;  
 }  
  
 *// load cofig tags from data xml file* **private void** loadConfiguration() {  
  
 *// get document element* Element docEle = **dom**.getDocumentElement();  
  
 *// get all node names for invalid messages* NodeList node = docEle.getElementsByTagName(**"InvalidMessages"**);  
  
 *// get all message nodes inside invalid messages node* NodeList nl = ((Element) node.item(0)).getElementsByTagName(**"message"**);  
  
 *// if node is not null and has children* **if** (nl != **null** && nl.getLength() > 0) {  
  
 *// loop through all children* **for** (**int** i = 0; i < nl.getLength(); i++) {  
  
 *// get message node* Element el = (Element) nl.item(i);  
  
 *// get message and add it to invalid messages array* String message = el.getFirstChild().getNodeValue();  
 **invalidMessages**.add(message);  
 }  
 }  
 }  
}

**package** by.gstu.chatbot.core;  
  
**import** java.util.regex.Matcher;  
**import** java.util.regex.Pattern;  
  
**public class** Regex {  
  
 **public static** String match(String pattern, String keyword) {  
 Pattern p = Pattern.*compile*(pattern.toLowerCase());  
 Matcher m = p.matcher(keyword.toLowerCase());  
 **if** (m.matches()) {  
 **return** m.group(1);  
 }  
 **return ""**;  
 }  
  
 **public static** String clear(String text) {  
 Pattern pattern = Pattern.*compile*(**"\\[.\*\\]"**);  
  
 *// Replace all occurrences of pattern in input* Matcher matcher = pattern.matcher(text);  
 **return** matcher.replaceAll(**""**);  
 }  
}

**package** by.gstu.chatbot.core;  
  
**import** java.util.ArrayList;  
**import** java.util.Random;  
  
**public class** State {  
  
 **private** String **id** = **""**;  
 **private** ArrayList<String> messages;  
 **private** ArrayList keywords;  
 **public** String argument = **""**;  
  
 *// default constructor, constructs State object* **public** State(String id, ArrayList messages, ArrayList keywords) {  
 **this**.id = id;  
 **this**.messages = messages;  
 **this**.keywords = keywords;  
 }  
  
 *// get state id* **public** String getId() {  
 **return** id;  
 }  
  
 *// get random state messages* **public** String getMessage() {  
 Random generator = **new** Random();  
 **return** messages.get(generator.nextInt(messages.size()));  
 }  
  
 *// set the argument from regex matcher* **public void** setRegex(String argument) {  
 **this**.argument = argument;  
 }  
  
 *// get state keywords* **public** ArrayList getKeywords() {  
 **return** keywords;  
 }  
}

login.jsp

<!DOCTYPE **html**>  
<**html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org"**>  
 <**head**>  
 <**title**>Chat</**title**>  
 <**meta http-equiv="Content-Type" content="text/html; charset=UTF-8"**/>  
 </**head**>  
 <**body**>  
 <**form name="joinChatForm" th:action="@{/login}" method="post"**>  
 <**fieldset**>  
 <**legend**>Chat</**legend**>  
 <**label for="user"**>User: </**label**>  
 <**input type="text" id="user" name="user"**/>  
 <**div class="form-actions"**>  
 <**button type="submit" class="btn"**>Join Chat</**button**>  
 </**div**>  
 </**fieldset**>  
 </**form**>  
 </**body**>  
</**html**>

Chat.html

<!DOCTYPE **html**>  
<**html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org"**>  
 <**head**>  
 <**title**>Chat</**title**>  
 <**meta http-equiv="Content-Type" content="text/html; charset=UTF-8"**/>  
 </**head**>  
 <**body**>  
 <**h1**>Chat</**h1**>  
  
 <**form id="joinChatForm" th:action="@{/chat}" data-bind="visible: activePollingXhr() == null"**>  
 <**p**>  
 <**label for="user"**>User: </**label**>  
 <**input id="user" name="user" type="text" data-bind="value: userName"**/>  
 <**input name="messageIndex" type="hidden" data-bind="value: messageIndex"**/>  
 <**button id="start" type="submit" data-bind="click: joinChat"**>Join Chat</**button**>  
 </**p**>  
 </**form**>  
  
 <**form id="leaveChatForm" th:action="@{/chat}" data-bind="visible: activePollingXhr() != null"**>  
 <**p**>  
 You're chatting as <**strong data-bind="text: userName"**></**strong**>  
 <**button id="leave" type="submit" data-bind="click: leaveChat"**>Leave Chat</**button**>  
 </**p**>  
 </**form**>  
  
 <**div data-bind="visible: activePollingXhr() != null"**>  
 <**textarea rows="15" cols="60" readonly="readonly" data-bind="text: chatContent"**></**textarea**>  
 </**div**>  
  
 <**form id="postMessageForm" th:action="@{/chat}" data-bind="visible: activePollingXhr() != null"**>  
 <**p**>  
 <**input id="message" name="message" type="text" data-bind="value: message"**/>  
 <**button id="post" type="submit" data-bind="click: postMessage"**>Post</**button**>  
 </**p**>  
 </**form**>  
 </**body**>  
 <**script type="text/javascript" src="../../../resources/js/jquery-1.7.2.min.js"  
 th:src="@{/resources/js/jquery-1.7.2.min.js}"**></**script**>  
 <**script type="text/javascript" src="../../../resources/js/knockout-2.0.0.js"  
 th:src="@{/resources/js/knockout-2.0.0.js}"**></**script**>  
 <**script type="text/javascript" src="../../../resources/js/chat.js" th:src="@{/resources/js/chat.js}"**></**script**>  
  
</**html**>

Chat.js

**$**(**document**).**ready**(**function** () {  
  
 **function** *ChatViewModel*() {  
  
 **var** that = **this**;  
  
 that.**userName** = **ko**.observable(**''**);  
 that.**chatContent** = **ko**.observable(**''**);  
 that.**message** = **ko**.observable(**''**);  
 that.**messageIndex** = **ko**.observable(0);  
 that.**activePollingXhr** = **ko**.observable(**null**);  
  
 **var** keepPolling = **false**;  
  
 that.joinChat = **function** () {  
 **if** (that.**userName**().trim() != **''**) {  
 keepPolling = **true**;  
 *pollForMessages*();  
 }  
 }  
  
 **function** *pollForMessages*() {  
 **if** (!keepPolling) {  
 **return**;  
 }  
 **var** form = **$**(**"#joinChatForm"**);  
 that.**activePollingXhr**(**$**.**ajax**({  
 **url**: form.attr(**"action"**), **type**: **"GET"**, **data**: form.serialize(), **cache**: **false**,  
 success: **function** (messages) {  
 **for** (**var** i = 0; i < messages.**length**; i++) {  
 that.**chatContent**(that.**chatContent**() + messages[i] + **"\n"**);  
 that.**messageIndex**(that.**messageIndex**() + 1);  
 }  
 },  
 error: **function** (xhr) {  
 **if** (xhr.**statusText** != **"abort"** && xhr.**status** != 503) {  
 *resetUI*();  
 **console**.error(**"Unable to retrieve chat messages. Chat ended."**);  
 }  
 },  
 complete: *pollForMessages* }));  
 **$**(**'#message'**).focus();  
 }  
  
 that.postMessage = **function** () {  
 **if** (that.**message**().trim() != **''**) {  
 **var** form = **$**(**"#postMessageForm"**);  
 **$**.**ajax**({  
 **url**: form.attr(**"action"**), **type**: **"POST"**,  
 **data**: **"message=["** + that.**userName**() + **"] "** + **$**(**"#postMessageForm input[name=message]"**).**val**(),  
 error: **function** (xhr) {  
 **console**.error(**"Error posting chat message: status="** + xhr.**status** + **", statusText="** + xhr.**statusText**);  
 }  
 });  
 that.**message**(**''**);  
 }  
 }  
  
 that.leaveChat = **function** () {  
 that.**activePollingXhr**(**null**);  
 *resetUI*();  
 **this**.**userName**(**''**);  
 }  
  
 **function** *resetUI*() {  
 keepPolling = **false**;  
 that.**activePollingXhr**(**null**);  
 that.**message**(**''**);  
 that.**messageIndex**(0);  
 that.**chatContent**(**''**);  
 }  
  
 }  
  
 *//Activate knockout.js* **ko**.applyBindings(**new** *ChatViewModel*());  
  
});