**ВВЕДЕНИЕ**

Мода на боты не появилась – она вернулась, это очередной эволюционный виток. Сами по себе боты существуют давно: первые были придуманы еще в 90-ых. В то время боты были, уделом исключительно технически подкованных людей. Но сейчас, с популяризацией *WhatsApp*, *Skype*, *Viber* и других мессенджеров, эта тема обрела второе дыхание. Благодаря платформам *iOS* и *Android* у каждого владельца смартфона установлен как минимум один мессенджер по умолчанию.

Боты практичны. Компании создают мобильные приложения для своего бизнеса. Это верный способ привязать пользователя к сервису. Каждый из нас не раз сталкивался с тем, что такие мини-приложения, которые используются всего несколько раз за месяц, засоряют экран мобильного устройства.

Боты решают эту проблему. Реализовать бота быстрей и проще. Плюс к тому, сегодня практически у каждого пользователя смартфона есть аккаунт в *Facebook*, соответственно – в *Facebook* *Messenger*. Аудитория *Telegram* также перевалила за 100 млн. активных пользователей. Человек охотней подпишется на бота, чем установит еще одно приложение.

У ботов, безусловно, всегда будет своя аудитория. Причем, не маленькая. Чаты, *WhatsApp*, *Telegram*, *Twitter* – очень зрелые решения. Эти платформы со временем будут становиться только лучше, что даст почву для создания новых ботов.

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. Однако при общении человеку свойственно ошибаться, и для успешного распознавания текста компьютеру необходимо проводить проверки правописания. Для этого можно использовать онлайн сервисы.

1. **ОБЩИЕ СВЕДЕНИЯ ОБ КОМПАНИИ ИООО «ЭПАМ СИСТЕМЗ»**
   1. **О компании**

*EPAM Systems* – крупнейший поставщик услуг в области разработки проектного (заказного) программного обеспечения и решений в Центральной и Восточной Европе. Реализовав тысячи сложных и масштабных проектов по всему миру, *EPAM Systems* неизменно остается признанным лидером в таких областях, как: разработка, тестирование, сопровождение и поддержка заказного программного обеспечения и бизнес-приложения; интеграция приложений на базе продуктов *SAP*, *Oracle*, *IBM*, *Microsoft*; миграция приложений на новую интеграционную платформу; создание выделенных центров разработки (центров компетенции), центров тестирования и контроля качества программного обеспечения.

Преимуществами сотрудничества с *EPAM Systems* пользуются сотни компаний из различных секторов экономики, в том числе банки и финансовые компании, страхование, поставщики программного обеспечения, розничная торговля и потребительские товары, информационный и медиа-бизнес, индустрия путешествий, телекоммуникации, нефтегаз и энергетика, автобизнес, государственный сектор и др.

Сильная команда талантливых разработчиков, богатый опыт работы с заказчиками по всему миру, партнерство с ведущими разработчиками программного обеспечения, выстроенная методология и наличие современных инструментов проектного управления, глубокая отраслевая и технологическая экспертиза являются неотъемлемыми составляющими успеха ИТ-проектов, реализуемых *EPAM* *Systems*. Для того чтобы лучше понимать и эффективнее решать задачи клиентов, в компании формируются центры компетенции, где консолидируются лучшие мировые практики и отечественный опыт по различным отраслям и технологиям. В целях обеспечения прозрачности и эффективности управления проектами в компании применяются не имеющие аналогов на рынке инструменты и системы метрик собственной разработки. В зависимости от задач проекта и предпочтений заказчика, *EPAM* предлагает различную методологию для разработки программного обеспечения.

## 1.2 Используемое оборудование

Отдел ИТ оснащен современным оборудованием, а также всеми сопутствующими необходимыми аксессуарами. Типичное рабочее место инженера-программиста состоит из следующих частей:

1. персональный компьютер на базе процессора *Intel Core i7*;
2. монитор 24” (по желанию работника могут быть установлены дополнительные мониторы);
3. клавиатура, мышь, наушники (выбирается работником исходя из его личных предпочтений).

Кроме этого, офис оснащён проводной и беспроводной (*WiFi*) сетью, объединяющей все компьютеры компании в рамках офиса. Данные сети, также, предоставляет доступ к сети Интернет.

## 1.3 Охрана труда на предприятии и техника безопасности на рабочем месте программиста.

Охрана труда – система обеспечения безопасности жизни и здоровья работающих в процессе трудовой деятельности, включающая правовые, социально-экономические, организационные, технические, психофизиологические, санитарно-гигиенические, лечебно-профилактические, реабилитационные и иные мероприятия и средства [1].

Руководством завода уделяется большое внимание улучшению эргономики рабочих мест, обеспечению гигиены и совершенствованию организации труда, регламентации режимов труда и отдыха.

Вся деятельность в области охраны труда на ИООО «*EPAM Systems*» регламентирована действующим законодательством Республики Беларусь, санитарными нормами и правилами, гигиеническими нормативами, предписаниями надзорных органов.

К комплексу мероприятий в области охраны труда, осуществляемых на ИООО «*EPAM Systems*» относятся:

* проведение производственного лабораторного контроля за условиями труда на рабочих местах;
* разработка инструкций по охране труда и ознакомление с ними персонала;
* модернизация рабочих мест и технологического оборудования;
* создание безопасных условий труда.

Важным фактором создания безопасных условий труда является оптимизация организации рабочих мест. Рабочее место, хорошо приспособленное к трудовой деятельности инженера, правильно и целесообразно организованное, в отношении пространства, формы, размера обеспечивает ему удобное положение при работе и высокую производительность труда при наименьшем физическом и психическом напряжении.

При правильной организации рабочего места производительность труда инженера возрастает с 8 до 20 процентов.

Основным рабочим местом инженера-программиста является стол для выполнения машинописных работ. Основная поза при выполнении работы – вынужденная, сидячая.

Рабочее место для выполнения работ в положении сидя организуется в соответствии с ГОСТ 12.2.032-78 «Система стандартов безопасности труда. Рабочее место при выполнении работ сидя. Общие эргономические требования» [2].

Требования к организации работы при использовании персонального компьютера и организационных средств определяются СанПиН 9-131 РБ 2000 «Гигиенические требования к видеодисплейным терминалам, электронно-вычислительным машинам и организации работы».

Согласно ГОСТ 12.2.032-78 конструкция рабочего места и взаимное расположение всех его элементов должно соответствовать антропометрическим, физическим и психологическим требованиям. Большое значение имеет также характер работы. В частности, при организации рабочего места программиста должны быть соблюдены следующие основные условия:

* оптимальное размещение оборудования, входящего в состав рабочего места;
* достаточное рабочее пространство, позволяющее осуществлять все необходимые движения и перемещения;
* уровень акустического шума не должен превышать допустимого значения.

Рабочая поза сидя вызывает минимальное утомление программиста. Рациональная планировка рабочего места предусматривает четкий порядок и постоянство размещения предметов, средств труда и документации. То, что требуется для выполнения работ чаще, расположено в зоне легкой досягаемости рабочего пространства.

Помещения для работы программиста должны иметь естественное и искусственное освещение.

Площадь на одно рабочее место с видео-дисплейным терминалом (ВДТ) и ПЭВМ для взрослых пользователей должна составлять не менее 6,0 м2, а объем не менее 20,0 м3.

Искусственное освещение в помещениях эксплуатации ВДТ и ПЭВМ должно осуществляться системой общего равномерного освещения. В административно-общественных помещениях, в случаях преимуществен­ной работы с документами, допускается применение системы комбинированного освещения (к общему освещению дополнительно устанавливаются светильники местного освещения, предназначенные для освещения зоны расположения документов).

Освещенность на поверхности стола в зоне размещения рабочего документа должна быть 300-500 лк. Местное освещение не должно создавать бликов на поверхности экрана и увеличивать освещенность экрана более 300 лк.

В качестве источников света при искусственном освещении должны применяться преимущественно люминесцентные лампы.

Конструкция рабочего стола должна обеспечивать оптимальное размещение на рабочей поверхности используемого оборудования с учетом его количества и конструктивных особенностей (размер ВДТ и ПЭВМ, клавиатуры и др.), характера выполняемой работы. При этом допускается использование рабочих столов различных конструкций, отвечающих современным требованиям эргономики.

Конструкция рабочего стула (кресла) должна обеспечивать поддержание рациональной рабочей позы при работе на ВДТ и ПЭВМ, позволять изменять позу с целью снижения статического напряжения мышц шейно-плечевой области истины для предупреждения развития утомления.

Тип рабочего стула (кресла) должен выбираться в зависимости от характера и продолжительности работы.

Рабочий стул (кресло) должен быть подъемно-поворотным и регулируе­мым по высоте и углам наклона сиденья и спинки, а также расстоянию спинки от переднего края сиденья, при этом регулировка каждого параметра должна быть независимой, легко осуществляемой и иметь надежную фиксацию.

Экран видеомонитора должен находиться от глаз пользователя на опти­мальном расстоянии 600–700 мм, но не ближе 500 мм с учетом размеров алфавитно-цифровых знаков и символов.

Высота рабочей поверхности стола должна регулироваться в пределах 680-800 мм, при отсутствии такой возможности высота рабочей поверхности стола должна составлять 725 мм.

Рабочий стол должен иметь пространство для ног высотой не менее 600 мм, шириной – не менее 500 мм, глубиной на уровне колен – не менее 450 мм и на уровне вытянутых ног – не менее 650 мм.

Клавиатуру следует располагать на поверхности стола на расстоянии не менее чем 300 мм от края, обращенного к пользователю или на специальной, регулируемой по высоте рабочей поверхности, отделенной от основной столешницы.

Помимо требований к организации рабочего места СанПиН 9-131 РБ 2000 устанавливает требования к микроклимату рабочей зоны: влажности, температуре, скорости потока воздуха и пр.

Техника безопасности – это система организационных мероприятий и технических средств, предотвращающих воздействие на работающих опасных производственных факторов.

Для соблюдения техники безопасности в своей деятельности инженер-программист должен руководствоваться «Инструкцией по охране труда для программиста при выполнении работ с применением ПЭВМ и ВДТ».

## Должностная инструкция инженера-программиста.

Инженер-программист относится к категории специалистов, принимается на работу и увольняется приказом начальника вычислительного центра (ВЦ) по представлению начальника производственного отдела ВЦ, начальника отдела разработок и внедрения ВЦ, руководителя иного структурного подразделения предприятия. При отсутствии вычислительного центра и иного специального подразделения, инженер-программист принимается на работу и увольняется приказом директора и подчиняется непосредственно ему.

На должность:

* инженера-программиста назначается лицо, имеющее высшее профессиональное (техническое или инженерно-экономическое) образование без предъявления требований к стажу работы или среднее профессиональное (техническое или инженерно-экономическое) образование и стаж работы в должности техника 1 категории не менее 3 лет, либо других должностях, замещаемых специалистами со средним специальным образованием, не менее 5 лет.
* инженера-программиста III категории назначается лицо, имеющее высшее профессиональное (техническое или инженерно-экономическое) образование и опыт работы по специальности, приобретенный в период обучения, или стаж работы на инженерно-технических должностях без квалификационной категории.
* инженера-программиста II категории – лицо, имеющее высшее профессиональное (техническое или инженерно-экономическое) образование и стаж работы в должности инженера-программиста III категории не менее 3 лет.
* инженера-программиста I категории – лицо, имеющее высшее профессиональное (техническое или инженерно-экономическое) образование и стаж работы в должности инженера-программиста II категории не менее 3 лет.

В своей деятельности инженер-программист руководствуется:

* постановлениями, распоряжениями, приказами, другими руководящими и нормативными документами, касающимися методов программирования и использования вычислительной техники при обработке информации;
* уставом предприятия;
* приказами и распоряжениями директора предприятия;
* должностной инструкцией.

Инженер-программист должен знать:

* руководящие и нормативные материалы, регламентирующие методы разработки алгоритмов и программ, и использования вычислительной техники при обработке информации;
* основные принципы структурного программирования;
* виды программного обеспечения;
* технико-эксплуатационные характеристики, конструктивные особенности, назначение и режимы работы ЭВМ, правила ее технической эксплуатации;
* технологию автоматической обработки и кодирования информации;
* формализованные языки программирования;
* действующие стандарты, системы счислений, шифров и кодов;
* порядок оформления технической документации;
* передовой отечественный и зарубежный опыт программирования и использования вычислительной техники;
* основы экономики, организации производства, труда и управления;
* основы трудового законодательства;
* правила внутреннего трудового распорядка;
* правила и нормы охраны труда.

На время отсутствия инженера-программиста (отпуск, болезнь, командировка, пр.) его обязанности исполняет лицо, назначенное в установленном порядке, которое несет ответственность за качественное исполнение возложенных на него обязанностей.

На инженера-программиста возлагаются следующие функции:

* разработка программ, обеспечивающих на основе анализа математических моделей и алгоритмов решения экономических и других задач возможность выполнения алгоритма, и, соответственно, поставленной задачи средствами вычислительной техники; проведение их тестирования и отладки;
* разработка технологии решения задач по всем этапам обработки информации.

Для выполнения возложенных на него функций инженер-программист обязан:

* определять информацию, подлежащую обработке средствами вычислительной техники, ее объемы, структуру, макеты и схемы ввода, обработки, хранения и вывода, методы ее контроля;
* осуществлять выбор языка программирования для описания алгоритмов и структур данных;
* выполнять работу по подготовке программ к отладке и проводить отладку;
* определять объем и содержание данных контрольных примеров, обеспечивающих наиболее полную проверку соответствия программ их функциональному значению;
* осуществлять запуск отлаженных программ и ввод исходных данных, определяемых условиями поставленных задач;
* проводить корректировку разработанной программы на основе анализа выходных данных;
* разрабатывать инструкции по работе с программами, оформлять необходимую техническую документацию;
* определять возможность использования готовых программных продуктов;
* осуществлять сопровождение внедрения программ и программных средств;
* разрабатывать и внедрять системы автоматической правильности проверки программ, типовые и стандартные программные средства, составлять технологию обработки информации;
* выполнять работу по унификации и типизации вычислительных процессов;
* принимать участие в создании каталогов и картотек стандартных программ, в разработке форм документов, подлежащих машинной обработке, в проектировании программ, позволяющих расширить область применения вычислительной техники.

Инженер-программист имеет право:

* знакомиться с проектами решений руководства предприятия, касающимися его деятельности;
* вносить на рассмотрение руководства предприятия предложения по совершенствованию работы связанной с обязанностями, предусмотренными настоящей инструкцией;
* запрашивать лично или по поручению непосредственного руководителя от руководителей структурных подразделений, специалистов информацию и документы, необходимые для выполнения своих должностных обязанностей;
* в пределах своей компетенции сообщать непосредственному руководителю о всех недостатках в деятельности предприятия (его структурных подразделениях) и вносить предложения по их устранению;
* привлекать специалистов всех (отдельных) структурных подразделений предприятия для решения возложенных на него обязанностей (если это предусмотрено положениями о структурных подразделениях, если нет – с разрешения их руководителей);
* требовать от своего непосредственного руководителя, руководства предприятия оказания содействия в исполнении им своих должностных обязанностей и прав.

Инженер-программист несет ответственность:

* за неисполнение (ненадлежащее исполнение) своих должностных обязанностей, предусмотренных настоящей должностной инструкцией, в пределах, определенных действующим трудовым законодательством Республики Беларусь;
* за совершение в процессе осуществления своей деятельности правонарушения в пределах, определенных действующим административным, уголовным и гражданским законодательством Республики Беларусь;
* за причинение материального ущерба – в пределах, определенных действующим трудовым, уголовным и гражданским законодательством Республики Беларусь.

# ИНФОРМАЦИОННОЕ ОБЕСПЕЧЕНИЕ

**2.1 Постановка задачи**

В процессе прохождения преддипломной практики бы ли поставлены следующие задачи:

* изучение требований, предъявляемых к безопасной и комфортной работе инженеров-программистов;
* знакомство со спецификой этапов жизненного цикла программного продукта;
* изучение принципов работы программ для имитации речевого поведения человека при общении с одним или многими собеседниками;
* изучение языка разметки искусственного интеллекта *AIML* с целью анализа его возможностей для построения интеллектуального чат-бота;
* изучение существующих фреймворков (*API*) для построения чат-ботов на базе распространенных текстовых менеджеров (*Telegram*, *Facebook*, *Skype* и др);
* проектирование структуры программного комплекса и базы данных для хранения информации в соответствии с темой дипломной работы;
* разработка программного модуля для запроса информации из базы данных заданной предметной области на основе текстового пользовательского запроса;
* формирование информационной базы для выполнения дипломной работы и тестирования предполагаемых к созданию программных продуктов;
* создание прототипа программного продукта в соответствии с темой дипломной работы.

**2.2 Программы имитации речевого поведения человека**

Вопросы применения систем виртуального общения на основе искусственного интеллекта исследуют на протяжении многих лет. На сегодняшний день проблема виртуального общения актуальна из-за быстрого доступа к информации, возможности одновременной работы в системе многих пользователей, обмена информацией, взаимодействия с целью решения любых вопросов, поддержки обучения, коммуникации с клиентами и партнерами по бизнесу, проведения аналитических исследований, сбора необходимой информации, повышения квалификации и других преимуществ.

Основными вопросами в создании систем общения являются разработка модели общения, модели участника общения, развитие средств, в первую очередь, семантических и прагматических, описания окружающей среды (модели языка, модели пользователя, модели окружающей среды, модели системы общения). Поэтому для решения этих вопросов необходимо определение принципов работы, особенностей имитации речевого поведения человека в процессе общения, разработка модели общения, написания чат-бота.

Среди программ-собеседников есть программы, созданные на основе искусственного интеллекта. При разработке таких программ необходимо знать психологию, а также принципы построения фраз человеческой речи. Более того, если правильно определить языковые ограничения и предметную область, то существующими методами можно получить системы, пригодные для общения. С точки зрения теории языка и общения необходима разработка семантического описания структур текстов и предложений. С точки зрения модели окружающей среды основным ограничением является отсутствие средств для представления динамически меня

Для необходимо определить, что такое бот. Бот – это некоторый программный комплекс для решения каких-либо задач, обладающий признаками искусственного интеллекта. К ним относятся: способность принимать решения на основе эмпирически полученных данных, понимать семантику данных, проводить анализ и выдавать оптимальное решение. Все слышали о ботах в компьютерных играх. Это небольшие библиотеки, встроенные или настраиваемые. Они анализируют математическую модель игры и получают решение, которое становится их стратегией. Другие боты – это виртуальные пользователи. Программа сама перемещает мышь по экрану, нажимает клавиши и выполняет прочие функции, присущие биологическому организму, сидящему перед монитором. Ещё одна разновидность ботов: чат-боты.

Чат-бот – это специальный бот, который позволяет вести диалог с человеком путём вычленения информации на основе определённых шаблонов поиска и приведения её к стандартизированному виду. При этом стандарт в каждом чат-боте задаётся по-разному. Семантический разбор естественного языка считается достаточно сложной задачей. Чтобы её облегчить, были придуманы шаблоны семантического анализа. Проще говоря, это некоторые алгоритмы, применяемые к тексту, которые возвращают наличие или отсутствие в нём определённого признака.

Чат-боты в основном используются для общения с людьми в чатах, на форумах и пр. Однако некоторая их категория применяется для анализа текста и выявления в нём информации с последующим помещением её в реляционные БД.

Принцип работы чат-бота заключается в реализации этапов: бот принимает входящие сообщения, анализирует их и отсылает результат выполнения и/или выполняет команду. Следовательно, общение в чат-ботах осуществляется путем ввода сообщений и вывода ответа (мнения) собеседника. Здесь возможны два вида ведения разговора: обычная беседа или обсуждение важного вопроса. Но, в отличие от разговора людей, программа не обладает гибким интеллектом, поэтому большинство виртуальных собеседников запрограммированы на ведение несложной беседы. Такие программы относят к классу программ с естественным языковым интерфейсом. Обработка естественного языка человека, особенно разговорного стиля, является проблемой, касающейся искусственного интеллекта. Проблема создания программ собеседников на базе искусственного интеллекта, которые могут моделировать интеллектуальную деятельность человека, на сегодняшний день остается открытым.

К сожалению, современные виртуальные собеседники лишь частично решают вопрос имитации разговора человека. Основу их функционирования составляет база знаний. В простейшем случае она содержит наборы возможных вопросов пользователя и соответствующих ответов на них. Наиболее распространенные методы выбора ответов в данном случае следующие: реакция на ключевые слова; совпадение фразы пользователя с той, которая есть в базе знаний; также программа может учитывать порядок слов. Программы-собеседники не могут использовать фразы, насыщенные местоимениями. В таких случаях программы анализируют предыдущие фразы пользователя и выбирают наиболее приемлемый ответ. Также проблематичным может быть подбор слов-синонимов.

На сегодняшний день разработано большое количество ботов. Среди них можно выделить наиболее распространенные: *A*.*L*.*I*.*C*.*E*., *ChatMaster*, *Electronic* *Brain*, *ELIZA*, *George*, *NAI*, *SkypeTalk* и другие.

**2.3 Язык разметки искусственного интеллекта *AIML***

На сегодняшний день остается актуальным создание программ имитирующих общение человека. Простейшей моделью общения является база вопросов и ответов к ним. В данном случае возникает проблема описания базы знаний и реализация программы интерпретатора. Язык разметки базы знаний может включать в себя паттерны вопросов и соответствующие им шаблоны ответов, также предысторию диалогов к ним и название соответствующей темы общения.

*AIML* (*Artificial* *Intelligence* *Markup* *Language*) – язык разметки для искусственного интеллекта. *AIML* – диалект *XML* для создания виртуальных собеседников.

Файл *AIML* – содержит набор категорий (*category*). Категория открывается тегом *<category>* и закрывается тегом *</category>.* Каждому тегу открытия должен соответствовать тег закрытия.

Категория (*category*) включает паттерны (*pattern*, обращения) и темплейты (*template*, ответные реплики). Паттерн пишется заглавными буквами.

Для определения произвольного слова или группы слов используется знак \*

Примеры паттернов:

1. <pattern> ПРИВЕТ*</pattern>*
2. *<pattern>* ПРИВЕТ *\*</pattern>*
3. *<pattern> \** ПРИВЕТ *\*</pattern>*
4. *<pattern> \** ПРИВЕТ*</pattern>*

Паттерн 2.3 а подойдет только для входящей реплики содержащей единственное слово «Привет» – и ни для какой другой.

Паттерн 2.3 б подойдет только для входящей реплики, начинающейся со слова «Привет» и содержащей еще какие-нибудь слова после слова «Привет», например, «Привет Бот!».

Паттерн 2.3 в подойдет только для входящей реплики, начинающейся с 1 или нескольких слов, за которым следует слово «Привет» и содержащей еще какие-нибудь слова после слова «Привет», например, «Эй ты привет красавчик!».

Паттерн 2.3 г подойдет только для входящей реплики, начинающейся с одного или нескольких слов, за которым следует слово «Привет», например, «Бот, привет!».

Темплейт может включать одну реплику, например:

*<template>* Добрый день!*</template>*

Или несколько реплик. В этом случае необходимо использовать дополнительный тег *<random>.* Пример:

*<template>*

*<random>*

*<li>* Добрый день! *</li>*

*<li>* Здравствуйте *</li>*

*<li>* Приветик… *</li>*

*</random>*

*</template>*

В этом случае в качестве ответной реплики будет произвольно выбрана одна из реплик, находящаяся между тегами *<li> </li>.*

Рандом можно комбинировать с обычной репликой. Например:

*<template>*

*<random>*

*<li>* Добрый день! *</li>*

*<li>* Здравствуйте *</li>*

*<li>* Приветик… *</li>*

*</random>*

Как дела?

*</template>*

В этом случае бот ответит «Добрый день! Как дела?» или «Здравствуйте Как дела?» или «Приветик… Как дела?»

**2.4 Основные свойства платформы *Java***

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* 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.

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.

***2.4.1*** 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*.

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 [6].

***2.4.2*** *Java Virtual Machine* – виртуальная машина *Java* – основная часть исполняющей системы *Java*, так называемой *Java Runtime Environment* (*JRE*). Виртуальная машина *Java* исполняет байт-код *Java*, предварительно созданный из исходного текста *Java*-программы компилятором *Java*. *JVM* может также использоваться для выполнения программ, написанных на других языках программирования. Например, исходный код на языке *Ada* может быть откомпилирован в байт-код *Java*, который затем может выполниться с помощью *JVM*.

*JVM* является ключевым компонентом платформы *Java*. Так как виртуальные машины *Java* доступны для многих аппаратных и программных платформ, *Java* может рассматриваться и как связующее программное обеспечение, и как самостоятельная платформа. Использование одного байт-кода для многих платформ позволяет описать *Java* как “скомпилировано однажды, запускается везде”.

Виртуальные машины *Java* обычно содержат интерпретатор байт-кода, однако, для повышения производительности во многих машинах также применяется *JIT*-компиляция часто исполняемых фрагментов байт-кода в машинный код.

Программы, предназначенные для запуска на *JVM*, должны быть скомпилированы в стандартизированном переносимом двоичном формате, который обычно представляется в виде файлов *.class*. Программа может состоять из множества классов, размещенных в различных файлах. Для облегчения размещения больших программ часть файлов вида *.class* могут быть упакованы вместе в так называемый .*jar*-файл.

Виртуальная машина *JVM* исполняет файлы *.class* или *.jar*, эмулируя инструкции, написанные для *JVM*, путем интерпретирования или использования *just-in-time* компилятора (*JIT*), такого, как *HotSpot* от *Sun* *microsystems*. В наши дни *JIT* компиляция используется в большинстве *JVM* в целях достижения большей скорости. Существуют также *ahead-of-time* компиляторы, позволяющие разработчикам приложений перекомпилировать файлы классов в родной для конкретной платформы код.

*JVM*, которая является экземпляром *JRE* (*Java Runtime Environment*), вступает в действие при исполнении программ *Java*. После завершения исполнения этот экземпляр удаляется сборщиком мусора. *JIT* является частью виртуальной машины *Java*, которая используется для ускорения выполнения приложений. *JIT* одновременно компилирует части байт-кода, которые имеют аналогичную функциональность, и, следовательно, уменьшает количество времени, необходимого для компиляции [7].

* 1. **Описание аналогов**

Существует множество возможностей для организации своего досуга, что порой очень трудно найти что-то подходящее. Невозможно знать все места отдыха, ведь постоянно открывается что-то новое, поэтому и необходимо как-то хранить и структурировать эту информацию.

Наиболее распространенным способом представления большого количества информации является вебсайт.

Недостатком всех сайтов является сложность в их разработке и поддержке. В связи с большим разнообразием размеров и разрешений экранов, бывает сложно сделать корректное отображение контента на конечных устройствах пользователей. А к преимуществам всех сайтов можно отнести их легкодоступность, т.к. они не требуют никакой дополнительной установки, а лишь наличие браузера на устройстве конечного пользователя.

Одним из аналогов системы описывающей услуги города Гомеля является сайт *www.orangesmile.com*. Данный сервис содержит структурированное описание большого количества стран и городов, что является его не оспоримым преимуществом. На рисунке 2.1 изображена страница с описанием города Гомеля.



Рисунок 2.1 – Сервис *www.orangesmile.com*

Однако к недостаткам данного аналога можно отнести что поиск по высоко загруженным сайтам может быть сложен и может требовать не всегда очевидных шагов. Так же к недостаткам можно отнести большое количество рекламы в контенте.

Альтернативным способом доставки контента является мобильное приложение. Преимуществом всех мобильных приложений является их визуальная составляющая. Аналог, разрабатываемого бота, в данном сегменте – приложение «Гомель – городской гид». Данный продукт предоставляет систематизированную информацию о заведениях и услугах города Гомеля. Внешний вид приложения изображен на рисунке 2.2.

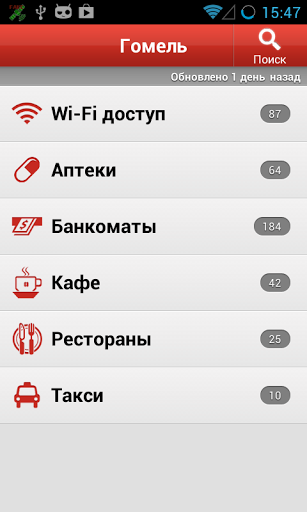


Рисунок 2.2 – Приложение «Гомель – городской гид»

К преимуществам данного приложения можно отнести богатый визуальный интерфейс, широкий ассортимент описываемых услуг. А так же хранение всей информации на устройстве пользователя, что позволяет полноценно пользоваться приложением без необходимости подключения к сети Интернет.

В связи с широким распространением мобильных приложений и перегрузкой магазинов приложений, необходимость установки на конечное устройство с ограниченным объемом памяти является существенным минусом. Так же к минусам можно отнести, что данный аналог является Android приложением, следовательно пользователи других мобильных платформ не могут его использовать.

Недостатком всех мобильных приложений является сложность их обновлений и поддержки, т.к. это заставляет пользователя постоянно поддерживать приложение в актуальном состоянии, посредством отдельного скачивания новой версии приложения или его обновлений.

**3 РАЗРАБОТКА ПРОГРАММНЫХ СРЕДСТВ ДЛЯ РЕШЕНИЯ ЗАДАЧИ**

**3.1 Особенности разработки приложения**

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 3.1.

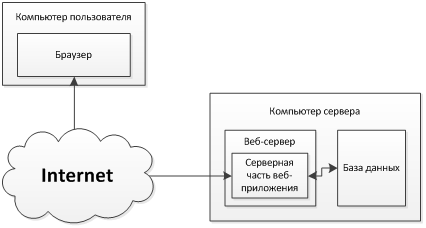


Рисунок 3.1 – Общая структура веб-приложения

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 [8].

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.

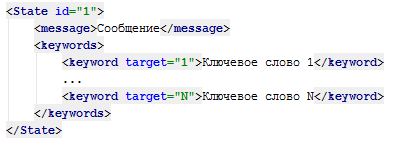


Рисунок 3.2 – Структура хранимых данных

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.

**3.2 Описание структуры приложения**

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.

***3.2.1*** 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. На иллюстрации 3.3 отображена схема фреймворка.

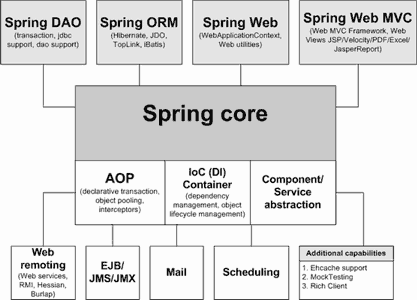


Рисунок 3.3 – Структура *Spring Framework*

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

*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 Б. 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.

***3.2.2*** 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. [9].

Общую схему взаимодействия упрощенно представлена на рисунке 3.4.

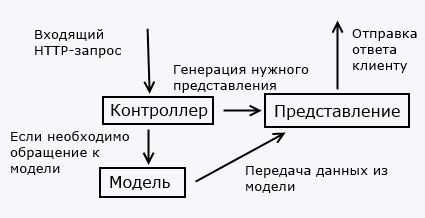


Рисунок 3.4 – Структура паттерна *MVC*

***3.2.3*** В разрабатываемом приложение роль представления выполняют *jsp* страницы, а так же интеграция с одним из самых прогрессивный и функциональный мессенджер на сегодняшний день –*Telegram*.

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.

***3.2.4*** *Telegram* – бесплатный кроссплатформенный мессенджер для смартфонов и других устройств, позволяющий обмениваться текстовыми сообщениями и медиафайлами различных форматов. Используются проприетарная серверная часть c закрытым кодом и несколько клиентов с открытым исходным кодом, в том числе под *GNU GPL*.

Учётные записи пользователей привязываются к телефонным номерам. При регистрации в сервисе и последующих авторизациях новых устройств, производится проверка телефонного номера через отправку *SMS*-сообщения с или телефонный вызов.

При помощи специального *API* сторонние разработчики могут создавать «ботов», специальные аккаунты, управляемые программами. Типичные боты отвечают на специальные команды в персональных и групповых чатах, также они могут осуществлять поиск в интернете или выполнять иные задачи, применяются в развлекательных целях или в бизнесе.

Для интеграции бота в Telegram, было создано отдельное приложение использующее библиотеку *TelegramBots*. Это простая в использовании библиотека для создания ботов в *Java* для *Telegram*, поддерживающая встроенный режим.

***3.2.5*** Для проверки и исправления приходящих от пользователя сообщений используется сервис проверки правописания Яндекс.Спеллер (коротко Спеллер). Он предлагает веб-разработчикам использовать возможность интерактивной проверки орфографии на страницах своих сайтов.

Спеллер позволяет выявлять и исправлять орфографические ошибки в текстах, набираемых посетителями сайтов в различных *HTML*-формах.

Спеллер анализирует слова, основываясь на правилах орфографии и лексике современного языка. В качестве словарного источника используется орфографический словарь, содержащий правильные написания большинства наиболее употребимых слов.

Сервис в настоящее время поддерживает три языка:

1. русский – словарь содержит 3.6 миллиона словоформ (разработка компании «Информатик»);
2. украинский – словарь содержит 1.8 миллиона словоформ (разработка компании «Информатик»);
3. английский – словарь содержит 150000 словоформ (разработка компании «Яндекс»).

Яндекс.Спеллер выявляет и исправляет только орфографические ошибки, то есть ошибки в отдельных словах, такие как:

* неправильные, пропущенные или лишние буквы, например, «карова», «велоспед», «караввай»;
* ошибки капитализации (неправильное употребление прописных и строчных букв), например, «мОСКВА»;
* повторы слов, например, «поезд поехал поехал быстрее».

Пунктуационные, грамматические (ошибки согласования слов) и стилистические ошибки не исправляются.

Яндекс.Спеллер является *HTTP*-службой и предоставляет *Web Service API* для обработки клиентских запросов на проверку орфографии. Яндекс.Спеллер поддерживает *HTTP* *GET*- и *POST*-запросы и работает в кодировке *UTF-8*.

Для доступа к Яндекс.Спеллеру по *HTTP* предлагаются *XML*-, *SOAP*-, *JSON*- и *JSONP*-интерфейсы. Все интерфейсы обеспечивают одинаковую функциональность и используют одни и те же входные параметры.

*XML*- и *SOAP*- интерфейсы возвращают ответ в виде *XML*-документа, *JSON*-интерфейс вместо *XML*-элементов возвращает *JavaScript*-объекты с теми же именами и семантикой, а *JSONP*-интерфейс возвращает те же самые *JavaScript*-объекты, но в виде вызова *callback*-функции с заданным именем.

Доступ к *API* предоставляется по следующим *URL*:

1. *XML*- и *SOAP*-интерфейс *http://speller.yandex.net/services/spellservice*;
2. *WSDL*-документ для *SOAP* доступен по адресу *http://speller.yandex.net/services/spellservice?WSDL*;
3. *JSON*- и *JSONP*-интерфейс *http://speller.yandex.net/services/spellservice.json*

Для включения *JSONP*-интерфейса требуется передать дополнительный параметр callback с именем функции обратного вызова.

*Web Service API* включает в себя два метода, которые позволяют проверять правописание в одном или нескольких фрагментах текстов: *checkText* и *checkTexts.*

***3.2.6*** Для автоматизации сборки проекта использовался фреймворк *Maven*. Это инструмент для сборки *Java* проекта: компиляции, создания *jar*(*war*), создания дистрибутива программы, генерации документации. Простые проекты можно собрать в командной строке. Если собирать большие проекты с командной строки, то команда для сборки будет очень длинной, поэтому её иногда записывают в *bat/sh* скрипт. Но такие скрипты зависят от платформы. Для того чтобы избавиться от этой зависимостии и упростить написание скрипта используют инструменты для сборки проекта.

Для платформы *Java* существуют два основных инструмента для сборки: *Ant* и *Maven*.

*Maven* не зависит от *OS*. Сборка проекта происходит в любой операционной системе. Файл проекта один и тот же. Он позволяет управлять зависимостями. Редко какие проекты пишутся без использования сторонних библиотек. Эти сторонние библиотеки зачастую тоже в свою очередь используют библиотеки разных версий. *Maven* позволяет управлять такими сложными зависимостями. Что позволяет разрешать конфликты версий и в случае необходимости легко переходить на новые версии библиотек.

Возможна сборка из командной строки. Такое часто необходимо для автоматической сборки проекта на сервере (*Continuous Integration*).

Имеет хорошая интеграцию со средами разработки. Основные среды разработки на *Java* легко открывают проекты которые собираются c помощью *Maven*. При этом зачастую проект настраивать не нужно – он сразу готов к дальнейшей разработке.

Как следствие – если с проектом работают в разных средах разработки, то *Maven* удобный способ хранения настроек. Настроечный файл среды разработки и для сборки один и тот же – меньше дублирования данных и соответственно ошибок.

*Maven* обеспечивает декларативную, а не императивную (в отличие от средства автоматизации сборки *Apache* *Ant*) сборку проекта. В файлах описания проекта содержится его спецификация, а не отдельные команды выполнения. Все задачи по обработке файлов, описанные в спецификации, *Maven* выполняет посредством их обработки последовательностью встроенных и внешних плагинов.

Информация для сборки проекта, поддерживаемого *Apache* *Maven*, содержится в *XML*-файле с названием *pom*.*xml*. При запуске *Maven* проверяет, содержит ли конфигурационный файл все необходимые данные и все ли данные синтаксически правильно записаны. Минимальная конфигурация включает версию конфигурационного файла, имя проекта, его автора и версию. С помощью *pom.xml* конфигурируются зависимости от других проектов, индивидуальные фазы процесса построения проекта (*build process*), список плагинов, реализующих порядок сборки.

## 3.3 Описание разработанного приложения

Приложение можно условно разделить на серверную и клиентскую части. Серверная часть представляет собой набор *java* классов. Структура этихклассов представлена на рисунке 3.5.

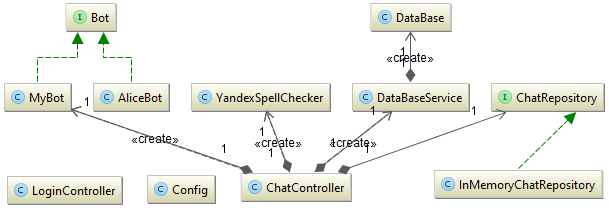


Рисунок 3.5 – *Java* классы приложения

Отображенные на рисунке выше классы, отвечают за работу приложения на сервере. *Java* код классов находится в приложении.

Класс *YandexSpellerCheck* представляет собой интерфейс для работы с сервисом проверки правописания *Yandex* *Speller*. Структура класс изображена на рисунке 3.6.

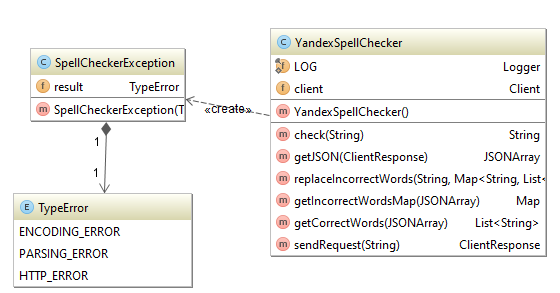


Рисунок 3.6 – Методы класса *YandexSpellerCheck*

Класс *YandexSpellerCheck* представляет собой интерфейс для работы с сервисом проверки правописания *Yandex* *Speller*.

Метод *sendRequest* получает строку и отвечает за формирование *http* cсоединения и отправку запроса к серверу *Yandex* *Speller*. И возвращает полученный *response*.

Метод *getJSON* входным параметром получает *response*. Выполняет парсинг ответа в объект типа *JSONArray*.

Метод *getIncorrectWordsMap* из объекта типа *JSONArray* формирует объект *Map*, в котором ключом являются слова, в которых допущена ошибка, а значением набор правильных вариантов написания этого слова.

Метод *replaceIncorrectWords* заменяет неправильные слова в исходном сообщении их корректными версиями на основе переданного объекта *Map*.

Метод *check* в качестве входного параметра получает сооб \*щение пользователя, вызовом методов описанных выше, проверяет это сообщение и исправляет ошибки.

Класс *DataBase* отвечает за работу с базой данных. Содержит методы установление соединения с базой, закрытие соединения и отправку запроса к базе и получения. Структура данного класса изображена на рисунке 3.7.

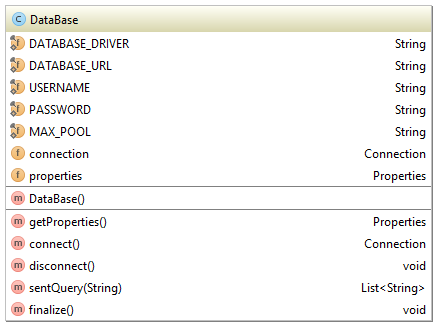


Рисунок 3.7 – Методы класса *DataBase*

Метод *connect* использует описанные в классе константы для создания соединения с базой данных *MySQL*. А метод *disconnect* соответственно закрывает соединение с базой данных.

Метод *sendQuery* формирует запрос к базе данных на основе полученной строки и выполняет этот запрос. После чего конвертирует полученный результат в набор строк и возвращает его.

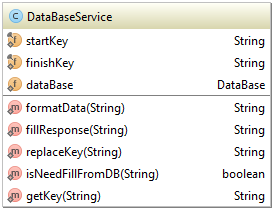


Рисунок 3.13 – Методы класса *DataBaseService*

Класс *DataBaseService* является посредником между контроллером и классом *DataBase*. Позволяет динамически добавлять в сообщения бота персистентную информацию, хранящуюся в базе данных.

Место инжекции дополнительной информации определяется фигурными скобками. Метод *getKey* выделяет маркер персистентной информации. А метод *replaceKey* заменяет маркер на полученную из базы данных информацию, предварительно форматированную в удобочитаемый пользователю формат, методом *formatData*.

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

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.

Приложение *TelegramBot* – это консольное *Java* приложение, представляющее собой обертку для разрабатываемого чат-бота и отвечающее за интеграцию этого бота в *Telegram*. Данное приложение написано с использованием *API* *TelegramBots.* Для работы приложения предварительно был создан и зарегистрирован бот в мессенджере *Telegram*. Структура приложения изображена на рисунке 3.14.

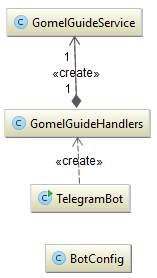


Рисунок 3.14 – Структура приложения *TelegramBot*

При регистрации нового бота в *Telegram*, необходимо было указать имя бота, после чего был выдан *token* для подключения к каналу созданного бота. Эти настройки хранятся в классе *BotConfig*.

Класс *TelegramBot* отвечает за запуск приложения и подключение его к серверам *Telegram*. Структура классов представлена на рисунке 3.15

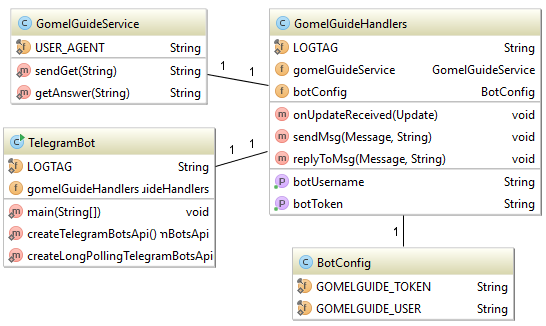


Рисунок 3.15 – Структура классов приложения *TelegramBot*

Класс *GomelGuideHandlers* получает сообщения пользователя и отправляет ответ. Содержит методы *onUpdateReceived* и *sendMsg*.

Метод *onUpdateReceived* вызывается при получении сообщения от пользователя, передает сообщение пользователя в метод *getAnswer* класса *GomelGuideService* и вызывает метод *sendMsg*.

Метод *sendMsg* формирует ответное сообщение и отправляет его пользователю.

Передает сообщение пользователя в метод *getAnswer* класса *GomelGuideService.* Формирует ответное сообщение и отравляет.

Класс *GomelGuideService* содержит метод *getAnswer* принимающий в качестве входного параметра строку *message*. Этот метод создает *http* соединение к созданному веб-приложению (чат-боту). Отправляет *post* запрос с параметром *message,* в рамках этого соединения и возвращает полученный ответ.

**3.5 Описание интерфейса пользователя**

Для работы с ботом через мессенджер *Telegram*, необходимо открыть соответствующее приложение и найти в поиске бота *GomelGuide*. После чего откроется страница диалога с ботом. Пример переписки с ботом изображен на рисунке 3.11.

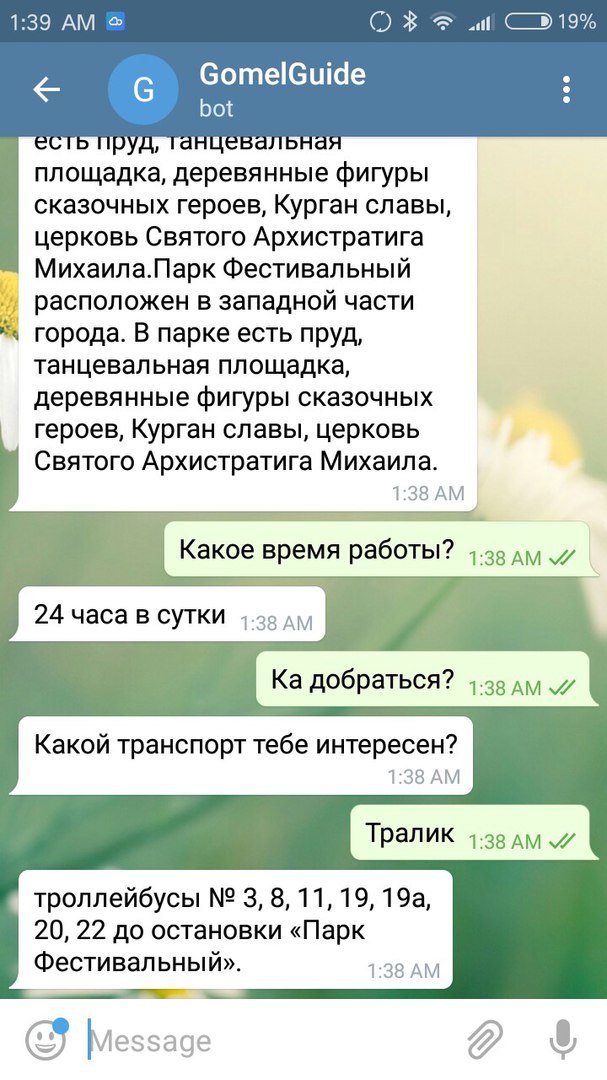
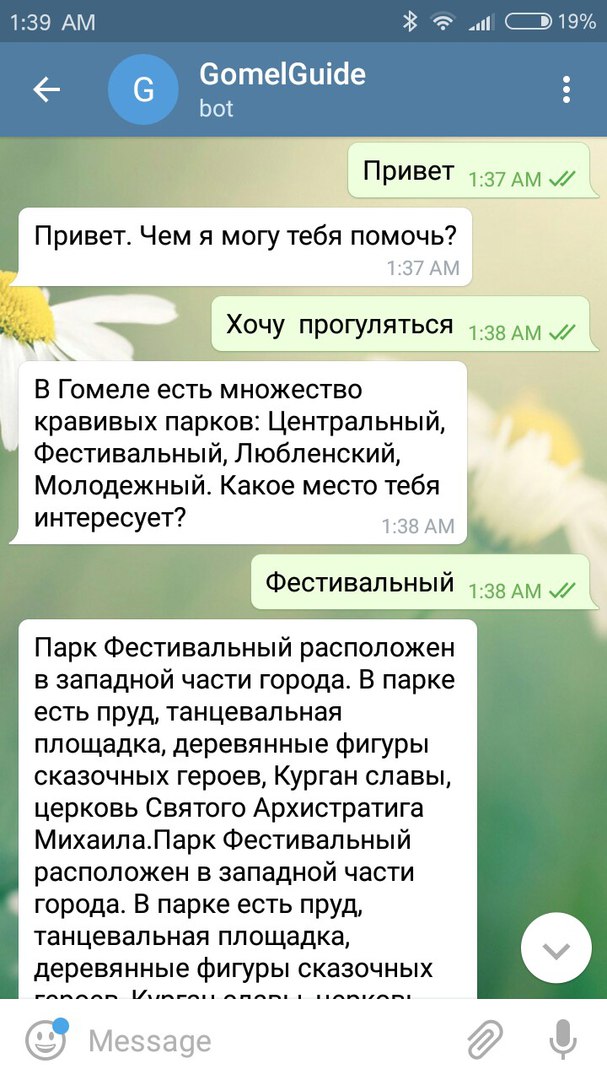


Рисунок 3.11 – Окно диалога в *Telegram*

Отправка сообщений в данном мессенджере происходит посредством ввода текста в соответствующее поле и нажатием клавиши ввода на экранной клавиатуре. Для инициации разговора с ботом необходимо отправить ему любое сообщение.

Для прекращения разговора достаточно выйти из окна диалога с ботом, либо закрыть приложение.

# ЗАКЛЮЧЕНИЕ

Уникальные возможности Интернет такие, как скорость, оперативность, доступность коммуникации между пользователями – позволяют использовать сеть как средство общения и создавать интерактивные формы общения: чаты, форумы, телеконференции, электронную почту и другие. На смену реальным собеседникам приходят программы искусственного интеллекта, такие как чаты, консультанты, помощники, развлекательные программы и другие. Но, в отличие от разговора людей, программа не обладает гибким умственным интеллектом. К сожалению, современные виртуальные собеседники лишь частично решают вопрос имитации разговора человека. Словарный запас большинства виртуальных собеседников ограничен, кроме этого, у них отсутствует эмоциональная окраска, тембр голоса и пр., поэтому большинство виртуальных собеседников запрограммированы на ведение несложной беседы. Обработка естественного языка человека, особенно разговорного стиля, является проблемой, касающейся искусственного интеллекта. В основу функционирования виртуальных собеседников положена база знаний. В простейшем случае она содержит наборы возможных вопросов пользователя и соответствующих ответов на них. Некоторые программы могут учиться, а именно: пополнять словарный запас, учитывать определенные особенности языка, стиля общения. Но проблема создания программ собеседников на базе искусственного интеллекта, которые могут моделировать интеллектуальную деятельность человека, на сегодняшний день остается открытой. Несмотря на преимущества, виртуальные собеседники в настоящее время не могут пройти тест Тьюринга на соответствие интеллекта компьютера человеческому интеллекту.

К сожалению, на современном этапе своего развития виртуальное общение не может заменить реальное, но компьютерные технологии постоянно совершенствуются, поэтому, возможно, через какое-то время виртуальные собеседники смогут заменить реальное общение.

**Список использованных источников**

1. Система стандартов безопасности труда. Рабочее место при выполнении работ сидя. Общие эргономические требования: ГОСТ 12.2.032-78. – Введ. 01.01.79. – Мн. : Межгос. совет по стандартизации, метрологии и сертификации: Белорус. гос. ин-т стандартизации и сертификации, 2004. – 12 с.
2. Санитарные правила и нормы. Гигиенические требования к видеодисплейным терминалам, электронно-вычислительным машинам и организации работы : СанПиН 9-131 РБ 2000 : утв. Постановлением Главного гос. санитарного врача Республики Беларусь № 53 от 10.11.2000 г. : ввод. в действие с 10.11.2000 г. – Мн. : ГУ НИИ санитарии и гигиены, 2000. – 64 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. Java Virtual Machine [Электронный ресурс]. – Режим доступа: https://ru.wikipedia.org/wiki/Java\_Virtual\_Machine. – Дата доступа: 12.02.2017.
7. Брюс Э. Философия *Java*. Библиотека программиста / Э. Брюс – СПб.: Питер, 2009. – 640с.
8. 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: 15.02.2017.
9. Паттерны проектирования. Паттерн *MVС* [Электронный ресурс]. – Режим доступа: http://metanit.com/sharp/mvc5/1.1.php. – Дата доступа: 15.02.2017.

**ПРИЛОЖЕНИЕ**

(Обязательное)

**Листинг классов**

**package** org.telegram;  
  
**import** org.telegram.telegrambots.ApiContextInitializer;  
**import** org.telegram.telegrambots.TelegramBotsApi;  
**import** org.telegram.telegrambots.exceptions.TelegramApiException;  
**import** org.telegram.telegrambots.logging.BotLogger;  
**import** org.telegram.telegrambots.logging.BotsFileHandler;  
**import** org.telegram.updateshandlers.\*;  
  
**import** java.io.IOException;  
**import** java.util.logging.ConsoleHandler;  
**import** java.util.logging.Level;  
  
*/\*\*  
 \* TelegramBot class to create all bots  
 \*/***public class** TelegramBot {  
 **private static final** String ***LOGTAG*** = **"MAIN"**;  
  
 **public static void** main(String[] args) {  
 BotLogger.*setLevel*(Level.***ALL***);  
 BotLogger.*registerLogger*(**new** ConsoleHandler());  
 **try** {  
 BotLogger.*registerLogger*(**new** BotsFileHandler());  
 } **catch** (IOException e) {  
 BotLogger.*severe*(***LOGTAG***, e);  
 }  
  
 **try** {  
 ApiContextInitializer.*init*();  
 TelegramBotsApi telegramBotsApi = *createTelegramBotsApi*();  
 **try** {  
 telegramBotsApi.registerBot(**new** GomelGuideHandlers());  
 } **catch** (TelegramApiException e) {  
 BotLogger.*error*(***LOGTAG***, e);  
 }  
 } **catch** (Exception e) {  
 BotLogger.*error*(***LOGTAG***, e);  
 }  
 }  
  
 **private static** TelegramBotsApi createTelegramBotsApi() **throws** TelegramApiException {  
 TelegramBotsApi telegramBotsApi;  
 telegramBotsApi = createLongPollingTelegramBotsApi();  
 **return** telegramBotsApi;  
 }  
  
 */\*\*  
 \** ***@brief*** *Creates a Telegram Bots Api to use Long Polling (getUpdates) bots.  
 \** ***@return*** *TelegramBotsApi to register the bots.  
 \*/* **private static** TelegramBotsApi createLongPollingTelegramBotsApi() {  
 **return new** TelegramBotsApi();  
 }  
  
}

**package** org.telegram.updateshandlers;  
  
**import** org.telegram.BotConfig;  
**import** org.telegram.services.\*;  
**import** org.telegram.telegrambots.api.methods.send.SendMessage;  
**import** org.telegram.telegrambots.api.objects.Message;  
**import** org.telegram.telegrambots.api.objects.Update;  
**import** org.telegram.telegrambots.bots.TelegramLongPollingBot;  
**import** org.telegram.telegrambots.exceptions.TelegramApiException;  
  
**public class** GomelGuideHandlers **extends** TelegramLongPollingBot {  
 **private static final** String ***LOGTAG*** = **"GOMELGUIDEHANDLERS"**;  
  
 @Override  
 **public** String getBotToken() {  
 **return** BotConfig.***GOMELGUIDE\_TOKEN***;  
 }  
  
 @Override  
 **public** String getBotUsername() {  
 **return** BotConfig.***GOMELGUIDE\_USER***;  
 }  
//вызывается при получении сообщения от пользователя  
 @Override  
 **public void** onUpdateReceived(Update update) {  
 Message message = update.getMessage();  
 **if** (message != **null** && message.hasText()) {  
 sendMsg(message, GomelGuideService.*getAnswer*(message.getText()));  
 }  
 }  
//формирует сообщение, отпраляемое пользователю  
 **private void** sendMsg(Message message, String text) {  
 SendMessage sendMessage = **new** SendMessage();  
 sendMessage.enableMarkdown(**true**);  
 sendMessage.setChatId(message.getChatId().toString());  
 sendMessage.setText(text);  
 **try** {  
 sendMessage(sendMessage);  
 } **catch** (TelegramApiException e) {  
 e.printStackTrace();  
 }  
 }  
//формирование ответа содержащее сообщение пользователя  
 **private void** replyToMsg(Message message, String text) {  
 SendMessage sendMessage = **new** SendMessage();  
 sendMessage.enableMarkdown(**true**);  
 sendMessage.setChatId(message.getChatId().toString());  
 sendMessage.setReplyToMessageId(message.getMessageId());  
 sendMessage.setText(text);  
 **try** {  
 sendMessage(sendMessage);  
 } **catch** (TelegramApiException e) {  
 e.printStackTrace();  
 }  
 }  
}

**package** org.telegram.services;  
  
**import** org.apache.http.HttpEntity;  
**import** org.apache.http.HttpResponse;  
**import** org.apache.http.NameValuePair;  
**import** org.apache.http.client.entity.UrlEncodedFormEntity;  
**import** org.apache.http.client.methods.HttpPost;  
**import** org.apache.http.impl.client.CloseableHttpClient;  
**import** org.apache.http.impl.client.HttpClients;  
**import** org.apache.http.message.BasicNameValuePair;  
**import** org.apache.http.util.EntityUtils;  
  
**import** javax.net.ssl.HttpsURLConnection;  
**import** java.io.\*;  
**import** java.net.MalformedURLException;  
**import** java.net.ProtocolException;  
**import** java.net.URL;  
**import** java.net.URLDecoder;  
**import** java.util.ArrayList;  
**import** java.util.List;  
  
**public class** GomelGuideService {  
  
 **private static final** String ***USER\_AGENT*** = **"Mozilla/5.0"**;  
  
 *// HTTP GET request* **public static** String sendGet(String parameter) {  
  
 String url = **"https://localhost:8080/chat/post"**;  
 **try** {  
 URL obj = **new** URL(url);  
 HttpsURLConnection con = (HttpsURLConnection) obj.openConnection();  
  
 *// optional default is GET* con.setRequestMethod(**"GET"**);  
  
 *//add request header* con.setRequestProperty(**"User-Agent"**, ***USER\_AGENT***);  
  
 **int** responseCode = con.getResponseCode();  
 System.***out***.println(**"\nSending 'GET' request to URL : "** + url);  
 System.***out***.println(**"Response Code : "** + responseCode);  
  
 BufferedReader in = **new** BufferedReader(  
 **new** InputStreamReader(con.getInputStream()));  
 String inputLine;  
 StringBuffer response = **new** StringBuffer();  
  
 **while** ((inputLine = in.readLine()) != **null**) {  
 response.append(inputLine);  
 }  
 in.close();  
  
 *//print result* System.***out***.println(response.toString());  
 **return** response.toString();  
 } **catch** (MalformedURLException e) {  
 e.printStackTrace();  
 } **catch** (ProtocolException e) {  
 e.printStackTrace();  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }  
 **return** parameter;  
 }  
  
 *// HTTP POST request* **public static** String getAnswer(String message) {  
 **try** {

//формирование pоst запроса  
 CloseableHttpClient httpclient = HttpClients.*createDefault*();  
 HttpPost httppost = **new** HttpPost(**"http://localhost:8080/chat"**);  
  
 List<NameValuePair> params = **new** ArrayList();  
 params.add(**new** BasicNameValuePair(**"message"**, message));  
 httppost.setEntity(**new** UrlEncodedFormEntity(params, **"UTF-8"**));  
//отправка запроса и получение ответа  
 HttpResponse response = httpclient.execute(httppost);  
 HttpEntity entity = response.getEntity();  
  
 **if** (entity != **null**) {  
 InputStream instream = entity.getContent();  
 **try** {

//возврат полученного декодированного ответа  
 **return** URLDecoder.*decode*(EntityUtils.*toString*(entity), **"UTF-8"**);  
 } **finally** {  
 instream.close();  
 }  
 }  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }  
 **return** message;  
 }  
}

**package** by.gstu.chatbot.core;  
  
**import** com.sun.jersey.api.client.Client;  
**import** com.sun.jersey.api.client.ClientResponse;  
**import** com.sun.jersey.api.client.WebResource;  
**import** org.apache.log4j.Logger;  
**import** org.json.simple.JSONArray;  
**import** org.json.simple.JSONObject;  
**import** org.json.simple.parser.JSONParser;  
**import** org.json.simple.parser.ParseException;  
  
**import** java.io.UnsupportedEncodingException;  
**import** java.net.URLEncoder;  
**import** java.util.\*;  
  
**public class** YandexSpellChecker {  
  
 **final static** Logger ***LOG*** = Logger.*getLogger*(YandexSpellChecker.**class**);  
  
 **private** Client **client**;  
  
 **public** YandexSpellChecker() {  
 **client** = Client.*create*();  
 }  
**public** String check(String text) {  
 **try** {  
 //формирование http соединения и отправка запроса

ClientResponse response = sendRequest(text);

//парсинг полученного ответа в объект json  
 JSONArray array = getJSON(response);  
 **if**(array.isEmpty()){  
 **return** text;  
 }

//получаем все слова, содержащие ошибки  
 Map incorrectWords = getIncorrectWordsMap(array);

//заменяем все неправильные слова на их правильную версию  
 String correctedText = replaceIncorrectWords(text, incorrectWords);  
 **return** correctedText;  
 } **catch** (SpellCheckerException e) {  
 ***LOG***.debug(e.getMessage());  
 **return** text;  
 }  
 }  
//парсит объект типа ClientResponse в объект типа JSONArray  
 **private** JSONArray getJSON(**final** ClientResponse response) **throws** SpellCheckerException {  
 String output = response.getEntity(String.**class**);  
 JSONParser parser = **new** JSONParser();  
 Object obj;  
 **try** {  
 obj = parser.parse(output);  
 } **catch** (ParseException e) {  
 **throw new** SpellCheckerException(TypeError.***PARSING\_ERROR***, **"PARSING\_ERROR"**);  
 }  
  
 JSONArray array = (JSONArray) obj;  
 **if** (array == **null**) {  
 **throw new** SpellCheckerException(TypeError.***PARSING\_ERROR***, **"PARSING\_ERROR"**);  
 }  
 **return** array;  
 }  
//заменяет все неправильные слова в сообщении на правильные  
 **private** String replaceIncorrectWords(**final** String text, **final** Map<String, List<String>> replacementWords) **throws** SpellCheckerException {  
 String correctText = text;  
 **for** (Map.Entry<String, List<String>> entry : replacementWords.entrySet()) {  
 **if**(!entry.getValue().isEmpty()) {  
 correctText = correctText.replace(entry.getKey(), entry.getValue().get(0));  
 }  
 }  
 **return** correctText;  
 }  
//получает все неправильно написанные слова в сообщении  
 **private** Map getIncorrectWordsMap(**final** JSONArray array) **throws** SpellCheckerException {  
 Map<String, List<String>> incorrectWords = **new** LinkedHashMap();  
 Iterator<Object> it = array.iterator();  
 **while** (it.hasNext()) {  
 JSONObject jo = (JSONObject) it.next();  
 String incorrectWord = (String) jo.get(**"word"**);  
 List<String> correctWords = getCorrectWords((JSONArray) jo.get(**"s"**));  
 incorrectWords.put(incorrectWord, correctWords);  
 }  
 **return** incorrectWords;  
 }  
//достает слова с правильным написание  
 **private** List<String> getCorrectWords(**final** JSONArray array) **throws** SpellCheckerException {  
 List<String> correctWords = **new** ArrayList<>();Iterator<Object> sit = array.iterator();  
 **while** (sit.hasNext()) {  
 String v = (String) sit.next();  
 correctWords.add(v);  
 }  
 **return** correctWords;  
 }  
//отправляет запрос yandex серверу  
 **private** ClientResponse sendRequest(**final** String text) **throws** SpellCheckerException {  
 String encodedText;  
 **try** {  
 encodedText = URLEncoder.*encode*(text, **"UTF8"**);  
 } **catch** (UnsupportedEncodingException e) {  
 **throw new** SpellCheckerException(TypeError.***ENCODING\_ERROR***, **"ENCODING\_ERROR"**);  
 }  
  
 WebResource webResource = **client** .resource(**"http://speller.yandex.net/services/spellservice.json/checkText"** + **"?text="** + encodedText);  
  
 ClientResponse response = webResource.accept(**"application/json"**)  
 .get(ClientResponse.**class**);  
  
 **if** (response.getStatus() != 200) {  
 **throw new** SpellCheckerException(TypeError.***HTTP\_ERROR***, **"HTTP\_ERROR"**);  
 }  
 **return** response;  
 }  
}  
  
**class** SpellCheckerException **extends** Exception {  
 TypeError **result**;  
 **public** SpellCheckerException(**final** TypeError result, **final** String message) {  
 **super**(result.name());  
 }  
}  
  
**enum** TypeError {  
 ***ENCODING\_ERROR***,  
 ***PARSING\_ERROR***,  
 ***HTTP\_ERROR***;  
}

**package** by.gstu.chatbot.core.datalayer;  
  
**import** java.sql.\*;  
**import** java.util.ArrayList;  
**import** java.util.List;  
**import** java.util.Properties;  
  
**public class** DataBase {  
 *// init database constants* **private static final** String ***DATABASE\_DRIVER*** = **"com.mysql.jdbc.Driver"**;  
 **private static final** String ***DATABASE\_URL*** = **"jdbc:mysql://localhost:3306/gomel"**;  
 **private static final** String ***USERNAME*** = **"root"**;  
 **private static final** String ***PASSWORD*** = **"1234"**;  
 **private static final** String ***MAX\_POOL*** = **"250"**;  
  
 *// init connection object* **private** Connection **connection**;  
 *// init properties object* **private** Properties **properties**;  
  
 **public** DataBase() {  
 connect();  
 }  
  
 *// create properties* **private** Properties getProperties() {  
 **if** (**properties** == **null**) {  
 **properties** = **new** Properties();  
 **properties**.setProperty(**"user"**, ***USERNAME***);  
 **properties**.setProperty(**"password"**, ***PASSWORD***);  
 **properties**.setProperty(**"MaxPooledStatements"**, ***MAX\_POOL***);  
 }  
 **return properties**;  
 }  
  
 *// connect database* **public** Connection connect() {  
 **if** (**connection** == **null**) {  
 **try** {  
 Class.*forName*(***DATABASE\_DRIVER***);  
 **connection** = DriverManager.*getConnection*(***DATABASE\_URL***, getProperties());  
 } **catch** (ClassNotFoundException | SQLException e) {  
 e.printStackTrace();  
 }  
 }  
 **return connection**;  
 }  
  
 *// disconnect database* **public void** disconnect() {  
 **if** (**connection** != **null**) {  
 **try** {  
 **connection**.close();  
 **connection** = **null**;  
 } **catch** (SQLException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 //sent query to database   
 **public** List<String> sentQuery(**final** String query) {  
 List<String> response = **new** ArrayList<>();  
 Statement statement = **null**;  
 **try** {  
 statement = **connection**.createStatement();  
 ResultSet resultSet = statement.executeQuery(query);  
 **while** (resultSet.next()) {  
 response.add(resultSet.getString(1));  
 }  
 statement.close();  
 } **catch** (SQLException e) {  
 e.printStackTrace();  
 }  
  
 **return** response;  
 }  
  
 @Override  
 **protected void** finalize() **throws** Throwable {  
 **this**.disconnect();  
 **super**.finalize();  
 }  
}

**package** by.gstu.chatbot.core.datalayer;  
**public class** DataBaseService {  
  
 **private final static** String ***startKey*** = **"{"**;  
 **private final static** String ***finishKey*** = **"}"**;  
 **private static** DataBase *dataBase* = **new** DataBase();  
//форматирует полученный из базы данных результат в отображаемый вид  
 **public static** String formatData(**final** String key) {  
 **final** List<String> data = *dataBase*.sentQuery(key);  
 **final** StringBuilder response = **new** StringBuilder();  
 **for** (**int** i = 0; i < data.size() - 1; i++) {  
 response.append(data.get(i)).append(**", "**);  
 }  
 response.append(data.get(data.size() - 1));  
 **return** response.toString();  
 }  
  
 **public static** String fillResponse(String response) {  
 **if** (!StringUtils.*isEmptyOrWhitespaceOnly*(response) && *isNeedFillFromDB*(response)) {  
 response = *replaceKey*(response);  
 }  
 **return** response;  
 }  
//заменяет маркер на данные из базы данных  
 **public static** String replaceKey(String response) {  
 **final** String key = *getKey*(response);  
 **return** response.replace(***startKey*** + key + ***finishKey***, *formatData*(key));  
 }  
//проверяет необходимо ли наполнять запрос данными из базы данных  
 **private static boolean** isNeedFillFromDB(**final** String response) {**return** response.contains(***startKey***) && response.contains(***finishKey***);  
 }  
//извлекает из сообщения маркер  
 **private static** String getKey(**final** String response) {  
 String keyword = response.substring(response.indexOf(***startKey***) + 1, response.indexOf(***finishKey***));  
 **return** keyword;  
 }  
}