

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 быстрее 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 распознавании 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»

1.1 О компании

Компания **EPAM** – ведущий разработчик заказного программного обеспечения и один из ведущих игроков в области консалтинга в Центральной и Восточной Европе, на настоящий момент она имеет филиалы более чем в 20 странах мира и продолжает стабильный рост.

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.

Преимущества 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 История развития компании

Компания *EPAM* была основана в 1993 году в Принстоне (штат Нью-Джерси) двумя одноклассниками Аркадием Добкиным (он ранее эмигрировал в США и работал в компании *SAP*) и Леонидом Лознером. Название компании

происходило от «*Effective Programming for America*». Первые офисы были открыты в США и Беларуси. Позже были открыты центральный североамериканский офис в Лоренсвилле, США, штат Нью-Джерси и центральный европейский офис в Будапеште, Венгрия, а также офисы по обслуживанию клиентов в Великобритании, Германии, России, Казахстане, Украине, Польше, Швеции, Швейцарии, Болгарии и Беларуси.

В марте 2004 года *EPAM* приобрела компанию *Fathom Technology* в Венгрии, а в сентябре 2006 *VDI* в России, образовав единую компанию под именем *EPAM Systems* со штатом сотрудников в 2200 человек.

В 2012 году компания совершает ряд приобретений на североамериканском рынке, в числе которых канадская компания *Thoughtcorp* и крупный поставщик услуг по разработке цифровых стратегий и организации мультимедийного взаимодействия *Empathy Lab*.

В 2014 году *EPAM* приобрела китайскую ИТ-компанию *Jointech (Joint Technology Development Limited)*, за счет чего, как следует из пресс-релиза *EPAM*, расширила свои возможности в Азиатско-Тихоокеанском регионе и купила американского поставщика услуг для здравоохранения и медико-биологического сектора *GGA Software Services*.

В 2015 *EPAM Systems* поглотил американские компании: *Navigation Arts*, специализирующуюся на цифровом консалтинге и дизайне, а также *Alliance Global Services*, которая специализируется на выпуске ПО и решений для автоматизированного тестирования. В связи с этим приобретением руководство *EPAM Systems* пересмотрело прогноз по выручке в сторону увеличения, ожидая её на уровне не ниже 905 млн долларов в 2015 году против 730 млн годом ранее.

В 2016 году *EPAM* поглотила китайскую компанию *Dextrys*, со штатом в 1400 сотрудников

1.3 Заказчики компании и предоставляемые услуги

EPAM работает с крупнейшими международными заказчиками, что позволяет приобрести опыт разработки и внедрения передовых решений для знаковых компаний глобального масштаба, получить уникальные знания об устройстве бизнеса ведущих представителей мировой экономики.

EPAM обладает обширным списком престижных клиентов по всему миру, среди которых многие члены списка *Fortune 500* и ведущие компании-разработчики программного обеспечения. Основными заказчиками компании являются: *SAP, Microsoft, Oracle, Thomson Reuters, The Coca-Cola Company*, Лондонская фондовая биржа, ММВБ, *Viacom/MTV Networks, Wolters Kluwer*, «Ренессанс-Капитал», Американский Красный Крест, *Clarins Group* и многие другие.

К основным партнерам фирмы можно отнести *SAP, Microsoft, Oracle, Thomson Reuters, Salesforce, IBM, Datalex, Adobe, Dell, ADAM, Amazon Web Services*.

EPAM предоставляет широкий спектр услуг и продуктов: ИТ-аутсорсинг, разработка программных продуктов, разработка приложений, создание платформ для корпоративных приложений, тестирование приложений, поддержка и сопровождение приложений, *ASM* и инфраструктурные сервисы, *Business Intelligence*, *SAP*-технологии, электронная коммерция, порталы и контент-менеджмент, облачные технологии, мобильные решения, разработка цифровых стратегий, дизайн и проектирование, разработка технологических решений, бизнес-анализ, встроенные решения.

Сегодня *EPAM* объединяет более 13 200 высококвалифицированных ИТ-профессионалов: программистов, руководителей проектов и бизнес-аналитиков, специалистов по обеспечению качества программных продуктов, архитекторов программного обеспечения, переводчиков и дизайнеров.

Компания располагает всеми возможностями для предоставления интересной, перспективной и стабильной работы как опытным профессионалам, так и начинающим специалистам.

1.4 Работа в компании

Сегодня *EPAM* объединяет более 13 200 высококвалифицированных ИТ-профессионалов: программистов, руководителей проектов и бизнес-аналитиков, специалистов по обеспечению качества программных продуктов, архитекторов программного обеспечения, переводчиков и дизайнеров.

Компания располагает всеми возможностями для предоставления интересной, перспективной и стабильной работы как опытным профессионалам, так и начинающим специалистам.

В *EPAM* созданы условия для возможности карьерного роста в направлениях технической экспертизы, управления проектами, построения архитектуры сложных систем, консалтинга и управления командами и отделами разработки, тестирования и поддержки. В процессе работы практикуются командировки к крупным зарубежным заказчикам.

В компании успешно реализуется *Career Development Program* – последовательный процесс обучения молодых специалистов, позволяющий закрепить теорию практикой в предметной области. Участие в *CDP* создает возможность для ускоренного приобретения необходимого опыта и построения успешной карьеры.

1.5 Используемое оборудование

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

- а) персональный компьютер на базе процессора *AMD Athlon II X4*;
- б) монитор 24” (по желанию работника могут быть установлены дополнительные мониторы);

с) 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.

1.6 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.

1.7 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.

1.8 Используемые методологии разработки программного обеспечения

Бизнес компаний часто зависит от качества работы информационных систем. Ошибки в ИТ-приложениях, некорректное выполнение функций могут привести к финансовым потерям. Для снижения подобных рисков *EPAM Systems* использует при разработке программного обеспечения методологию *Continuous Integration* (CI, непрерывная интеграция).

Continuous Integration – это полностью автоматизированная практика создания, сборки и тестирования программного кода. При ее применении компиляция и компоновка кода начинает проводиться как можно раньше и повторяется как можно чаще. Специальное программное обеспечение отслеживает процесс разработки: при наличии изменений в коде (например, добавилась новая часть) автоматически запускается процесс сборки и тестирования. Это позволяет найти дефекты и противоречия в компонентах системы еще на ранних стадиях ее создания. В результате – обеспечивается высокое качество программного обеспечения.

Преимущества *Continuous Integration*:

- сокращение ручных операций - этапы создания, сборки и тестирования программного обеспечения проводятся в автоматическом режиме;
- наличие рабочей ИТ-системы на всем протяжении процесса разработки – у проектной команды всегда есть свежая версия решения для демонстрации заказчику, получения обратной связи и быстрой доработки;
- качество программного обеспечения – в рамках *Continuous Integration* используются различные программные средства для контроля качества кода, что позволяет сократить количество ошибок;
- минимизация рисков – дефекты выявляются на ранних стадиях разработки информационной системы, что помогает избежать увеличения сроков и стоимости проекта;
- окупаемость инвестиций в ИТ – автоматизация процесса разработки обеспечивает высокую эффективность и надежность информационной системы.

Для разработки программного обеспечения в рамках *Continuous Integration* в *EPAM* используются различные программные и аппаратные средства. Выбор конкретных инструментов зависит от задач и специфики проекта.

- развертывание и подготовка к работе: *VMWare, Microsoft Hyper-V, Citrix Xen, Parallels*;
- средства разработки: *Eclipse, MS Visual Studio, Borland Delphi* и т.д;
- системы контроля версий: *StarTeam, Perforce, CVS, PVCS, VSS, Synergy, Subversion, GIT* и т.д;
- среды и системы автоматической сборки: *CruiseControl, CC.net, TeamCity, Build Forge, Team Foundation Server, Electric Commander*;

- компиляторы и средства сборки: *Compilers, Linkers, Ant, Make, NAnt MSBuild, Maven*;
- контроль качества: *Mercury Quality Center, LoadRunner, TestDirector, WinRunner, Xunit, Clover, IBM Functional, Performance & Manual Tester*.

Процесс *Continuous Integration* может быть полностью – от написания кода до запуска готового решения в эксплуатацию – выстроен как облачный сервис. Требуемое качество программного обеспечения и соблюдение сроков разработки и поддержки ИТ-систем обеспечиваются с помощью заключения *SLA (Service Level Agreement)* – соглашения об уровне услуг) между заказчиком и EPAM.

1.9 Используемые методы управления жизненным циклом проекта

Для управления жизненным циклом разрабатываемых проектов в компании «*Epam Systems*» используется *Subversion*, который очень эффективно и хорошо реализуется на базе *BitBucket*.

Subversion – (также известная как «*SVN*») – свободная централизованная система управления версиями, официально выпущенная в 2004 году компанией *CollabNet Inc.*

Цель проекта – заменить собой распространенную на тот момент систему *Concurrent Versions System (CVS)*, которая ныне считается устаревшей. *Subversion* реализует все основные функции *CVS* и свободна от ряда недостатков последней.

В настоящее время *Subversion* используется многими сообществами разработчиков открытого программного обеспечения (в том числе сообществами, ранее использовавшими *CVS*). В их числе такие известные проекты, как *Apache, GCC, FreePascal, Python, Ruby, FreeBSD, AROS, Blender, Boost, Tor, OGRE*. *Subversion* также широко используется в закрытых проектах и корпоративной сфере. Хостинг *Subversion*, в том числе для проектов с открытым кодом, также предоставляют популярные хостинг-проекты *SourceForge.net, Tigris.org, Google Code* и *BountySource*.

BitBucket («ведро битов») – веб-сервис для хостинга проектов и их совместной разработки, основанный на системе контроля версий *Mercurial* и *Git*. По назначению и предлагаемым функциям аналогичен *GitHub* (однако *GitHub* не предоставляет бесплатные «закрытые» репозитории, в отличие от *Bitbucket*), который поддерживает *Git* и *Subversion*.

Git – распределённая система управления версиями файлов. Проект был создан Линусом Торвалдсом для управления разработкой ядра *Linux*, первая версия выпущена 7 апреля 2005 года.

Ядро *Git* представляет собой набор утилит командной строки с параметрами. Все настройки хранятся в текстовых файлах конфигурации. Такая реализация делает *Git* легко портируемым на любую платформу и даёт возможность легко интегрировать *Git* в другие системы (в частности, создавать графические *git*-клиенты с любым желаемым интерфейсом).

Репозиторий *Git* представляет собой каталог файловой системы, в котором находятся файлы конфигурации репозитория, файлы журналов, хранящие операции, выполняемые над репозиторием, индекс, описывающий расположение файлов и хранилище, содержащее собственно файлы. Структура хранилища файлов не отражает реальную структуру хранящегося в репозитории файлового дерева, она ориентирована на повышение скорости выполнения операций с репозиторием. Когда ядро обрабатывает команду изменения (неважно, при локальных изменениях или при получении патча от другого узла), оно создаёт в хранилище новые файлы, соответствующие новым состояниям изменённых файлов. Существенно, что никакие операции не изменяют содержимого уже существующих в хранилище файлов.

По умолчанию репозиторий хранится в подкаталоге с названием «*.git*» в корневом каталоге рабочей копии дерева файлов, хранящегося в репозитории. Любое файловое дерево в системе можно превратить в репозиторий *git*, отдав команду создания репозитория из корневого каталога этого дерева (или указав корневой каталог в параметрах программы). Репозиторий может быть импортирован с другого узла, доступного по сети. При импорте нового репозитория автоматически создаётся рабочая копия, соответствующая последнему зафиксированному состоянию импортируемого репозитория (то есть не копируются изменения в рабочей копии исходного узла, для которых на том узле не была выполнена команда *commit*).

Репозиторий *git* бывает локальный и удаленный. Локальный репозиторий – это поддиректория *.git*, создается (в пустом виде) командой *git init* и (в непустом виде с немедленным копированием содержимого родительского удаленного репозитория и простановкой ссылки на родителя) командой *git clone*.

Практически все обычные операции с системой контроля версий, такие, как коммит и слияние, производятся только с локальным репозиторием. Удаленный репозиторий можно только синхронизировать с локальным как «вверх» (*push*), так и «вниз» (*pull*).

Наличие полностью всего репозитория проекта локально у каждого разработчика дает *git* ряд преимуществ перед *SVN*. Так, например, все операции, кроме *push* и *pull*, можно осуществлять без наличия Интернет-соединения.

Очень мощной возможностью *git* являются ветви, реализованные куда более полно, чем в *SVN*. Создать новую ветвь так же просто, как и совершить коммит. По сути, ветвь *git* есть не более чем читаемое человеком имя, «навешенное» на некий коммит в репозитории (используется поддиректория *refs*). Коммит без создания новой ветви всего лишь передвигает эту ссылку на себя, а коммит с созданием ветви – оставляет старую ссылку на месте, но создает новую на новый коммит, и объявляет её текущей. Заменить локальные девелоперские файлы на набор файлов из иной ветви, тем самым перейдя к работе с ней – также тривиально.

2 ИНФОРМАЦИОННОЕ ОБЕСПЕЧЕНИЕ

2.1 Жизненный цикл программного продукта

Методология проектирования информационных систем (ИС) описывает процесс создания и сопровождения систем в виде жизненного цикла информационной системы, представляя его как некоторую последовательность стадий и выполняемых на них процессов. Для каждого этапа определяются состав и последовательность выполняемых работ, получаемые результаты, методы и средства, необходимые для выполнения работ, роли и ответственность участников и т.д. Такое формальное описание жизненного цикла информационной системы позволяет спланировать и организовать процесс коллективной разработки и обеспечить управление этим процессом.

Жизненный цикл (ЖЦ) информационной системы можно представить как ряд событий, происходящих с системой в процессе ее создания и использования.

Модель жизненного цикла отражает различные состояния системы, начиная с момента возникновения необходимости в данной информационной системе и заканчивая моментом ее полного выхода из употребления. Модель жизненного цикла – это структура, содержащая процессы, действия и задачи, которые осуществляются в ходе разработки, функционирования и сопровождения программного продукта в течение всей жизни системы, от определения требований до завершения ее использования.

В зависимости от потребностей проекта выбирается соответствующий подход к циклу разработки. В процессе создания программного обеспечения (ПО) используются разные виды жизненных циклов.

Каскадная стратегия (однократный проход, водопадная или классическая модель) подразумевает линейную последовательность выполнения стадий создания информационной системы (рис. 2.1). Другими словами, переход с одной стадии на следующую происходит только после того, как будет полностью завершена работа на текущей. Каждый этап завершается выпуском полного комплекта документации, достаточной для того, чтобы разработка могла быть продолжена командой специалистов на следующем этапе.



Рисунок 2.1 – Каскадная схема разработки ПО

К положительным сторонам применения каскадного подхода можно отнести то, что на каждом этапе формируется законченный набор проектной документации, отвечающий критериям полноты и согласованности. Выполняемые в логичной последовательности этапы работ позволяют планировать сроки завершения всех работ и соответствующие затраты.

Каскадный подход хорошо зарекомендовал себя при построении информационных систем, для которых в самом начале разработки можно достаточно точно и полно сформулировать все требования, с тем чтобы предоставить разработчикам свободу реализовать их как можно лучше с технической точки зрения. В эту категорию попадают сложные расчетные системы, системы реального времени и другие подобные задачи. Однако, в процессе использования этого подхода обнаружился ряд его недостатков, вызванных прежде всего тем, что реальный процесс создания ПО полностью не укладывался в такую жесткую схему.

В процессе создания ПО постоянно возникала потребность в возврате к предыдущим этапам и уточнении или пересмотре ранее принятых решений. В результате реальный процесс создания ПО принимал вид, представленный на рисунке 2.2.

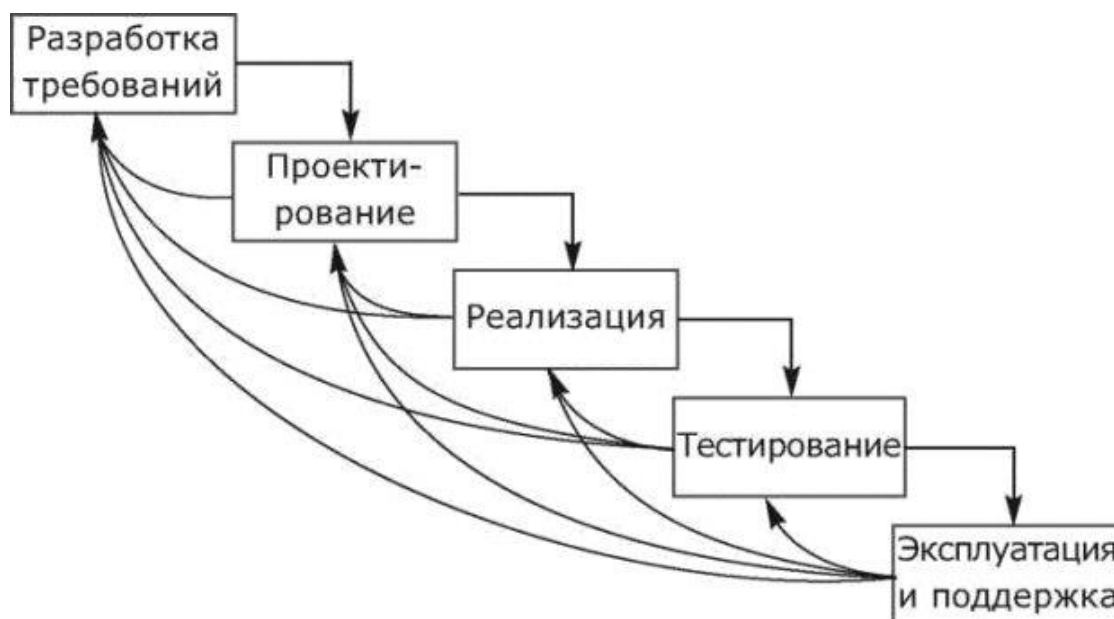


Рисунок 2.2 – Классическая итерационная модель жизненного цикла программного обеспечения

Так выглядит поэтапная модель с промежуточным контролем, где разработка ИС ведется итерациями с циклами обратной связи между этапами. Межэтапные корректировки позволяют учитывать реально существующее взаимовлияние результатов разработки на различных этапах. Время жизни каждого из этапов растягивается на весь период разработки.

Основным недостатком такого подхода является существенное запаздывание с получением результатов. Согласование результатов с пользователями производится только в точках, планируемых после завершения каждого этапа работ, требования к ИС сохраняются в виде технического задания на все время ее создания. Таким образом, пользователи могут внести свои замечания только после того, как работа над системой будет полностью завершена. В случае неточного изложения требований или их изменения в течение длительного периода создания ПО, пользователи получают систему, не удовлетворяющую их потребностям. Модели автоматизируемого объекта могут устареть одновременно с их утверждением.

Для преодоления перечисленных проблем была предложена спиральная модель ЖЦ, изображенная на рисунке 2.3. На этапах анализа и проектирования реализуемость технических решений и степень удовлетворения потребностей заказчика проверяется путем создания прототипов. Каждый виток спирали соответствует созданию работоспособного фрагмента или версии системы. Это позволяет уточнить требования, цели и характеристики проекта, определить качество разработки, спланировать работы следующего витка спирали. Таким образом углубляются и последовательно конкретизируются детали проекта и в результате выбирается обоснованный вариант, который удовлетворяет действительным требованиям заказчика и доводится до реализации.



Рисунок 2.3 – Спиральная модель ЖЦ ИС

Итеративная разработка отражает объективно существующий спиральный цикл создания сложных систем. Она позволяет переходить на следующий этап, не дожидаясь полного завершения работы на текущем и решить главную задачу – как можно быстрее показать пользователям системы работоспособный продукт, тем самым активизируя процесс уточнения и дополнения требований.

Основная проблема спирального цикла – определение момента перехода на следующий этап. Для ее решения вводятся временные ограничения на каждый из этапов жизненного цикла, и переход осуществляется в соответствии с планом, даже если не вся запланированная работа закончена. Планирование производится на основе статистических данных, полученных в предыдущих проектах, и личного опыта разработчиков.

Одной из самых современных и актуальных методологий разработки является разновидность итерационного подхода – гибкая методология разработки (англ. *Agile software development*, *agile-методы*).

Agile – это серия подходов к разработке программного обеспечения, ориентированных на использование интерактивной разработки, динамическое формирование требований и обеспечение их реализации в результате постоянного взаимодействия внутри самоорганизующихся рабочих групп, состоящих из специалистов различного профиля. Существует несколько методик, относящихся к классу гибких методологий разработки, в частности экстремальное программирование, *DSDM*, *Scrum*, *FDD*.

Большинство гибких методологий нацелены на минимизацию рисков путём сведения разработки к серии коротких циклов, называемых итерациями,

которые обычно длятся две-три недели. Каждая итерация сама по себе выглядит как программный проект в миниатюре и включает все задачи, необходимые для выдачи мини-прироста по функциональности: планирование, анализ требований, проектирование, программирование, тестирование и документирование. Хотя отдельная итерация, как правило, недостаточна для выпуска новой версии продукта, подразумевается, что гибкий программный проект готов к выпуску в конце каждой итерации. По окончании каждой итерации команда выполняет переоценку приоритетов разработки.

Методы *Agile* – это такие гибкие методологии, как *Lean Development* («Бережливая разработка ПО»), *Scrum* и др. Они были разработаны еще в начале 2000-х как альтернатива малоэффективным традиционным *IT* методам.

Практически все *agile*-команды сконцентрированы в одном офисе (*bullpen*). Офис включает *product owner* – заказчика, который и определяет требования к продукту. В качестве заказчика может выступать бизнес-аналитик, менеджер проекта или клиент. Кроме того, в офис могут входить и дизайнеры интерфейса, тестировщики, технические писатели. То есть методы *Agile* направлены в первую очередь на непосредственное общение.

Основной метрикой *agile*-методов является рабочий продукт. Отдавая предпочтение непосредственному общению, *agile*-методы уменьшают объём письменной документации по сравнению с другими методами.

Основные идеи:

- люди и взаимодействие важнее процессов и инструментов;
- работающий продукт важнее исчерпывающей документации;
- сотрудничество с заказчиком важнее согласования условий контракта;
- готовность к изменениям важнее следования первоначальному плану.

Принципы *Agile*:

- а) удовлетворение клиента за счёт ранней и бесперебойной поставки ценного программного обеспечения;
- б) приветствие изменений требований даже в конце разработки (это может повысить конкурентоспособность полученного продукта);
- в) частая поставка рабочего программного обеспечения (каждый месяц или неделю или ещё чаще);
- г) тесное, ежедневное общение заказчика с разработчиками на протяжении всего проекта;
- д) проектом занимаются мотивированные личности, которые обеспечены нужными условиями работы, поддержкой и доверием;
- е) рекомендуемый метод передачи информации – личный разговор (лицом к лицу);
- ж) работающее программное обеспечение – лучший измеритель прогресса;

- з) постоянное внимание улучшению технического мастерства и удобному дизайну;
- и) простота – искусство не делать лишней работы;
- к) лучшие технические требования, дизайн и архитектура получаются у самоорганизованной команды;
- л) постоянная адаптация к изменяющимся обстоятельствам.

Главные преимущества *Agile*:

- качество продукта. Вовлечение заказчика в процесс каждой итерации дает возможность корректировать процесс, что неизменно повышает качество;
- высокая скорость разработки. Итерация длится не более 3-х недель, к концу этого срока обязательно есть результат;
- минимизация рисков. Крупный проект дает возможность заказчику оплатить несколько итераций и в ходе работы понять, что он вовремя получит именно то, что хочет и за приемлемую цену. Водопадные модели (с применением спецификаций и технических заданий) таких возможностей не дают;
- заказчик всегда имеет возможность наблюдать за ходом разработки, корректировать функциональность проекта, тестировать или запускать его, даже может остановить его в любой момент.

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 отправляет результат выполнения и/или выполняет команду. Следовательно, общение в чат-ботах осуществляется путём 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.

Так, британская компьютерная программа *George* победила в *Loebner Prize* – конкурсе чат-ботов на наилучшее соответствие языку человека, который проводят ежегодно.

SkypeTalk ведет диалог на украинском языке. Возможности и знания можно расширить, поскольку основная часть написана на специальном диалекте *XML*, что достаточно гибко позволяет переписывать бот под свои потребности и обновлять базу. Версию *XML*, используемую в *A.L.I.C.E.*, распространяется под лицензией *GNU GPL*, поэтому существует много свободных реализаций этого языка, а также программ, использующих этот язык (интерпретаторов). Наиболее активно используемая версия интерпретатора, написанная на языке *Java*. Существует также формальная спецификация языка и *XML*-схема, данная консорциумом *W3C*.

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:

- а) `<pattern> ПРИВЕТ </pattern>`
- б) `<pattern> ПРИВЕТ * </pattern>`
- в) `<pattern> * ПРИВЕТ * </pattern>`
- г) `<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 тегам `` ``.

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?»

2.5 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* 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 *C*.

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.

2.5.1 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*.

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 [6].

2.5.2 *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 выполняться 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].

3 РАЗРАБОТКА ПРОГРАММНЫХ СРЕДСТВ ДЛЯ РЕШЕНИЯ ЗАДАЧИ

3.1 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.

3.2 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.

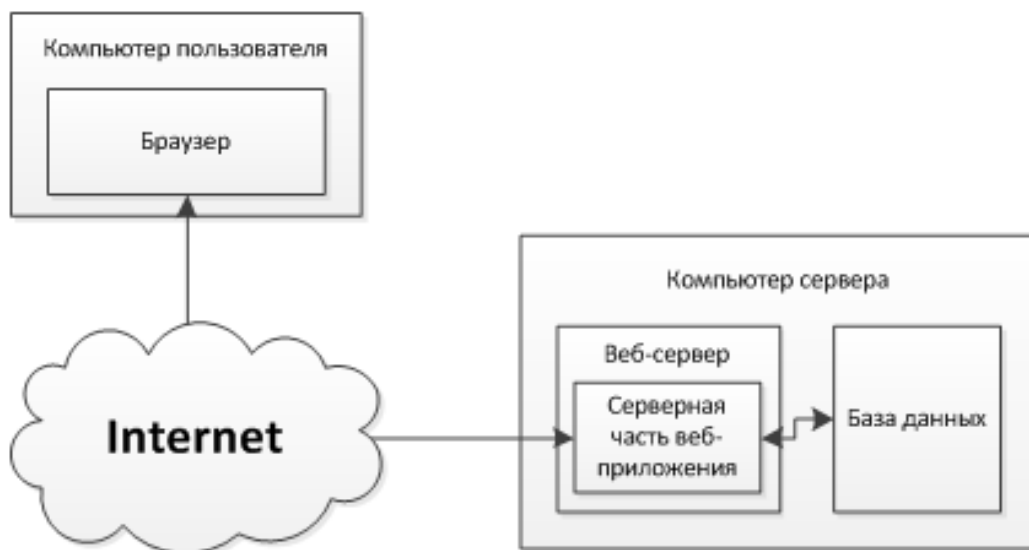


Рисунок 3.1 – Общая структура веб-приложения

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 отправляет клиенту по сети. Браузер тут же отображает полученный в виде очередной веб-страницы результат [9].

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 текстовым 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.

```

<State id="1">
  <message>Сообщение</message>
  <keywords>
    <keyword target="1">Ключевое слово 1</keyword>
    ...
    <keyword target="N">Ключевое слово N</keyword>
  </keywords>
</State>

```

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.

3.3 Описание разработанного 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.

3.3.1 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.

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 3.3 отображена схема фреймворка.

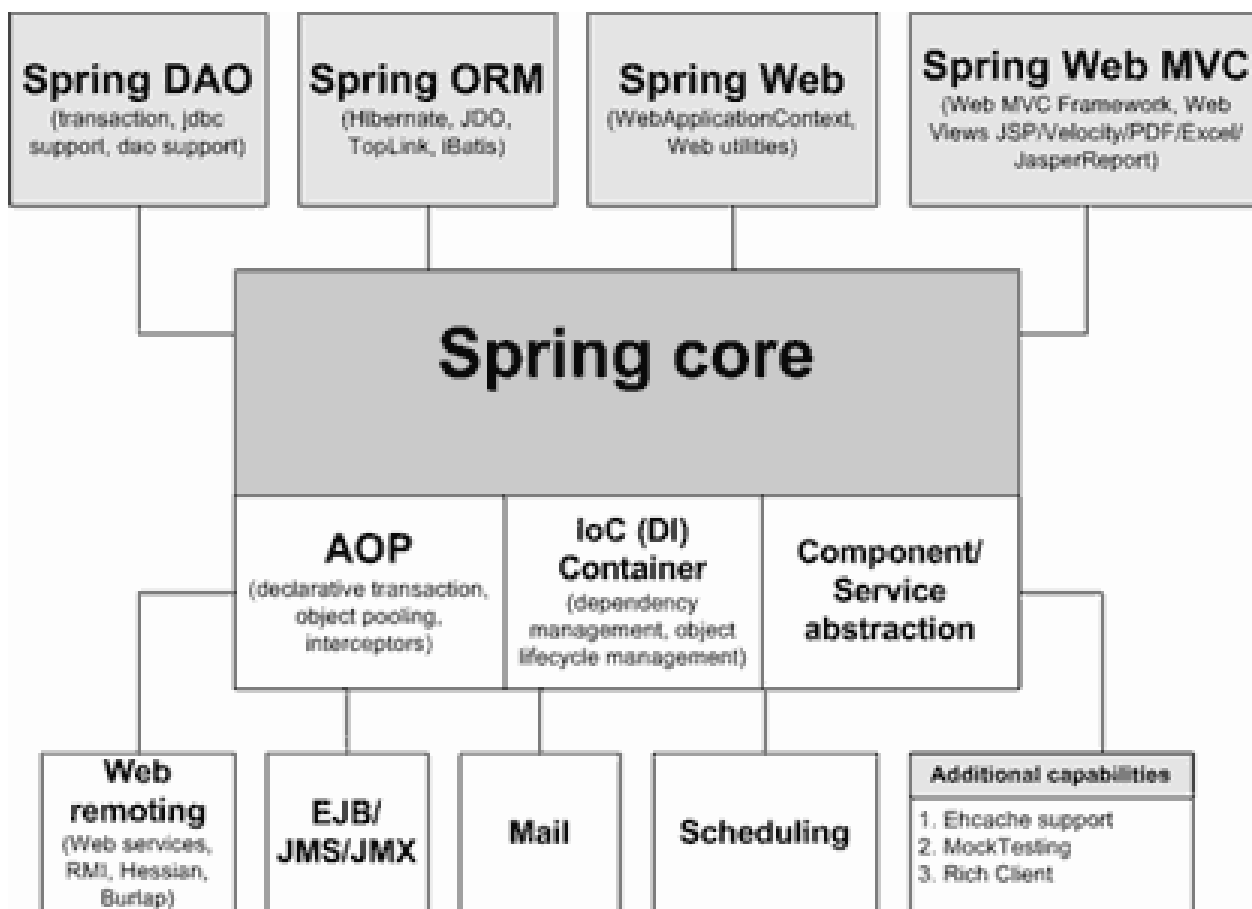


Рисунок 3.3 – Структура *Spring Framework*

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 А и В. Причём класс А зависит от класса В (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 В, которые нужны классу А, и быть реализованным с помощью класса В. 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.

3.3.2 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. [10].

Общую схему взаимодействия упрощенно представлена на рисунке 3.4.

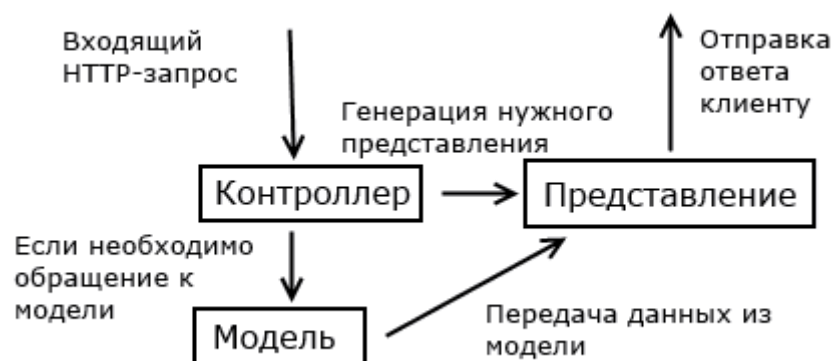


Рисунок 3.4 – Структура паттерна *MVC*

3.3.3 В разрабатываемом приложении 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.

3.3.4 *Telegram* – бесплатный кроссплатформенный мессенджер для смартфонов и других устройств, позволяющий обмениваться текстовыми сообщениями и медиафайлами различных форматов. Используются проприетарная серверная часть с 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.

3.3.5 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:

- а) русский – словарь содержит 3.6 миллиона словоформ (разработка компании «Информатик»);
- б) украинский – словарь содержит 1.8 миллиона словоформ (разработка компании «Информатик»);
- в) 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*:

- а) *XML*- и *SOAP*-интерфейс <http://speller.yandex.net/services/spellservice>;
- б) *WSDL*-документ для *SOAP* доступен по адресу <http://speller.yandex.net/services/spellservice?WSDL>;
- в) *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*.

3.3.6 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 зависимости 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 с 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.

3.4 Описание основной структуры приложения

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

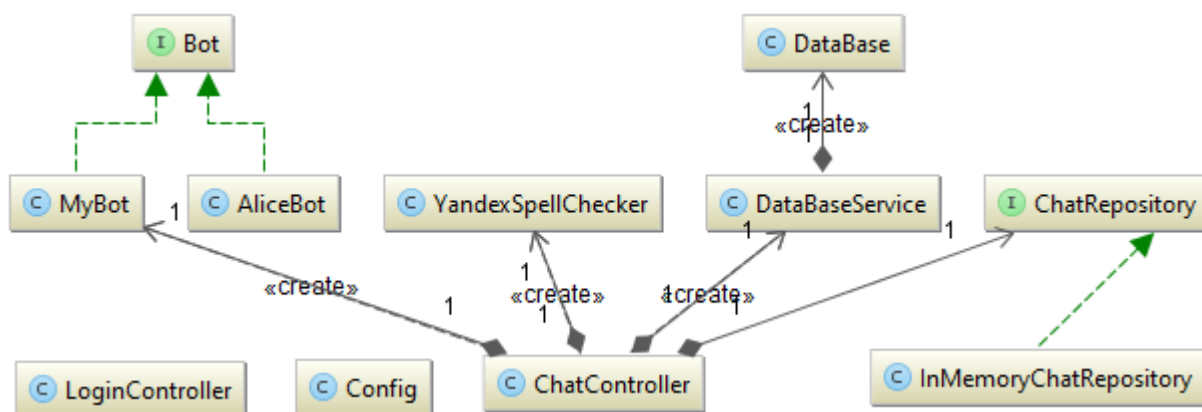


Рисунок 3.5 – *Java* классы приложения

Отображенные на рисунке выше классы, отвечают за работу приложения на сервере. Подробная структура основных классов изображена на рисунках 3.6 – 3.10. *Java* код классов находится в приложении А.

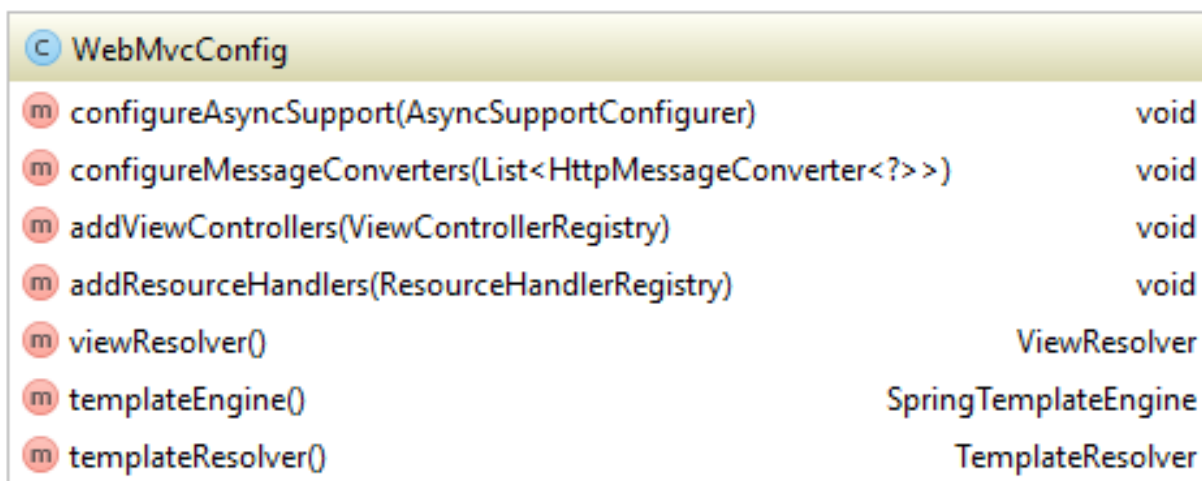


Рисунок 3.6 – Методы класса *WebMvcConfig*

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

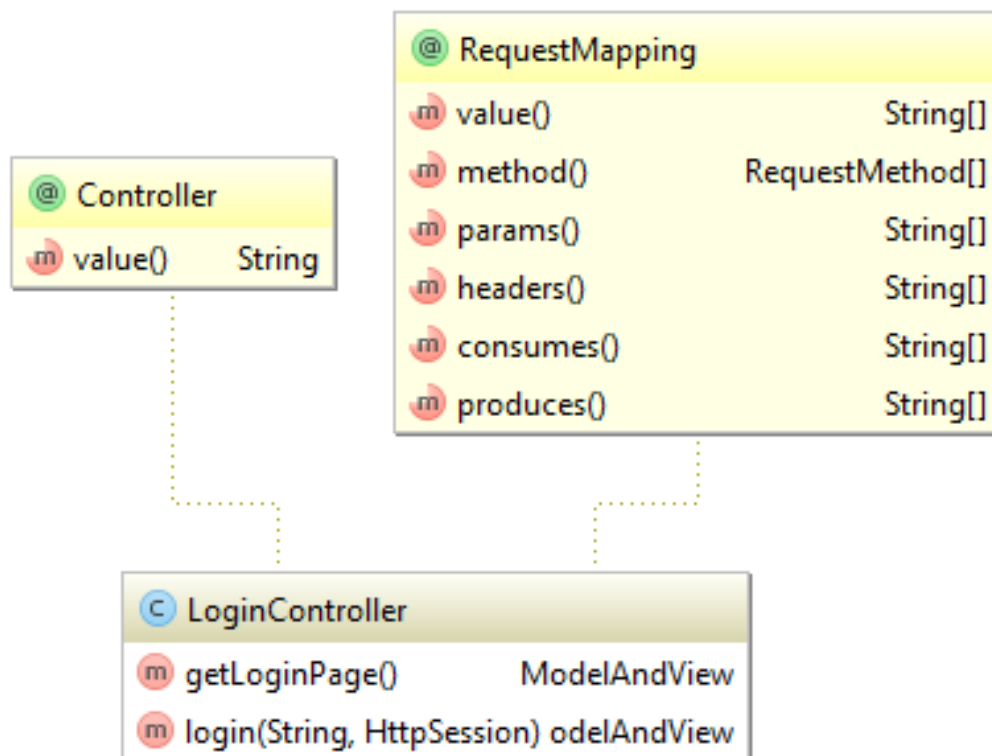


Рисунок 3.7 – Зависимости и методы класса *LoginController*

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.

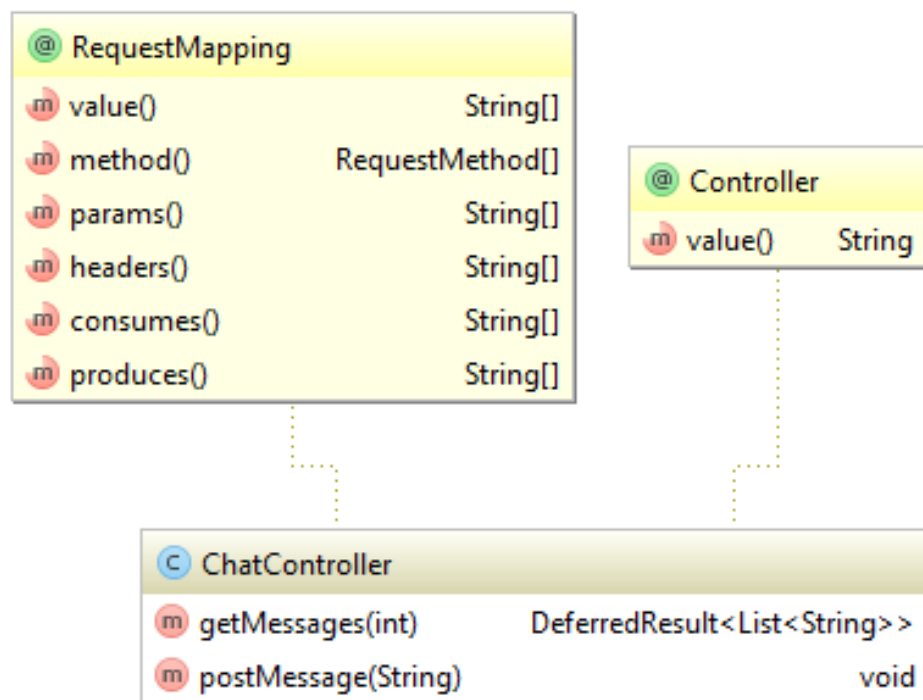


Рисунок 3.8 – Зависимости и методы класса *ChatController*

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

Метод *postMessage* получает вопрос пользователя. Для проверки и исправления ошибок в введенном пользователем сообщении, оно передается в метод *check* класса *YandexSpellChecker*. Затем корректное сообщение отправляется в класс *Bot* для получения ответа. Полученный ответ передается методу *fillResponse* класса *DataBaseService* для дополнения ответа информацией из базы данных. Вернувшаяся строка добавляется в ответ и ответ отправляет пользователю.

C DataParser	
f dom	Document
f states	HashMap<String, State>
f invalidMessages	ArrayList<String>
f invalidMessageIndex	int
f stateCounter	int
m DataParser()	
m loadStates()	void
m getState(String)	State
m addState(State)	void
m getKeywords(Element)	ArrayList
m loadConfiguration()	void
P invalidAnswer	String

Рисунок 3.9 – Методы класса *Parser*

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

C Bot	
f name	String
f dictionary	HashMap<String, String>
f level	String
f parser	DataParser
m Bot(String, DataParser)	
m send(String)	String
m parse(String, ArrayList<Keyword>)	Keyword
m getMatches(String, Keyword)	int
m replaceMatches(String)	String
P message	String

Рисунок 3.10 – Методы класса *Bot*

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

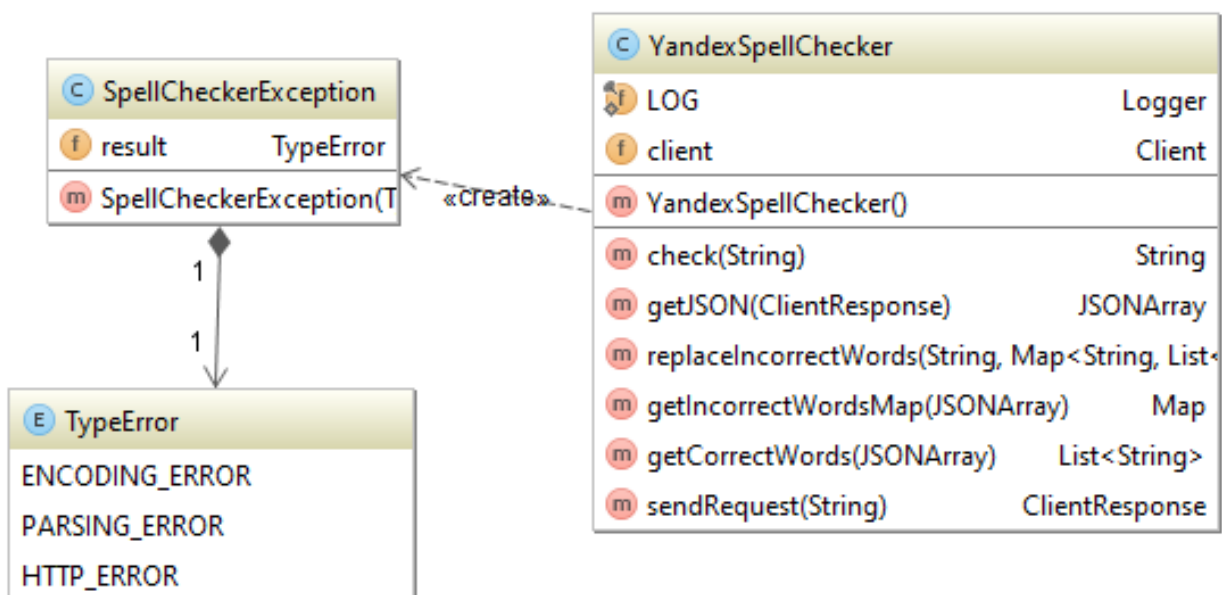


Рисунок 3.11 – Методы класса *YandexSpellerCheck*

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

Метод *sendRequest* получает строку и отвечает за формирование *http* соединения 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 сообщение пользователя, вызовом методов описанных выше, проверяет это сообщение и исправляет ошибки.

C DataBase	
DATABASE_DRIVER	String
DATABASE_URL	String
USERNAME	String
PASSWORD	String
MAX_POOL	String
connection	Connection
properties	Properties
m DataBase()	
m getProperties()	Properties
m connect()	Connection
m disconnect()	void
m sentQuery(String)	List<String>
m finalize()	void

Рисунок 3.12 – Методы класса *DataBase*

Класс *DataBase* отвечает за работу с базой данных. Содержит методы установление соединения с базой, закрытие соединения и отправку запроса к базе и получения.

C DataBaseService	
startKey	String
finishKey	String
dataBase	DataBase
m formatData(String)	String
m fillResponse(String)	String
m replaceKey(String)	String
m isNeedFillFromDB(String)	boolean
m getKey(String)	String

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 инъекции 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* – 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*.

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 3.14.

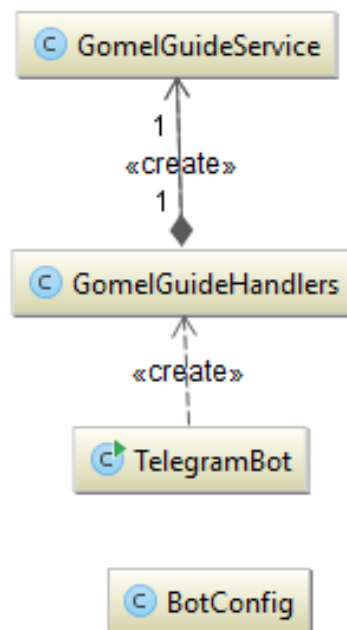


Рисунок 3.14 – Структура классов *TelegramBot*

Класс *GomeIGuideHandlers* 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 отправляет.

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 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 3.11.



Рисунок 3.11 – Стартовая страница приложения

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

Chat

You're chatting as **Anna**

[bot]Привет, я Bot. Я могу тебе помочь?
[Anna] Где я могу отдохнуть?
[bot]В Гомеле есть много красивейших парков: Центральный, Фестивальный, Любенский, Молодёжный. Какое место тебе интересно?
[Anna] центральный
[bot]Это один из красивейших пейзажных парков - дворцово-парковый комплекс в Гомеле является одной из главных достопримечательностей и настоящей жемчужиной белорусской земли. Расположен на обрывистом берегу реки Сож и оврага с ручьем Гомий.
[Anna] как я могу туда добраться?
[bot]общественным транспортом: автобусы № 1, 4, 11, 12, 20, 30; троллейбусы № 1, 5, 7, 10, 15, 16, 22 до остановки «Площадь Ленина». От ж/д вокзала до Дворцово-паркового ансамбля всего две остановки можно и пешком по проспекту Ленина.
автомобилем: координаты GPS: 52.422133,31.016917

Рисунок 3.12 – 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 приложением достаточно закрыть окно браузера.

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 3.13.

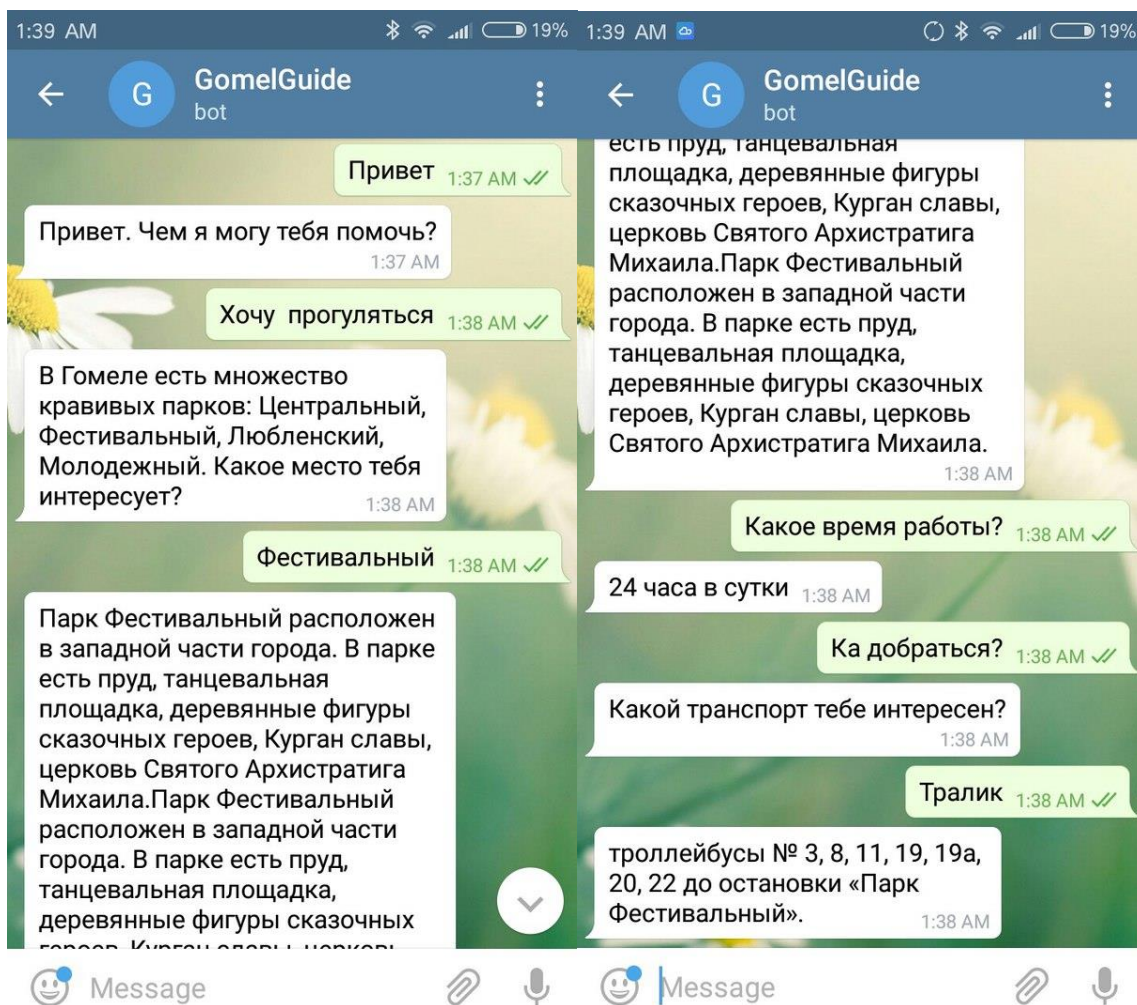


Рисунок 3.13 – Окно диалога в *Telegram*

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. : введ. 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. Хабибулин, И.Ш. Самоучитель Java / И.Ш. Хабибулин – СПб.: БХВ-Петербург, 2001. – 464с
9. 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.
10. Паттерны проектирования. Паттерн MVC [Электронный ресурс]. – Режим доступа: <http://metanit.com/sharp/mvc5/1.1.php>. – Дата доступа: 15.02.2017.

ПРИЛОЖЕНИЕ А
(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.updatehandlers.*;

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.updatehandlers;

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 GomeIGuideHandlers extends TelegramLongPollingBot {
    private static final String LOGTAG = "GOMELGUIDEHANDLERS";

    @Override
    public String getBotToken() {
        return BotConfig.GOMELGUIDE_TOKEN;
    }

    @Override
    public String getBotUsername() {
        return BotConfig.GOMELGUIDE_USER;
    }

    @Override
    public void onUpdateReceived(Update update) {
        Message message = update.getMessage();
        if (message != null && message.hasText()) {
            sendMsg(message,
GomeIGuideService.getAnswer(message.getText()));
        }
    }

    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();
        }
    }

    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 {
        CloseableHttpClient httpclient =
HttpClientBuilder.create().build();
        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"));

        HttpResponse response =
httpClient.execute(httppost);
        HttpEntity entity = response.getEntity();

        if (entity != null) {
            InputStream instream = entity.getContent();
            try {
                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 {
            ClientResponse response = sendRequest(text);
            JSONArray array = getJSON(response);

```

```

        if(array.isEmpty()){
            return text;
        }
        Map incorrectWords = getIncorrectWordsMap(array);
        String correctedText = replaceIncorrectWords(text,
incorrectWords);
        return correctedText;
    } catch (SpellCheckerException e) {
        LOG.debug(e.getMessage());
        return text;
    }
}

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;
}

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;
}

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;
    }

    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;
    }

    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();
            }
        }
    }

    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;

```

```

import com.mysql.jdbc.StringUtils;

import java.util.List;

public class DataBaseService {

    private final static String startKey = "{";
    private final static String finishKey = "}";
    private static DataBase dataBase = new DataBase();

    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;
    }

    public static String replaceKey(String response) {
        final String key = getKey(response);
        return response.replace(startKey + key + finishKey,
formatData(key));
    }

    private static boolean isNeedFillFromDB(final String
response) {
        return response.contains(startKey) &&
response.contains(finishKey);
    }

    private static String getKey(final String response) {
        String keyword =
response.substring(response.indexOf(startKey) + 1,
response.indexOf(finishKey));
        return keyword;
    }
}

package by.gstu.chatbot.controllers;

```



```

@Controller
@RequestMapping("/chat")
public class ChatController {

    private final ChatRepository chatRepository;
    private static MyBot bot;
    private final Map<DeferredResult<List<String>>, Integer>
chatRequests =
        new ConcurrentHashMap<DeferredResult<List<String>>,
Integer>();
    YandexSpellChecker spellChecker = new YandexSpellChecker();

    @Autowired
    public ChatController(ChatRepository chatRepository) {
        this.chatRepository = chatRepository;
        DataParser dp = new DataParser();
        bot = new MyBot("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,
HttpServletResponse rsp) {
        this.chatRepository.addMessage(message);
        String userMessage =

```

```

spellChecker.check(message.substring(message.indexOf('']') + 1));
    String response =
DataBaseService.fillResponse(bot.sendMessage());
    if (response.length() == 0) {
        response =
DataBaseService.fillResponse(bot.getMessage());
    }
    this.chatRepository.addMessage("[bot]" + response);

    for (Entry<DeferredResult<List<String>>, Integer> entry
: this.chatRequests.entrySet()) {
        List<String> messages =
this.chatRepository.getMessages(entry.getValue());
        entry.getKey().setResult(messages);
    }

    try {
        PrintWriter out = rsp.getWriter();
        out.println(URLEncoder.encode(response, "UTF-8"));
        out.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

@RequestMapping("/reload")
public void reload() {
    (bot.setParser(new DataParser()));
}
}

```