# ВВЕДЕНИЕ

Современные предприятия для сокращения расходов стремятся максимально ускорить и автоматизировать производство. Это выражается не только в воде новых техническо-аппаратных средств, но и в модернизации, инновации и повсеместного внедрения нового программного обеспечения. Фирмы и предприятия могут быть сколь угодно велики, в то время как область их деятельности может быть достаточно узконаправленна, что требует создания специализированного программного обеспечения. Создание такого рода ПО – достаточно дорогой процесс, при этом основную часть стоимости часто составляет оплата труда штата программистов. Как правило, в целях оптимизации расходов на создание ПО, на крупных предприятиях организуется штат квалифицированных работников, которые выполняют основную работу по созданию и внедрению необходимых программных средств.

*EPAM* *Systems* – крупнейший поставщик услуг в области разработки проектного (заказного) программного обеспечения и решений в Центральной и Восточной Европе. Созданная в 1993 году, сегодня компания имеет представительства в 17 странах мира, более 13 000 высококвалифицированных специалистов в штате, и продолжает активно расти. Реализовав тысячи сложных и масштабных решений для заказчиков по всему миру, *EPAM* *Systems* неизменно остается признанным лидером в таких областях, как:

* разработка, тестирование, сопровождение и поддержка заказного программного обеспечения и бизнес-приложений;
* 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.

***Место практики:*** ИООО «ЭПАМ Системз».

* *Цели практики:* изучить методы и технологии, применяемые при разработке реальных приложений корпоративного масштаба на языке программирования *Java*, спроектировать компьютерную модель работы пассажирского лифта, реализовать многопоточное приложение симулирующее пассажирские перевозки.

.

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

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

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

* 1. **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 северо-американском рынке, в числе которых канадская компания *Thoughtcorp* и крупный поставщик услуг по разработке цифровых стратегий и организации мультиканального 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 продуктв: 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:

1. персональный компьютер на базе процессора *AMD Athlon II X4*;
2. монитор 24” (по желанию работника могут быть установлены дополнительные мониторы);
3. клавиатура, мышь, наушники (выбирается работником исходя из его личных предпочтений).

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

## **1.6 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.

## **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* – соглашение 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 Торвальдсом 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*).

# ИНФОРМАЦИОННОЕ ОБЕСПЕЧЕНИЕ

**2.1 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 *С*.

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

Основные возможности:

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

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

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

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

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

**2.2 Кроссплатформенность *Java*-приложений**

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

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

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

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

Абстрагируясь от аппаратуры на уровне библиотек классов, программисты могут больше не заботиться о различиях в реализации программного интерфейса конкретных операционных систем. Это позволяет создавать по-настоящему мобильные приложения, не требующие при переносе на различные платформы перетрансляции и изменения исходного текста [6].

**2.3 Виртуальная машина *Java***

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

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

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

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

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

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

1. **РАЗРАБОТКА ПРОГРАММНЫХ СРЕДСТВ ДЛЯ РЕШЕНИЯ ЗАДАЧИ**

**3.1 Постановка задачи**

В процессе прохождения технологической практики бы ли поставлены следующие задачи:

* знакомство со структурой ИООО «Эпам Системз»;
* изучить методы и технологии разработки приложений корпоративного масштаба на языке программирования *Java*;
* проектирование компьютерной модели работы пассажирского лифта;
* разработка многопоточного приложения, симулирующего пассажирские перевозки.

**3.2 Особенности разработки приложения**

При написании приложения использовалась методология разработки через тестирование (англ. *test-driven development, TDD*). *TDD* — техника разработки программного обеспечения, которая основывается на повторении очень коротких циклов разработки: сначала пишется тест, покрывающий желаемое изменение, затем пишется код, который позволит пройти тест, и под конец проводится рефакторинг нового кода к соответствующим стандартам. Тест — это процедура, которая позволяет либо подтвердить, либо опровергнуть работоспособность кода. Когда программист проверяет работоспособность разработанного им кода, он выполняет тестирование вручную.

Создавая тесты до реализации кода, мы создаем модель предметной области в уме, управляем процессом разработки кода, и, наконец, обеспечиваем себя средствами для автоматической проверки корректности кода. В результате мы получаем более безопасный, структурированный, легко читаемый код, уменьшаем количество дефектов и т.д. Этот способ программирования полностью отличается от тех, к которым мы привыкли, и намного приятнее.

На рисунке 3.1 представлен цикл разработки через тестирование.

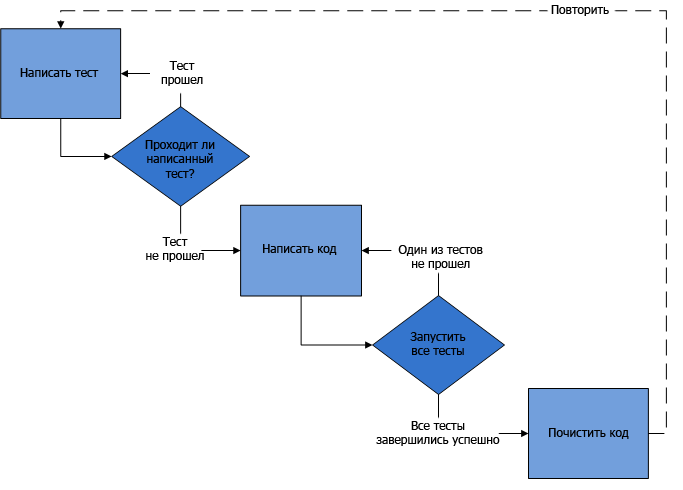


Рисунок 3.1 – Графическое представление цикла разработки, в виде блок-схемы

Подробное описание алгоритма разработки программного продукта.

Добавление теста. При разработке через тестирование, добавление каждой новой функциональности в программу начинается с написания теста. Неизбежно этот тест не будет проходить, поскольку соответствующий код ещё не написан. (Если же написанный тест прошёл, это означает, что-либо предложенная «новая» функциональность уже существует, либо тест имеет недостатки). Чтобы написать тест, разработчик должен чётко понимать предъявляемые к новой возможности требования. Для этого рассматриваются возможные сценарии использования и пользовательские истории. Новые требования могут также повлечь изменение существующих тестов. Это отличает разработку через тестирование от техник, когда тесты пишутся после того, как код уже написан: она заставляет разработчика сфокусироваться на требованиях до написания кода — тонкое, но важное отличие.

Запуск всех тестов, чтобы убедиться, что новые тесты не проходят. На этом этапе проверяют, что только что написанные тесты не проходят. Этот этап также проверяет сами тесты: написанный тест может проходить всегда и соответственно быть бесполезным. Новые тесты должны не проходить по объяснимым причинам. Это увеличит уверенность (хотя не будет гарантировать полностью), что тест действительно тестирует то, для чего он был разработан.

Написание кода. На этом этапе пишется новый код так, что тест будет проходить. Этот код не обязательно должен быть идеален. Допустимо, чтобы он проходил тест каким-то неэлегантным способом. Это приемлемо, поскольку последующие этапы улучшат и отполируют его. Важно писать код, предназначенный именно для прохождения теста. Не следует добавлять лишней и, соответственно, не тестируемой функциональности.

Запуск всех тестов, чтобы убедиться, что все тесты проходят. Если все тесты проходят, программист может быть уверен, что код удовлетворяет всем тестируемым требованиям. После этого можно приступить к заключительному этапу цикла.

Рефакторинг. Когда достигнута требуемая функциональность, на этом этапе код может быть почищен. Рефакторинг — процесс изменения внутренней структуры программы, не затрагивающий её внешнего поведения и имеющий целью облегчить понимание её работы, устранить дублирование кода, облегчить внесение изменений в ближайшем будущем.

Повторить цикл. Описанный цикл повторяется, реализуя всё новую и новую функциональность. Шаги следует делать небольшими, от 1 до 10 изменений между запусками тестов. Если новый код не удовлетворяет новым тестам или старые тесты перестают проходить, программист должен вернуться к отладке. При использовании сторонних библиотек не следует делать настолько небольшие изменения, которые буквально тестируют саму стороннюю библиотеку, а не код, её использующий, если только нет подозрений, что библиотека содержит ошибки.

Для написания тестов в разрабатываемом приложении использовалась библиотека *JUnit*. *JUnit* – это самый популярный в настоящее время инструмент модульного тестирования приложений в мире *Java*. Существует ряд других достаточно мощных инструментов, например, *TestNG*, но им пока не удалось достигнуть такой широкой популярности у разработчиков, как *JUnit*. С выходом четвертой версии, Кент Бэк и Эрик Гамма впервые за последние несколько лет представили значительные изменения программного интерфейса. В то время, когда свет увидел первый релиз проекта, в 2005 году, его было не так просто использовать в существующих проектах из-за отсутствия достаточной поддержки в инструментах для разработки приложений. На данный момент абсолютное большинство инструментов автоматической сборки приложений и средств разработки (*IDE*) включают полную поддержку *JUnit 4*.

При разработке приложения для упрощения процесса сборки проекта использовалась утилита *Ant.*

*Apache Ant* – утилита для автоматизации процесса сборки программного продукта. Является платформонезависимым аналогом утилиты *make*, где все команды записываются в XML-формате.

*Ant* был создан в рамках проекта *Jakarta*, сегодня — самостоятельный проект первого уровня *Apache Software Foundation*. Первая версия была разработана инженером *Sun Microsystems* Джеймсом Дэвидсоном.

*Ant*, в отличие от другого сборщика проектов, обеспечивает императивную, а не декларативную сборку проекта.

Утилита *Ant* полностью независима от платформы, требуется лишь наличие на применяемой системе установленной рабочей среды *Java* – *JRE*. Отказ от использования команд операционной системы и формат *XML* обеспечивают переносимость сценариев.

Управление процессом сборки происходит посредством *XML*-сценария, также называемого *Build*-файлом. В первую очередь этот файл содержит определение проекта, состоящего из отдельных целей (*Targets*). Цели сравнимы с процедурами в языках программирования и содержат вызовы команд-заданий (*Tasks*). Каждое задание представляет собой неделимую, атомарную команду, выполняющую некоторое элементарное действие.

Между целями могут быть определены зависимости – каждая цель выполняется только после того, как выполнены все цели, от которых она зависит (если они уже были выполнены ранее, повторного выполнения не производится).

Типичными примерами целей являются *clean* (удаление промежуточных файлов), *compile* (компиляция всех классов), *jar* (создать *jar* архив). Конкретный набор целей и их взаимосвязи зависят от специфики проекта.

* 1. **Структура программного комплекса**

Перед написание программного кода приложения, была спроектирована модель, описывающая выбранную предметную область, и отображающая будущие сущности, и их взаимосвязи. На рисунке 3.2 изображена логическая модель проектируемого приложения.

Для решения поставленной задачи была выбрана [среда разработки](http://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D0%B3%D1%80%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D1%80%D0%B5%D0%B4%D0%B0_%D1%80%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B8) *IntelliJ IDEA.* Этоинтегрированная среда разработки программного обеспечения на многих языках программирования, в частности *Java, JavaScript, Python,* разработанная компанией *JetBrains*. *IntelliJ IDEA* – первая *Java IDE* с широким набором интегрированных инструментов для рефакторинга, которые позволяют программистам быстро реорганизовывать исходные тексты программ. Данная среда разработки хорошо совместима со многими популярными свободными инструментами разработчиков, такими как *Apache* *Ant*, *Maven* и *JUnit*, а также в ней реализована интеграция с наиболее популярными системами управления версиями.

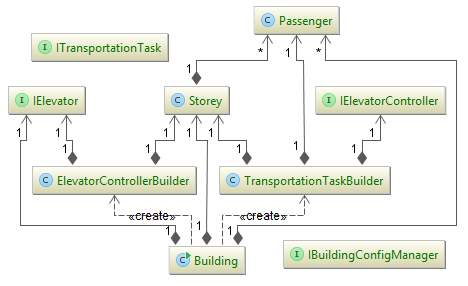


Рисунок 3.2 – Диаграмма зависимостей базовых сущностей модели

На данной диаграмме видно, что зависимости основаны на интерфейсах, что делает программу легко масштабируемой и убирает необходимость при изменении функциональности одного класса изменять работу других классов.

Исходные данные для работы приложения хранятся в файле «*config.properties»*. Значение свойства *storiesNumber* задает количество этажей в здании. Вместимость кабины лифта задается свойством – *elevatorCapacity*, а количество людей для перевозки – *passengersNumber*.

На рисунке 3.3 приведена файловая структура разработанного приложения.

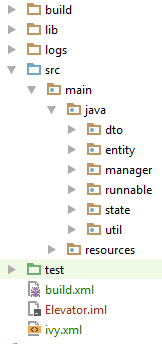


Рисунок 3.3 – Структура приложения.

В каталоге «*src.main.java»* содержатся исходный *java*-файлы. Все файлы классов помещены в каталоги, описывающие их предназначение. В папке «*resources*» хранятся конфигурационные файлы, необходимые для начала работы приложения. Каталог «*test*» содержит файлы *unit*-тестов, которые запускаются перед началом работы приложения.

* 1. **Реализация задачи на языке программирования**

В разработанной программе реализовано 27 классов. В классах реализованы различные методы, которые предназначенные для упрощения общих задач программирования. Описание основных классов и их методов приведено ниже.

Класс *Passenger*, описывает сущность пассажира. Содержит информацию о индивидуальном номере пассажира, номере этажа назначения и текущем состоянии, а также стандартные методы для их простора и редактирования.

Перечисление TransportationState, описывает состояние пассажира одним из значений: поток, описывающий перемещение пассажира еще не стартовал(*NOT\_STARTED*); пассажир в процессе перевозки(*IN\_PROGRESS*); пассажир достиг этажа назначения(*COMPLETED*), поток прерван в процессе перевозки(*ABORTED*).

Класс *ElevatorTest*, предназначен для тестирования класса *Elevator*. Элементы класса описаны в таблице 3.1.

Таблица 3.1 – Методы класса «*ElevatorTest*»

|  |  |
| --- | --- |
| Название метода | Описание |
| *addPassengerInElevatorContainer* | Метод для тестирования функции добавления пассажира в контейнер лифта. |
| *deletePassengerFromElevatorContainer* | Метод для тестирования функции удаления пассажира из контейнера лифта. |
| *checkIsNotFullElevatorContainer* | Метод тестирующий проверку на наполненность контейнера лифта. |
| *checkIsEmptyElevatorContainer* | Метод тестирующий проверку на пустоту контейнера лифта. |
| *moveElevatorOnStorey* | Метод для тестирования функции перемещения кабины лифта на заданный этаж. |
| *getNumberOfCurrentStorey* | Метод для тестирования функции возвращающей номер текущего этажа. |

Класс *StoreyTest*, предназначен для тестирования функциональности класса *Storey*. Элементы класса описаны в таблице 3.2.

Таблица 3.2 – Методы класса «*StoreyTest*»

|  |  |
| --- | --- |
| Название метода | Описание |
| *checkStoreyIsEmpty* | Тестирует правильность определения пустоты этажа. |
| *addNewPassengerInDispatchContainer* | Тестирует добавление пассажира в контейнер отправления этажа. |
| *addNewPassengerInArriveContainer* | Тестирует добавление пассажира в контейнер прибытия этажа. |
| *deletePassengerFromDispatchContainer* | Тестирует удаления пассажира из контейнера отправления. |

Класс *ElevatorControllerTest*, класс тестирующий функционал класса *ElevatorController*, выполняющегося в отдельном потоке. Элементы класса описаны в таблице 3.3.

Таблица 3.3 – Методы класса «*ElevatorControllerTest*»

|  |  |
| --- | --- |
| Названиеметода | Описание |
| *movingPassenger*  *FromDispatchContainerOfStorey*  *InElevatorContainer* | Тестирует перемещение пассажира из этажа оправления в кабину лифта. |
| *movingPassenger*  *FromElevatorContainer*  *InArrivalContainerOfStorey* | Тестирует перемещение пассажира из кабины лифта на этаж назначения. |
| *fallingAsleepThreadToWainingForElevator* | Проверяет что все потоки, ожидающие лифт, засыпают до его прибытия. |
| *fallingAsleepThreadToWainingForStorey* | Проверяет что все потоки, находящиеся в кабине лифта, засыпают до его прибытия на этаж назначения. |

Класс *Elevator*, описывает сущность лифта, содержит данные о вместимости кабины лифта, номере текущего этажа, списке пассажиров, находящихся в кабине лифта, а также объекты, предназначенные для синхронизации потоков. Для всех полей класса имеются стандартные методы, позволяющие просматривать и задавать их состояние. Методы, описывающие поведение объекта данного класса описаны в таблице 3.4.

Таблица 3.4 – Методы класса «*Elevator*»

|  |  |  |
| --- | --- | --- |
| Название метода | Описание | |
| *addPassenger* | Добавляет переданного пассажира в контейнер, описывающий кабину лифта. | |
| *deletePassenger* | Удаляет переданного пассажира из контейнера лифта. | |
| *isNotFull* | | Возвращает «*true»,* если количество пассажиров в лифте меньше заданной вместимости лифта. |
| *isEmpty* | | Возвращает «*true*», если размер контейнера лифта равен 0. |
| *move* | | Изменяет номер текущего этажа. |

Класс *Storey*, описывает сущность этажа, содержит данные о номере этажа, пассажирах, ожидающих лифт на данном этаже, пассажирах, прибывших на этот этаж и средства синхронизации потоков. Для всех полей класса имеются стандартные методы, позволяющие проверять и задавать их состояние. Методы, описывающие поведение объекта данного класса описаны в таблице 3.5.

Таблица 3.5 – Методы класса «*Storey*»

|  |  |
| --- | --- |
| Название метода | Описание |
| *comeInPassenger* | Добавляющий переданного пассажира в контейнер отправления. |
| *comeOffPassenger* | Удаляющий переданного пассажира из контейнера отправления. |
| *arrivePassenger* | Добавляющий переданного пассажира в контейнер прибытия. |
| *isStoreyEmpty* | Проверяющий есть ли пассажиры в контейнере отправления. Возвращает «*true*» если контейнер пуст. |

Класс *Building*, описывающий сущность здания, содержит объекты, описывающие список этажей, список пассажиров и лифт. Данный класс является точкой старта приложения. Инициализирует все данные запускает работу по перевозке пассажиров. Методы класса описаны в таблице 3.6.

Таблица 3.6 – Методы класса «*Building*»

|  |  |
| --- | --- |
| Название метода | Описание |
| *main* | Главный метод программы. Инициализирует объект типа *Building*, после чего вызывает метод *start.* |
| *start* | Создает потоки для пассажиров и лифта, запускает их, и по окончании проверяет правильность их работы. |
| *checkOnAllPassengersOnRightStories* | Проверяет все ли пассажиры прибыли на свой этаж назначения. |
| *checkOnAllPassengersAreCompleted* | Проверяет все ли пассажиры имеют состояние «*COMPLETED»*. |
| *checkOnAllDispatchContainersAreEmpty* | Проверяет на всех ли этажах контейнеры отправления пусты |
| *checkOnRightCountPassengers* | Проверяет совпадает ли начальное количество пассажиров с конечным. |
| *verify* | Запускает все методы проверки и возвращает «*true*» только если все проверки выполнены успешно. |

Класс *TranportationTask* создается отдельно для каждого пассажира, и имплементируется средствами *Java*, как отдельный *Thread*. Он описывает поведение пассажира во время перевозки, а именно вызывает методы, описывающие ожидание лифта, вход пассажира в лифт, ожидание в лифте своего этажа назначения и выход из лифта по прибытию на такой этаж.

Класс *ElevatorController* описывает сущность оператора и выполняет всю работу по управлению лифтом и также имплементируется средствами *Java*, как отдельный *Thread.* Методы класса описаны в таблице 3.7.

Таблица 3.7 – Методы класса «*ElevatorController*»

|  |  |
| --- | --- |
| Название метода | Описание |
| *exitPassenger* | Перемещает пассажира из лифта на этаж назначения. |
| *enterPassenger* | Перемещает пассажира с этажа отправления в лифт. |
| *isTheSameDirectionForPassenger* | Проверяет совпадает ли направление движения пассажира с направление движения лифта. |
| *countEnterPassengers* | Досчитывает количество входящих в лифт пассажиров. |
| *getNumberOfCurrentStorey* | Возвращает номер текущего этажа. |
| *getNumberOfExitPassengerOnStorey* | Возвращает число выходящих пассажиров. |
| *isComplete* | Проверяет можно ли закончить перевозку. |
| *getNumberOfFreePlacesInElevator* | Возвращает количество свободных мест в лифте. |
| *notifyElevator* | Оповещает, ожидающие возможности выхода из лифта, потоки пассажиров. |
| *notifyStorey* | Оповещает, ожидающие возможности войти в лифт, потоки пассажиров. |
| *waitEndMovingPassengers* | Приостанавливает работу потока пока не выйдут все пассажиры. |
| *moveElevator* | Перемещает лифт на заданный этаж. |
| *stopElevator* | Останавливает лифт и начинает работу по высадке и посадке пассажиров. |
| *run* | Вызывается при запуске ассоциированного потока. |
| *waitDestinationStorey* | Приостанавливает, вызвавший данный метод, поток до достижения лифтом переданного в параметре этажа. |
| *waitElevator* | Приостанавливает, вызвавший данный метод, поток до остановки лифта. |

Для работы с *property*-файлами создан класс *PropertiesReader*. Класс Writer отвечает за вывод информации в консоль и *log*-файл.

Исходные тексты разработанного приложения приведены в приложении А.

Запуск *unit*-тестов, компиляция и сборка приложения в *jar*-файл осуществлялась посредством *Ant*. Для этого в командной строке необходимо ввести команду «*ant»* и указать название *build*-файла. Если явно не указать название файла, то по умолчанию будет вызван файл «*build.xml»*. В разработанном проекте файл сборки хранится в корневом каталоге и называется «*build.xml*». Файл сборки приведен в приложении Б.

Так как при создании приложения использовался инструмент разрешения зависимостей библиотек *Apache Ivy*, то при сборке все используемые библиотеки будут скачены автоматически. Для этого был создан файл, описывающий зависимости приложения *ivy.xml.*

Для начала работы приложения необходимо запустить файл *Elevator.jar*.

После чего приложение запустится и будет выводить в консоль и лог файл сообщения, описывающие все этапы перемещения пассажиров в здании. По завершении перевозок будет выполнена верификация результата, состоящая из нескольких проверок. Результат по каждой проверке также выведется в консоль и в файл «*log.txt*» каталога «*logs*».

Пример работы приложения с исходными данными: *storeysNumber* = 4, *elevatorCapacity* = 2, *passengersNumber* = 10, приведен в приложении В.

# ЗАКЛЮЧЕНИЕ

В процессе прохождения практики ознакомилась с компанией ИООО «ЭПАМ СИСТЕМЗ» её сотрудниками, особенностями, применяемыми технологиями, принципами управления и работы.

При прохождении технологической практики, были подробно изучены: *JUnit* – библиотека для модульного тестирования программного обеспечения на языке *Java*, *Apache Ant* – утилита для автоматизации процесса сборки программного продукта, а также средства *Java* для разработки многопоточного приложения.

Приложение разрабатывалось через тестирование что позволило более эффективно писать понятный и легко читаемый код.

Результатом пройденной практики является, разработанное средствами *Java API,* многопоточное приложение, симулирующее работу пассажирского лифта. Приложение прошло проверку у разработчиков компании, что подтверждает правильность выполненнго задания.

Об компании можно сделать вывод: «*Epam Systems*» – это большая, быстроразвивающаяся компания широкого профиля деятельности. Которая активно инвестирует в подготовку будущих профессионалов и оказывает особую поддержку тем, кто только начинает карьеру. Она предлагает бесплатные тренинги по различным направлениям для студентов старших курсов и выпускников для дальнейшего трудоустройства в компании.

**СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

1. Система стандартов безопасности труда. Рабочее место при выполнении работ сидя. Общие эргономические требования: ГОСТ 12.2.032-78. – Введ. 01.01.79. – Мн. : Межгос. совет по стандартизации, метрологии и сертификации: Белорус. гос. ин-т стандартизации и сертификации, 2004. – 12 с.
2. Санитарные правила и нормы. Гигиенические требования к видеодисплейным терминалам, электронно-вычислительным машинам и организации работы : СанПиН 9-131 РБ 2000 : утв. Постановлением Главного гос. санитарного врача Республики Беларусь № 53 от 10.11.2000 г. : ввод. в действие с 10.11.2000 г. – Мн. : ГУ НИИ санитарии и гигиены, 2000. – 64 с.
3. Блинов, И.Н. Java промышленное программирован
4. ие: практ. Пособие / И. Н. Блинов, В.С. Романчик – Минск: УниверсалПресс, 2007. – 704с.
5. Брюс Э. Философия Java. Библиотека программиста / Э. Брюс – СПб.: Питер, 2009. – 640с.
6. Хабибулин, И.Ш. Самоучитель Java / И.Ш. Хабибулин – Спб.: БХВ-Петербург, 2001. – 464с
7. Java Virtual Machine [Электронный ресурс]. – Режим доступа: https://ru.wikipedia.org/wiki/Java\_Virtual\_Machine. – Дата доступа: 03.07.2016.

**ПРИЛОЖЕНИЕ А**

(Обязательное)

**Листинг классов**

Файл Elevator.java

**package** main.java.entity.impl;  
  
**import** main.java.entity.IElevator;  
**import** main.java.state.ElevatorState;  
**import** java.util.ArrayList;  
**import** java.util.List;  
**import** java.util.Objects;  
**import** java.util.concurrent.locks.Condition;  
**import** java.util.concurrent.locks.Lock;  
**import** java.util.concurrent.locks.ReentrantLock;  
**public class** Elevator **implements** IElevator {  
 **private int elevatorCapacity**;  
 **private** List<Passenger> **elevatorContainer**;  
 **private int numberOfCurrentStorey**;  
 **private final** Lock **elevatorLock** = **new** ReentrantLock();  
 **private final** Condition **elevatorStopped** = **elevatorLock**.newCondition();  
 **private** ElevatorState **elevatorState**;  
  
 **public** Elevator(**final int** elevatorCapacity) {  
 **this**.**elevatorCapacity** = elevatorCapacity;  
 **this**.**elevatorContainer** = **new** ArrayList<>(elevatorCapacity);  
 **this**.**numberOfCurrentStorey** = 0;  
 **elevatorState** = ElevatorState.***STOP***;  
 }  
  
 **public boolean** addPassenger(**final** Passenger passenger) {  
 **return elevatorContainer**.add(passenger);  
 }  
  
 **public boolean** deletePassenger(**final** Passenger passenger) {  
 **return elevatorContainer**.remove(passenger);  
 }  
  
 **public boolean** isNotFull() {  
 **return elevatorContainer**.size() < **elevatorCapacity**;  
 }  
  
 **public boolean** isEmpty() {  
 **return elevatorContainer**.isEmpty();  
 }  
  
 **public int** move(**final int** storey) {  
 **numberOfCurrentStorey** = storey;  
 **return** getNumberOfCurrentStorey();  
 }  
  
 **public** ElevatorState getElevatorState() {  
 **return elevatorState**;  
 }  
  
 **public void** setElevatorState(ElevatorState elevatorState) {  
 **this**.**elevatorState** = elevatorState;  
 }  
  
 **public** Lock getElevatorLock() {  
 **return elevatorLock**;  
 }  
  
 **public** Condition getElevatorStopped() {  
 **return elevatorStopped**;  
 }  
  
 **public int** getElevatorCapacity() {  
 **return elevatorCapacity**;  
 }  
  
 **public void** setElevatorCapacity(**int** elevatorCapacity) {  
 **this**.**elevatorCapacity** = elevatorCapacity;  
 }  
  
 **public** List<Passenger> getElevatorContainer() {  
 **return elevatorContainer**;  
 }  
  
 **public void** setElevatorContainer(List<Passenger> elevatorContainer) {  
 **this**.**elevatorContainer** = elevatorContainer;  
 }  
  
 **public int** getNumberOfCurrentStorey() {  
 **return numberOfCurrentStorey**;  
 }  
  
 @Override  
 **public boolean** equals(Object o) {  
 **if** (**this** == o) **return true**;  
 **if** (!(o **instanceof** Elevator)) **return false**;  
 Elevator elevator = (Elevator) o;  
 **return** getElevatorCapacity() == elevator.getElevatorCapacity() &&  
 Objects.*equals*(getElevatorContainer(), elevator.getElevatorContainer());  
 }  
  
 @Override  
 **public int** hashCode() {  
 **return** Objects.*hash*(getElevatorCapacity(), getElevatorContainer());  
 }  
}

Файл Storey.java

**package** main.java.entity.impl;  
  
**import** java.util.ArrayList;  
**import** java.util.List;  
**import** java.util.Objects;  
**import** java.util.concurrent.locks.Condition;  
**import** java.util.concurrent.locks.Lock;  
**import** java.util.concurrent.locks.ReentrantLock;  
**public class** Storey {  
 **private** List<Passenger> **dispatchStoreyContainer**;  
 **private** List<Passenger> **arrivalStoreyContainer**;  
 **private int numberOfStorey**;  
 **private** Lock **storeyLock** = **new** ReentrantLock();  
 **private** Condition **arrivalElevator** = **storeyLock**.newCondition();  
  
 **public** Storey(**int** numberOfStorey) {  
 **this**.**numberOfStorey** = numberOfStorey;  
 **this**.**dispatchStoreyContainer** = **new** ArrayList<>();  
 **this**.**arrivalStoreyContainer** = **new** ArrayList<>();  
 }  
  
 **public boolean** comeInPassenger(**final** Passenger passenger) {  
 **return this**.**dispatchStoreyContainer**.add(passenger);  
 }  
  
 **public boolean** comeOffPassenger(**final** Passenger passenger) {  
 **return dispatchStoreyContainer**.remove(passenger);  
 }  
  
 **public boolean** arrivePassenger(**final** Passenger passenger) {  
 **return this**.**arrivalStoreyContainer**.add(passenger);  
 }  
  
 **public boolean** isStoreyEmpty() {  
 **return dispatchStoreyContainer**.isEmpty();  
 }  
  
 **public int** getNumberOfStorey() {  
 **return numberOfStorey**;  
 }  
  
 **public** List<Passenger> getDispatchStoreyContainer() {  
 **return dispatchStoreyContainer**;  
 }  
  
 **public void** setDispatchStoreyContainer(List<Passenger> dispatchStoreyContainer) {  
 **this**.**dispatchStoreyContainer** = dispatchStoreyContainer;  
 }  
  
 **public** List<Passenger> getArrivalStoreyContainer() {  
 **return arrivalStoreyContainer**;  
 }  
  
 **public void** setArrivalStoreyContainer(List<Passenger> arrivalStoreyContainer) {  
 **this**.**arrivalStoreyContainer** = arrivalStoreyContainer;  
 }  
  
 **public** Condition getArrivalElevator() {  
 **return arrivalElevator**;  
 }  
  
 **public** Lock getStoreyLock() {  
 **return storeyLock**;  
 }  
  
 @Override  
 **public boolean** equals(Object o) {  
 **if** (**this** == o) **return true**;  
 **if** (!(o **instanceof** Storey)) **return false**;  
 Storey storey = (Storey) o;  
 **return** Objects.*equals*(getDispatchStoreyContainer(), storey.getDispatchStoreyContainer()) &&  
 Objects.*equals*(getArrivalStoreyContainer(), storey.getArrivalStoreyContainer());  
 }  
  
 @Override  
 **public int** hashCode() {  
 **return** Objects.*hash*(getDispatchStoreyContainer(), getArrivalStoreyContainer());  
 }  
}

**ПРИЛОЖЕНИЕ Б**

(Обязательное)

**Файл «*build.xml*»**

<**project name="Elevator" default="build" xmlns:ivy="antlib:org.apache.ivy.ant"**>  
<**property name="src.dir" location="src/main/java"**/>  
 <**property name="resources.dir" location="src/main/resources"**/>  
 <**property name="test.src.dir" location="test"**/>  
 <**property name="build.dir" location="build"**/>  
 <**property name="classes.dir" location="${build.dir}/classes"**/>  
 <**property name="test.classes.dir" location="${build.dir}/test-classes"**/>  
 <**property name="ivy.reports.dir" location="${build.dir}/ivy-reports"**/>  
 <**property name="test.reports.dir" location="${build.dir}/test-reports"**/>  
 <**property name="dist.dir" location="${build.dir}/dist"**/>  
 <**property name="jar.main.class" value="main.java.entity.impl.Building"**/>  
 <**property name="jar.file" value="${dist.dir}/${ant.project.name}.jar"**/>  
  
 <**available classname="org.apache.ivy.Main" property="ivy.installed"**/>  
<**target name="install-ivy" description="Install ivy" unless="ivy.installed"**>  
 <**mkdir dir="${user.home}/.ant/lib"**/>  
 <**get dest="${user.home}/.ant/lib/ivy.jar" src="http://search.maven.org/remotecontent?filepath=org/apache/ivy/ivy/2.3.0/ivy-2.3.0.jar"**/>  
 <**fail message="Ivy has been installed. Run the build again"**/>  
 </**target**>  
  
 <**target name="resolve" depends="" description="Use ivy to resolve classpaths"**>  
 <**ivy:resolve**/>  
  
 <**ivy:report todir='${ivy.reports.dir}' graph='false' xml='false'**/>  
  
 <**ivy:cachepath pathid="compile.path" conf="compile"**/>  
 <**ivy:cachepath pathid="test.path" conf="test"**/>  
 </**target**><**target name="resources" description="Copy resources into classpath"**>  
 <**mkdir dir="${classes.dir}/main/resources"**/>  
 <**copy todir="${classes.dir}/main/resources"**>  
 <**fileset dir="${resources.dir}"**/>  
 </**copy**>  
 <**copy todir="${classes.dir}"**>  
 <**fileset dir="${resources.dir}"**>  
 <**include name="log4j.properties"**/>  
 </**fileset**>  
 </**copy**>  
 <**copy todir="${test.classes.dir}"**>  
 <**fileset dir="${test.src.dir}"**>  
 <**include name="\*.properties"**/>  
 <**exclude name="\*.java"**/>  
 </**fileset**>  
 </**copy**>  
 </**target**>  
  
 <**target name="compile" depends="resolve,resources" description="Compile code"**>  
 <**mkdir dir="${classes.dir}"**/>  
 <**javac srcdir="${src.dir}" destdir="${classes.dir}" includeantruntime="false" debug="true" classpathref="compile.path"**/>  
 </**target**>  
  
 <**target name="compile-tests" depends="compile" description="Compile tests"**>  
 <**mkdir dir="${test.classes.dir}"**/>  
 <**javac srcdir="${test.src.dir}" destdir="${test.classes.dir}" includeantruntime="false" debug="true"**>  
 <**classpath**>  
 <**path refid="test.path"**/>  
 <**pathelement path="${classes.dir}"**/>  
 </**classpath**>  
 </**javac**>  
 </**target**>  
  
<**target name="test" depends="compile-tests" description="Run unit tests"**>  
 <**mkdir dir="${test.reports.dir}"**/>  
 <**junit printsummary="yes" haltonfailure="yes"**>  
 <**classpath**>  
 <**path refid="test.path"**/>  
 <**pathelement path="${classes.dir}"**/>  
 <**pathelement path="${test.classes.dir}"**/>  
 </**classpath**>  
 <**formatter type="xml"**/>  
 <**batchtest fork="yes" todir="${test.reports.dir}"**>  
 <**fileset dir="${test.src.dir}"**>  
 <**include name="\*\*/\*Test\*.java"**/>  
 <**exclude name="BuildingTest.java"**/>  
 </**fileset**>  
 </**batchtest**>  
 </**junit**>  
 </**target**>  
<**target name="build" depends="test" description="Create executable jar archive"**>  
 <**ivy:retrieve pattern="${dist.dir}/lib/[artifact]-[revision](-[classifier]).[ext]" conf="runtime"**/>  
  
 <**manifestclasspath property="jar.classpath" jarfile="${jar.file}"**>  
 <**classpath**>  
 <**fileset dir="${dist.dir}/lib" includes="\*.jar"**/>  
 </**classpath**>  
 </**manifestclasspath**>  
  
 <**jar destfile="${jar.file}" basedir="${classes.dir}"**>  
 <**manifest**>  
 <**attribute name="Main-Class" value="${jar.main.class}"** />  
 <**attribute name="Class-Path" value="${jar.classpath}"** />  
 </**manifest**>  
 </**jar**>  
 </**target**>  
  
 <**target name="run" depends="build" description="Run code"**>  
 <**java jar="${jar.file}" fork="true"**/>  
 </**target**>  
<**target name="clean" description="Cleanup build files"**>  
 <**delete dir="${build.dir}"**/>  
 </**target**>  
  
 <**target name="clean-all" depends="clean" description="Additionally purge ivy cache"**>  
 <**ivy:cleancache**/>  
 </**target**>  
  
</**project**>

**ПРИЛОЖЕНИЕ В**

(Обязательное)

**Пример работы приложения**

Содержимое файла «log.xml»

BOARDING\_OF\_PASSENGER passenger6 on 0 storey  
MOVING\_ELEVATOR from 0 to 1 storey  
BOARDING\_OF\_PASSENGER passenger10 on 1 storey  
MOVING\_ELEVATOR from 1 to 2 storey  
DEBOARDING\_OF\_PASSENGER passenger10 on 2 storey  
MOVING\_ELEVATOR from 2 to 3 storey  
BOARDING\_OF\_PASSENGER passenger3 on 3 storey  
MOVING\_ELEVATOR from 3 to 2 storey  
MOVING\_ELEVATOR from 2 to 1 storey  
DEBOARDING\_OF\_PASSENGER passenger3 on 1 storey  
DEBOARDING\_OF\_PASSENGER passenger6 on 1 storey  
BOARDING\_OF\_PASSENGER passenger8 on 1 storey  
BOARDING\_OF\_PASSENGER passenger9 on 1 storey  
MOVING\_ELEVATOR from 1 to 0 storey  
DEBOARDING\_OF\_PASSENGER passenger8 on 0 storey  
DEBOARDING\_OF\_PASSENGER passenger9 on 0 storey  
BOARDING\_OF\_PASSENGER passenger1 on 0 storey  
BOARDING\_OF\_PASSENGER passenger7 on 0 storey  
MOVING\_ELEVATOR from 0 to 1 storey  
DEBOARDING\_OF\_PASSENGER passenger1 on 1 storey  
DEBOARDING\_OF\_PASSENGER passenger7 on 1 storey  
MOVING\_ELEVATOR from 1 to 2 storey  
MOVING\_ELEVATOR from 2 to 3 storey  
BOARDING\_OF\_PASSENGER passenger4 on 3 storey  
BOARDING\_OF\_PASSENGER passenger5 on 3 storey  
MOVING\_ELEVATOR from 3 to 2 storey  
MOVING\_ELEVATOR from 2 to 1 storey  
MOVING\_ELEVATOR from 1 to 0 storey  
DEBOARDING\_OF\_PASSENGER passenger4 on 0 storey  
DEBOARDING\_OF\_PASSENGER passenger5 on 0 storey  
MOVING\_ELEVATOR from 0 to 1 storey  
MOVING\_ELEVATOR from 1 to 2 storey  
MOVING\_ELEVATOR from 2 to 3 storey  
MOVING\_ELEVATOR from 3 to 2 storey  
MOVING\_ELEVATOR from 2 to 1 storey  
BOARDING\_OF\_PASSENGER passenger2 on 1 storey  
MOVING\_ELEVATOR from 1 to 0 storey  
DEBOARDING\_OF\_PASSENGER passenger2 on 0 storey  
COMPLETION\_TRANSPORTATION  
Elevator is empty.  
All storeys are empty.  
All passengers have completed state.  
All passengers arrived on destination storey.  
Initial number of passengers match with the final.