ДМИТРИЙ
ЧЕРЕМНОВ

IT

ПРОФЕССИОНАЛЬНЫЕ КОМПЕТЕНЦИИ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

12+

Дмитрий Черемнов

Профессиональные компетенции разработки программного обеспечения

«ЛитРес: Самиздат»

2019

Черемнов Д. Н.

Профессиональные компетенции разработки программного обеспечения / Д. Н. Черемнов — «ЛитРес: Самиздат», 2019

Одного знания языка программирования недостаточно для профессиональной разработки программного обеспечения. Нужны ряд компетенций - знаний, методик, навыков владения инструментами. Ключевая информация и рекомендации для будущих разработчиков программного обеспечения и других информационно-технологических специалистов. Лайфхаки для ИТ юниоров, желающих значительно повысить свою квалификацию и зарплату. ИТ "кухня" - технологии, процессы и методологии, необходимые для создания качественного программного продукта.

Содержание

Введение

Программисты учатся на ошибках других программистов, а потом обучают новых программистов тем же ошибкам.

Я фанат разработки программного обеспечения…

У меня интересная работа технического лидера в компании, разрабатывающей программное обеспечение на заказ для зарубежных и отечественных клиентов.

За долгие годы путем обучения, проб, ошибок, дошел до уровня, когда мне доверяют начинать разработку проекта и доводить его с командой до продакшена.

Я поделюсь своими мыслями, знаниями и опытом в разработке программного обеспечения.

Почему…?

Я задумался, почему несмотря на множество людей с горящими глазами, изучающими или знающими какой-либо язык программирования, остается острая нехватка программистов (например, в нашей и других компаниях дают бонус за успешную рекомендацию кандидата)?

Почему, несмотря на большой объем информации в интернете, сложно понять, что требуется для того, чтобы тебя приняли на работу в компанию, разрабатывающую программное обеспечение?

Почему сейчас, когда компании лояльно относятся к сотрудникам (в нашей компании десяток человек работают удаленно), которые находятся за сотни километров в маленьких городках и успешно работают на ключевых позициях, остается кадровый голод?

Злосчастный круг

К сожалению, проблема студентов и молодых специалистов известна – компании не хотят рисковать и брать на работу людей без опыта, без нужных компетенций.

А без работы невозможно (а точнее очень трудно) получить необходимый опыт.

Злосчастный круг замыкается и его сложно разорвать.

Тебя интересуют информационные технологии?

Ты изучаешь язык программирования и уже пишешь (пусть и простые) программы?

Ты хочешь связать свою профессиональную жизнь с ИТ?

А может ты уже работаешь в ИТ, но тебе нужно значительно повысить свою квалификацию?

Ты сможешь получить концентрат знаний и опыта, тебе не придется блуждать в дебрях интернета и собирать все по крупицам. Ты сможешь за год усвоить и понять основное, что в ином случае достигается годами. Но не обещаю, что это будет легко.

Стремись к профессиональной цели!

Если тебя интересует программирование, информационные технологии и ты находишься в начале профессионального пути – стремись к своей цели!

Получи Чек лист по профессиональным компетенциям, требуемыми для разработки программного обеспечения.

Чек лист поможет оценить навыки, наметить векторы по развитию профессиональных компетенций разработчика программного обеспечения, которые значительно повысят ИТ квалификацию и помогут открыть дверь в компанию твоей мечты или получить повышение на текущей должности.

Определив свой текущий уровень, сформируй карьерный план и определи ближайшие и долгосрочные цели. Не трать на планирование много времени! План – это список намерений, но не более того.

Профессиональные компетенции разработки программного обеспечения помогут достичь поставленные цели при любой текущем уровне – для уровня курсанта, юниора или разработчика с опытом. Наибольший эффект получите на начальных уровнях, но разработчики среднего уровня, специализируясь на определенных технологиях и уделяя внимание отстающим компетенциям, могут достичь экспертного уровня.

Путь до юниора может занять много времени – от полугода до 2 лет, в зависимости от начального уровня и интенсивности подготовки. Нужно много изучать и практиковать, не менее 10 часов и нескольких дней в неделю (например, 2-3 дня в неделю по 2 часа вечером в будни и 4-6 часов в выходной теории и практики). Такой график сложно, но можно соблюдать по совместительству для учащегося, студента или занятого на основной работе. Естественно, у каждого человека своя скорость усвоения материала и получения практических навыков, кто-то достигнет цели ранее, а кому-то потребуется больше времени.

Звезд с неба не хватал

Может у тебя возникают сомнения, сможешь ли ты достичь своей профессиональной цели – начать работу в ИТ или повысить свою квалификацию?

Может ты думаешь, что многим все дается легко и просто? Вероятно, такие люди есть, но я не из их числа…

Знания мне даются непросто, свой опыт я набирал долго с "кровью и потом" – на работе, вечерами дома, а если меня посетит вдохновение, то иногда и ночью :)

Многие задают вопрос – есть ли польза в книгах, курсах и наставниках? Курсы, книги и учебные материалы, менторы – это ускорители, мотиваторы и тотализаторы, но все зависит от вас самих – вы должны изучать и практиковать!

Первый опыт я получил в школе, кодируя алгоритмы на микрокалькуляторе “Электроника МК-52”, программируя простые игры на микрокомпьютерах “БК-0010” и “ZX Spectrum” на языке Basic. Помимо этого, я провел много времени за компьютерными играми. В университете работал с системой управления базой данных dBase, создавая программу расчета полезных веществ в продуктах питания.

Моя первая работа на должности программист-инженер была в маленьком закрытом городке Казахстана в Курчатове в Институте Атомной Энергии в Национальном Ядерном центре. Курчатов расположен недалеко от ядерного полигона, на полигоне я видел огромные скрюченные, погнутые железобетонные столбы и “атомное” озеро – последствия наземных термоядерных испытаний. Первая реализованная мною база данных, ушедшая в продакшен – “Система учета радиоактивных материалов” для Международного агентства по атомной энергии. Ценным источником знаний по языку программирования C в 1996 году, являлась бумажная распечатка руководства “Язык программирования C” Брайана Кернигана и Денниса Ритчи, по ней я обучался с моим другом и коллегой Цай Евгением. Нашими наставниками по разработке были Инков Александр, Петренко Андрей (передаю им персональный привет!). Позже по документации и книгам изучали C++ и Delphi. В дальнейшем мы работали на C, C++ и Delphi в основном над созданием информационно-управляющих систем для эксперементальных стендов Ангара и EAGLE.

В 2006 году выиграл грант и в Омске в компании Luxoft прошел 4 месячный интенсивный курс по методикам разработки Rational Unified Process, языку программирования Java, Java EE технологиям и базе данных Oracle. После этого сменил специализацию с C++, Delphi на Java стек. Большинство ребят после курса также стали работать Java разработчиками. Переехал в кремневую долину Сибири – в Новосибирск. (Передаю всем знакомым и друзьям из Курчатова, Омска и Новосибирска пламенный привет!).

Начало карьеры может быть сложным, но вам должно нравиться программировать, чтобы получать удовольствие от ИТ учебы в настоящее время и в будущем от работы в ИТ! Чертовски приятно иметь работу, которая интересна и захватывает словно хобби! Иначе, возможно имеет смысл вам найти другое занятие по душе…

Естественно, каждый человек рано или поздно имеет тенденцию терять мотивацию. Практически каждый из нас нуждается периодически в “волшебном пенделе” ;) Рекомендую найти сообщество единомышленников среди изучающих ИТ и ментора из числа опытных разработчиков, которые могут оказать как моральную поддержку, так и дать практические подсказки, советы и помощь в освоении информационных технологий. Разработчики с опытом в свою очередь, оказывая помощь курсантам, могут избежать “выгорания” и прокачать свои навыки по управлению командой и проектом и вырасти до Лидера команды.

Через много лет я достиг своей цели – разработка программного обеспечения в классной ИТ компании в профессиональной команде!

А какова твоя цель?

Я знаю, что ничего не знаю

Оглядываясь назад, я начал обобщать и фиксировать свой опыт.

Теперь я понимаю, что могу указать более короткую дорогу к твоей профессиональной цели, передать свои знания и опыт – это моя новая цель!

Почему я точно знаю, какие знания и опыт требуется в ИТ?

Почему я смог выделить самое важное для юниоров из огромного количества информации?

Потому что я писал программы (Delphi, C, C++) и кодирую сейчас (Java, JavaScript, SQL) на нескольких языках программирования.

Потому что проектировал, программировал, тестировал, внедрял и поддерживал системы с различными технологиями:

REST Services, Web Services, Micro services, JSON, XML…

Для хранения данных использовал различные базы данных:

Oracle, MySQL, MariaDB, PostgreSQL, MongoDB, Cassandra, Redis…

Потому что мы используем самые современные средства разработки:

Jira, Redmine, GitLab, Git, Jenkins, TeamCity, IDEA, Eclipse и пр.

Потому что участвовал в десятке ИТ проектов для стартапов, электронной коммерции, банков, бизнеса в одиночку и в командах от 2 до 10 человек, продолжительностью от 3 месяцев до 3 лет, на роли рядового разработчика, ведущего разработчика или технического лидера.

Потому что у меня есть некоторый опыт фриланса и "домашние" ИТ проекты, на которых я изучаю незнакомые мне технологии.

Потому что я знаю и применяю современные методики разработки:

паттерны, рефакторинг, код ревью, юнит и интеграционное тестирование, основные методологии разработки: XP, Scrum.

Потому что помимо успешных проектов были и провалы, а на ошибках учатся.

Потому что имею несколько сертификатов по ИТ, значимые их них Sun Certified Programmer for the Java 2 Platform (SCP) и Oracle PL/SQL Developer Certified Associate (OCA) – интересен факт, что Oracle проглотил солнце в 2009 году.

Потому что прочитал десятки книг и сотни статей по ИТ, часто буду ссылаться на внешние источники знаний, готов сам учиться у других и перенимать опыт.

Нужно бежать со всех ног, чтобы только оставаться на месте

Есть 2 "новости" для новичков в ИТ. Одна – хорошая, вторая – плохая:

• "Плохая новость" – уровень входа в ИТ за последнее десятилетие значительно вырос. Ранее в большинстве случаев требовалось знание одного языка программирования и навык кодирования. Сейчас ИТ компании даже к юниорам выдвигают ряд обязательных требований.

• "Хорошая новость" – некоторые ИТ специалисты по инерции игнорируют важные ИТ навыки. Вы же можете, начав с чистого листа, взять все самое лучшее в свой профессиональный арсенал.

Информационные технологии меняются с поразительной быстротой – чтобы оставаться профессионалом, нужно постоянно изучать, экспериментировать, использовать на практике.

Одного знания языка программирования недостаточно, чтобы вести командную разработку программного обеспечения. Нужны еще ряд профессиональных компетенций – знаний, методик, навыков владения инструментами, которые позволят сделать значимые огромные шаги в профессии разработчика ПО.

У меня эти шаги заняли годы, ты пройдешь их за год – но придется приложить большие усилия.

Не существует "золотой пилюли" для успеха. С 2016 года я прежде всего для себя фиксировал в Wiki конспект ИТ технологий, методологий, инструментов и ссылок на полезные ресурсы. С 2017 года на основе этих материалов, с добавлением практических заданий, постепенно формирую курс, который повысит ИТ квалификацию и поможет открыть дверь в компанию твоей мечты или получить повышение на текущей должности, повысить рейт. Неоднократно материалы курса обновлялись и дополняются в настоящее время. На основе содержания курса сформирована книга, которая указывает ключевые направления профессионального роста для информационно-технологических специальностей и показывает всю современную "кухню" процесса разработки. Материалы курса и книги прежде всего ориентированы на разработчиков ПО, но будут несомненно полезны для тестеров, менеджеров, ИТ администраторов, дизайнеров, владельцев ИТ продуктов – всех участников разработки программного обеспечения.

Готов ли ты приложить усилия, вместе с нами изучать информационные технологии, набор инструментов, методик и применять их на практике, чтобы технически "вырасти" и профессионально выделиться среди тысяч молодых ИТ специалистов?

Мне показалось, или кто-то ответил – "НЕТ"? Вы можете не верить мне, ведь я могу ошибаться. Но главное, чтобы вы поверили в себя…

Если твой ответ "ДА", то ты на верном пути.

Добро пожаловать в ИТ сообщество профессионалов!

Благодарности

Книга посвящается моим родителям.

Это страница адресована моим близким и родным, друзьям и коллегам, преподавателям и наставникам. Я хочу выразить слова благодарности многим людям.

Благодарю своих родителей: Черемнова Николая Георгиевича и Валентину Федоровну, которые воспитали и поддерживали меня.

С уважением и любовью к старшим: дяде Боре Марченко и тете Саше в Актобе, теще Григорьевой Татьяне Дмитриевне, тете Люде в Семее, дяде Саше Бондарчук и тете Тане в Тюмене, дяде Андрею Альберт и тете Нелли в Германии, дяде Виктору и тете Оле в Канаде, и многим другим. Вы далеки, но я часто о вас вспоминаю.

С горячим приветом к сестре Ирине и ее мужу Олегу Пенкиным, троюродным братьям и сестрам (и их половинкам): Танк Вовчику в Германии, Султанову Тофику, Лейле в Семее, Бондарчук Денису, Наташе и Ирине в Тюмене, Залесову Евгению в Москве, сестре жены Лене и ее брату Григорьеву Денису, и всем другим. Спасибо, что принимаете меня таким, какой я есть.

С юношеским приветом к племянницам Шенделевой Саше и ее мужу Никите, Деревянко Вике и ее мужу Артему, и всей молодежи и детворе. С вами я чувствую себя молодым :)

Благодарю своих школьных и университетских учителей и товарищей. Не забыть школьные события с друзьями: Жуковым Игорем, Лисовцовым Александром, Шацким Аркадием и Ябсом Владимиром.

Спасибо моим друзьям, товарищам и преподавателям со времен университета. Персональная благодарность Цай Евгению, за 15 незабываемых лет юности в универе, множеству приключений и совместной работе в Курчатове. Благодарю Сайдашева Тахира за практичный драйв и за попытку воплощения амбициозного проекта. Я до сих пор с ними на связи.

Большое спасибо моим наставникам и руководителям по работе в Курчатове в Национальном Ядерном центре: Инкову Александру, Петренко Андрею, Дзалбо Виктору за опыт и поддержку. Передаю привет бывшим коллегам: Кривцову Павлу, Щербаку Игорю, Кошненко Игорю, Коровикову Александру, Ольховику Дмитрию и другим.

Моя благодарность всем коллегам в компании Азофт – я приобрел бесценный опыт, многому научился у вас и надеюсь, чему-то научил вас. Особая благодарность: Ожиганову Ивану за поддержку и воплощение моих идей, Лихачеву Олегу – за доверие сотрудникам и делегирование полномочий.

Благодарен своим друзьям в Бердске, со времен проживания и работы в Курчатове: Клименко Алексею за аналитический взгляд, но принятие моих безумных идей, Ястребкову Дмитрию – за здоровую критику идей, но поддержку, Пасько Александру – за критичный и осмысленный взгляд на жизнь. Отдельный привет их лучшим половинкам: Тане, Наташе и Гуле.

Отдельная благодарность Шенделевой Александре за дизайн, который она готовит по моим просьбам, в частности за обложку ИТ книги.

Прошу не огорчаться тем, кто не встретил на странице упоминания о себе – я о вас не забыл и благодарен вам…

Особая благодарность моей семье, любимым: жене Олесе и сыну Ивану, которые рядом со мной и которые помогали мне в подготовке книги.

Профессиональные компетенции разработки программного обеспечения

Лучше научите людей, рискуя, что они уйдут, чем не делайте ничего, рискуя, что они останутся.

Факты

• Каждая профессиональная компетенция разработки программного обеспечения повышает квалификацию ИТ специалиста.

• Даже одна компетенция может повысить эффективность разработки ПО.

• Комплексное применение компетенций многократно усиливает продуктивность работы и качество ПО.

• Без ряда компетенций командная, удаленная разработка невозможна.

• Компетентная команда ИТ специалистов – залог успешного проекта.

Цель

• Познакомиться с профессиональными компетенциями разработки программного обеспечения.

• Получить рекомендации по формированию компетенций.

• Изучить дополнительные материалы и документацию для повышения квалификации.

• Ознакомиться с инструментами, используемыми в разработке программного обеспечения и получить базовые навыки по работе с ними.

• Подготовить резюме, пройти собеседование и получить работу в ИТ компании или получить повышение по должности на текущей работе.

Проект и задачи

Цель и содержание проекта

Самая большая проблема с программистами в том, что ты никогда не сможешь понять, чем он занимается, пока не будет слишком поздно.

Описание

Каждый проект имеет набор документации. Есть документы, которые формируются иногда задолго до начала реализации, ряд других формируют по мере работы над проектом. Некоторые документы модифицируются со временем, иные становятся неактуальными. Одними из первых документов являются Цель и содержание проекта (часто они включаются в техническое задание в качестве раздела).

Цель и содержание проекта – это краткое описание, которое дает общее представление о назначении проекта и конечного планируемого результата разработки.

Цель проекта описывает какие задачи должны быть решены в результате проекта, а содержание проекта – что именно является результатом проекта.

Описание цели и содержания проекта (Project Scope) на примере проекта "Универсальная модульная платформа", в реализации которого принимают участие некоторые "выпускники" курса.

Проект "Универсальная модульная платформа"

Цель проекта

Много проектов имеют схожую многомодульную структуру, до 25% общего функционала.

Если выделить часто используемый общий функционал в модули, подключаемые по необходимости в разные проекты, то можно решить следующие задачи:

- быстрый старт разработки проекта на базе платформы;
- получение востребованного опыта и навыков разработки участниками;
- легкое вхождение участников команды разработки в однотипный проект;
- эффективное участие юниоров в разработке однотипных проектов;
- упрощение разработки и поддержки однотипных проектов;
- улучшение качества за счет многократного тестирования общего кода на разных проектах;
- уменьшение периода разработки за счет подключаемых модулей;
- финансовая экономия.

***Описание проекта* – Project Scope**

Описание проекта – многомодульной платформы, предоставляющей базовый функционал наиболее часто востребованных нефункциональных и функциональных требований. Модули данной платформы могут быть подключены по необходимости для реализации систем различных назначений, реализующих конкретные бизнес требования:

- система заказов услуг или продуктов;
- система бронирования и продажи билетов;
- система логистики;
- e-commerce система;
- информационная система и прочие.

Можно выделить основные компоненты систем:

- SQL база данных;
- Backend с бизнес логикой;

• приложение администратора;
• REST (JSON) API сервер;
• Frontend с веб интерфейсом;
• мобильные приложения.

Список наиболее востребованных нефункциональных и функциональных требований:
• аутентификация и авторизация;
• логирование;
• уровень доступа к SQL базе данных;
• планировщики для запуска периодических процессов;
• инфраструктура и настройка REST контроллеров;
• создание, редактирование и просмотр администраторов и пользователей системы;
• загрузка в систему и скачивание из системы файлов (фото, документов и т.п.).

Нужно отметить, что платформа предъявляет более строгие требования к проектированию, реализации, тестированию системы. Важно соблюдать баланс между минимальностью и достаточностью платформы, с учетом использования в разных приложениях.

Открытыми для обсуждения с участниками проекта остаются аспекты:

• масштабируемость платформы – должна ли система, построенная на платформе легко масштабироваться при необходимости;

• мультиарендность платформы (multitenancy) – должна ли архитектура поддерживать множество арендаторов-владельцев.

Данные аспекты являются интересными, но повышают сложность и увеличивают время разработки. Для MVP (minimum viable product – минимально жизнеспособный продукт) данными аспектами можно пренебречь. Но часто заказчики программного обеспечения планируют маштабируемость системы. Мультиарендная платформа используется обычно в программном обеспечении как услуге (software as a service – SaaS).

Техническое задание или Спецификация

Specification

Ходить по воде и разрабатывать ПО из спецификации легко. Просто нужно заморозить и то, и другое.

Описание

До разработки требуется определить требования и набор функциональности, которое необходимо реализовать в программном обеспечении, для этого предварительно проводится анализ бизнеса, идей и "хотелок" заказчика.

Аналитики изучают бизнес требования и формируют **Техническое задание** (ТЗ) или **Спецификацию**, с участием дизайнеров, которое включает:

• функциональные требования – описание основной и дополнительной функциональности, которое требуется реализовать в программном обеспечении;

• не функциональные требования – требования к среде исполнения, производительности, защищенности, наличия мониторинга, логирования и т.п.

• мокапы – схематическое изображение экранов или страниц с графическими элементами представления (надписи, таблицы и т.п.), ввода (поля и окна ввода, таблицы и т.п.) и управления (кнопки, радио кнопки, чек-боксы и т.п.), со схемой переходов между экранами;

• дизайн – графическое представление страниц и экранов программного обеспечения может дополнительно прилагаться к ТЗ (либо создаваться дизайнерами позже).

На базе **Технического задания** проводится предварительная оценка проекта: оценивается время реализации каждого требования, суммарное время, с учетом времени аналитики, дизайна, менеджмента, тестирования, поставки и рисков.

С учетом как функциональных, так и не функциональных требований формируется архитектура проекта, проводится выбор технологий, инструментов разработки, баз данных и пр., определяется требуемый состав и число ИТ специалистов.

В классическом процессе разработки **ТЗ** или **Спецификация** для программиста является исходным документом, используемым в процессе разработки.

На основе **Технического задания** (или **Спецификации**) формируется список задач программистам для реализации.

Пользовательские истории

User stories

Без требований программирование – это искусство добавлять баги в пустой файл.

Описание

В "гибких" методологиях (позже ты изучишь их подробнее), заменой **Технического задания**, **Спецификации** (или его дополнением) являются **Пользовательские истории** (**User stories**).

User story – фиксация требования, функциональности посредством краткого описания и ряда атрибутов:

• ID – уникальный идентификатор (в системе ведения проекта ID генерируется автоматически).

• Название – краткое описание истории. Например, “Просмотр журнала своих транзакций”.

• Важность (Importance) – степень важности данной задачи, по мнению заказчика. Например,

10 или 150. Другой вариант параметра – приоритет, например, P1, P2, P3.

• Предварительная оценка (initial estimate) – начальная оценка объема работ, необходимого для реализации истории по сравнению с другими историями.

• Как продемонстрировать (how to demo) – краткое пояснение того, как завершённая задача будет продемонстрирована.

• Примечания – любая другая информация: пояснения, ссылки на дополнительные источники информации, и т.д.

Часто **User story** дополнительно разделяется на отдельные задачи. В таком случае реализация каждой истории требует выполнения одной или нескольких задач.

Управление проектом и задачами

Чтобы руководить программистами, нужен Д. Кнут и пряник.

Описание

ИТ-проекты проходят через 5 основных фаз жизненного цикла:

• инициация;

• планирование;

• выполнение;

• мониторинг и контроль;

• завершение.

В начале жизненного цикла проекта проводится аналитика, формирование требований для программного обеспечения (в документах **Техническое задание** или **Спецификация**, или **Пользовательские истории**).

Далее следуют: планирование работ по проекту, формирование списка задач, их распределение, контроль над выполнением, тестированием задач, сборка и поставка ПО, анализ метрик по проведенной работе – все фазы, это неотъемлемая часть процесса разработки ПО.

Каким образом можно координировать, согласовывать и отслеживать проводимую работу? Для этого предназначены системы управления проектами и задачами, с разной функциональностью и возможностями.

Использование системы управления задачами – это не дань моде, а конкурентная необходимость, инструмент координации и эффективности работ, один из гарантов успешной реализации проекта.

Реальный отзыв команды, которая начала использовать современную систему управления проектами и задачами:

"Только при создании нового сайта мы смогли сократить срок разработки в 1,5 раза".

Для вас, как для разработчика, работа над задачей в общем случае выглядит следующим образом (реальный процесс может несколько отличаться):

Менеджером проекта вам назначается тикет (Issue), где описано, какую функциональность надо добавить или какую ошибку надо исправить.

Основные этапы выполнения задачи (тикета):

• Анализ задачи – вы знакомитесь с задачей, с документами и анализируете требования, имеющие отношение к вашей задаче. Если возникают вопросы по задаче, обсуждаете их с менеджером.

• Проектирование и оценка задачи – вы знакомитесь с кодом, который будет затронут в ходе кодирования. Обдумываете, как будете реализовывать задачу, какой паттерн программирования сможете использовать. Оцениваете задачу, а именно указываете количество времени, за которое планируете выполнить задачу, например: /estimate 1d 2h.

• Реализация задачи – рефакторинг (если требуется) текущего кода, написание нового кода, добавление юнит/интеграционных тестов. В ходе выполнения задачи или(и) после завершения задачи вы устанавливаете в задаче количество фактически затраченного времени, например: /spend 4h 30m.

• Деплоймент проекта – сборка проекта, установка в локальное рабочее окружение, ручное тестирование задачи.

• Контроль задачи – ревью кода, мерж кода в основную ветку разработки, проверка статуса сборки на сервере интеграции (проверка стилистики кода, качественных метрик кода, автоматических тестов).

В зависимости от системы управления и установленного процесса разработки задаче на каждом этапе устанавливаются различные статусы.

Следование процессу разработки значительно повышает эффективность работ, обеспечивает и поддерживает качество программного продукта, допускает взаимозаменяемость между членами команды, позволяет планировать и прогнозировать сроки выполнения итерации (этапа работ).

К процессу разработки ПО неоднократно будем возвращаться при описании последующих компетенций.

Инструменты

Десятки систем управления проектами и задачами, с разной функциональностью и разной доступностью:

Jira, Redmine, Asana, Basecamp, Trello, Wrike, GitLab и др.

Рекомендации

В случае, если вы самостоятельно разрабатываете проект, может показаться, что система управления не нужна.

Это заблуждение – в таком случае вы сами выполняете роль менеджера проекта и принимаете решения, что делать и какую функциональность включить в разработку.

Система управления проектами позволит проводить планируемую работу:

• Записывайте задачи, которые требуется решить в ходе работ над проектом.

• Записывайте функциональность, которую планируете реализовать.

• Фиксируйте сразу обнаруженные ошибки в ПО, чтобы позже не забыть их исправить.

Навыки применения системы управления задачами пригодятся вам, когда будете участвовать в командных разработках.

Советую для изучения и практического применения Open Source системы управления проектами и задачами (с потенциальной возможностью установки на собственный сервер):

Redmine – функциональна, гибко настраивается, имеется возможность подключения плагинов с дополнительной функциональностью.

GitLab – содержит базовые функции управления проектами и задачами.

Помимо этого, в ИТ компаниях часто используют мощную (и дорогую) коммерческую систему:

Jira

Дополнительные материалы

Сравнительная статья о системах управления проектами:

43 полезных сервиса для управления проектами. Без эпитетов

Резюме

Вы должны понимать и уметь рассказать о назначении систем управления проектами и задачами.

Необходимо иметь практические навыки работы хотя бы с одной из систем.

В результате добавится новая строчка в вашем резюме, например:

Project management: GitLab

Проектирование программного обеспечения

Проектирование ПО

Если бы строители строили дома так, как программисты пишут код, то первый же дятел, присевший на фасад, уничтожил бы цивилизацию.

Описание

При проектировании систем надо учитывать весь спектр требований:

функциональные – требования, отражающие бизнес логику приложения;

нефункциональные – требования, влияющие на работу приложения, такие как аутентификация и авторизация, масштабирование, производительность и др.

Проектирование включает выбор рабочего окружения и среды разработки:

- платформа;
- операционная система;
- технологии;
- база данных;
- язык программирования;
- библиотеки;
- фреймворки;
- …

Необходимо предварительно согласовать с заказчиком поддерживаемые платформы, ОС, веб браузеры (в случае веб разработки) с учетом версий.

UML

Для проектирования программного обеспечения помимо текстовых документов используют UML (**Unified Modeling Language** – унифицированный язык моделирования), который позволяет посредством графических изображений описать систему. Детальное проектирование применяют в сферах, где цена ошибки высока: космическая, авиационная, военная сфера.

В реальности описывают только самые важные или сложные аспекты.

Наиболее часто используемые диаграммы:

- Class diagram – Диаграмма классов для описания классов и связи между ними;
- Component diagram – Диаграмма компонентов для описания компонентов и связи между ними;
- Deployment diagram – Диаграмма развёртывания для описания физического развертывания программ на узлах;
- Activity diagram – Диаграмма деятельности для описания последовательности действий;
- Use case diagram – Диаграмма вариантов использования, описывает типичный способ взаимодействия пользователя с системой;
- Sequence diagram – Диаграмма последовательности, показывающий жизненный цикл (создание-деятельность-уничтожение) и взаимодействие для набора объектов.
- Statechart diagram – Диаграмма состояний, описывает возможные последовательности состояний и переходов модели.

Рекомендации

Если вы в начале профессионального пути, то проектированием программного обеспечения для заказного проекта будет заниматься более опытный сотрудник.

Но для своего "домашнего" проекта разработки, вы сами и менеджер, и архитектор и … Дерзайте!

Нужно понимать часто используемые UML диаграммы, в задачах иногда ссылаются на них.

Несмотря на многообразие технологий и инструментов разработки,

я планирую выделить узкий вектор наиболее доступных для изучения и использования, и чаще востребованных на рынке ИТ.

Инструменты

На курсе мы с курсантами провели выбор инструмента проектирования. Для построения диаграмм по ряду характеристик выбран Papyrus.

Дополнительные материалы

Вводная статья об основных диаграммах UML:

Проектирование программного обеспечения

Резюме

Вы должны понимать основные диаграммы **UML**.

Хорошо иметь практические навыки проектирования.

В результате добавится новая строчка в вашем резюме:

Software design: UML

Шаблоны проектирования

Самая сложная часть в дизайне… держаться подальше от фич.

Описание

Шаблоны (или **паттерны**) **проектирования** описывают типичные способы решения часто встречающихся проблем при проектировании программ – классов, компонентов.

Паттерны упрощают проектирование и поддержку программ.

Шаблоны проектирования в процессе разработки программного обеспечения:

• Предоставляют оптимальные, проверенные решения (вы эффективно используете время, если паттерн выбран грамотно).

• Обеспечивают стандартизацию кода (вы меньше совершаете ошибок, другой разработчик лучше понимает код).

• Формируют словарь программиста (одно название паттерна, сообщенное коллеге, раскрывает основную суть).

Основные типы шаблонов проектирования:

• **Основные шаблоны (Fundamental)** включают общие паттерны.

• **Порождающие шаблоны (Creational)** шаблоны проектирования, которые описывают механизмы создания и инициализации объектов.

• **Структурные шаблоны (Structural)** определяют различные сложные структуры.

• **Поведенческие шаблоны (Behavioral)** определяют взаимодействие между объектами.

Рекомендации

Для начала изучите основные шаблоны проектирования GOF. SOLID, GRASP, SAGA и иные – группы шаблонов, с которыми вы должны постепенно ознакомиться.

До кодирования – на этапе проектирования компонента или класса, до того, как "изобретать велосипед", найдите паттерн, который позволит решить задачу. Кодируйте согласно шаблону.

Даже если шаблон не найден – кодируйте! Важна практика – не ошибается тот, кто ничего не делает :)

Дополнительные материалы

Описание назначения и шаблонов проектирования простым языком:

Каталог паттернов проектирования

Более широкий список шаблонов:

Справочник "Паттерны проектирования"

Описание 23-х шаблонов проектирования GOF:

Шпаргалка по шаблонам проектирования

Блог о GRASP:

Шаблоны распределения обязанностей (GRASP)

Объяснение принципов SOLID на PHP:

Простое объяснение принципов SOLID

Описание паттерна Saga:

Паттерн: Сага

Для тех, кто предпочитает видео:

Шаблоны проектирования на языке Java

Книга и примеры шаблонов на Java:

Стивен Стелтинг, Олав Маассен "Применение шаблонов JAVA".

В дальнейшем можете изучить книгу-классику:

«Приёмы объектно-ориентированного проектирования. Паттерны проектирования» (англ. Design Patterns: Elements of Reusable Object-Oriented Software) от Gang of Four – Эрих Гамма, Ричард Хелм, Ральф Джонсон, Джон Влиссидес (для юниора книга сложна для восприятия).

Резюме

Шаблоны проектирования являются "шпаргалками" проектирования для программиста. Они позволяют изучить и использовать многолетний накопленный опыт проектирования программного обеспечения. Неразумно их игнорировать.

После изучения основных паттернов проектирования добавьте строчку в резюме:

Software design: Design pattern

Инструменты разработки

Коммуникация

Программисты на работе общаются двумя фразами: "непонятно" и "вроде работает".

Описание

Разработка программного обеспечения – это командная игра. Конечно есть исключения из правил – "домашние" проекты, небольшой Open Source проект, библиотека, инди-игра вполне по силам отдельному разработчику. Но в большинстве случаев, современная реальность требует участия сплоченной группы – менеджера, дизайнера, разработчиков, тестировщиков. Не зря ИТ гиганты покупают небольшие ИТ компании – помимо технологий или продукта они приобретают сработавшуюся команду.

Коммуникация – ключевой фактор командной работы.

Форматы для передачи информации:

• текст;

• звук;

• видео;

• файлы.

Инструменты

Для коммуникации и передачи информации используют множество инструментов:

• система управления проектами и задачами – обязательный и приоритетный инструмент взаимодействия команды;

• система управления версиями – (данной системе посвящен отдельный раздел) может содержать помимо исходного кода информационные файлы (README и др.);

• Wiki – web система позволяет оперативно документировать проект и процесс разработки;

• мессенджер – механизм более оперативного и менее формализованного общения;

• форум – система, где можно обсудить проблему и получить подсказки от внешнего сообщества;

• email – классический способ обмена информацией;

• голосовые чаты – многие предпочитают голосовое обсуждение;

• телефония – классический голосовой канал связи, актуален для связи с заказчиками и для оперативной связи, в случае недоступности других каналов связи;

• файловое хранилище – может рассматриваться как инструмент коммуникации для передачи больших файлов;

• прочие системы.

Рекомендации

Для эффективной командной работы с четким регламентом достаточно системы управления проектами и задачами и системы управления версиями.

Wiki – система для быстрого документирования с высокой степенью оперативности и доступности. Поддерживается версионность, часто интегрирована с системой управления проектами.

Классический email допустим для корпоративной коммуникации и как механизм связи с внешним миром, но мало эффективен для командной работы над проектом.

Голос важный коммуникационный канал, но его серьезная проблема в том, что он не способен фиксировать соглашения, потому важно использовать его наряду с другими системами (например, формирование задач или документации Wiki по итогам обсуждения). В некоторых

формализованных процессах разработки стараются исключать голос из процесса разработки, а во многих методиках это важный и постоянный источник информации.

Мессенджер – социальная составляющая для работы имеет значение, необходима в случае отсутствия четкого регламента работы. В некоторых случаях может быть интегрирована с системой управления проектами.

Корпоративный мессенджер Slack используют большое количество ИТ команд для коллективной работы над проектами, но его бесплатные функции ограничены.

В качестве его альтернативы рекомендую Open Source мессенджер Mattermost, имеющий тесную интеграцию с GitLab и множеством других систем. Его используем также в системе ИТ курса.

Резюме

Время одиночек давно прошло – программное обеспечение в основном создаются командами. Требуется эффективная передача информации между участниками команды.

Для коммуникации наибольший приоритет отдавайте специализированным инструментам командного взаимодействия.

Языки программирования

Есть два типа языков программирования: те, которые люди постоянно ругают, и те, которыми никто не пользуется.

Описание

Существует множество языков программирования, они имеют различную классификацию:

• языки низкого и высокого уровня;
• компилируемые и интерпретируемые языки;
• различные парадигмы программирования;
• …

Для разработки нативных мобильных приложений доступны, например,
для Android: Java, Kotlin;
для iOS: Objective-C, Swift.

Для разработки серверных приложений спектр языков (и технологий) много шире:
Java, Python, JavaScript, Ruby, PHP, C#, C++, C, Scala, Go и др.

Для создания Web и мобильных приложений помимо прочего бывают необходимы:
HTML, CSS и др.

Рекомендации

Если вы в начале профессионального пути, возникает вопрос – какой язык программирования изучать?

Советую выбрать язык программирования для изучения из числа наиболее распространенных. Эти языки являются наиболее востребованными. Для них представлено наибольшее число вакансий и есть больше возможностей найти работу.

Если вы заглянете в Дополнительные материалы, ты вы увидите ссылку на страницу индекса популярности языков.

Помимо этого, надо учитывать ряд факторов, а именно: платформу, тип ПО, поддержку кросс-платформенности, наличие Open Source библиотек и фреймворков.

Мои рекомендации, в зависимости от факторов:

• Web applications: Java, JavaScript или Python;
• Desktop, Console applications: C++ или Python;
• Android: Java и Kotlin;
• iOS: Swift;
• SQL Databases: SQL.

Но в случае, если вы владеете иным широко распространенным языком программирования, например:

C#, C, PHP, Objective-C, Go, Ruby… – это также хороший выбор, продолжайте его углубленное изучение.

Дополнительные материалы

TIOBE

Индекс популярности языков программирования:
TIOBE Index

ИТ библиотека

По инициативе курсантов в системе курса сформирована страница с ресурсами по информационным технологиям – ИТ библиотека: Обучающие материалы по языкам программирования и ИТ, которые можно использовать в качестве дополнительного образования.

Резюме

В первые годы не нужно браться за молодые или редко используемые языки программирования.

Лучше сконцентрируйтесь на одном основном языке программирования и овладейте базовыми знаниями о вспомогательных языках, тогда в вашем резюме появится новая строчка, например,

Programming languages: Java, JavaScript, SQL

Базы данных

Самые дешевые, быстрые и надежные базы данных те, которых тут нет.

Описание

Базы данных можно разделить на 2 большие группы:

• SQL: Oracle, MS SQL, PostgreSQL, MySQL, MariaDB и др.;

• NoSQL: Apache HBase, Apache Cassandra, MongoDB, CouchDB, Redis, ctOS и др.

Выбор базы данных осуществляется на основании типов хранимых данных, вариантов использования, необходимости масштабирования и пр. факторов.

Несмотря на многообразие баз данных, активно используются в проектах десяток баз данных.

Последние годы Open Source базы данных достойно конкурируют с коммерческими монстрами.

Мейнстримом является область noSQL баз данных и Big Data.

Рекомендации

Даже крайне консервативный финансовый сектор (банки) уже спокойно принимает в качестве SQL базы данных PostgreSQL,

вместо доминирующей ранее Oracle. В реализовываемых нами в последние годы проектах более 50 процентов потребностей закрывает PostgreSQL.

noSQL базы данных обычно непросто установить и сложно администрировать.

Но есть приятные исключения, одним из них является MongoDB.

Мои рекомендации выбора баз данных для изучения и применения:

• SQL database: PostgreSQL;

• noSQL database: MongoDB.

Если вы используете MySQL, рекомендую вместо нее альтернативу MariaDB с оптимизированными движками и потенциальной возможностью кластеризации.

В версии 4 MongoDB появилась поддержка механизма транзакций, что расширяет область применения этой базы данных.

Дополнительные материалы

PostgreSQL

Официальная документация:

PostgreSQL Documentation

Документация по PostgreSQL на русском языке:

Документация к PostgreSQL 11

MariaDB

Официальные материалы для изучения

MariaDB

MongoDB

Официальная документация:

The MongoDB Manual

Резюме

Если вы изучите SQL и овладеете навыками работы с PostgreSQL, то сможете комфортно работать

с любой другой SQL базой данных.

Мощная и простая noSQL база данных MongoDB дает возможность ознакомиться с принципом Map-Reduce, используемом в области Big Data.

После набора знаний и навыков добавьте в резюме строки:

SQL database: PostgreSQL

noSQL database: MongoDB

Open Source

Открытое программное обеспечение

Прием сотрудника в новую Open Source компанию:

Мы еще не решили, что будем разрабатывать, но уже начинаем… Ты с нами?

Описание

Open Source – программное обеспечение с открытым исходным кодом. Исходный код таких программ доступен для просмотра, изучения и изменения. Большинство программ с открытыми исходниками является одновременно свободными, пользователи которого имеют права на его неограниченную установку, запуск, свободное использование, изучение, распространение и изменение (совершенствование), а также распространение копий и результатов изменения.

Фактически, **Open Source** является фундаментом большинства программных проектов, создаваемых разработчиками коммерческого программного обеспечения, от стартапов до крупных предприятий. В недавнем исследовании профессиональные разработчики сообщили, что в среднем 92% их проектов содержат компоненты с открытым исходным кодом. И 68% сообщили, что все их проекты содержат компоненты с открытым исходным кодом. И это неудивительно. Открытый исходный код дает любому, кто пытается разработать новое или внедрить существующее программное обеспечение мощный старт и развитие. Разработчики могут выбирать из множества проектов открытого исходного кода, разработанного и совместно используемого открытым сообществом сотрудников.

Рекомендации

Использование открытого программного обеспечения позволяет использовать огромный потенциал на всех этапах разработки:

• для организации процесса разработки (Redmine, GitLab, Git, Docker, Eclipse и другое);

• в качестве библиотек (Spring, Hibernate, Log4j и многие тысячи других) для разрабатываемого продукта;

• базы данных и хранилища данных (PostgreSQL, MariaDB, MySQL, Apache Cassandra, MongoDB, ElasticSearch и пр.);

• системы и компоненты (Flowable, Activiti, Kibana, Beats, Logstash и др.)

Основными площадками **Open Source** проектов являются GitLab, GitHub и SourceForge.

При выборе **Open Source** следует обращать внимание:

• на лицензию (ПО должно быть также свободным – Apache, MIT, BSD, LGPL, Eclipse Public License и некоторые другие лицензии);

• на активность проекта (ведется ли разработка и насколько активно, как часто выпускаются релизы, сколько активных контрибьюторов);

• на документацию (имеется ли документация или придется изначально "читать" код, насколько она актуальна, понятна и детализирована);

• на сообщество (число форков разработчиками и закачек ПО пользователями);

• на техподдержку (имеется ли форумы или чаты, на которых можно получить ответы на возникающие вопросы и решить проблемы).

Выбор "мертвого" проекта следует избегать – вы получите большие издержки на его внедрение без последующих перспектив. Использование же развивающегося проекта снизит ваши риски и позволит в будущем воспользоваться новыми наработками.

Для разработчика хороший **Open Source** код является дополнительным учебным материалом – проводя ревью кода вы можете улучшить свои знания языка программирования и изучить применение шаблонов проектирования.

Участие в **Open Source** проекте в качестве контрибьютора или тестера позволяет получить хороший опыт командной распределенной разработки.

Дополнительные материалы

Веб ресурсы, содержащие **Open Source** проекты:

SourceForge

GitLab Explore

GitHub Explore

Если вы планируете запустить свой **Open Source** проект, прочтите книгу:

Producing Open Source Software, Karl Fogel (eng)

Создание Свободного Программного Обеспечения, Карл Фогель (ru – частично переведена)

Если вы хотите подробнее узнать об Open Source лицензиях:

Лицензии

Резюме

Разработка программного обеспечения без использования **Open Source** в настоящее время немыслима. Но следует внимательно подходить к выбору свободного программного обеспечения, чтобы получить преимущества при ее внедрении и использовании.

Управление версиями

Версия ПО и сборки

В этой версии программы исправлены старые баги и добавлены новые.

Описание

Программное обеспечение помимо наименования имеет номер версии – под этим номером ПО представлено пользователям.

Формат версии может быть различным, например:

• 3.4 (версия 3.4)

• 6.5.1-C-CE (версия 6.5.1, кандидат в релиз, community edition)

• ULTIMATE 2016.2 (ultimate edition 2 от 2016 года)

Помимо этого, программное обеспечение имеет версию сборки – ее используют разработчики, тестировщики, администраторы, например:

• 3.4.8 (сборка 3.4.8)

• 128 (сборка 128)

• U-162.18.17 (сборка 162.18.17, ultimate edition)

Таким образом в версиях программного обеспечения и сборки могут участвовать числа, дата, символы, либо их комбинация, в которых отражены порядковый номер, дата выпуска или сборки, редакция ПО.

Рекомендации

Рекомендую 3-х числовое обозначение номера версии ПО, например:

2.14.1 (версия 2.14.1)

Первое число (2) указывает на порядковый номер мажорного (значимого, существенного) изменения программного обеспечения, зачастую не совместимого с предыдущей версией.

Второе число (14) указывает на порядковый номер минорного изменения ПО – небольшого изменения с добавлением функциональности и/или исправлением багов.

Третье число (1), это порядковый номер пакета исправления (hotfix) – исправление критических ошибок, обнаруженных в ранее выпущенном ПО.

Номер сборки будет присваиваться автоматически при создании сборки.

В общем случае версионирование и командная работа по непосредственной реализации ПО может строиться с условным разбиением на 4 этапа

(основное будет подробнее раскрыто в последующих материалах).

Этап инициализации:

На старте проекта по разработке ПО ему присваивается версия 0.1.0.

В системе управления проектами уже существует проект.

Создается проект в системе управления версиями.

На этом этапе работу ведут 1-2 человека в одной ветке, предназначенной для разработки.

Формируется структура проекта ПО, настраиваются конфигурационные файлы, добавляются классы, реализующие основу проекта.

Настраивается интеграционный сервер. Запускаются первые сборки.

Настраивается тестовый сервер, на него устанавливается 0.1.0 версия.

Данный этап длится обычно до 10 рабочих дней.

Этап реализации:

Выбирается период итерации (обычно от 1 до 4 недель).

С началом новой итерации разработки наращивается минорная версия ПО (0.2.0, 0.3.0 и т.д.).

Разработка фич и багфиксинг (разработчикам на заметку: "Это не баг – это незадокументированная фича") разработчиками проводятся в отдельных ветках.

На данном этапе к проекту могут быть подключены тестировщики и остальные разработчики.

Данный этап продолжается до первого релиза на продакшен, может длиться несколько месяцев.

Этап реализации и поддержки:

Номер первого продакшен релиза устанавливается как 1.0.0.

Дальнейшее присваивание версий с привязкой к итерациям разработки согласно соглашению.

Команда продолжает наращивать функционал, осуществлять багфиксинг.

Помимо этого, правятся критичные баги путем формирования патчей (пакета исправлений).

Этап поддержки:

На этой фазе в основном идет багфиксинг, редко добавление функций. Ввиду этого идет наращивание только минорной версии или пакета исправлений.

Команда может сократиться до 1-2 человек.

Дополнительные материалы

Подробнее о Семантическом Версионировании:

Семантическое Версионирование

Резюме

Каждое программное обеспечение имеет свой формат версий, важно понимать, что в нем заключен определенный значимый смысл.

Формат и правила версионирования устанавливаются на уровне компании/команды/проекта.

Версия ПО и сборки является уникальным идентификатором артефакта, которым оперируют на всех жизненных этапах программного обеспечения. На версию ПО и сборки ссылаются при реализации, тестировании, документировании, поддержке ПО.

После изучения документа **Семантическое Версионирование** добавьте в резюме:

Version control: Semantic Versioning

Система управления версиями

Version Control System

– Почему ваши дети все время ссорятся?

– Конфликт версий, – отвечает программист.

Описание

Современная разработка программного кода в одиночку или в команде не мыслима без использования системы управления версиями. Использование системы управление версиями значительно повышает эффективность одиночной работы, и многократно – при работе над проектом в команде.

Кратко поясним основные возможности системы управления версиями:

• **Система управления версиями** позволяет в ходе кодирования сохранять изменения кода, с возможностью вернуть любой ранее зафиксированный срез данных – это означает, что, если при кодировании была внесена ошибка, можно оперативно вернуть работающий код.

• В **VCS** можно создавать ветки (копировать срез данных) для нового разрабатываемого кода, для параллельного изменения кода, управлять ветками (после изменения данных сливать ветки и удалять при ненадобности).

• При командной разработке система управления версиями при возможности автоматически объединяет версии кода или указывает на возникающие конфликты слияний кода и помогает их разрешить – в случае, если один участок кода меняет одновременно несколько человек, то это будет зафиксировано системой и можно будет выбрать нужный вариант кода.

• Система управления версиями является своего рода резервным хранилищем – если возникли проблемы с локальным диском, то можно вернуть ранее сохраненные на сервере или на других компьютерах разработчиков данные.

• Общедоступные версионные хранилища выступают в качестве источника Open Source проектов для изучения, для практического применения в реальных проектах.

• Кроме того, версионное хранилище с открытыми проектами, в которых принимали участие, можно предоставить для презентации компетентному заказчику или работодателю.

Инструменты

Существуют множество систем контроля версий, призванных решать задачи управления версиями:

SVN, CVS, Git, Mercurial, Bazaar, Darcs и др.

Некоторые из них являются устаревшими, с более ограниченными возможностями.

Современные системы различаются некоторыми функциональными возможностями, наличием средств интеграции с системами разработки, степенью распространенности среди разработчиков, в разной степени документированы, потому ее выбор является важным фактором.

Рекомендации

В качестве системы управления версиями рекомендую **Git**.

Эта система распределенная, имеет мощный, гибкий функционал, широко распространена, интегрируется со многими системами разработки и хорошо документирована.

Мое мнение подтверждают результаты исследований лучших VCS систем.

Значимый фактор, наличие помимо консольного клиента дополнительных инструментов для этой системы контроля версий:

• Графические клиенты с удобными функциями управления и просмотра: git-gui, SmartGit, TortoiseGit и др.

• Web серверы, расширяющие возможности Git функциями администрирования, просмотра, доступа, ревью через веб-интерфейс: GitHub, GitLab.

Приоритет отдаю Open Source web серверу **GitLab**.

Основные варианты и нюансы их использования будут подробно раскрыты в дополнительных материалах.

Дополнительные материалы

Результаты исследований лучших VCS систем:

Best Version Control Systems

Git

Книга по системе Git:

Git Book

GitLab

Краткое описание:

Git на сервере – GitLab

Официальная документация:

Help

Резюме

Изучите для начала базовые возможности и получите практические навыки работы с Git и функциональностью GitLab по управлению версиями.

Обязательно укажите в резюме полученную компетенцию:

Version control: Git, GitLab

Модель ветвления

GitLab Flow

Жена спрашивает мужа-программиста: – Дорогой, почему до свадьбы ты был такой хороший и вежливый? Помню, даже цветы мне дарил…

– Ну, милая, то была demo-version…

Описание

Git представляет собой инструмент для управления версиями. Модель ветвления и правила управления версиями устанавливаются на уровне компании/команды/проекта. Для git существуют несколько моделей ветвления, но основные: Git Flow, GitHub Flow, GitLab Flow.

Рекомендации

Часто следуют простым моделям, типа GitHub Flow, но более полными являются Git Flow и GitLab Flow.

Рекомендую GitLab Flow, она несколько проще модели Git Flow, но исключает возможные проблемы, присущие простым моделям.

Помимо этого, можно будет воспользоваться функциональностью, предоставляемой системой GitLab.

Этап инициализации:

На этом этапе работа по формированию структуры проекта ведется в master – основной ветки разработки.

Этап реализации:

Приступая к работе над задачей, создайте новую ветку от ветки master. Её название должно начинаться с номера тикета (в нашем случае номер тикета в системе GitLab), например:

– 268-loading-categories (тикет 268, загрузка категорий)

По завершении работы после ревью ветка мержится назад в master.

Этапы реализации и поддержки/этап поддержки:

От основной master ветки со стабильным кодом создается production ветка для выпуска кода в продакшен.

Реализация фич и багфиксинг ведется по-прежнему в отдельных ветках.

Обращаю внимание, что номер версии ПО может присутствовать в наименовании веток и тегов в системе управлении версиями, например:

– release-2.14.0 (ветка релиза версии 2.14.0)

– 5346-hotfix-2.14.1 (тикет 5346, ветка пакета исправления 2.14.1)

– release-2.14.1 (ветка релиза с патчем версии 2.14.1)

Полное описание GitLab Flow и других моделей доступно по ссылкам в дополнительных материалах.

Дополнительные материалы

GitLab Flow

Модель ветвления для GitLab.

GitLab Flow

Перевод

GitHub Flow

Для ознакомления, простая модель ветвления GitHub Flow.

GitHub Flow

Git Flow

Для ознакомления, полная модель ветвления для git.

Git Flow: A successful Git branching model

Перевод: Удачная модель ветвления для Git

Резюме

На старте проекта или для одиночного проекта вы можете использовать простую модель ветвления типа GitHub Flow.

GitLab Flow обеспечивает наиболее полную и оптимальную модель ветвления при управлении версиями ПО в командном режиме, уменьшает число возможных конфликтов при мерже версий, позволяет с наименьшими затратами поддерживать рабочую версию ПО.

В итоге можно дополнить резюме:

Version control: Git, GitLab, GitLab Flow

Качество кода

Стандарт кодирования

Coding standart, **coding convention**, **code style** или **programming style**

Всегда пиши код так, как будто человек, который будет его саппортить – психопат-убийца, который знает, где ты живешь.

Описание

Под стандартом кодирования принято понимать свод правил и соглашений, которые используются (или неблагополучно игнорируются :с) при написании исходного текста на определённом языке программирования.

Стандарт кодирования может описывать:

• способы выбора названий и используемый регистр символов для имён классов, переменных, констант, методов и других идентификаторов;

• стиль отступов при оформлении логических блоков, ширину отступа;

• способ расстановки скобок, ограничивающих логические блоки;

• использование пробелов при оформлении логических и арифметических выражений;

• стиль комментариев и использование документирующих комментариев.

• многое другое…

Стандарт кодирования улучшает качество кода и решает следующие задачи:

• облегчает написание и поддержку кода: проще понимать и редактировать стандартизированный код;

• уменьшает число ошибок при кодировании;

• облегчает проведение ревью кода;

• упрощает управление версиями кода, слияние кода в большем числе случаев проходит автоматически.

Таким образом стандарт кодирования улучшает качество кода и программного продукта.

Но к сожалению, не во всех компаниях ему следуют, что приводит к ряду проблем:

• Ухудшается визуальное восприятие кода, разработчик быстрее устает;

• Чаще возникают ошибки;

• Больше времени тратится на ревью кода;

• Управление версиями кода усложняется, превращается в ад на больших проектах.

В результате отсутствия стандарта качество кода и продукта ухудшается, стоимость поддержки проекта значительно возрастает.

Рекомендации

Для каждого языка программирования может существовать более одного стандарта, важно чтобы в компании или на проекте придерживались одного из них.

Стандарт кодирования проще изучать на примерах кода или в процессе **Рецензирования кода**.

Желательно, чтобы среда разработки позволяла форматировать код согласно стандарту кодирования. Если редактор кода не имеет функций формата, не позволяет проводить базовый **Рефакторинг** кода, имеет смысл его поменять.

Хорошей привычкой является форматирование кода после завершения работы над классом, блоком кода, перед коммитом в **Систему управления версиями**.

Желательно, чтобы **Непрерывная интеграция** на интеграционном сервере проводила проверки стилистики и качества кода.

Дополнительные материалы

Если вы программируете на Java, знайте, что широко распространен
Java Code Conventions
Его часто используют с некоторыми дополнениями или изменениями:
Рекомендации к стилю кода
В качестве альтернативы можно использовать стиль кодирования Java, принятый в Google:
Google Java Style Guide
Стиль кодирования C++, принятый в Google:
Google C++ Style Guide

Резюме

Первое, что бросается в глаза на ревью кода, это формат и стилистика кода (или их отсутствие).

В любых проектах соблюдайте стандарт кодирования.

Не всегда в резюме пишут о стандарте кодирования (опытные разработчики предполагают, что это аксиома), но для юниора это будет еще одним плюсом.

Добавьте в резюме стандарт кодирования (только один для каждого языка программирования!), которому вы следуете, например:

Code quality: coding standart (Code Conventions for the Java Programming Language)

Рецензирование кода

Code review

Плохой код на самом деле не плохой. Его просто не так поняли.

Описание

Code review – инженерная практика в разработке. Это анализ (инспекция) кода с целью выявить ошибки, недочеты, расхождения в соответствии написанного кода и поставленной задачи.

К очевидным плюсам этой практики можно отнести:

• Обнаружение ошибок в реализации – опечатки, логические ошибки, ошибки дизайна кода правятся по результатам рецензирования кода.

• Повышение степени совместного владения кодом – участники команды лучше узнают код программы, что в дальнейшем упрощает разработку.

• Разделение ответственности – разработчик, принимая код на ревью от другого разработчика, разделяет с ним ответственность, что повышает сплоченность команды.

• Эффективный метод взаимного обучения – учиться не только тот, чей код просматривают, но и сам ревьювер.

Инструменты

Инструменты, используемые для ревью кода:

TFS, Atlassian Stash, GitHub, GitLab, Gerrit, Upsource и др.

Рекомендации

Рецензирование кода проводится путем просмотра вносимых изменений (в некоторых случаях не в лучшую сторону) в код программы и формирования замечаний по необходимости. В простом случае это осуществляется путем анализа коммитов в системе управления версиями и написанием сообщений в чат.

Но эффективнее использовать инструменты для ревью. Специализированные инструменты, например, Upsource, предоставляют удобные средства для этого. Но даже интегрированных в систему **GitLab** средств для **code review** достаточно для эффективного проведения ревью.

Ревью кода – это важный этап в процессе разработки. Обычно он осуществляется перед мержем ветки задач или багфикса в основную ветку разработки.

Рецензирование кода может быть централизованным (ревью осуществляет всегда один более опытный разработчик, а свой код может отдать на ревью любому разработчику), перекрестным (ревью проводится произвольным программистом). Централизованный подход применим, например, в команде, если участники имеют сильно разный уровень подготовки. В ином случае эффективен перекрестный подход.

Я при возможности отдаю свой код на рецензирование другому участнику команды и зачастую получаю ценные замечания и указания на ошибки. При проведении ревью нацелен не только на поиск недочетов, но и беру на заметку удачные и интересные реализации классов, методов, изучаю использование незнакомых для меня технологий и библиотек.

Дополнительные материалы

Статья, более подробно раскрывает назначение рецензирования кода:

Эффективные ревью кода: 9 советов от исправившегося скептика

Статья, с примерным чек-листом для проведения ревью кода:

Уменьшаем количество ошибок с помощью чек-листа Code Review

Вернитесь к статье по GitLab Flow с акцентом на мерж-реквест и ревью кода:

GitLab Flow

Перевод

Резюме

Code review улучшает качество кода.

Рецензирование кода – это хороший метод обучения программированию.

После получения навыка ревью сможете дополнить резюме:

Code quality: coding standart (Code Conventions for the Java Programming Language), code review

Рефакторинг

Refactoring

У программиста как у повара – первая версия программы всегда комом. Потому совсем не важно, чтобы получилось хорошо с первого раза. Важно, чтобы хорошо получилось с последнего.

Описание

Refactoring – инженерная практика в разработке. Это изменения исходного кода без изменения функциональности для улучшения внутреннего качества (упрощение кода, повышение гибкости архитектуры, устранение дублирования кода и тому подобное). Для проведения переработки кода желательно знать «запахи кода» и непосредственно приемы рефакторинга.

Инструменты

В современных средствах разработки встраивают механизмы для проведения рефакторинга, анализаторы с подсказками (например, при "Выделении метода" идет автоматическое обнаружение дублирующего кода с предложением провести в них также "Выделении метода").

Рекомендации

Рефакторинг кода зачастую является необходимым условием выполнения задачи (осуществляется в самом начале или в ходе процесса добавления функциональности или багфиксинга).

Реже, если рефакторинг предполагает значительное изменение структуры программы, он может быть вынесен в отдельную задачу.

Для эффективного и быстрого рефакторинга требуется соблюдения ряда методик:

• **Шаблон проектирования** – часто показывает, какой результат должен быть после рефакторинга.

• **Стандарт кодирования** – упрощает проведение рефакторинга.

• **Рецензирования кода** – результатом рецензирования может быть предложение проведения рефакторинга.

• Коллективное владение – каждый может улучшить любой фрагмент кода.

• **Модульное тестирование** и **Интеграционное тестирование** – наличие тестов позволяет быстро проконтролировать рабочее состояние кода после рефакторинга.

• **Непрерывная интеграция** позволяет проверить сборку, метрики кода, все тесты проекта.

В случае отсутствия стандарта кодирования, модульных тестов, рефакторинг значительно затрудняется или становится невозможным.

Если используемая вами IDE или редактор не имеет механизма рефакторинга, стоит его заменить. Например, редактор Vim с подсветкой кода можно использовать для изучения синтаксиса языка, учебной разработки, но для профессиональной разработки этого мало.

Десяток лет назад я познакомился с парнем, который лихо строчил код C++ в Vim. На мой вопрос "Как поступаешь, если требуется переименовать класс?" получил ответ "А зачем?". Он давно возмужал и стал профи, пользуется профессиональной IDE и на своем опыте получил ответ на свой вопрос.

Дополнительные материалы

Описание назначения и видов рефакторинга простым языком:

Рефакторинг

Книга, подробно раскрывает признаки необходимости переработки кода и приемы рефакторинга:

"Рефакторинг. Улучшение существующего кода" Мартин Фаулер, Кент Бек, Джон Брант, Дон Робертс, Уильям Апдайк

Примеры рефакторинга на Java, но принципы применимы для любых языков программирования.

Как пишут в предисловии, это книга для тех:

• Если вы хотите понять, что такое рефакторинг.

• Если вы хотите понять, для чего следует производить рефакторинг.

• Если вы хотите узнать, что должно подлежать рефакторингу.

• Если вы хотите реально заняться рефакторингом.

Могу добавить, что эта книга:

• Показывает принципы применения ООП.

• Описывает некоторые паттерны GoF (**Шаблон проектирования**).

• Содержит **UML** диаграммы.

• Описывает, как осуществлять **Модульное тестирование**.

Резюме

Refactoring улучшает качество кода.

Рефакторинг – это неотъемлемая обязательная практика, используемая при кодировании.

После получения навыка рефакторинга сможете дополнить резюме:

Code quality: coding standart (Code Conventions for the Java Programming Language), code review, refactoring.

Тестирование программного обеспечения

Тестирование ПО

Не волнуйся, если не работает. Если бы все всегда работало, у тебя бы не было работы.

Описание

Тестирование – процесс исследования ПО с целью получения информации о качестве продукта.

Существует множество классификаций видов тестирования. Для разработчиков наибольшее значение имеет классификация по степени изолированности компонентов:

- Компонентное (модульное) тестирование (component/unit testing)
- Интеграционное тестирование (integration testing)
- Системное тестирование (system/end-to-end testing)
- Тестирование игры, где ты прыгаешь в 1 стену 2 часа чтобы найти баг.

Модульное и интеграционное тестирование – это область прямой ответственности разработчиков ПО.

Обычно разработчики первичное системное тестирование разработанной функциональности проводят самостоятельно путем запуска программ, а далее системное тестирование выполняют сотрудники тестового отдела.

Иногда разработчики принимают участие в других видах тестирования.

Рекомендации

Модульное и интеграционное тестирование – это важные компетенции разработчика программного обеспечения.

Создание модульных и интеграционных тестов рекомендую проводить в процессе разработки программного обеспечения.

Следует изучить и получить навыки создания юнит-тестов, интеграционных тестов.

Модульное и интеграционное тестирование необходимо автоматизировать, для этого предназначены методики и инструменты, которые рассмотрим в следующих материалах.

Дополнительные материалы

Статья описывает различные виды тестирования программного обеспечения:

Тестирование программного обеспечения

Резюме

Не следует считать, что тестированием занимаются только тестировщики.

Определенные виды тестов – модульные и интеграционные, следует создавать именно разработчикам.

Модульное тестирование

Unit testing (**юнит-тестирование**)

Достаточно сложно найти ошибку в вашем коде, когда вы ее ищете. Это еще сложнее, если вы предполагаете, что ваш код не содержит ошибок.

Описание

Unit testing – инженерная практика тестирования в разработке, осуществляемая программистом. Это тестирование путем предварительного написания модульных тестов для отдельных модулей программы – классов, их методов.

Цель юнит-тестирования – изолировать отдельные части программы и показать, что по-отдельности эти части работоспособны.

Отличным навыком является, при реализации классов, написание юнит-тестов на наиболее часто используемые варианты использования классов и на граничные случаи использования.

В методологиях разработки **XP (Extreme Programming)** и TDD (Test-driven development) данная практика продвигается как разработка через тестирование (сначала пишутся тесты, затем функционал).

К очевидным плюсам модульного тестирования можно отнести:

• Раннее обнаружение ошибок при реализации – юнит-тесты отлавливают большую часть ошибок во время реализации классов, задолго до того, как можно будет запустить программу для функционального тестирования.

• Быстрая проверка работоспособности модулей – автоматический прогон тестов в ходе непрерывной интеграции позволяет быстро проверить работоспособность отдельных частей кода.

• Упрощение изменения кода – ручная и автоматическая проверка юнит-тестов позволяет проводить **Рефакторинг** (**refactoring**) кода без боязни что-нибудь сломать в программе.

• Формирование хорошей архитектуры – неочевидным "побочным" эффектом от внедрения юнит-тестирования является необходимость поддержания хорошей архитектуры (разделение реализации и интерфейсов, обеспечение низкой связности и высокого зацепления модулей и пр.).

• Документирование кода – является своеобразным способом документирования, показывающим возможные тест кейсы (варианты) использования классов.

Инструменты

Для модульного тестирования предназначены фреймворки семейства xUnit для разных языков программирования:

• JUnit для Java;

• gtest для C++;

• NUnit для .NET;

• phpUnit для PHP;

• и т. д.

Каждый язык программирования и библиотека может рекомендовать свои соглашения о наименовании тестовых классов и тестовых методов, но общий принцип модульного тестирования совпадает.

Помимо **xUnit** существуют специализированные библиотеки для модульного тестирования. Например, библиотеки для создания моков (mock) и стабов (stub), позволяющие изолировать тестируемый класс и подменять внешние классы.

Современное средство разработки имеет хорошую интеграцию с библиотекой **xUnit** тестирования и зачастую графически отображает результаты тестов – показывает зеленые успешные тесты и красные "упавшие" тесты с комментариями.

Рекомендации

Какой подход вы бы не выбрали – юнит-тестирование с разработкой или разработку через тестирование, при грамотном применении это будет являться вашим гарантом реализации работоспособного модуля.

В первое время разработки данная практика может незначительно задержать разработку, за счет необходимости получения навыков и создания инфраструктуры (настройки, создания вспомогательных классов для модульного тестирования). Но чем больше и продолжительней проект, чем чаще вы используете модульное тестирование, тем большая выгода от его применения.

Требование слияния временной ветки разработки в основную ветку разработки только при наличии юнит-тестов для разрабатываемых классов считаю вполне обоснованным, если команда придерживается данной практики.

К сожалению, не всегда этой практики следуют даже опытные разработчики, т. к. не получили требуемых навыков в работе. Вам выпадает отличный шанс получить конкурентное преимущество еще на позиции юниора.

Дополнительные материалы

Статья, раскрывает общий смысл и особенности модульного тестирования:

Юнит-тестирование для чайников

Примеры в статье юнит-тестов на .NET, для вашего языка программирования синтаксис и соглашение об наименовании будут отличаться. Я не призываю полностью понять суть приводимый примеров, главное осознать важность модульного тестирования.

Резюме

Unit testing помогает качественно разрабатывать и облегчает поддержку программного обеспечения.

Владение технологией и практикой модульного тестирования – важный навык профессионального разработчика.

Зрелые ИТ компании с хорошо поставленным процессом разработки не отложат в сторону резюме с данным навыком:

Testing: Unit testing

Интеграционное тестирование

Integration testing

Плохое ПО одного человека – постоянная работа другого.

Описание

Integration testing – тестирование, осуществляемое разработчиком, логически связанных модулей или тестирование модуля во взаимодействии с внешней системой, например, базой данных.

Технически написание интеграционных тестов схоже с написанием модульных тестов. Но, в отличие от юнит-тестов, нет изолированности – тесты затрагивают не один, а ряд классов, модулей и, возможно, другие системы. Обычно требуется отдельная конфигурация приложения для интеграционного тестирования.

Признаки, когда имеет смысл применить интеграционное тестирование:

• Когда нет возможности изолировать и протестировать функциональность одного модуля.

• Когда не имеет смысла изолировать функциональность. Например, при тестировании Data Access Object нет смысла изолировать уровень доступа к базе данных – запросы к базе данных это основная бизнес-логика уровня DAO.

• Когда имеет смысл проверить работу логически связанных модулей – когда на выходе получаем ясные и предсказуемые результаты, которые можно покрыть тестами.

Инструменты

Для интеграционного тестирования применяют фреймворки семейства **xUnit** и специализированные библиотеки тестирования.

Для каждого языка программирования существует свой набор инструментов для тестирования.

Но есть ряд библиотек, имеющих реализацию на многих языках, например, ранее озвученные библиотеки семейства **xUnit** или Hamcrest.

Для Java хорошая подборка описана в статье:

12 инструментов для интеграционных и unit-тестов в Java,

среди которых хотел бы отметить:

• Mockito

• Hamcrest

Этим списком набор инструментов тестирования конечно же не ограничивается.

Рекомендации

Для тестирования уровня доступа к базе данных (Data Access Object – DAO) можно применить следующие тактики:

• зачастую вместо "промышленной" БД можно подключить легкую встроенную базу данных типа H2.

• использовать специализированную библиотеку (например, в Java библиотеку DbUnit – расширение для JUnit), которая может быть использована для инициализации БД в известное состояние, перед выполнением каждого интеграционного теста и заполнения БД нужными данными.

Для интеграционного тестирования, также как и для модульного, может понадобиться реализация вспомогательной инфраструктуры, но затраченное время на средних и больших проектах оправдано.

Если время выполнения модульного теста редко превышает секунду, то интеграционный тест в некоторых случаях может выполняться продолжительное время (порядка десятка секунд). А когда общее время прогона всех тестов превышает десяток минут, то в таком случае

полный прогон имеет смысл осуществлять на интеграционном сервере. Помимо этого, применяют ряд мер по оптимизации тестов.

Резюме

Integration testing является следующим уровнем проверки функционала после модульного тестирования, позволяет проверить работу нескольких модулей или их взаимодействие с внешней системой.

После практики добавится навык:

Testing: Unit testing, Integration testing

Интеграция и поставка программного обеспечения

Непрерывная интеграция

Continuous Integration

Электронные мозги ошибаются гораздо точнее…

Описание

Continuous Integration (**CI**) – это практика разработки ПО, при котором на сервере интеграции регулярно автоматизировано выполняется ряд операций над проектом, результатом чего являются отчеты и готовое для установки программное обеспечение.

Список операций **Continuous Integration** может варьироваться в зависимости от цели, от используемого сервера интеграции и его настроек. Полный список этапов непрерывной интеграции:

• скачивается код из **Системы управления версиями** (по одному из событий – например, изменению кода или нажатию кнопки);

• подготавливаются файлы проекта для сборки;

• компилируется код (в случае компилируемого языка программирования);

• проводится **Модульное тестирование**;

• собирается проект;

• проводится **Интеграционное тестирование**;

• проводится функциональное, нагрузочное и прочее тестирование;

• проверяются метрики (например, проверяется **Стандарт кодирования**, проводится автоматический поиск багов и т. п.);

• рассчитывается статистика (например, статистика покрытия кода тестами и пр.)

• строятся отчеты (графические, табличные, текстовые по тестам, метрикам, статистике);

• отсылаются нотификации (email, SMS, чат нотификации).

• архивируется код и результаты интеграционной сборки.

В случае возникновения критической проблемы на операции (например, "падение" модульного теста), процесс прерывается и высылается нотификация с предупреждением (чаще на email всей команде). При успешной сборке нотификация обычно не отсылается.

Преимущества использования непрерывной интеграции:

• раннее обнаружение ошибок и проблем в ПО;

• регулярный полный прогон модульных и интеграционных тестов;

• наличие метрик и статистики по коду;

• постоянное наличие текущей стабильной версии ПО для деплоймента (установки).

Минусы также имеются (но значимость плюсов их перекрывает):

• цена оборудования/хостинга/облака интеграционного сервера;

• трудозатраты на настройку и поддержку системы;

• дополнительные правила и активности процесса разработки.

Инструменты

В настоящее время есть выбор между множеством серверов непрерывной интеграции:

• GitLab CI/CD, Bitbucket Pipelines – системы интеграционной сборки как дополнение для веб-систем контроля версий, предоставляют широкие возможности.

• Travis CI, Buildbot, Strider-cd, GoCD – простые, но самодостаточные интеграционные сервера с основной функциональностью.

• Jenkins, Hudson, TeamCity, Bamboo – интеграционные сервера, имеющие расширенные функциональные возможности.

Рекомендации

Какой бы интеграционный сервер вы не использовали, при грамотном встраивании в процесс разработки это улучшит качество ПО и оптимизирует процесс разработки.

В командной разработке CI используем более 10 лет. Если встает вопрос выбора интеграционного сервера, рекомендую 2 варианта, что вполне согласуется с результатами аналитики – оба продукта входит в число лидеров The Forrester Wave™: Continuous Integration Tools, Q3 2017:

• **Jenkins –** Open Source, наличие огромного количества плагинов, позволяющих значительно расширить функциональность и гибко настроить интеграционную сборку для любого проекта.

• **GitLab CI/CD –** в 2017-2018 годах команда GitLab много времени и сил вложила в создание инструментария для CI/CD автоматизации – это очень хороший вектор развития продукта и достойный кандидат на внедрение в процессы компании (что означает CD вы узнаете в следующей статье).

Вам не потребуется устанавливать сервер непрерывной интеграции, настраивать и встраивать непрерывную интеграцию в процесс разработки в начале своей карьеры. Но есть необходимость следовать установленным правилам в компании/команде и использовать сервер непрерывной интеграции на пользовательском уровне.

Правил обычно немного, например:

• Нотификация системы интеграционной сборки – это сигнал команде немедленно проверить метрики интеграционной сборки проекта и устранить причину нотификации.

• Кто "ломает" сборку (например, проект не собирается или падает тест), тот должен ее в кратчайшие сроки починить. Но если необходимо, сборку исправляет любой участник команды.

• Checkstyle Warnings, FindBugs Warnings и прочие предупреждения должны быть в наиболее короткие сроки исправлены.

• Слияние кода разработчиком в ветку, из которой скачивается код для интеграционной сборки, это повод проверить метрики проекта на интеграционном сервере.

Дополнительные материалы

Для чего программисту Continuous Integration и с чего начинать

Еще в одной статье неплохо описано, какие проблемы решает CI:

Непрерывная интеграция в Селектеле

Изначально в Селектеле выбрали коммерческий TeamCity. Затем стали использовать Jenkins, об этом статья.

Jenkins

Официальная документация:

Jenkins Documentation

Резюме

После того, как поймете назначение **Continuous Integration** и ознакомитесь с основной функциональностью сервера непрерывной интеграции, укажите в резюме:

Continuous Integration: Jenkins

Непрерывная поставка

Continuous Delivery

Пора бы уже и врачам взять на вооружение золотое правило программистов: "Если оно работает – лучше не трогай!".

Описание

Continuous Delivery (**CD**) – это продолжение практики **Continuous Integration** разработки ПО, при котором готовится рабочее окружение, собранный проект конфигурируется и поставляется на сервер тестирования в автоматизированном режиме, а в дальнейшем на продакшен (рабочий сервер).

Список операций **Continuous Delivery** расширяет список **Continuous Integration** и может включать в себя как автоматические, так и ручные операции:

• подготовка рабочего окружения – например: операционной системы, баз данных, дополнительного ПО;

• конфигурирование, настройка разрабатываемого программного обеспечения;

• деплоймент (развертывание) на тестовые сервера;

• прогон автоматических тестов (UI, нагрузочных и др.);

• ручное тестирование;

• процесс принятия или отказ от деплоймента на продакшен;

• деплоймент на производственные сервера.

Основным принципом Continuous Delivery является быстрая поставка программного обеспечения пользователям. Чем быстрее пользователь получит доступ к новым баг-фиксам, исправляющим ошибки или новому функционалу, тем быстрее пользователи смогут этим воспользоваться. В итоге могут быть собраны отзывы пользователи, на основе которых планируется последующая работа, новая функциональность. Появляется возможность быстрой адаптации программы под меняющиеся бизнес требования.

Преимущества **CD**:

• быстрые поставки разрабатываемого ПО пользователям;

• снижение рисков для заказчика;

• быстрая адаптация ПО под обновленные требования;

• включение активных пользователей в процесс тестирования и апробации;

• повышение конкурентоспособности программного продукта.

Недостатки **CD**:

• усложнение процесса разработки на этапе внедрения CD;

• повышение "уровня входа" в разработку (дополнительные технологии/инструменты поставки и развертывания).

Инструменты

В настоящее время имеется ряд серверов, реализующие непрерывную поставку:

• GitLab CI/CD, Jenkins X – полная поддержка CI/CD;

• Bitbucket Pipelines, TeamCity – включают механизмы поставки или расширяемые посредством плагинов.

GitLab уделяет особое внимание функциям CI/CD в своем продукте.

Jenkins X – это вариант Jenkins с более полной реализацией CI/CD.

Летом 2018 компания Microsoft приобрела GitHub. Видимо можно ожидать в ближайшем будущем интеграцию GitHub с Azure и такие функции, как миграция разрабатываемого ПО в это облако. Но эта сделка уменьшила популярность хранилища кода. Тысячи ИТ компаний и разработчиков мигрировали свой код из GitHub в GitLab или Bitbucket.

Рекомендации

Каждая компания/команда создает/адаптирует процессы **CI/CD** под свои проекты. Нет единого эталона или стандарта, необходимо учитывать множество факторов. Для начала достаточно иметь общее представление об этих практиках.

Возможно, вы работаете или вам повезет, и вы станете работать в компании, где **CI/CD** являются финальными ключевыми принципами разработки ПО. В ином случае вы имеете возможность в будущем принять участие во внедрении этих практик и инструментов.

Дополнительные материалы

В статье QA-инженером рассказано о процессе **Continuous Delivery / Continuous Integration** в компании:

Ежедневные релизы – это не так уж страшно

GitLab CI/CD

Официальная документация:

GitLab Continuous Integration (GitLab CI/CD)

Интересно взглянуть на GitLab Community Edition – Pipelines

Jenkins X

Официальная документация:

Jenkins X Documentation

Резюме

Понимание основ **CI/CD** все чаще требуется в ИТ компаниях.

Изучив принципы **Continuous Delivery** поправьте в резюме:

Continuous Integration / Continuous Delivery: GitLab CI/CD

DevOps

Кому и командная строка – дружественный интерфейс

Описание

DevOps – это объединение практик **Continuous Integration**, **Continuous Delivery** разработки ПО, с добавлением механизма непрерывного деплоймента **Continuous Deployment** и мониторинга.

В общий список операций добавляются:

• деплоймент (развертывание) на производственные сервера (в автоматическом режиме);

• мониторинг – мониторинг систем и устройств, сбор статистики и аналитика данных.

Инструменты

Помимо систем CD/CI в поставке программного обеспечения должны использоваться дополнительно различные инструменты, позволяющие подготовить рабочее окружение, развернуть ПО, проводить мониторинг и собрать статистику для последующего анализа:

• системы виртуализации – VirtualBox, Vargant, Docker и другие;

• инструменты для управления конфигурацией – Chef, Puppet, Ansible и SaltStack;

• инструменты для кластерной конфигурации – например, для Docker, Docker Compose это Docker Swarm, Kubernetes или OpenShift.

• системы для мониторинга устройств и систем, сбора и анализа данных – Zabbix, Nagios, Prometheus, Graphite, ELK и пр.

Рекомендации

Если вы не планируете стать специалистом DevOps, то скорее всего вы начнете изучать и использовать инструментарий, который в ходу в компании.

В любом случае советую вам обратить внимание на линейку виртуализации и управления:

Docker, Docker Compose, Docker Swarm или Kubernetes, OpenShift. Docker – программа, позволяющая операционной системе запускать процессы в изолированном окружении на базе специально созданных образов. Docker Compose используется для управления многоконтейнерных приложений. Docker Swarm – это родная кластеризация для Docker. Он превращает пул хостов Docker в единый виртуальный хост Docker. Альтернативу Kubernetes или OpenShift применяют также для построения из контейнеров докера кластера, содержащего различные компоненты системы (пример кластера: база данных, контейнер веб приложения и собственно веб приложение).

Для сбора статистики (в основном данных из логов приложений) и анализа данных, мониторинга работы устройств можно использовать – ELK (ElasticSearch, Logstash, Kibana).

ELK позволяет собирать, обрабатывать и анализировать неструктурированные данные и параметры устройств, визуально представляют их в веб интерфейсе браузера в виде дашбордов с графическими элементами (графиками, гистограммами, таблицами и числовыми панелями).

Технологический стек ELK состоит из ряда компонент:

• компоненты сбора данных (FileBeat, MetricBeat и другие);

• приложение обработки первичных данных и построение индексов (Logstash);

• движок полнотекстового поиска – мощная система хранения и аналитической обработки текста (ElasticSearch);

• веб система конфигурирования дашбордов и отображения данных (Kibana).

Система ELK позволяет проводить анализ собираемой информации и на основе этого отлаживать приложения, оптимизировать настройки компонент системы, мониторить работу устройств.

Дополнительные материалы

Статья освещает проблемы, который решает докер:

Зачем нам Docker?

Резюме

Не так давно появился новый класс специалистов – **DevOps**, которые занимаются в основном внедрением/настройкой инструментария Continuous Integration / Continuous Delivery / Continuous Deployment и поддерживают полный процесс разработки ПО. Но часто сама команда внедряет и использует эти практики в своем проекте.

Если вы желаете развиваться в этом направлении, то вам следует начать изучение Docker, Docker Compose, Docker Swarm – это будет хорошим стартом и тогда в ближайшее время вы сможете расширить свое резюме:

DevOps: Docker, Docker Compose, Docker Swarm

Методологии разработки программного обеспечения

Методологии разработки ПО

Цель компьютерных наук – построить что-то, что простоит хотя бы до того момента, когда мы закончим это строить.

Описание

Для эффективной работы, чтобы делать продукт качественно, вовремя и с минимальными затратами, люди придумывали и использовали разные процессы разработки.

Практики, методики разработки программного обеспечения, со многими из которых вы уже знакомы, объединяются в методологии.

Существуют множество методологий разработки программного обеспечения – они постепенно эволюционируют:

1970 Waterfall
1980 V-Model
1982 CASE
1986 Spiral
1987 PMBOK, Cleanroom
1989 PRINCE2
1991 RAD, CMM
1993 MSF
1994 OOAD, MIL-STD-498
1995 UML, Chaos, Scrum
1997 FDD
1998 IEEE 12207
1999 XP, CI
2001 Agile, AOP
2002 PFE
2003 RUP, TDD, Lean, BDD
2004 DSM
2006 MDD
2009 Continuous Delivery, DevOps
2010 XDSD

Методологии, общими свойствами которых являются короткие итерации, динамическое формирование требований и реализация в самоорганизующейся команде, получили общее название Agile ("гибких") методологий:

Scrum, XP, Kanban, Lean и др.

Agile методологии самодостаточны, но часто взаимодополняют и усиливают друг друга:

- Scrum акцентируется на управленческих методиках.
- XP содержит мощные инженерные практики.
- Kanban упрощает организацию поддержки.
- Lean оптимизирует производство.

В тренде практически ценные для бизнеса молодые методологии:

Continuous Delivery, **DevOps –** методики, нацеленные на тесное сотрудничество отделов разработки, тестирования, эксплуатации и быстрый ввод программного обеспечения в продакшен посредством автоматизации операций.

Любопытны новейшие методологии, например:

XDSD (eXtremely Distributed Software Development) – крайне распределенная разработка программного обеспечения, применяемая распределенными (в пространстве и времени) командами, подразумевает высокий уровень самостоятельности и профессионализма участников.

Рекомендации

Рекомендую сделать акцент и изучить ряд **Agile** методологий и их методики: **Scrum**, **XP**.

Возможно, вы станете работать в компании, где разработка не придерживается строго определенной методологии, но факт, что там будет использоваться ряд методик из списка:

- Небольшие версии
- Коллективное владение
- **Стандарт кодирования**
- Рецензирование кода
- **Рефакторинг**
- **Разработка с тестами/Разработка через тестирование**
- **Непрерывная интеграция**
- **Непрерывная поставка**
- **DevOps**

А если это не так, то для вас открывается широкий горизонт оптимизации процесса разработки в рамках компании.

Весной 2010 года меня пригласили в небольшую компанию из двух десятков человек, специализирующейся на заказной разработке программного обеспечения (бизнес приложения, мобильные приложения и сервера мобильных клиентов и многое другое), на позицию Senior Java developer для создания игрового сервера игры покер. В первые дни я заикнулся о некоторых методиках: модульном, интеграционном тестировании, интеграционной сборке – на что получил ответ, что заказчик не платит за все это… Что мне оставалось делать?

На свой страх и риск поднял на локальном сервере интеграционный сервер, стал по ходу реализации игрового сервера писать модульные, интеграционные, нагрузочные тесты. В течение месяца подключились 2 java-разработчика, флеш-программист начал создавать клиентское приложения для социальных сетей. Через 4 месяца мы успешно опубликовали первую версию в социальной сети "Одноклассники". Смогли бы мы запустить проект без этих методик? Смогли, но много позже…

Еще пару лет я поднимал под проекты интеграционный сервер на своей рабочей станции, провел в компании презентацию по **XP** методикам.

Есть опыт проведения проектов по **Scrum**. В заказной разработке с фиксированной ценой за проект Scrum методология в классическом варианте неприменима, но в модифицированном варианте возможна.

Сейчас в компании в сотню человек, множество методик прочно прижились в компании, проект на интеграционном сервере поднимается в первые дни запуска проекта в производство. Все меняется к лучшему!

Дополнительные материалы

Материалы могут содержать ссылки на книги, прочтение которых может занять недели, но оно того стоит.

Не переживайте, если не все сразу будет понятно, на осмысление ряда методик требуется время.

Книга по **Agile** методологиям разработки:

Гибкие методологии разработки. Вольфсон Борис

Резюме

Выбирайте для проекта гибкую методологию и методики, которые соответствуют реалиям, поэтапно встраивайте их в процесс разработки, следуйте им, адаптируйте их в ходе разработки ПО.

Да пребудет с вами сила :)

После изучения и апробации основных методик из списка **Agile** методологий сможете указать в резюме:

Agile software development

Extreme Programming (XP)

Extreme Programming – Экстремальное программирование

Программирование – на 10% наука, на 20% изобретательность и на 70% попытка заставить изобретательность работать вместе с наукой.

Описание

Extreme Programming одна из Agile методологий.

Именно из **XP** (ни в коем случае не из Windows) методологии почерпнул инженерные практики разработки, которые эффективны и очень актуальны.

Многие методики вам уже знакомы по предыдущим материалам – теперь раскрывается их взаимосвязь.

Методология **XP** базируется на ряде методик:

• **Игра в планирование** (**planning game**) – быстро определяет перечень задач (объем работ), которые необходимо реализовать в следующей версии продукта. Для этого рассматриваются бизнес-приоритеты и технические оценки. Если со временем план перестает соответствовать действительности, происходит обновление плана.

• **Небольшие версии** (**small releases**, иное название – итерации) – самая первая упрощенная версия системы быстро вводится в эксплуатацию, после этого через относительно короткие промежутки времени происходит выпуск версии за версией.

• **Метафора** (**metaphor**) – эта простая общедоступная и общеизвестная история, которая коротко описывает, как работает вся система. Эта история управляет всем процессом разработки.

• **Простой дизайн** (**simple design**) – в каждый момент времени система должна быть спроектирована так просто, как это возможно. Чрезмерная сложность устраняется, как только ее обнаруживают.

• **Модульное** и **Интеграционное тестирование** (**testing**) – программисты постоянно пишут тесты для модулей. Для того чтобы разработка продолжалась, все тесты должны срабатывать. Заказчики пишут тесты, которые демонстрируют работоспособность и завершенность той или иной возможности системы.

• **Рефакторинг** (**refactoring**) – программисты реструктурируют систему, не изменяя при этом ее поведения. При этом они устраняют дублирование кода, улучшают коммуникацию, упрощают код и повышают его гибкость.

• **Программирование парами** (**pair programming**) – весь разрабатываемый код пишется двумя программистами на одном компьютере.

• **Коллективное владение** (**collective ownership**) – в любой момент времени любой член команды может изменить любой код в любом месте системы.

• **Непрерывная интеграция** (**continuous integration**) – система интегрируется и собирается множество раз в день. Это происходит каждый раз, когда завершается решение очередной задачи.

• **40-часовая неделя** (**40-hour week**) – программисты работают не более 40 часов в неделю. Это правило. Никогда нельзя работать сверхурочно две недели подряд.

• **Заказчик на месте разработки** (**on-site customer**) – в состав команды входит реальный живой пользователь системы. Он доступен в течение всего рабочего дня и способен отвечать на вопросы о системе.

• **Стандарты кодирования** (**coding standards**) – программисты пишут весь код в соответствии с правилами, которые обеспечивают коммуникацию при помощи кода.

Extreme Programming, по моему мнению, наиболее интересная и "техническая" **Agile** методология благодаря методикам **Тестирование**, **Рефакторинг**, **Непрерывная интеграция**.

Рекомендации

Не всегда удается применять все методики **XP**, но иногда удается их адаптировать:

• Если заказчик географически удален, невозможно соблюдать принцип Заказчик на месте разработки, в этом случае интересы заказчика может представлять менеджер проекта.

• Не всегда удается следовать 40-часовой недели – на старте проекта и перед сдачей проекта, но основная разработка проводится обычно по рабочему графику.

• Парное программирование недоступно при удаленном разработке в распределенной команде. Но хорошей альтернативой парному программированию является **Рецензирование кода**.

• Вместо разработки через тестирование (сначала тесты, затем функционал), можно применять разработку с тестами (параллельная разработка функционала и тестов).

Дополнительные материалы

Бек Кент, один из авторов методологии **XP**, рассказывает подробно о ней в своей книге.

Бек Кент – **Экстремальное программирование**

Резюме

Даже использование ряда методик значительно усиливает процесс разработки, повышает качество разрабатываемого ПО.

После изучения и практического применения **XP** методик сможете расширить резюме:

Agile software development: XP (Extreme Programming)

Scrum

Первые 90% кода занимают первые 90% времени на разработку… Оставшиеся 10% кода занимают еще 90% времени на разработку.

Описание

Scrum – одна из **Agile** методологий.

Scrum делает акцент на управленческих методиках – решает вопросы управления и организации.

Основные роли (**Core roles**) в методологии скрам:

• **Скрам-мастер** (**Scrum Master**) – следит за соблюдением всех принципов скрама.

• **Владелец продукта** (**Product Owner**) – представляет интересы конечных пользователей и других заинтересованных в продукте сторон.

• **Команда Разработки** (**Development Team**) – кросс-функциональная команда разработчиков проекта (от 3 до 9 человек).

Методология **Scrum** включает следующие артефакты:

• **Журнал пожеланий проекта** (**Product Backlog**) – приоритезированный список имеющихся на данный момент бизнес требований и технических требований к системе.

• **Журнал пожеланий спринта** (**Sprint Backlog**) – содержит функциональность, выбранную **Product Owner** из **Журнала пожеланий проекта** для **Спринта**.

• **Спринт** (**Sprint**) – итерация в скраме, в ходе которой создаётся функциональный рост программного обеспечения. Жёстко фиксирован по времени. Длительность одного спринта от 1 до 4 недель.

• **Диаграмма сгорания задач** (**Burndown chart**) – диаграмма, демонстрирующая количество сделанной и оставшейся работы относительно времени на разработку проекта.

• **Покер планирования** (**Planning Poker**) – техника оценки, основанная на достижении договорённости, главным образом используемая для оценки сложности предстоящей работы или относительного объёма решаемых задач.

• **Планирование спринта** (**Sprint Planning Meeting**) – происходит в начале новой итерации **Спринта**.

• **Ежедневное совещание** (**Daily Scrum meeting**) – короткое совещание, где каждый отвечает на 3 вопроса:

– Что сделано вчера?

– Что будет сделано сегодня?

– С какими проблемами столкнулся?

• **Обзор итогов спринта** (**Sprint review meeting**) – проводится в конце спринта, команда демонстрирует прирост функциональности продукта всем заинтересованным лицам.

• **Ретроспективное совещание** (**Retrospective meeting**) – проводится в конце спринта, команда улучшает процесс разработки.

• ряд других артефактов.

Рекомендации

Scrum является адаптивной методологией, может и должен модифицироваться и комбинироваться с инженерными методиками из **XP**.

Часто используется одна из следующих моделей финансового взаимодействия (Software Development Pricing Models):

• Фиксированная цена (Fixed Price) – заказчик платит за объем работы в разработке ПО. Подходит для минимально жизнеспособного продукта (MVP) или малого проекта с четкой целью и полным описанием, стоимость и масштабы которого прогнозируются.

• Оплата по факту (Time-and-Materials – T&M) – заказчик платит за человеко-часы, потраченные командой подрядчика на разработку ПО. Модель работает для средних и крупных проектов с меняющимися требованиями.

• Выделенная команда (Dedicated team model) – для проекта формируется и выделяется команда. Используется в долгосрочной разработке программного обеспечения, с постоянным потоком ИТ задач.

Scrum хорошо подходит для разработки ПО с оплатой по факту и в выделенной команде.

Формальный подход применения методик приводит к некоторому "разочарованию" в Agile. Важно помнить, что важными критериями является эффективность работы и качество продукта, потому требуется практичный взгляд. Наряду с бизнес требованиями нужно реализовывать и нефункциональные требования программного обеспечения. Классической ошибкой является управление проектом только на уровне спринта и погоня за краткосрочными изменениями. В таком случае быстро накапливается технический долг, что ведет к снижению качества продукта и замедлению разработки. Нужно понимать, что не всем и не всегда подходит Scrum (на это есть ряд объективных причин). Если у вас выбор между командой или Scrum, то ответ очевиден (команда без Scrum работает, Scrum без команды не имеет смысла). Это согласуется с Agile-манифестом разработки программного обеспечения:

• 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

Дополнительные материалы

Henrik Kniberg в книге Scrum and XP from the Trenches (2007) описал опыт применения **Scrum** and **XP**:

Хенрик Книберг – Scrum и XP: заметки с передовой

Резюме

Адаптированный к компании/проекту **Scrum** в комбинации с другими методиками (например, из **XP**) широко применяется при разработке программного обеспечения.

После изучения и практического применения **Scrum** методик на проекте можете расширить резюме:

Agile software development: Scrum

Заключение

Тактика и стратегия ИТ карьеры

Нужно быть лучше, чем вчера, а не лучше, чем другие. Хоть эта стратегия жизни и не самая легкая, зато самая беспроигрышная…

Карьера – самая простая и очевидная цель, для тебя она может заключаться в трудоустройстве в ИТ компании, или для работающих – повышении должности на текущем месте работы.

В случае поиска своей первой работы, ваша первоочередная задача – получить приглашение на собеседование. Для этого вам необходимо выделиться среди тысяч кандидатов.

В вашем арсенале для преимущества должны быть:

• желание и готовность изучать и практиковать информационные технологии.

• хорошее теоретическое знание одного языка программирования и практика программирования на нем;

• специализация – вы должны изучать и практиковать одну область информационных технологий (например: мобильную, фронтэнд или бэкэнд разработку).

• знание базовых технологий и минимальные практические навыки в выбранной специализации.

• понимание процесса командной разработки и базовые теоретические знания методологий разработки.

• базовое владение инструментами разработки и командной работы.

Я хотел бы, чтобы ты заглянул на несколько лет вперед… Повышение ИТ квалификации должно привести к долгосрочной стратегической цели.

Специфика информационных технологий такова, что с большой скоростью постоянно появляются новые технологии. Информационные технологии устаревают и в конце концов исключаются из разработки. Навыки изучения и применения новых технологий, фреймворков, библиотек – самые актуальные для всех разработчиков.

Другой эффективной стратегией является – специализация.

Важно найти свое призвание, и тогда вы будете конкурентноспособны. Подход фул стек разработчика хорош для создания прототипа приложения или минимально жизнеспособного приложения (minimum viable product, MVP), но в долгосрочной перспективе является, по моему мнению, не эффективным. Я сам являюсь отличным бэкэнд разработчиком, но посредственным во фронтэнде (создании UI интерфейса) – и считаю это нормальным (с детства не любил рисование и черчение, плохо воспринимаю дизайн).

Не возможно знать и уметь все одинаково хорошо, у каждого должны быть свои личные предпочтения. Важно только чтобы ваша специализация была востребована на рынке. Например, экспертное владение Фортраном – для настоящего это не актуально.

Для студента и юниора специализация важна, чтобы быстрее достичь востребованного уровня. Если ты будешь распыляться, то на это уйдут многие годы. Специалист среднего уровня может позволить себе эксперементировать, и найти для себя новую нишу. Для эксперта специализация необходима, чтобы постоянно поддерживать свой высокий уровень, который постоянно стремится скатиться вниз.

По аналогии со спортом – ты начинаешь заниматься фитнесом, и нагрузки должны быть дозированными, но постоянными. Твоя выносливость повышается, наращивается мышечная масса, связки укрепляются. Через пару лет ты можешь попробовать силы в

каком-либо виде спорта – бодибилдинге или силовом пауэрлифтинге. Но профессионалы бодибилдеры редко достигают высот в пауэрлифтинге, и наоборот, и это естественно.

Уровень специалиста определяется 3 факторами:

- *теория*
- *практика*
- *опыт*

Теоретические знания формируются при чтении и изучении книг, статей, документации, при прохождении курсов. Большое количество материалов, документации доступно только на английском языке, потому важно знание английского языка. Но все больше появляется качественного материала на русском. Практические навыки, умения приобретаются путем повторения практических примеров, при выполнении практических заданий учебных материалов или интерактивных курсов, на "домашних проектах", при работе над поставленными задачами проекта на работе.

Опыт, это единство знаний и навыков, которые сотрудник получил в результате своей учебной и рабочей деятельности. Интересно, что опыт может быть положительным или отрицательным. И часто отрицательный опыт имеет весомую ценность – ранее допущенные ошибки могут быть осознанно исключены в последующей работе, проектах.

Теория – забывается, требуется периодическое повторение материалов. Знания устаревают – часть информации становится не актуальной и невостребованной.

Требуется постоянная практика, навыки теряются после продолжительного периода. Повторение реализации однотипной задачи через короткий промежуток времени проводится быстро и эффективно. После продолжительного перерыва требуется период восстановления умений.

Опыт – это единственный капитал, который навсегда остается с нами. Все пережитое может быть использовано в последующей работе (в большей или меньшей степени). Опыт программирования с одним языком программирования зачастую облегчает изучение другого языка. Опыт разработки определенных систем может быть применен при проектировании и разработке новых, с вносимыми изменениями и улучшениями. Навыки работы с одной из системы управления проектами упрощает освоения другой системы и т.п.

Эти факторы учитывайте при построении своей карьеры – полезно изучать и опробовать востребованные, не знакомые для себя и новые, часто используемые технологии. И может быть пустой тратой времени изучения экзотических языков программирования, не имеющих практических применений.

Впереди у тебя много нового и интересного.

Я желаю тебе успеха и всего наилучшего!

Курс

В единстве сила!

Получите поддержку ментора и сокурсников. Одноименный интерактивный курс "Компетенции профессиональной разработки программного обеспечения" помимо теоретических материалов содержит практические задания в GitLab, коммуникация с ментором и сокурсниками осуществляется в Mattermost – используются системы, которые применяют в работе сотня тысяч ИТ компаний.

Кратко о курсе

Проект

Практика ведет к совершенству.

Всякая теория должна подкрепляться практикой. Проект с использованием современных технологий, в котором каждый, кто владеет Java на базовой уровне, может принять участие. Требуется знание и готовность следовать установленным правилам процесса разработки программного обеспечения, понимание методологий, базовые навыки владения инструментами разработки на уровне курса "Компетенции профессиональной разработки программного обеспечения". Подробнее о проекте:

Универсальная модульная платформа

Об авторе

Творец книги – автор, творец ее судьбы – общество.
Черемнов Дмитрий Николаевич – технический лидер ИТ компании.
Email: d.cheremnov@asvoip.com
Специализация:
• Консультирование по архитектуре и разработке программного продукта
• Проектирование программного продукта
• Разработка серверного программного обеспечения
• Формирование и оптимизация процесса разработки программного обеспечения
Экспертиза:
• Web и Java технологии (13 лет): REST, Web Services, Microservices, Spring, Hibernate и другие.
• Методологии разработки: Agile, Scrum, Extreme Programming (XP).
• Языки программирования: Java, JavaScript, SQL.
• Базы данных: PostgreSQL, MySql, MariaDB, Oracle, MongoDB и иные.
Проекты:
• Сервисы мобильных платежей
• Банковские системы
• Системы электронной коммерции
• Сервисы для бизнеса
• Информационные базы данных
• Информационно-управляющие системы и другие
Опыт и сертификаты:
• Более 20 лет участвует в проектировании и разработке программного обеспечения.
• Сертификаты (наиболее значимые):
– Sun Certified Programmer for the Java 2 Platform (SCP)
– Oracle PL/SQL Developer Certified Associate (OCA)