1.0 - Introduction

- Информация по УЦ и процессу обучения
- Стек технологий
  - High Level архитектура Web приложения
- Обзор средств разработки
  - Apache Maven
  - Git&Svn
    - Svn
    - Git
  - Tools
  - IntelliJ IDEA
  - Apache Tomcat
  - Code review
  - Continuous Integration/Continuous Delivery(CI/CD)
- Для самостоятельного изучения

Введение в разработку ПО

Информация по УЦ и процессу обучения

Компания Netcracker Technology – мировой лидер в области создания и внедрения комплексных программных решений для провайдеров услуг связи, крупных предприятий и государственных учреждений. Мы разрабатываем программное обеспечение (коммерческие порталы, онлайн системы управления услугами, CRM, системы мониторинга, биллинг, системы поддержки виртулизации, системы по учёту инфраструктуры, визуализации, провижонинга, активации и проч.), которыми ежедневно напрямую или косвенно пользуются миллионы человек по всему миру. К таким системам предъявляются повышенные требования по безопасности, производительности, соответствию международным стандартам, а в разработку таких систем в рамках проекта может быть вовлечено от нескольких десятков до нескольких сотен разработчиков.

Мы - продуктовая компания. У нас большое портфолио собственных продуктов, разработкой, внедрением и развитием которых мы занимаемся. Вместе с тем мир постоянно меняется, на смену одним технологиям приходят новые; успешные в прошлом архитектуры устаревают, появляются новые тренды, потребности, для покрытия которых мы разрабатываем совершенно новые системы.

Учебный Центр Netcracker - это наша школа по подготовке разработчиков (и не только), в которой мы даём общее понимание о процессах разработки программного обеспечения, учим понимать и использовать технологии на практике. Продолжительность обучения в УЦ - 2.5 месяца, что включает в себя 24 3-х часовых занятия (2 занятия в неделю).

Занятия будут кобинированные: лекция + практическая часть по прочитанному материалу. Итоговая задача обучающихся - написать веб-приложение, пройдя все этапы от фронт-энда до бэк-энда и моделирования базы данных. На практической части с вами будут работать кураторы - наши ведущие разработчики, которые будут помогать в разработке проекта, помогут исправить ошибки и прояснить непонятные моменты. Совместно с кураторами необходимо согласовать расписание, выбрать проект для реализации.

Завершающим этапом обучения является открытая защита проекта, посетить которую сможет любой сотрудник компании Netcracker: от разработчика до технического директора. Защита проекта - это демонстрация работы веб-приложения, его функциональных возможностей, объяснение этапов его создания, а также ответы на технические вопросы. Все выпускники Учебного Центра получат сертификат о его окончании. Тем кто проявит себя во время обучения, зарекомендует себя и успешно защитит проект, мы сделаем предложение о работе и пригласим стать частью нашей команды.

Стек технологий

В процессе обучения будут использованы следующие технологий:

#	Название	Описание
1	Язык разработки	Java 8

2	Средство сборки и деплоймента	Maven
3	Система контроля версий	Git
4	IDE	IntelliJ IDEA
5	Back-end часть	• Spring • Spring MVC • Spring Data • Spring Security • Spring Boot
6	Front-end	• HTML • CSS/Bootstrap • JavaScript/jQuery • Angular/TypeScript
7	СУБД	MySql

High Level архитектура Web приложения

Большинство существующих Web приложений можно определить с помощью следующей схемы:

Данная архитектура предполагает следующий сценарий взаимодействия конечного пользователя с Web приложением:

1. Пользователь вводит в строку браузера адрес и выбирает услугу, которой хотел бы воспользоваться (шаг 1). В качестве примера можно привести просмотр каталога товаров в интернет магазине или сервисов на интернет портале. Стоит отметить, что персональные компьютеры и различные мобильные устройства с установленными на них операционными системами и браузерами являются клиентами для данного Web приложения. Браузер формирует реквест и используя сетевую инфраструктуру (Интернет/Интранет) передает его на сервер по тому адресу, что указан в адресной строке браузера.
2. Опуская делали можно сказать, что сервер приложений/сервисов получает реквест от браузера(клиента) и вызывает определенную бизнес логику на нашем Web приложении (Шаг 2).
3. В зависимости от сложности запрашиваемого результата, наше Web приложение может обратиться как к базе данных (она в тоже время может быть реляционной или объектной) так и к другому web приложению, используя сервисы интеграции (Шаг 3). После получения ответа (в нашем случае это будет набор данных по товарам или услугам) наше web приложение сформирует ответ и вернет его клиенту в браузер, используя сервер приложений и интернет инфраструктуру.

Итак можно выделить 3 основных части нашей архитектуры:

1. **Клиент** - это чаще всего интернет браузер или приложение (client application). Вид клиента в основном будет зависеть от типа устройства на котором он запущен. Для упрощения будем считать что это интернет браузер.
2. **Сервер** - программа, в которой запускается и работает наше web приложение.
3. **Сетевая инфраструктура**, благодаря которой есть возможность взаимодействия клиента с сервером.

С точки зрения проектирования и разработки web приложений нас будут интересовать первые две части архитектуры: клиент и сервер. В процессе взаимодействия с пользователем web приложение обязано предоставить некоторый **UI** или **front-end** (**User Interface** - интерфейс пользователя/клиентская часть), с помощью которого пользователь смог бы вызывать бизнес логику приложения и посмотрть результаты этого вызова. Также web приложение обязано иметь **back-end** (серверную) часть, которая будет реализовывать необходимую бизнес логику и предоставлять ответ клиенту, который мог бы отобразиться в UI части. Из этих двух определений: front-end и back-end видно, что наше web приложение (**application**) должно включать эти две части.

Также в качестве вводной части можно отметить, что существуют различные варианты реализации или исполнения для нашего web application. Рассмотрим следующие 2:

- **Монолит** - когда front-end и back-end части тесно связанны между собой и реализованны в виде одного исполняемого ресурса (**jar** или **war**). Вся бизнес логика сконцентрированна в этом одном ресурсе.
- **Микросервисы** - когда front-end и back-end части слабо связанны и теоретически могут вообще не знать друг о друге. Каждая часть - это микросервис, который реализован в виде самостоятельного ресурса (также может быть jar либо war)

В процессе выполнения тестового проекта по итогам этого курса, мы будем использовать микросервисную архитектуру. Более подробно о преимуществах и недостатках этих двух шаблонов проектирования можно будет узнать из следующих лекций.

Обзор средств разработки

Apache Maven

Перед тем как приступить к изучению **Apache Maven,** далее просто maven, рассмотрим сборку программы на Java "в ручном" режиме. Итак, для того, чтобы собрать и запустить java application нам необходимо в зависимости от операционной системы, где мы будем осуществлять процесс, выполнить следующие команды:

- для компиляции на Linux/Mac

  javac -cp ".:<path to dependency jar>" nc/test/Test.java
- для компиляции на Windows

  javac -cp ".;<path to dependency jar>" nc/test/Test.java

- для запуска на Linux/Mac/Windows

  java nc.test.Test (**в качестве параметра необходимо использовать имя скомпилированного класса, который содержит main метод**)
- для запуска jar
  java -jar <путь к .jar файлу>

Как видно все выглядит достаточно просто для одного класса или jar файла. Но что будет если необходимо собрать скажем 100 файлов которые расположены в 20 пакетах.

Основные неудобства, связанные с "ручной" сборкой:

1. Разработчику наобходимо самому следитть за всеми необходимымы зависимостями (**dependencies**)
2. Необходимо самому следить за правильностью заполнения опции classpath (-cp)
3. Для построения большого проекта придется использовать несколько шагов
4. Человеческий фактор
5. Как следствие - большие затраты времени

Чтобы избежать всех перечисленных проблемных моментов нам и необходим maven (https://maven.apache.org/).

Чтобы запустить проект с использованием maven необходимо сделать следующие шаги:

1. Создать POM(project object model) файл для java модуля
2. Используя основные команды построить проект.
   Например команда **mvn clean install** делает следующее:
   1. Чистит папку target, удаляет все ресурсы, которые остались с предыдущего **build** (сборки)
   2. Проверяет (валидирует) конфигурацию в POM файле проекта и/или модуля

3. Компилирует все java классы в проекте
4. Запускает unit tests при их наличии
5. Пакует скомпилированные классы в один из форматов: **jar, war, ear**
6. Помещает(инсталлирует) получившийся артифакт(файл) в **local Maven Repository** (локальное хранилище)

Пример файловой структуры модуля приложения:

/test-service

/src

/target

pom.xml

Так выглядит **pom.xml**:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.
w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.
apache.org/xsd/maven-4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>
        <groupId>nc.test</groupId>
        <artifactId>test-service</artifactId>
        <version>0.0.1-SNAPSHOT</version>
        <packaging>jar</packaging>
        <name>test-service</name>
        <description>Configuration service</description>
        <parent>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-parent</artifactId>
                <version>2.0.3.RELEASE</version>
                <relativePath/> <!-- lookup parent from repository -->
        </parent>
        <properties>
                <project.build.sourceEncoding>UTF-8</project.build.
sourceEncoding>
                <project.reporting.outputEncoding>UTF-8</project.reporting.
outputEncoding>
                <java.version>1.8</java.version>
                <spring-cloud.version>Finchley.RELEASE</spring-cloud.
version>
        </properties>
        <dependencies>
                <dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-actuator<
/artifactId>
                </dependency>
                <dependency>
                        <groupId>org.springframework.cloud</groupId>
                        <artifactId>spring-cloud-config-server</artifactId>
                </dependency>
```

```
                <dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-test</artifactId>
                        <scope>test</scope>
                </dependency>
        </dependencies>
        <dependencyManagement>
                <dependencies>
                        <dependency>
                                <groupId>org.springframework.cloud<
/groupId>
                                <artifactId>spring-cloud-dependencies<
/artifactId>
                                <version>${spring-cloud.version}</version>
                                <type>pom</type>
                                <scope>import</scope>
                        </dependency>
                </dependencies>
        </dependencyManagement>
        <build>
                <plugins>
                        <plugin>
                                <groupId>org.springframework.boot</groupId>
                                <artifactId>spring-boot-maven-plugin<
/artifactId>
                        </plugin>
                </plugins>
        </build>
</project>
```

На практическом занятии рассмотрим создание java модуля с помощью mave (https://maven.apache.org/download.cgi)

Git&Svn

Зачем нужны системы контроля версий. Вот несколько пунктов:

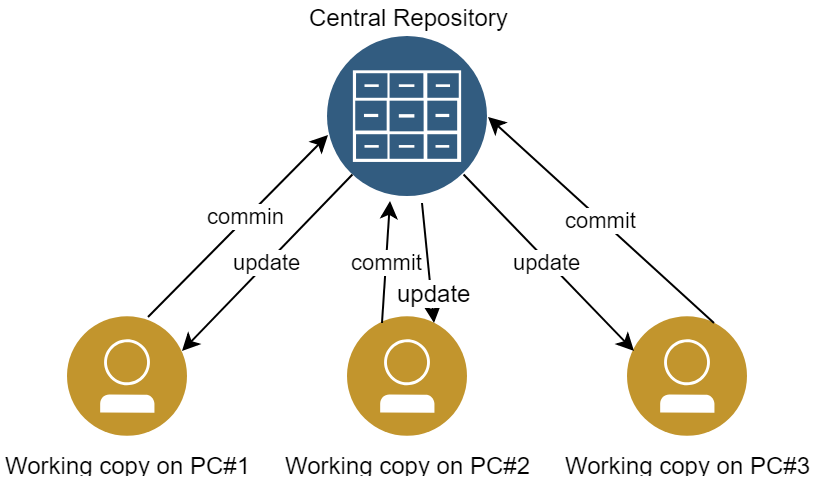
1. Возможность работать многочисленной команде разработчиков над одним и тем же кодом (**same code base**)
2. Сохраняется полная история изменений для файла, который помещен в систему контроля версий
3. Легко можно молучить список различий между версиями одного и того же файла или между ветками(**branches**) кода
4. Поддержка множество веток кода (**branch**)

Svn

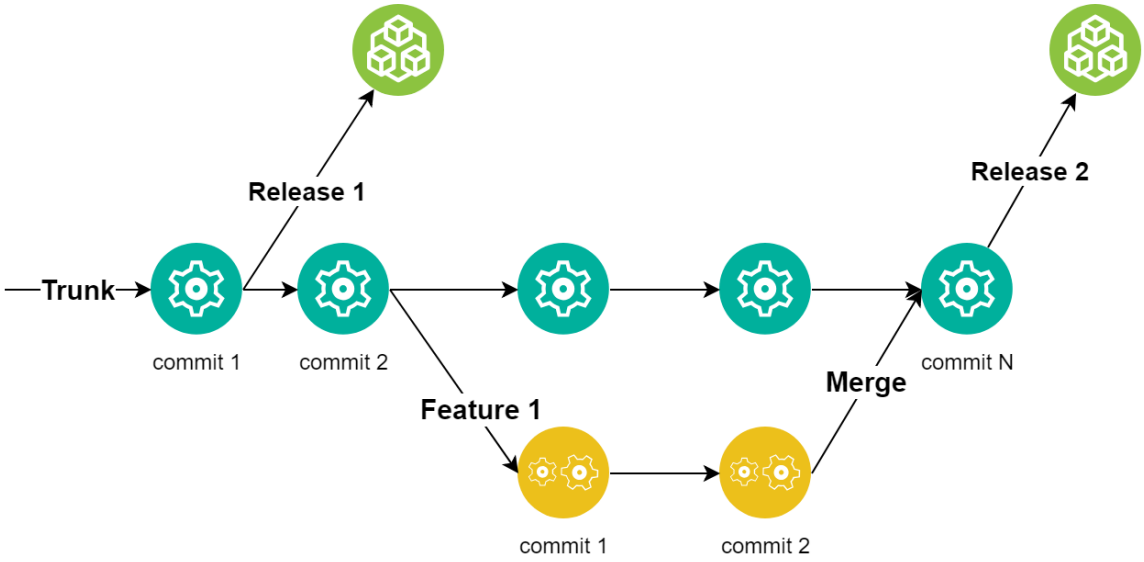
Основные характеристики:

1. Один центральный репозиторий на удаленном сервере
2. Каждый разработчик имеет свою рабочую копию этого репозитория
3. Чтобы выполнить изменение кода в репозиторий разработчик должен сделать следующее:
    1. Получить актуальное состояние центрального репозитория (выполнить **update**)
    2. Изменить код локально
    3. Внести изменения в центральный репозиторий (выполнить **commit**)

Схема показывает типовой алгоритм работы с Svn:



Большинство систем контроля версий имеют такое понятие как ветка кода (**code branch**). В Svn основной/главной веткой является **Trunk**, от нее отводятся все дополнительные ветки: для исправления ошибок (**bug fixing**), для создания версий ПО (**Release 1, Release 2**), а также для разработки определенного функционала (**Future 1**), который необходимо будет добавить в основную линию кода, путем соединения (**Merge**). Все перечисленное показана на схеме:



Работая с общей линией кода, разработчик рано или поздно сталкивается с таким понятием как кофликт (**conflict**) при добавлении своего кода в общую ветку. Чтобы решить конфликт и добавить свой код в общую ветку в большинсве случаев необходимо обновить свою локальную копию последними изменениями из удаленного центрального репозитория (ветка Trunk), устранить все конфликты локально с помощью средства сранения кода (например WinMerge) и закомитать обновленный локальный код в Trunk.

Вот пример возникшего конфликта и последовательности его устранения:

- User 1 сделал checkout версии commit 1
- User 2 сделал checkout той же версии commit 1
- User 1 работал над своей часть кода и делал несколько комитов до commit N, его изменения уже находтся на общей ветке Trunk в центральном репозитории
- User 2 работал параллельно над своей линией кода и пытается закомитить свои изменения в Trunk, но не может этого сделать из за конфликта в коде, так как User 1 уже внес свои изменения

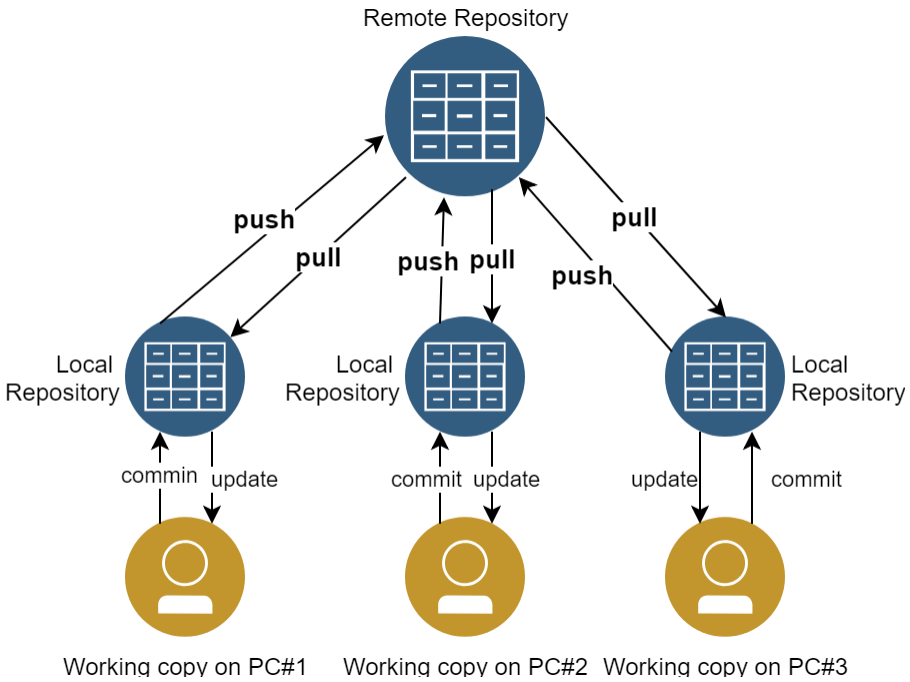
Чтобы решить конфлик User 2 должен сделать следующее:

1. Обновить свой локальный код (ветку) последнии изменениями из Trunk
2. Устранить все конфликты локально (WinMerge)

3. Закомитать свои изменения в Trunk

Git

Эту систему контроля версий можно охарактеризовать одним словом - распределенная (**Distributed**).



Кроме удаленного репозитория у каждого разработчика есть возможность иметь локальный репозиторий и историю его изменений. Над локальным репозиторием можно совершать все те же действия что и над удаленным: создавать ветки, делать комиты и т.д. Но при работе необходимо помнить, что для получения всех изменений из удаленного репозитория, необходимо выполнить команду **pull**, а для сохранения свои изменений в удаленный репозитория - команду **push**.

Вот типичный сценарий работы с описанием команд:

1. Клонируем удаленный репозиторий git clone **https://<путь к удаленному репозиторию>.git**
2. После клонирования по умолчанию вы на ветке **master**
3. создаем свою ветку от master **git checkout -b [ваше имя ветки]**
4. после создания вы переключитесь на эту ветку или для пеперключения на нее вы можете выполнить **git checkout [ваше имя ветки]**
5. сделать изменения в коде и с помощью **git add** добавить их в комит
6. закомитать изменения в локальный репозитория, выполнить **git commit -m "описание комита"**
7. пушнуть ветку в удаленный репозиторий **git push origin [ваше имя ветки]**
8. с помощью **pull request** замержить измения из вашей запушеной ветки в master

для создания удаленной ветки для вашей локальной ветки - **git remote add [ваше имя ветки]**

чтобы запушить изменения из локальной ветки в удаленную - **git push[ваше имя удаленной ветки][ваше имя ветки]**

обновить удаленную ветку, когда **master** в уделенном репозитории обновился **git fetch [ваше имя удаленной ветки]**

Ссылки на ресурсы: https://git-scm.com/downloads or https://git-for-windows.github.io/

Tools

Кроме перечисленных выше, существуют также другие инструменты менеджмента проекта. Например альтернативой maven могут быть: **Ant** (обычно legacy системы) или **Gradle**

В настоящее время существует много менеджеров сборки и деплоймента, можно сказать, что каждая технология предоставляет такой инструмент "из коробки". Альтернативы для Git и Svn может служить например **Perforce**. В тестовом проекте будет использоваться Git.

IntelliJ IDEA

Существует много различных IDE. IntelliJ IDEA является одним из самых удачных общепризнаных вариантов. Основные возможности:

1. Создание проекта "с нуля", из имеющихся исходных кодов или используя репозиторий системы контроля версий
2. Поддержка менеджеров проекта, таких как maven и gradle
3. автоматическое форматирование и широкие возможности по поиску и рефакторингу кода, автогенерации
4. локальный и удаленный режим отладки (**local/remote debug**)
5. Локальная история изменений ресурсов
6. Интеграция с Git/SVN/Mercuria

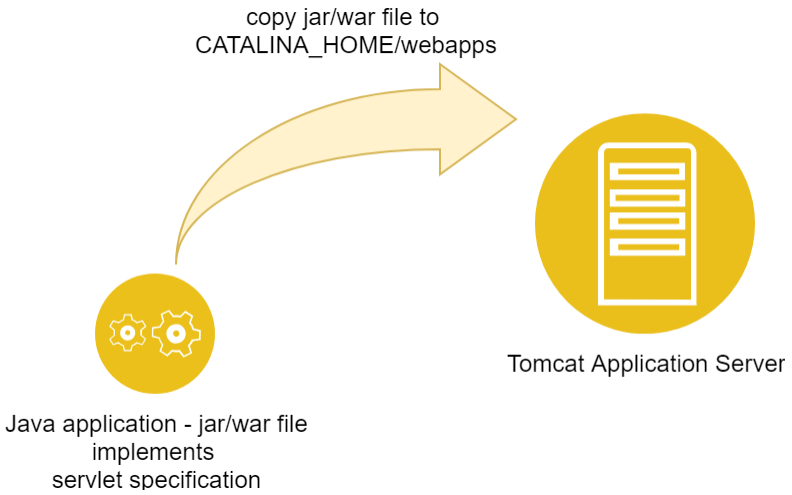
Алтернативой IDEA может служить Eclipse.

Apache Tomcat

Apache Tomcat является одним из самых распространенных серверов для java приложений. Он реализует следующие технологии:

- *Java Servlet*
- *JavaServer Pages*
- *Java Expression Language*
- *Java WebSocket*

Схематично процесс запуска java приложения на tomcat можно представить следующим образом:



Code review

Code review - процесс проверки (контроля) выполненой задачи. В зависимости от зрелости процессов на проекте code review может преследовать множество целей.

- Контроль за соблюдением общепринятых стандартов на проекте: нэйминг-конвенции, использование констант
- Обмен знаниями внутри команды, особенно для Jr-разработчиков: логические ошибки, отсутствие проверок на NPE, неоптимальное использование логгера, изобретение своих велосипедов вместо паттернов/оптимальных конструкций и т.д.

- Повышение дисциплины в команде по качеству оформления/документирования исходного кода
- Использование элемента "парного программирования" и понимания "что делают другие члены команды"
- Обнаружение ревьювером потенциальных проблем, неочевидных для того кто вносит изменения
- Как итог - повышение качества кода на проекте

Continuous Integration/Continuous Delivery(CI/CD)

Непрерывная интеграция и доставка продукта. Сущность этого процесса можно показать используя такую схему:

Схема описывает жизненный цикл продукта от стадии разработки до выпуска версии. Итак что происходи на каждой стадии:

1. Development - написание и компиляция кода, unit testing
2. Build - комит кода в систему контроля версий, запуска сборки проекта на удаленном сервере: компиляция исхдного кода + запуск unit tests
3. Если шаг 2 закончился успешно и проект скомпилировался - полученный артифакт или группа артифактов копируются или развертываются на тестовом сервере
4. Такое же развертывание происходит на тестовом сервере, запускающем тесты автоматизации (**automation tests**)
5. Если "ручное" (**manual**) тестирование закончилось успешно, все **test cases** пройдены успешно и **automation tests** также дали положительный результат, то можно запускать развертывание на подготовленных окружениях (**staging , pre-production, production environments**)
6. Установка на различные окружения (**environments**)
7. Все инфраструктурные проблемы решены, можно открывать доступ конечным пользователям на **production environment**. Выход (**Release**) продукта.

Несколько слов про тестирование и интеграцию:

- Unit-тесты (очень быстрые и независимые от среды) - их разрабатывают разработчики
- Качество кода – покрытие кода тестами, оформление кода, статический анализ
- Интеграционные тесты (эмуляция кликов мыши на элементах UI, запуск легковесной БД в памяти, эмуляция http запросов)
- Acceptance тесты – проверка того, что приложение удовлетворяет бизнес-требованиям – похоже на интеграционные тесты, но проверяется именно реализация бизнес-требований.
- Нефункциональные тесты – проверка производительности, security, memory footprint, надежности (High Availability) и т.д.

- Ручное тестирование – пропущенные фичи, незамеченные баги, ошибки оформления и usability.

Преимущества такого процесса:

- Все, что возможно работает автоматически – от коммита в репозиторий до развертывания фикса на продакшн
- Мгновенные уведомления о сбое в процессе
- Последняя полноценная сборка всегда существует и готова к релизу
- Единообразный процесс развертывания в любую среду – тестовую, продакшн, разработки

Для самостоятельного изучения

Ссылка	Краткое описание
https://maven.apache.org/	
https://git-scm.com/downloads	
https://git-for-windows.github.io/	