Gradle

1. Компилируемый язык программирования — язык программирования, исходный код которого преобразуется

компилятором в машинный код и записывается в файл с особым заголовком и/или расширением для

последующей идентификации этого файла, как исполняемого операционной системой.

1. Интерпретируемый язык программирования — язык программирования, исходный код на котором

выполняется методом интерпретации

Классифицируя языки программирования по способу исполнения, к группе интерпретируемых относят языки,

в которых операторы программы друг за другом отдельно транслируются и сразу выполняются

(интерпретируются) с помощью специальной программы-интерпретатора

1. Байт-код — стандартное промежуточное представление, в которое может быть переведена компьютерная

программа автоматическими средствами.

По сравнению с исходным кодом, удобным для создания и чтения человеком, байт-код — это компактное

представление программы, уже прошедшей синтаксический и семантический анализ. В нём в явном виде

закодированы типы, области видимости и другие конструкции. С технической точки зрения байт-код

представляет собой машинно-независимый код низкого уровня, генерируемый транслятором из исходного

кода.

1. Сборка ПО (англ. Software Build), также известная как интеграция - процесс преобразования файлов с

исходным кодом и их компиляция в артефакты, составляющие приложение, такие как бинарные и

исполняемые файлы.

Чаще всего включает компиляцию и компоновку, выполняется инструментами автоматизации.

1. Стандартные цели для сборки дистрибутивов GNU:

all — выполнить сборку пакета;

install — установить пакет из дистрибутива (производит копирование

исполняемых файлов, библиотек и документации в системные каталоги);

uninstall — удалить пакет (производит удаление исполняемых файлов и

библиотек из системных каталогов);

clean — очистить дистрибутив (удалить из дистрибутива объектные и

исполняемые файлы, созданные в процессе компиляции);

distclean — очистить все созданные при компиляции файлы и все

вспомогательные файлы, созданные утилитой ./configure в процессе настройки

параметров компиляции дистрибутива.

1. Автоматизация сборки — этап процесса разработки программного обеспечения, заключающийся в

автоматизации широкого спектра задач, решаемых программистами в их повседневной деятельности.

Включает такие действия, как:

• компиляция исходного кода в объектный модуль,

• сборка бинарного кода в исполняемый файл,

• выполнение тестов,

• развёртывание программы в целевой среде,

• написание сопроводительной документации или описание изменений новой версии

1. Виды автоматизации, применяемые в различных инструментах:

• автоматизация по запросу (on-demand automation): запуск пользователем сценария в командной строке,

• запланированная автоматизация (scheduled automation): непрерывная интеграция, происходящая в виде

ночных сборок,

• условная автоматизация (triggered automation): непрерывная интеграция, выполняющая сборку при

каждом подтверждении изменения кода (commit) в системе управления версиями.

1. Gradle — система автоматической сборки, построенная на принципах Apache Ant и Apache Maven, но

предоставляющая DSL на языках Groovy и Kotlin вместо традиционной XML-образной формы представления

конфигурации проекта

В отличие от Apache Maven, основанного на концепции жизненного цикла проекта, и Apache Ant, в котором

порядок выполнения задач (targets) определяется отношениями зависимости (depends-on), Gradle использует

направленный ациклический граф для определения порядка выполнения задач.

1. Предметно-ориентированный язык (англ. domain-specific language, DSL — «язык, специфический для

предметной области») — компьютерный язык, специализированный для конкретной области применения (в

противоположность языку общего назначения, применимому к широкому спектру областей и не

учитывающему особенности конкретных сфер знаний).

Построение такого языка и/или его структура данных отражают специфику решаемых с его помощью задач.

Является ключевым понятием языково-ориентированного программирования.

1. Ориентированный ациклический граф (направленный ациклический граф, DAG от англ. directed acyclic graph) — орграф, в котором отсутствуют направленные циклы, но могут быть «параллельные» пути, выходящие из одного узла и разными путями приходящие в конечный узел. Направленный ациклический граф является

обобщением дерева (точнее, их объединения — леса).

1. Сами по себе задачи состоят из:

Действия — действия, которые что-то

делают, например, копируют файлы или

компилируют исходные тексты

Входы — значения, файлы и каталоги,

которые используют или над которыми

работают действия

Выходы — файлы и каталоги, которые

изменяют или генерируют действия

1. Фазы сборки

Инициализация:

Gradle поддерживает одиночные и много-проектные сборки. На этапе инициализации Gradle определяет, какие проекты будут участвовать в сборке, и создает экземпляр Project для каждого из этих проектов.

Конфигурация:

На этом этапе происходит настройка объектов проекта. Выполняются скрипты сборки всех проектов, входящих в состав сборки.

Исполнение:

Gradle определяет подмножество задач, созданных и настроенных на этапе конфигурации, которые должны быть выполнены. Подмножество определяется аргументами имени задачи, переданными команде gradle и текущей директории. После этого gradle выполняет каждую из выбранных задач.

1. Процесс линковки представляет собой замену относительных адресов функций внешних библиотек на реальные адреса которые будут использоваться программой в процессе её выполнения.
2. make — утилита, автоматизирующая процесс преобразования файлов из одной формы в другую. объектные файлы и последующая компоновка Чаще всего это компиляция исходного кода в исполняемые файлы или библиотеки. Утилита использует специальные make-файлы, в которых указаны зависимости файлов друг от друга и правила для их удовлетворения. На основе изменения запускает информации о времени последнего каждого файла make определяет и необходимые программы.
3. Предметно-ориентированный язык (англ. domain-specific language, DSL — «язык, специфический для предметной области») — компьютерный язык, cистемы сборки специализированный для конкретной(в противоположность языку применимому к широкому области общего спектру применения назначения, областей и не учитывающему особенности конкретных сфер знаний). Построение такого языка и/или его структура данных помощью языково-отражают специфику решаемых с его задач. Является ключевым понятие ориентированного программирования.
4. Ресурс — это данные (изображения, звук, тексты т.д.), к которым программе необходимо получить доступ независимо от места расположения программного кода.

Стандартные задачи:

*clean*

Удаляет каталог сборки и все, что в нем находится, т.е. путь, указанный свойством проекта Project.getBuildDir().

*check* — задача жизненного цикла

К этой задаче прикрепляются к этой задаче свои задачи верификации, например, запускающие тесты

*assemble* — задача жизненного цикла

К этой задаче прикрепляются задачи, которые производят дистрибутивы и другие расходные артефакты. Например, jar производит артефакт для библиотек Java.

1. Типы плагинов

В Gradle есть два типа плагинов: плагины для скриптов и бинарные плагины. Плагины скриптов – это дополнительный скрипт сборки, который дает декларативный подход к манипулированию сборкой. Обычно это используется в сборке. Бинарные плагины – это классы, которые реализуют интерфейс плагина и применяют программный подход к управлению сборкой. Бинарные плагины могут находиться со скриптом сборки, с иерархией проекта или внешне в JAR плагина.

1. Жизненный цикл программного обеспечения (Software Life Cycle Model) — это период времени, который начинается с момента принятия решения о создании программного продукта и заканчивается в момент его полного изъятия из эксплуатации. Этот цикл — процесс построения и развития ПО. Модели Жизненного цикла программного обеспечения Жизненный цикл можно представить в виде моделей. В настоящее время наиболее распространенными являются: каскадная, инкрементная (поэтапная модель с промежуточным контролем) и спиральная модели жизненного цикла.
2. Каскадная модель

Каскадная модель (англ. waterfall model) — модель процесса разработки программного обеспечения, жизненный цикл которой выглядит как поток, последовательно проходящий фазы анализа требований, проектирования. реализации, тестирования, интеграции и поддержки. Процесс разработки реализуется с помощью упорядоченной последовательности независимых шагов. Модель предусматривает, что каждый последующий шаг начинается после полного завершения выполнения предыдущего шага. На всех шагах модели выполняются вспомогательные и организационные процессы и работы, включающие управление проектом, оценку и управление качеством, верификацию и аттестацию, менеджмент конфигурации, разработку документации. В результате завершения шагов формируются промежуточные продукты, которые не могут изменяться на последующих шагах.

Выработка системных требований -> выработка требований к ПО -> Анализ -> Проектирование -> Кодирование

->Тестирование -> Эксплуатация.

1. Достоинства модели:

• стабильность требований в течение всего жизненного цикла разработки;

• на каждой стадии формируется законченный набор проектной документации, отвечающий критериям полноты и согласованности;

• определенность и понятность шагов модели и простота её применения;

• выполняемые в логической последовательности этапы работ позволяют планировать сроки завершения всех работ и соответствующие ресурсы (денежные. материальные и людские).

Недостатки модели:

• сложность чёткого формулирования требований и невозможность их динамического изменения на протяжении пока идет полный жизненный цикл;

• низкая гибкость в управлении проектом;

• последовательность линейной структуры процесса разработки, в результате возврат к предыдущим шагам для решения возникающих проблем приводит к увеличению затрат и нарушению графика работ;

• непригодность промежуточного продукта для использования;

• невозможность гибкого моделирования уникальных систем;

• позднее обнаружение проблем, связанных со сборкой, в связи с одновременной интеграцией всех результатов в конце разработки;

• недостаточное участие пользователя в создании системы — в самом начале (при разработке требований) и в конце (во время приёмочных испытаний);

• пользователи не могут убедиться в качестве разрабатываемого продукта до окончания всего процесса разработки. Они не имеют возможности оценить качество, т.к.нельзя увидеть готовый продукт разработки;

• у пользователя нет возможности постепенно привыкнуть к системе.

Процесс обучения происходит в конце жизненного цикла, когда ПО уже запущено в эксплуатацию;

• каждая фаза является предпосылкой для выполнения последующих действий, что превращает такой метод в рискованный выбор для систем, не имеющих аналогов, т.к. он не поддается гибкому моделированию.

1. Область применения Каскадной модели

Ограничение области применения каскадной модели определяется её недостатками. Её использование наиболее эффективно в следующих случаях:

• при разработке проектов с четкими, неизменяемыми в течение жизненного цикла требованиями, понятными реализацией и техническими методиками;

• при разработке проекта, ориентированного на построение системы или продукта такого же типа, как уже разрабатывались разработчиками ранее;

• при разработке проекта, связанного с созданием и выпуском новой версии уже существующего продукта или системы;

• при разработке проекта, связанного с переносом уже существующего продукта или системы на новую платформу;

• при выполнении больших проектов, в которых задействовано несколько больших команд разработчиков.

1. Инкрементная модель

(поэтапная модель с промежуточным контролем) Инкрементная модель (англ. increment — увеличение, приращение) подразумевает разработку программного обеспечения с линейной последовательностью стадий, но в несколько инкрементов (версий), т.е. с запланированным улучшением продукта за все время пока Жизненный цикл разработки ПО не подойдет к окончанию. Сложный рисунок, это инкрементная модель.

1. Достоинства:

• затраты, которые получаются в связи с изменением требований пользователей, уменьшаются, повторный анализ и совокупность документации значительно сокращаются по сравнению с каскадной моделью;

• легче получить отзывы от клиента о проделанной работе — клиенты могут озвучить свои комментарии в отношении готовых частей и могут видеть, что уже сделано. Т.к. первые части системы являются прототипом системы в целом.

• у клиента есть возможность быстро получить и освоить программное обеспечение — клиенты могут получить реальные преимущества от системы раньше, чем это было бы возможно с каскадной моделью.

Недостатки модели:

• менеджеры должны постоянно измерять прогресс процесса. В случае быстрой разработки не стоит создавать документы для каждого минимального изменения версии;

• структура системы имеет тенденцию к ухудшению при добавлении новых компонентов — постоянные изменения нарушают структуру системы. Чтобы избежать этого требуется дополнительное время и деньги на рефакторинг. Плохая структура делает программное обеспечение сложным и дорогостоящим для последующих

изменений. А прерванный Жизненный цикл ПО приводит еще к большим потерям.

1. V модель (разработка через тестирование)

Данная модель имеет более приближенный к современным методам алгоритм, однако все еще имеет ряд недостатков. Является одной из основных практик экстремального программирования.

1. Модель на основе разработки прототипа

Данная модель основывается на разработки прототипов и прототипирования продукта. Прототипирование используется на ранних стадиях жизненного цикла программного обеспечения:

• Прояснить не ясные требования (прототип UI)

• Выбрать одно из ряда концептуальных решений (реализация сценариев)

• Проанализировать осуществимость проекта

Классификация прототипов:

• Горизонтальные и вертикальные

• Одноразовые и эволюционные

• бумажные и раскадровки

Горизонтальные прототипы — моделирует исключительно UI не затрагивая логику обработки и базу данных.

Вертикальные прототипы — проверка архитектурных решений.

Одноразовые прототипы — для быстрой разработки.

Эволюционные прототипы — первое приближение эволюционной системы.

1. Спиральная модель: Жизненный цикл — на каждом витке

спирали выполняется создание очередной версии продукта, уточняются требования проекта, определяется его качество и планируются работы следующего витка. Особое внимание уделяется начальным этапам разработки — анализу и проектированию, где реализуемость тех или иных технических решений проверяется и обосновывается посредством создания прототипов.

Достоинства модели:

• позволяет быстрее показать пользователям системы работоспособный продукт, тем самым, активизируя процесс уточнения и дополнения требований;

• допускает изменение требований при разработке программного обеспечения, что характерно для большинства разработок, в том числе и типовых;

• в модели предусмотрена возможность гибкого проектирования, поскольку в ней воплощены преимущества каскадной модели, и в тоже время разрешены итерации по всем фазам этой же модели;

• позволяет получить более надежную и устойчивую систему. По мере развития программного обеспечения ошибки и слабые места обнаруживаются и исправляются на каждой итерации;

• эта модель разрешает пользователям активно принимать участие при планировании, анализе рисков, разработке, а также при выполнении оценочных действий;

• уменьшаются риски заказчика. Заказчик может с минимальными для себя финансовыми потерями завершить развитие неперспективного проекта;

• обратная связь по направлению от пользователей к разработчикам выполняется с высокой частотой и на ранних этапах модели, что обеспечивает создание нужного продукта высокого качества. Недостатки модели:

• если проект имеет низкую степень риска или небольшие размеры, модель может оказаться дорогостоящей. Оценка рисков после прохождения каждой спирали связана с большими затратами;

• Жизненный цикл модели имеет усложненную структуру, поэтому может быть затруднено её применение разработчиками, менеджерами и заказчиками;

• спираль может продолжаться до бесконечности, поскольку каждая ответная реакция заказчика на созданную версию может порождать новый цикл, что отдаляет окончание работы над проектом;

• большое количество промежуточных циклов может привести к необходимости в обработке дополнительной документации;

• использование модели может оказаться дорогостоящим и даже недопустимым по средствам, т.к. время. затраченное на планирование, повторное определение целей, выполнение анализа рисков и прототипирование, может быть чрезмерным;

• могут возникнуть затруднения при определении целей и стадий, указывающих на готовность продолжать процесс разработки на следующей.

Область применения спиральной модели

• Применение спиральной модели целесообразно в следующих случаях:

• при разработке проектов, использующих новые технологии;

• при разработке новой серии продуктов или систем;

• при разработке проектов с ожидаемыми существенными изменениями или дополнениями требований;

• для выполнения долгосрочных проектов;

• при разработке проектов, требующих демонстрации качества и версий системы или продукта через короткий период времени;

• при разработке проектов. для которых необходим подсчет затрат, связанных с оценкой и разрешением рисков.

Gradle практическое

Gradle — система сборки проектов с открытым исходным кодом, в которой основной упор идет на гибкость и производительность. В настоящее время поддерживаются 2 языка программирования для написания скриптов сборки: groovy и kotlin.Для работы с gradle нужна только установленная jdk, для всех проектов хорошим тоном считается иметь gradle-wrapper. Это обертка из небольшого jar-файла (порядка 58 Кб), 1 файла конфигурации и 2 скриптов для запуска — windows и \*nix. При первом запуске враппера загружается gradle необходимой версии и после чего начинает свою штатную работу. Данная схема необходима, так как в зависимости от множества причин версия системы сборки у членов команды и билд-сервера может отличаться.

Выполнение задач в gradle:

Задачи в gradle выполняются следующим образом: вызов враппера gradle задача1 задача2 … задачаN. Выполнение задач будет именно в той последовательности, в которой вы из вызовете. Также некоторые задачи по умолчанию зависят друг от друга. Например, задачу compileJava вызывать нет никакой нужды, если вы собираете проект целиком командой build.

Самые популярные задачи:

● clean — очистка всех сборочных директорий

● build — собрать приложение целиком

● javadoc — сгенерировать JavaDoc документацию

● shadowJar — собрать UberJar (архив со всеми зависимостями, для использования необходимо импортировать плагин shadow)

● run — запуск приложения

● test — прогон тестов

Рассмотрим основные разделы:

1) plugins — здесь перечисляются плагины, которые будут использоваться при сборке проекта. В сам gradle включены только“Core Plugins”. Если для нужд проекта чего-либо не хватает, то можно воспользоваться поиском на https://plugins.gradle.org/, либо реализовать свой плагин, который будет решать поставленную задачу.

2) Пакет и версия — нужны для идентификации вашего проекта. Если кто-то публикует приложение или библиотеку в maven-репозиториях, то по пакету и названию проекта можно его найти (указывается в рядом лежащем settings.gradle, например: rootProject.name = 'sample').

3) repositories — здесь перечисляются репозитории, которые будут использоваться для поиска и загрузки артефактов. Самым популярным является mavenCentral(). Если у вас есть зависимости от других ваших проектов, которые вы пока нигде не публикуете, то не забудьте указать еще mavenLocal() - в таком случае поиск артефактов будет производиться и по вашим локально собранным пакетам. Если вам необходимо указывать зависимости, которые есть, например, только в вашем рабочем хранилище артефактов, то такие зависимости указываются следующим образом внутри блокаrepositories.

4) dependencies — зависимости проекта. Скорее всего, когда приложение становится чуть больше HelloWorld, возникает необходимость в импортировании кода от третьих лиц (возможно, это будет библиотека от ваших же коллег, которые вынесли все, что вы используете вместе с ними, в отдельный common-модуль, которые все подключают как зависимость). Наиболее популярные типы:

● annotationProcessor — обработка аннотаций в процессе компиляции (чаще всего используется для кодогенерации)

● implementation — зависимости, необходимые для компиляции проекта, которые не являются частью API, предоставляемого проектом. Например, проект использует Hibernate для работы с БД, но API для работы с ним нет.

● api — зависимости, необходимые для компиляции проекта, которые являются частью API, предоставляемого проектом. Например, проект использует Guava и предоставляет публичные интерфейсы с классамиGuava в их сигнатурах методов.