МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ

РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное бюджетное образовательное учреждение

высшего образования

«Кемеровский государственный университет» (КемГУ)

**Институт Цифры**

**Кафедра цифровых технологий**

**ОТЧЕТ**

**О ВЫПОЛНЕНИИ ЛАБОРАТОРНОЙ РАБОТЫ №4(3)**

**ПО ПРОГРАММНОЙ ИНЖЕНЕРИИ**

Студента 3 курса, группы ПИ-202

Балахонцева Владимира Игоревича

Направление подготовки

*09.03.03 Прикладная информатика*

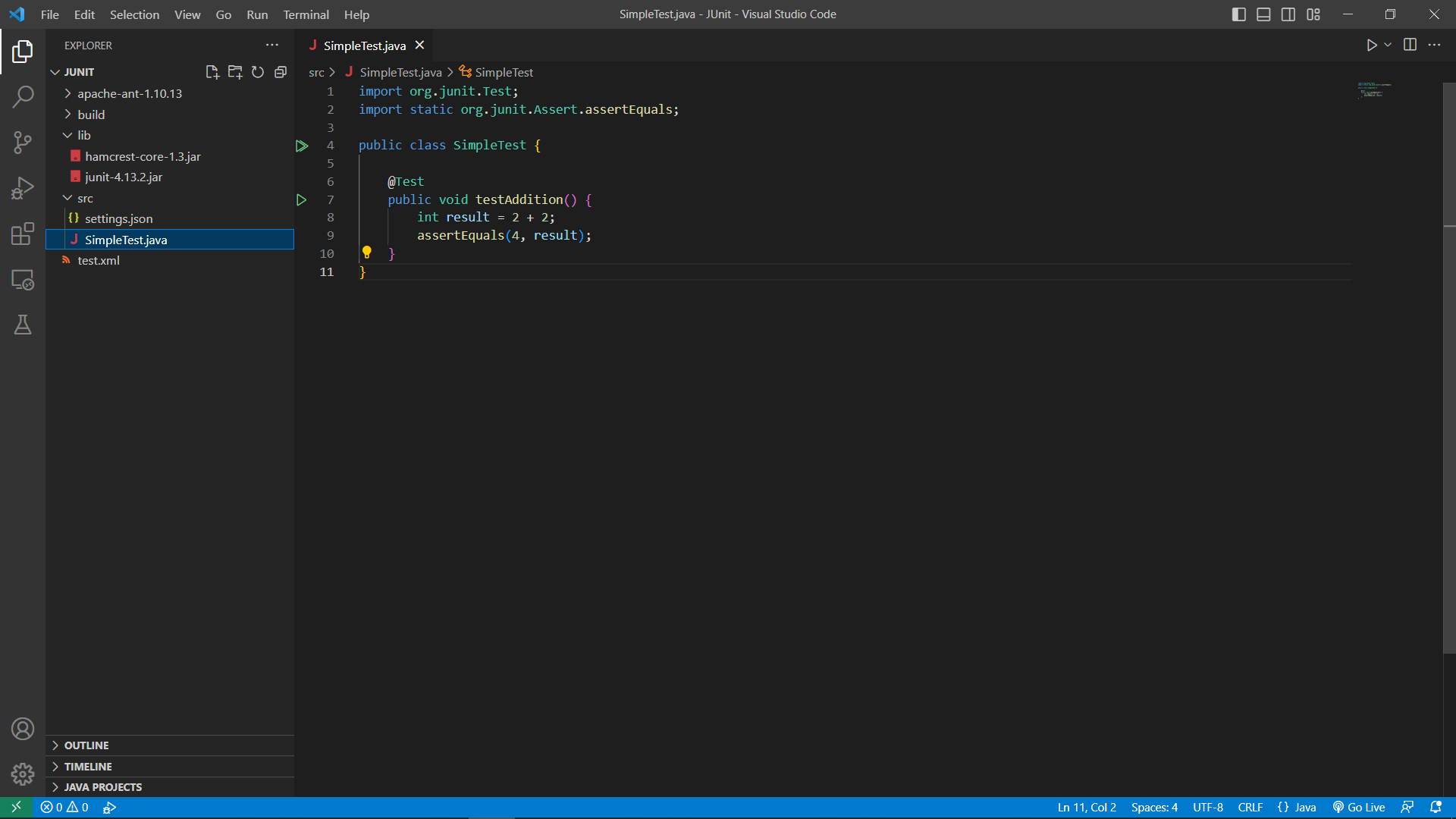
Руководитель:

ассистент

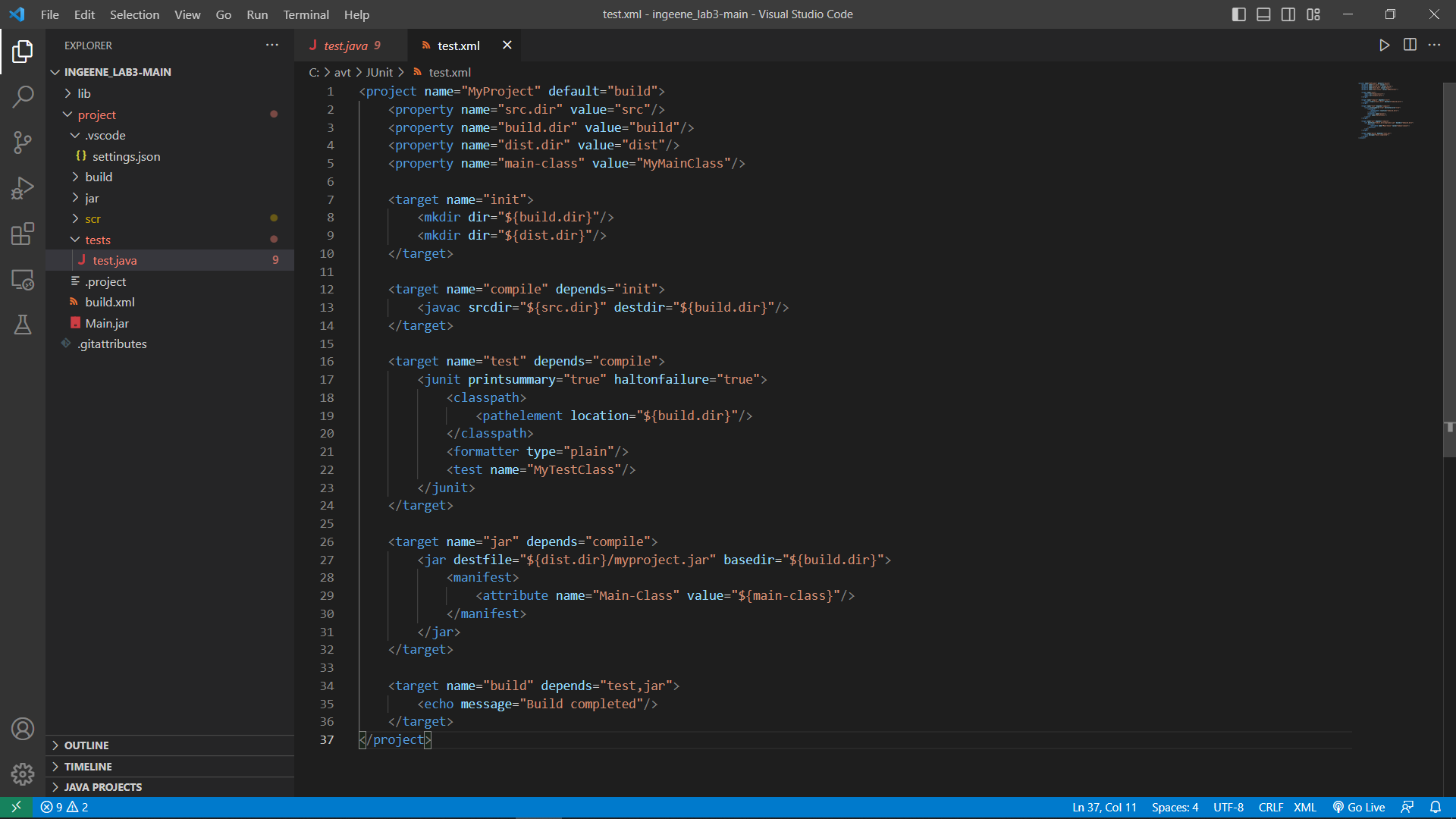
В.В. Илькевич

Кемерово 2023

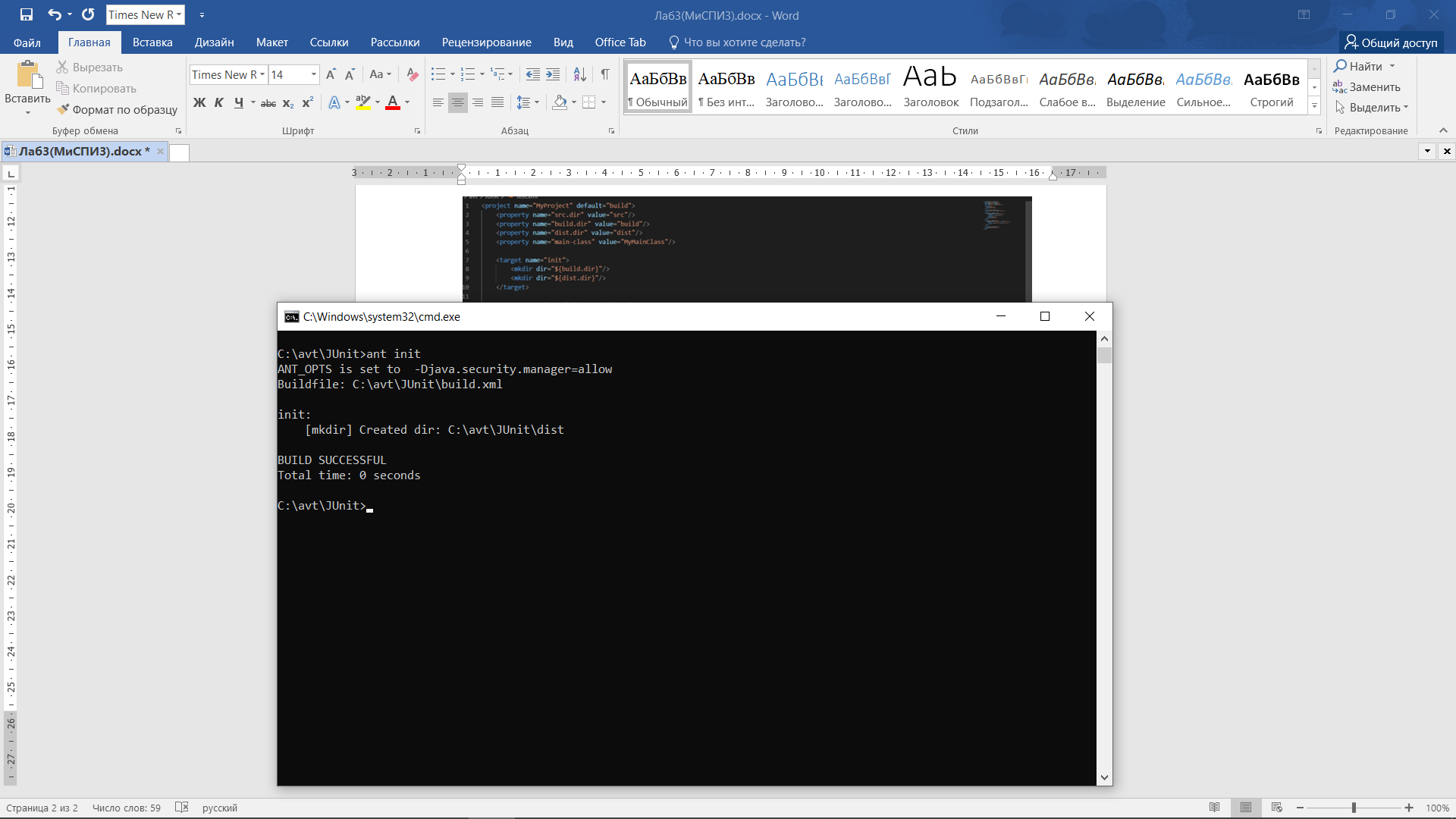
**Пример Junit теста на java:**



**Build код**



**Выполнение**



**Вывод:**

При выполнении данной лабораторной работы были изучены основы работы с Apache Ant и написаны первые junit-тесты.

**Вопросы:**

1. Тестирование ПО. Цель тестирования, виды тестирования.

Тестирование программного обеспечения — процесс исследования программного обеспечения (ПО) с целью получения информации о качестве продукта.

Цели тестирования: Обнаружение дефектов Повышение уверенности в уровне качества Предоставление информации для принятия решений Предотвращение дефектов.

Виды тестирования:

• Функциональное и нефункциональное

Функциональное тестирование — это проверка функций программы. Специалист нажимает на всевозможные клавиши и пытается вести себя необычно, дабы обнаружить недочеты проекта.

Нефункциональное тестирование представляет собой проверку производительности, надежности и отзывчивости приложения, а также ее соответствия нормам безопасности.

• Статическое и динамическое

Статическая проверка выполняется с выключенной программой. Специалисты открывают документацию приложения, анализируют указанные в ней функции, а затем изучают код для оценки качества реализации.

Динамическое тестирование выполняется после статического. В этом случае необходимо включить программу и на практике узнать, насколько работоспособными являются ее функции.

• Нагрузочное. Речь идёт о тестировании программы в условиях высоких нагрузок, которые могут быть больше, чем планировали разработчики. Эти тесты обязательны для онлайн-сервисов, которые должны правильно работать даже при наличии большого числа посетителей на пиковой или регулярной основе (онлайн-магазины во время распродаж, новостные ресурсы при резонансных событиях и т.д.).

• Тестирование UX. В этом случае специалист сосредотачивается на пользовательском опыте. Тестировщику необходимо поставить себя на место клиента. На основе составленных им замёток в процессе взаимодействия с приложением будут вноситься соответствующие изменения.

• Конфигурационное. Это проверка совместимости программы с аппаратным обеспечением и прочими software-элементами (различными версиями OS и процессоров). Конфигурационное тестирование необходимо для межплатформенных программ и в процессе перехода поставщика платформы на принципиально новую аппаратную базу (яркий пример — появление ноутбуков с чипами М1 от Apple).

2. Модульное тестирование, основные принципы и используемые подходы.

Модульное тестирование — это процесс проверки функциональности отдельных модулей программного обеспечения. Модуль — это независимый компонент программы, который может быть протестирован отдельно от других модулей.

В модульном тестировании программисты создают тестовые сценарии для каждого модуля, которые проверяют корректность его работы. Если тест не проходит, программисты находят и исправляют ошибки до тех пор, пока тест не будет пройден успешно.

Существует множество инструментов для модульного тестирования, таких как JUnit, NUnit, PHPUnit и другие. Они обеспечивают возможность создания тестовых сценариев и автоматического выполнения тестов.

«Ящичный» подход. Согласно этому подходу, все тесты ПО делятся на три вида ящиков:

• Тестирование типа «белый ящик» проверяет внутренние структуры и модули, игнорирует ожидаемую функциональность для конечных пользователей. Это может быть тестирование API, внесение неисправностей (fault injection), модульное тестирование, интеграционное тестирование.

• Тестирование типа «чёрный ящик» больше интересуется тем, что делает ПО, а не как делает. Это означает, что тестировщики не обязаны ни разбираться в объекте тестирования, ни понимать, как он работает под капотом. Такой тип тестирования нацелен на конечных пользователей, их опыт взаимодействия с видимым интерфейсом. К «чёрным ящикам» относится тестирование на основе моделей, тестирование способов использования, таблицы переходов состояний, спецификационное тестирование и т. д.

• Тестирование типа «серый ящик» проектируется со знанием программных алгоритмов и структур данных (белый ящик), но выполняется на пользовательском уровне (чёрный ящик). Сюда относится регрессионное тестирование и шаблонное тестирование (pattern testing).

3. Пакет JUnit, основные API.

JUnit — наиболее широко используемая среда тестирования для приложений Java. JUnit долгое время отлично справлялся со своей задачей.

org.junit.jupiter.api – JUnit Jupiter API для написания тестов.

org.junit.jupiter.api.condition – Условия включения или отключения тестов в JUnit Jupiter на основе аннотаций.

org.junit.jupiter.api.extension – JUnit Jupiter API для написания расширений.

org.junit.jupiter.api.function – Функциональные интерфейсы, используемые в JUnit Jupiter.

org.junit.jupiter.api.io – Поддержка ввода-вывода в JUnit Jupiter.

org.junit.jupiter.api.parallel – JUnit Jupiter API для влияния на параллельное выполнение тестов.

4. Системы автоматической сборки. Назначение, принципы работы, примеры систем.

Gradle — система автоматической сборки, которую используют для упрощения работы с Java. С помощью (условно) стандартизированных средств она помогает разработчикам собрать нужный продукт без потери его уникальности.

High-level архитектура. Архитектура всех build-систем сводится к следующему:

• конфигурации

• собственная конфигурация, где хранятся «личные» настройки системы. Например, такие как информация о месте установки или окружении, информация о репозиториях и прочее;

• конфигурация модуля, где описывается место расположения проекта, его зависимости и задачи, которые требуется выполнять для проекта;

• парсеры конфигураций

• парсер способный «прочитать» конфигурацию самой системы, для её настройки соответствующим образом;

• парсер конфигурации модуля, где некоторыми «понятными человеку» терминами описываются задачи для системы сборки;

• сама система — некоторая утилита + скрипт для её запуска в вашей ОС, которая после чтения всех конфигураций начнет выполнять тот или иной алгоритм, необходимый для реализации запущенной задачи;

• система плагинов — дополнительные подключаемые надстройки для системы, в которых описаны алгоритмы реализации типовых задач;

• локальный репозиторий — репозиторий, расположенный на локальной машине, для кэширования запрашиваемых файлов на удаленных репозиториях.

Подробнее об этом вы можете прочитать в инструкции к плагинам: JavaCompile (добавить новые задачи), SourceSet (добавить новые объекты домена). Также плагины работают с соглашениями и расширяют объекты. Например, с помощью плагина можно добавить новые элементы DSL и (или) настроить их.

Набор "Core Plugins" автоматически создаётся при установке Gradle. На первых этапах рекомендуют использовать категорию "Utility", а именно — плагин "Build Init Plugin". Он предоставляет задачи для инициализации проекта в системе.

После выбора задачи и её успешного завершения вы увидите внизу зелёную надпись "BUILD SUCCESSFUL". Это значит, что (вам повезло) всё прошло без проблем. Также внизу система выдаст краткий отчёт.

Если статус "executed" — задача действительно выполнена. Если "up-to-date" — нет. Это не значит, что произошёл какой-то сбой. В случае такого статуса задача не требует решения в принципе, т. е. её объект уже в актуальном состоянии.

Это произошло потому, что в Gradle автоматически формируется "Up-to-date checks" — инкрементальный билд, цель которого — оптимизация работы системы. Поэтому задачи, которые уже завершены или не требуют действий, не прорабатываются.

Отключить этот build можно. Вручную. Для этого необходимо выполнить задачу с указанием флага --rerun-tasks. Если всё сделано правильно, статус у всех задач изменится на "executed".

Также Gradle позволяет просматривать логи разного уровня. Например, команда gradle run -i позволит читать информационные сообщения о работе системы, команда run -q — включить режим тишины, run -d — показывать все сообщения журнала и т. д. Полный список логов и дополнительная информация о системе логирования тут.

5. Утилита make. Make-файлы, цели и правила.

Утилита make предназначена для автоматизации сборки проектов. Если какие-либо файлы проекта могут быть сгенерированы из других файлов, утилита позволяет выполнить процесс построения наиболее оптимальным способом, по возможности минимизируя количество обрабатываемых файлов.

На сегодняшний день наиболее распространены три варианта утилиты, объединенные общими принципами работы, но отличающиеся синтаксисом языка и возможностями:

• GNU make — самый распространенный и функциональный вариант

• BSD make (pmake) — используется в проектах BSD, по функциональности примерно соответствует GNU make

• nmake (Microsoft make) — работает под Windows, малофункционален, только базовые принципы make.

Утилита make работает по правилам (rules), записанным в специальном конфигурационном файле. Правила определяют цели (targets), завимости между целями и набор команд для выполнения каждой цели.

Правило make это ответы на три вопроса:

{Из чего делаем? (реквизиты)} ---> [Как делаем? (команды)] ---> {Что делаем? (цели)}

Цели могут соответствовать определенным файлам. Кроме того, цели могут не соответствовать ни одному файлу и использоваться для группировки других целей или определенной последовательности команд. Такие цели называются phony targets.

Каждая цель может зависеть от выполнения других целей. Выполнение цели требует предварительного выполнения других целей, от которых она зависит.

В случае зависимости между целями, соответствующими файлам, цель выполняется только в том случае, если файлы, от которых она зависит, новее, чем файл, соответствующий цели. Это позволяет перегенерировать только файлы, зависящие от измененных файлов, и не выполнять потенциально долгий процесс пересборки всех файлов проекта.

Таким образом, makefile определяет граф зависимостей, по которому утилита make выполняет ту или иную цель, по возможности минимизируя количество операций сборки.

6. Утилита Ant. Сценарии сборки, цели и команды.

Ant - это мощный платформо-независимый скриптовый инструмент, используемый для сборки приложений. Сценарий сборки приложения java оформляется в виде XML-файла подобно скриптам "make" при обработке файлов С/С++. По-умолчанию сценарий сборки извлекается из файла build.xml.

Пример описания сценария сборки:

• копирование \*.jar файлов в каталог релиза, но перед этим необходимо

• сформировать \*.jar файлы, но перед этим необходимо

• скомпилировать java-файлы в файлы \*.class

Сценарий сборки ant'у показывает что надо делать, чтобы превратить из того, что есть (как правило, исходный java-код) в то, что необходимо. Сценарий представляет собой детальный план сборки из частей единого целого, включающий ряд операндов, позволяющих выполнять команды копирования, удаления и перемещения файлов, компиляции java-файлов, формирование документации к коду и исполняемого jar-файла. Корневой элемент сценария project может содержать три необязательных атрибута:

• name - имя проекта;

• default - цель проекта по умолчанию;

• basedir - базовая директория, относительной которой будут вычисляться все пути.

Элемент, описывающий цель проекта target может содержать следующие атрибуты:

• name - имя цели, обязательный атрибут;

• depends - промежуточные цели, от которых зависит данная цель; имена перечисляются через запятую;

• if - определяет какие свойства должны быть равны true для запуска цели;

• unless - определяет какие свойства должны быть равны false для запуска цели;

• description - краткое описание цели, что она делает.

Параметр property определяет пару имя/значение, которая может многократно использоваться в сценарии подобно переменным. Свойства (настройки) можно определять как внутри build.xml файла, так и в отдельных файлах. При определении внутри xml файла свойства могут включать следующие атрибуты:

• name - имя свойства;

• value - значение свойства;

• location - устанавливает значение свойства в абсолютный путь. Если значение уже абсолютный путь, то ничего не меняется, если относительный, то подставляется базовая директория. Символы / и \ меняются автоматически в зависимости от платформы;

• refid - ссылка на другой объект, определенный где-либо;

• resource - имя ресурса содержащего настройки в формате настроечного файла;

• file - путь к файлу настройки (в нем свойства определяются как имя=значение на отдельной строке);

• url - адрес настройки;

• environment - префикс используемый для доступа к переменным окружения. Например, если определено myenv, то к переменным обращаются как "myenv.PATH";

• classpath - путь к ресурсам;

• prefix - префикс добавляемый к свойствам загруженных из файла, ресурса, или url. По умолчанию префикс ".".