

1. Make и написать makefile для сборки джавы

На слайде приведен пример файла, задающего конфигурацию сборки проекта при помощи утилиты make. Make впервые появилась в Unix-экосистеме и существует по сей день как средство сборки низкого уровня. В файле сборки (Makefile) указаны опции компиляции и зависимости, необходимые для работы make.

Внимательно изучив пример, можно заметить, что в makefile прописаны цели, которые нужно собрать, после них через двоеточие указаны файлы, от которых зависят данные цели, а на следующей строке — последовательность действий, которые нужно выполнить для сборки каждой, формируя показанное дерево зависимостей.

При запуске make, утилита анализирует файл, начиная с цели "all" (стандартная цель по умолчанию), после чего собирает программы с учетом зависимостей. Также учитывается время последней модификации файлов. Если время последней модификации файла позже, чем время модификации любой его зависимости, то данную цель в makefile необходимо пересобрать.

Как заюзать под джаву?

1. create a file named 'makefile' in your project directory with the same content
2. modify the CLASSES macro so that it has the names of your .java files;
3. run 'make', and if all goes well, it should compile all of your java source files that need to be re-built.

JFLAGS = -g

JC = javac

JVM= java

FILE=

.SUFFIXES: .java .class

```
CLASSES = \  
    Experiment.java \  
    Block.java \  
    Spring.java \  
    PhysicsElement.java \  
    Simulator.java
```

MAIN = Experiment

default: classes

classes: \$(CLASSES:.java=.class)

run: \$(MAIN).class

\$(JVM) \$(MAIN)

2. Gradle

Gradle — система автоматической сборки, построенная на принципах Apache Ant и Apache Maven, но предоставляющая на языках Groovy и Kotlin вместо традиционной XML-образной формы представления конфигурации проекта.

- Нейтрален к языкам программирования
- Базируется на Apache Ivy
- Подобно Maven использует плагины и жизненный цикл
 - Основные задачи по сборке определены в плагины
 - Плагины определены для многих типов проектов
- Описание зависимостей от Maven (GAV)
 - Может использовать репозитории Maven и Ivy
- В качестве скрипта сборки использует DSL (Domain Specific Language) на Groovy
- Инкрементальная и параллельная сборка

3. Непрерывная интеграция

Непрерывная интеграция (CI, англ. *Continuous Integration*) — практика разработки программного обеспечения, которая заключается в постоянном слиянии рабочих копий в общую основную ветвь разработки (до нескольких раз в день) и выполнении частых автоматизированных сборок проекта для скорейшего выявления потенциальных дефектов (тестирования) и решения интеграционных проблем.

4. Как написать плагин на мавен



Плагины

- Все операции над проектом выполняются плагинами.
- Плагины в качестве точек входа содержат цели, связанные с ЖЦ:
 - Core: clean compiler deploy failsafe install resources site surefire verifier.
 - Packaging: ear ejb jar rar war app-client/acr shade source verifier.
 - Reporting, Tools, and thirdparty.

<http://maven.apache.org/plugins/index.html>

- Плагины указываются в файле **pom.xml** внутри блока `<plugins>` `</plugins>`
- Каждый плагин может иметь несколько целей.
- Мы можем определять фазу, из которой мы можем начать выполнение плагина. В примере выше мы использовали фазу **compile**.

Плагин должен реализовывать метод `execute()` абстрактного класса `AbstractMojo` и на это все требования к плагину формально заканчиваются.

```
/**
 * Goal which changes a greeted name.
 */
@Mojo(name = "setname", defaultPhase = LifecyclePhase.PROCESS_CLASSES, threadSafe = true)
public class NamingMojo extends AbstractMojo {
    @Parameter(property = "name")
    private String name;


    @Parameter(defaultValue = "${project}", readonly = true)
    private MavenProject project;

    public void execute() throws MojoExecutionException {
        getLog().info("Updating greeter target");

        /* skipped *.
    }
}
```

```
<plugin>
  <groupId>ru.easyjava.maven</groupId>
  <artifactId>name-maven-plugin</artifactId>
  <version>1</version>
  <configuration>
    <name>Will greet:</name>
  </configuration>
  <executions>
    <execution>
      <id>process</id>
      <phase>process-classes</phase>
      <goals>
        <goal>setname</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

5. JUnit 4



JUnit

- JUnit фреймворк обеспечивает:
 - Аннотации для маркировки метода как теста `@Test`.
 - Аннотации для маркировки действий до и после теста `@Before`, `@After`, `@BeforeClass`, `@AfterClass`.
 - Методы для проверки (assertion).
 - UI, журнал тестов...

JUnit4 построен на аннотациях. Метод, который организует тестирование, помечается аннотацией `@Test`. При этом фреймворк тестирования последовательно просматривает загружаемые классы (при помощи reflection API) и ищет в них эту аннотацию. Все методы с найденной аннотацией запускаются (возможно, в разных потоках).

Внутри тестового метода проверяется тестовое покрытие на соответствие определенным условиям при помощи специальных функций проверки, которые называются assertion. В JUnit4 существует большой набор таких функций, которые могут проверять на равенство, совпадения, появление исключительной ситуации и так далее. Результаты тестов заносятся в специальный журнал для последующего анализа.

Для организации тестового окружения существуют аннотации, позволяющие выполнить код перед всеми тестами, после всех тестов, а также при загрузке тестового класса в память и выгрузки его из неё.

Последовательность тестов в JUnit не регламентирована (порядок можно искусственно определить, но по умолчанию считается, что все тесты выполняются параллельно и независимо). JUnit самостоятельно определяет, каким образом будет запущено тестирование.

- `assertEquals(int1, int2)` или утверждение эквивалентности. Проверяет на равенство двух значений любого примитивного типа;
- `assertFalse, assertTrue(condition)` или булевы утверждения. Вместо "condition" необходимо вставить проверяемое условие;
- `assertNull, assertNotNull(obj)` относятся к Null утверждениям и проверяет содержимое объектной переменной на Null значение;
- `assertSame(obj1, obj2)` утверждение позволяет сравнивать объектные переменные.

6. Gradle - сравнить с maven и ant

Проблемой декларативной сборки Maven является то, что она по сути своей является ограниченно-декларативной. Это значит, что пока ваше приложение подчиняется основной логике, которая заложена в плагины — проект конфигурируется быстро и собирается хорошо. Если по какой-либо причине вам нужно собрать что либо, не описываемое плагином — тут начинаются сложности. Скорее всего, вам придется разработать собственный плагин, что не является тривиальной задачей. Кроме того, бывает сложно разобраться в XML-файле проекта, который содержит много проектов и зависимостей. Де-факто, XML является удобным форматом для его программного анализа, а не для написания человеком.

Средством, которое является удачным последователем Ant и Maven, которое получило в современном мире разработчиков большое распространение, и которое сообщество считает «стильным, модным и молодежным», является Gradle. Основными достоинствами его являются:

- Нейтральность к языкам программирования. В отличие от Maven, который предназначен, в основном, для сборки Java-приложений, gradle никак не ограничивает вас в языках программирования. Вы можете указать, как собрать приложение, которое вы можете разрабатывать на множестве различных языков.
- Преемственность описания зависимостей и возможность использования настраиваемых репозиториях зависимостей, в том числе от Maven, используя различные нотации описания.
- Использование лаконичного DSL (Domain Specific Language — проблемно-специфичного языка) для описания сборки, который сам по себе разработан и использует возможности Groovy.
- Инкрементальная и параллельная сборка — позволяет запускать плагины только при наличии действительной необходимости выполнения задач по сборке, и экономит время сборки проекта. Инкрементальная сборка существует в различных системах сборки, но, например, в Maven все равно тратится достаточное количество времени на выполнение плагина, даже если собирать нечего.

7. JUnit - про тайм-аут спросил

Если тест занимает больше времени, чем определенное «время ожидания», для завершения будет выдано «`TestTimedOutException`» и тест, помеченный как неудачный. Есть 2 способа декларирования таймаута – глобальный и

```
public class TimeoutRuleTest {  
  
    //global timeout rule  
    @Rule  
    public Timeout globalTimeout = Timeout.seconds(1);  
  
    //This test will be failed, because it will take more than 1 second to finish!  
    @Test  
    public void testSlowMethod1() throws InterruptedException {  
        //...  
        TimeUnit.SECONDS.sleep(5000);  
    }  
  
    //passed  
    @Test  
    public void testSlowMethod2() {  
        //...  
    }  
}
```

Этот пример `timeout` относится только к одному методу тестирования. И значение тайм-аута в миллисекундах.

```
TimeoutTest.java  
package com.mkyong;  
  
import org.junit.Test;  
  
public class TimeoutTest {  
  
    //This test will always failed :)  
    @Test(timeout = 1000)  
    public void infinity() {  
        while (true) ;  
    }  
  
    //This test can't run more than 5 seconds, else failed  
    @Test(timeout = 5000)  
    public void testSlowMethod() {  
        //...  
    }  
}
```


локальный

8. Dependency hell

Dependency hell (англ. *ад зависимостей*) — это антипаттерн управления конфигурацией, разрастание графа взаимных зависимостей программных продуктов и библиотек, приводящее к сложности установки новых и удаления старых продуктов. В сложных случаях различные установленные программные продукты требуют наличия разных версий одной и той же библиотеки. В наиболее сложных случаях один продукт может косвенно требовать сразу две версии одной и той же библиотеки.

Виды проблем:

1. Множество зависимостей

Приложение зависит от большого числа объемных библиотек, которые требуют длительных скачиваний, занимают много дискового пространства.

2. Длинные цепочки зависимостей

Приложение зависит от библиотеки "А", которая зависит от библиотеки "Б", ... , которая в свою очередь зависит от библиотеки "Я".

3. Конфликтующие зависимости

Если "Приложение 1" зависит от библиотеки "А" версии 1.2, а "Приложение 2" зависит от той же библиотеки "А", но уже версии 1.3, и различные версии библиотеки "А" не могут быть одновременно установлены, то "Приложение 1" и "Приложение 2" нельзя одновременно использовать (или даже установить, если установщики проверяют зависимости).

4. Циклические зависимости

Ситуация, когда приложение "А" версии 2 зависит от приложения "Б", которое зависит от приложения "В", которое в свою очередь зависит от приложения "А", но версии 1.

9. Как организовать модульное тестирование в JUnit?

[СМОТРИ ДРУГОЙ ВОПРОС](#)

10. JUnit. Как писать тесты. Как проверить, что метод не завис.
Параметризация тестов (с параметрами)

Параметры в тестах

Проверка времени исполнения метода – установка таймаута

11. Что такое цели и зависимости в ant?

По мнению авторов, XML не является очень удобным для чтения человеком, хотя прекрасно приспособлен для машинной обработки. Как видно из примера, описание проекта состоит из именованных целей (target) с явным указанием зависимостей — других целей, что позволяет создать из зависимостей дерево. Кроме того, внутри каждой цели определены действия, которые необходимо выполнить в процессе сборки.

Цели можно вызывать (antcall) в явном виде (не через зависимости). Возможен вызов целей из другого файла, что позволяет собирать систему из модулей.

Каждый build-файл содержит один проект (project) и хотя бы одну цель (target). Цель содержит задачи (tasks), задачи могут использовать для своей работы отдельные ресурсы (resources) (например, файл) или коллекцию ресурсов (множество файлов). Кроме целей, задач и ресурсов есть свойства (properties). К свойствам можно относиться как к переменным, а точнее константам. У свойства есть имя и значение в виде строки. Значение свойства устанавливается один раз и любая попытка изменить это значение игнорируется.

Задачи представляют действия, которые могут осуществляться с какими-либо ресурсами или без их участия.

12. Чем отличается maven от ant? Плюсы и минусы.

Императивный — Ant, декларативный — Maven

Императивный подход хорош до тех пор, пока удобно последовательно описывать сборку каждой из частей программы. Со временем стало очевидно, что для особенно сложных проектов императивный подход создает путаницу из-за большого количества директив по сборке файлов. При этом становится сложно отслеживать правильную последовательность действий при сборке проекта.

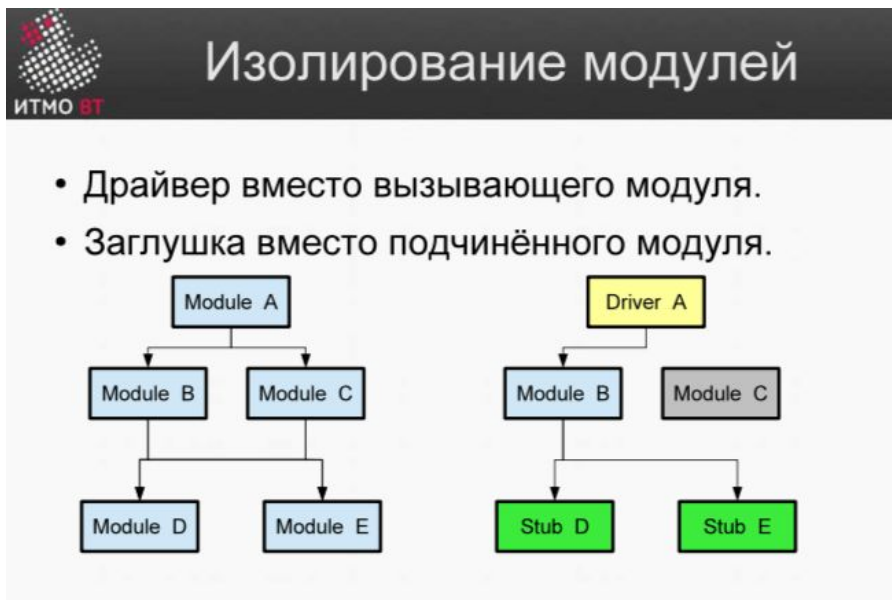
Как решение, появились декларативные средства разработки. Они построены на тех же принципах, что и императивные, но большинство действий по сборке выполняется прозрачно для разработчика, а сами действия вызываются через интерфейс верхнего уровня декларативно. Пользователь указывает, *что* он желает собрать, а не *как*, и средство сборки самостоятельно выполняет необходимую последовательность команд. При этом, у пользователя остаётся возможность модификации выполняемых команд сборки.

В Maven, в отличие от Ant, существует система каталогов проекта по умолчанию.

В ветке src/main расположены основные исходные файлы. Некоторые из них необходимо компилировать (src/main/java), некоторые непосредственно копируются в целевую директорию target. Отдельно выделяются зависимые от языка пользователя ресурсы в файле .properties. С точки зрения компиляции, для директорий исходных файлов и ресурсов и выполняются разные цели.

Аналогично разделены каталоги для тестов, относящихся к исходным файлам Java и к ресурсам.

13. Заглушки




- Драйвер вместо вызывающего модуля.
- Заглушка вместо подчинённого модуля.

Драйвер — компонент, вызывающий модули и обеспечивающий последовательность тестирования. Драйвер должен последовательно вызывать тестируемый модуль с различными входными параметрами и условиями.

Зاغлушка ведет себя подобно подчинённому модулю, имеет тот же интерфейс, но гораздо более простую реализацию. При вызове заглушка возвращает заранее определённые значения. Однако при вызове заглушки не предусматривается подача входных значений, отличающихся от предусмотренных в заглушке. Часто для создания заглушек используется табличный метод с соотнесением входных параметров с возвращаемыми заглушкой. Такой подход хорош своей наглядностью.


14. Как maven понимает, какие файлы использовать при сборке (структура maven проекта)




Структура проекта

- target — рабочая и целевая директории.
- src/main — основные исходные файлы:
 - src/main/java — исходные файлы java.
 - src/main/webapp — web-страницы, jsp, js, css...
 - src/main/resources — файлы, которые нет необходимости компилировать.
- src/test — исходные файлы для тестов:
 - src/test/java
 - src/test/resources


15. модульное, функциональное, интеграционное тестирование

 **Модуль**

- Модульное (компонентное) тестирование - тестирование отдельных компонентов программного обеспечения [IEEE 610].
 - Метод или класс.
 - Программный модуль.
- Модули описаны в дизайне.
- Для тестирования необходимо изолировать модуль из системы.

 **Интеграционное тестирование**

- Проверяет интерфейсы и взаимодействие модулей (компонент) или систем:
 - Вызовы API, сообщения между ОО компонентами.
 - Базы данных, пользовательский графический интерфейс.
 - Интерфейсы взаимодействия (сетевые, аппаратные, локальные, ...).
 - Инфраструктурные.
- Может проводиться, когда два компонента разработаны (спроектированы):
 - Остальные добавляются по готовности.

 **Функциональное тестирование**

- На базе сценариев использования.
- Ручное/автоматическое.
- На готовой системе, в рамках модульного и интеграционного.
- Проверяются функции системы, начиная с интерфейса пользователя.
- Средства автоматизации:
 - Открытые: Selenium, Sahi, Watir.
 - Коммерческие: от HP, Rational (IBM)...



Статическое и динамическое тестирование

- Статическое (рецензирование):
 - Не включает выполнения кода.
 - Ручное, автоматизированное.
 - Неформальное, сквозной контроль, инспекция.
- Динамическое:
 - Запуск модулей, групп модулей, всей системы.
 - После появления первого кода (при TDD — иногда перед!)

В V-model каждой стадии разработки ПО соответствует свой уровень тестирования. Код ПО, иерархически организованный в виде модулей, покрывается модульными тестами, а архитектура и взаимодействие слоёв архитектуры между собой — интеграционным тестированием.

Формирование и анализ требований покрываются системным тестированием. Реализующая эти требования программа принимается заказчиком в эксплуатацию во время приёмочного тестирования, где проверяются основные характеристики программы, которые были заложены в требованиях.

16. Автоматизация тестирования: регрессионное, повторное, приемочное
- 17.

Приемочное тестирование – это комплексное тестирование, необходимое для определения уровня готовности системы к последующей эксплуатации. Тестирование проводится на основании набора тестовых сценариев, покрывающих основные бизнес-операции системы.

Автоматизация тестов является положительным явлением, но часто ручное тестирование оказывается дешевле и проще, особенно при простых задачах, где легче нанять низкоквалифицированный персонал, чем написать сложную программу, формирующую сценарии тестирования и меняющую эти сценарии при изменении функционала.

Отдельный вид тестирования — это т.н. регрессионное тестирование, которое заключается в том, что при изменении программы запускаются старые тесты. Это позволяет проверить, не повлияли ли внесённые изменения на работу всей программы и не появились ли нарушения в этой работе.

Чтобы автоматизировать тесты и пользоваться регрессионным тестированием, необходимо обеспечить однозначное повторение тестового сценария (внутри ПО не должно быть вариативности в поведении, одинаковые входные данные должны порождать одинаковые выходные данные).

Важной является проверка одного и того же приложения в разных окружениях (т.н. тестирование совместимости). Необходимость повторять тесты в различных окружениях порождает большую сложность всего процесса. Необходимо выполнить много тестов, поэтому чем меньше будет сформировано изначальных функциональных тестов, тем меньше общая сложность и длительность выполнения таких тестов.

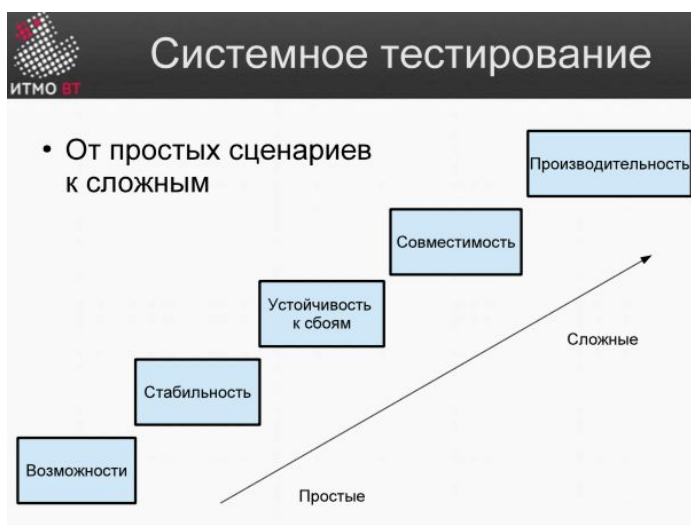
18. Системное, Альфа/Бета, приемочное тестирования

- *Системное тестирование*. Оно обычно выполняется внутри организации-разработчика без привлечения сторонних лиц.

- *Альфа- и Бета-тестирование* — выполняется пользователем под контролем разработчика. За счёт этого разработчики получают полезные для завершения разработки отзывы. Кроме того, так как тестирование выполняют не разработчики, то тестирующие могут пойти нестандартными путями, которые разработчики не учли. При этом альфа-тестирование производится на окружении разработчиков, а бета-тестирование — в реальном пользовательском окружении (но всё равно под контролем разработчиков).

- *Приёмочное тестирование* — выполняется пользователем в его собственном окружении без контроля разработчика. На этом этапе решается вопрос о выплате разработчикам гонорара.

Методики всех этапов тестирования системы в целом практически одинаковые, отличия есть только в строгости интерпретации результатов.



19. Цели тестирования ПО

повышение пользовательского доверия, предотвращение дефектов, проверка соответствия заявленным требованиям, нахождение дефектов, валидация, верификация, получение результатов


The slide, titled "Основная цель тестирования", lists the primary goals of testing:

- Увеличение приемлемого уровня пользовательского доверия в том, что программа функционирует корректно во всех необходимых обстоятельствах.
 - Корректное поведение.
 - Уровень доверия.
 - Необходимые обстоятельства — требование реального окружения.

Цели тестирования (ISTQB)

- ISTQB — International Software Testing Qualifications Board
 - www.istqb.org
 - www.rstqb.org
- Цели тестирования:
 - Обнаружение дефектов.
 - Повышение уверенности в уровне качества.
 - Предоставление информации для принятия решений.

20. Ошибка Дефект Failure Error



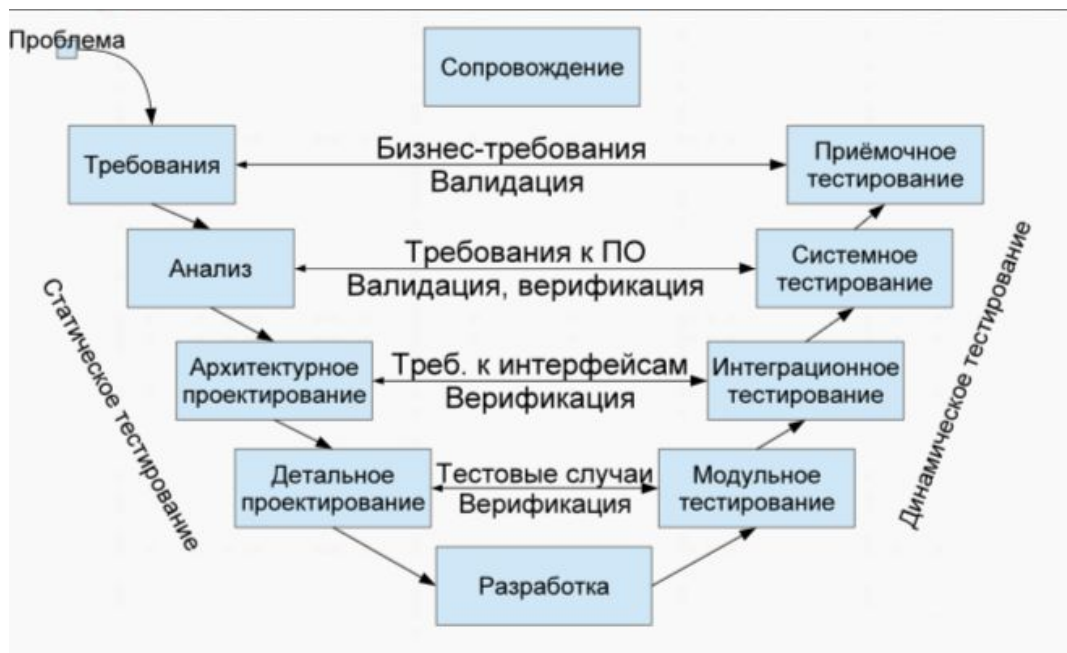
Термины

Люди ошибаются

- Mistake (Error) — ошибка, просчёт (человека).
- Fault — дефект, изъян (ПО в результате ошибки).
- Failure — неисправность, отказ, сбой (внешнее проявление дефекта).
- Error — невозможность выполнить задачу вследствие отказа.
- BUG — используется неформально. Может обозначать: дефект, отказ, невозможность выполнить задачу.
 - Что-то другое или ничего не обозначать.

Отказ м.б. следствием окружающей среды

21. Верификация, Валидация



Валидация — мероприятия по проверке корректности требований к программному обеспечению. Верификация — проверка соответствия ПО сформулированным требованиям.

22. Software bill of materials

A software bill of materials is a list of all the open source and third-party components present in a codebase. A software BOM also lists the licenses that govern those components, the versions of the components used in the codebase, and their patch status.

Software composition analysis tools can generate a complete software BOM that tracks third-party and open source components and identifies known security vulnerabilities, associated licenses, and code quality risks.