Software technology

Build systems

Máté Cserép

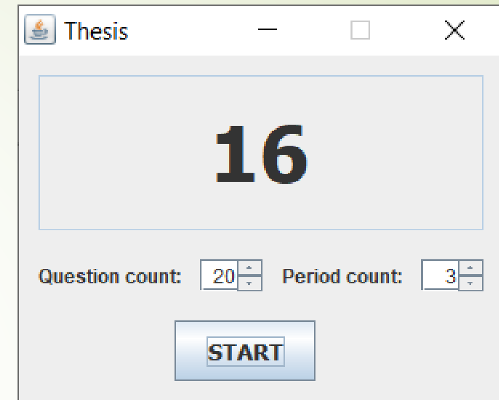
ELTE, Faculty of Informatics

2020.

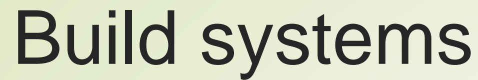
Build systems

Sample application

- Let's have a sample Java application.
- The ThesisGenerator application can generate thesis serial number for a verbal examination.
  - https://szofttech.inf.elte.hu/mate/thesisgenerator-java
- Model-View architecture:
  - `thesisGenerator.model` package: UI independent business logic
  - `thesisGenerator.view` package: Swing based UI
  - `thesisGenerator` package: Main program



Build systems



Compiling Java programs

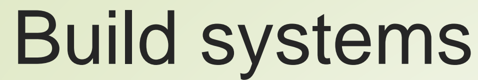
```
mkdir dist
javac -d dist
  src\thesisGenerator\*.java
  src\thesisGenerator\model\*.java
  src\thesisGenerator\view\*.java


cd dist
jar -cfe thesis-generator.jar
  thesisGenerator.ThesisGenerator
  thesisGenerator\*.class
  thesisGenerator\model\*.class
  thesisGenerator\view\*.class


java -jar thesis-generator.jar
```

Build systems

Compiling Java programs

- That was not simple for such a basic program with a few source files all together.

- **Problem statement:**
  - Compiling program manually with console commands can easily can get difficult to manage even for smaller applications with a couple source files.
  - Working with larger programs is becomes untrackable which translation units require recompilation.
    - Recompiling the complete application can take a long time for an enterprise application.



Build systems

Requirements towards build systems

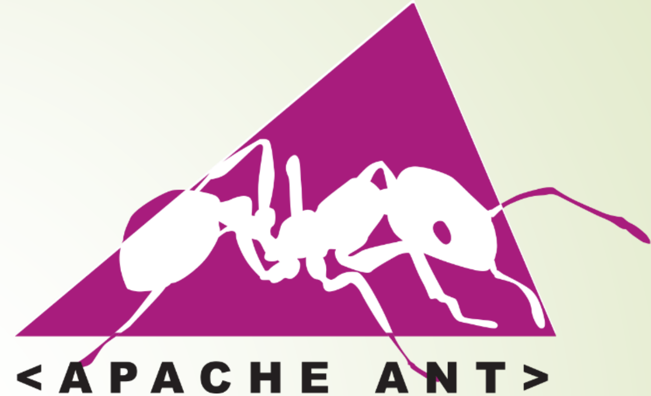
- Compiling the code
  - Manage dependencies of compilation targets
- Packaging the binaries
  - Support multiple release options
- Perform automatized tests
- Deploying the binaries to the test server
- Copying the code from one location to another

- Management of package repositories

Build systems - Ant

Features

- Imperative approach
- Typically used for Java projects
- XML-based build file
  - Named *build.xml* by default
- Official website and tutorial:
  - https://ant.apache.org/
  - https://ant.apache.org/manual/tutorial-HelloWorldWithAnt.html
- Installation:
  - Windows installer can be downloaded from official website.
  - UNIX systems: from package repository.
    - Debian/Ubuntu: apt-get install ant
  - Together with an IDE, e.g. NetBeans installs Ant.



Build systems - Ant

Build file (build.xml)

- The *build.xml* file contains a *project* root element, which sets:
  - Name of the project
  - Default target (discussed later)
  - The base directory of the project, typically the current folder.

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="projname"
         default="deftarget"
         basedir=".">
  ...
</project>
```

Build systems - Ant

Define a target

- Inside the project element multiple type of elements can be defined. The most important are *targets*:

```
<project ...>
  <target name="compile">
    ...
  </target>
</project>
```

  - Targets can be executed on the command line: `ant compile`

Build systems - Ant

Directory creation

- Create a target which creates the *classes* directory for the *.class* files to be compiled:

```
<project ...>
  <target name="prepare">
    <mkdir dir="classes"/>
  </target>
</project>
```

  - Command line:
    `ant prepare`

Build systems - Ant

Target dependencies

- Define a target to compile all Java source files in the *src* directory. Place the output inside the *classes* folder.
  - Make the *compile* target depend on the *prepare* target!

```
<project ...>
  <target name="compile" depends="prepare">
    <javac srcdir="src" destdir="classes"/>
  </target>
</project>
```

  - Command line:
    `ant compile`

Build systems - Ant

Cleanup target

- Add a cleanup target, which removes all compilation binaries.

```
<project ...>
  <target name="clean">
    <delete>
      <fileset dir="classes" includes="*"/>
    </delete>
    <delete dir="classes"/>
  </target>
</project>
```

  - Deleting the files are not required, removing the classes folder removes its content recursively.
  - Command line:
    `ant clean`

Build systems - Ant

Cleanup target

- The `failonerror` attribute configures the target whether to fail the complete process if that target fails.

```
<project ...>
  <target name="clean" failonerror="false">
    <delete dir="classes"/>
  </target>
</project>
```

- Command line:
  `ant clean`

Build systems - Ant

Properties

- Inside a project we can also defines properties.
  - Properties are key-value pairs.
  - Evaluated at runtime with the `${name}` syntax.

```
<project ...>
  <property name="jarname"
            value="filename.jar" />
  ...
</project>
```

  - There are also built-in properties, e.g. the `${basedir}` is the base project directory.
    - https://ant.apache.org/manual/properties.html

Build systems - Ant

Packaging

```
<project ...>
  <target name="jar" depends="compile">
    <jar destfile="${jarname}">
      <fileset dir="classes">
        <include name="*.class"/>
      </fileset>
      <manifest>
        <attribute name="Main-Class" value="Main"/>
      </manifest>
    </jar>
  </target>
</project>
```

- **Note:** it is important to set the entry point in the manifest!

Build systems - Ant

Target: complex example

```
<target name="compile" depends="prepare,init">
  <javac destdir="build/classes" debug="on">
    <src path="src1/java"/>
    <src path="src2/java"/>
    <include name="**/*.java"/>
    <exclude name="com/comp/xyz/applet/*.java"/>
    <classpath>
      <fileset dir="lib">
        <include name="*.jar"/>
      </fileset>
    </classpath>
  </javac>
</target>
```

Build systems - Ant

Deploying (file operations)

- Deploy by copying the final binaries to a target destination:

```
<target name="install" depends="jar">
  <mkdir dir="build/war/WEB-INF/lib"/>
  <copy todir="build/war/WEB-INF/lib">
    <fileset dir="lib">
      <include name="*.jar"/>
      <exclude name="servlet-api.jar"/>
      <exclude name="catalina-ant.jar"/>
      <exclude name="el-api.jar"/>
    </fileset>
  </copy>
</target>
```

  - Command line:
    `ant install`

Build systems - Ant

JVM launch

- A target for executing the compiled and packaged JAR file can also be defined:

```
<target name="run" depends="compile">
  <java jar="${jarname}" fork="true" />
</target>
```

  - The `fork` attribute causes the task to run in a different process, and a different Java virtual machine (JVM).
  - Command line:
    `ant run`

Build systems - Ant

JVM launch

- More complex example:

```
<target name="run">
  <java classname="com.comp.foo.TestClient"
        jvmargs="-Xdebug server=y,suspend=n">
    <classpath>
      <fileset dir="lib">
        <include name="*.jar"/>
      </fileset>
    </classpath>
  </java>
</target>
```

  - Command line:
    `ant run`

Build systems - Ant

Generating API documentation

```
<target name="doc">
  <tstamp>
    <format property="timestamp" pattern="d.M.yyyy"
            locale="en"/>
  </tstamp>
  <mkdir dir="doc"/>
  <javadoc sourcepath="src" destdir="doc"
           windowtitle="Project documentation">
    <header>Very Important Project</header>
    <footer>Javadocs compiled ${timestamp}></footer>
    <fileset dir="src/" includes="**/*.java" />
  </javadoc>
</target>
```

Build systems - Ant

Complete build.xml file for the ThesisGenerator app

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project name="ThesisGenerator" default="jar" basedir=".">
    <target name="clean">
        <delete dir="build"/>
    </target>
    <target name="compile">
        <mkdir dir="build/classes"/>
        <javac srcdir="src" destdir="build/classes"/>
    </target>
    <target name="jar" depends="compile">
        <mkdir dir="build/jar"/>
        <jar destfile="build/jar/ThesisGenerator.jar" basedir="build/classes">
            <manifest>
                <attribute name="Main-Class" value="thesisGenerator.ThesisGenerator"/>
            </manifest>
        </jar>
    </target>
    <target name="run" depends="jar">
        <java jar="build/jar/ThesisGenerator.jar" fork="true"/>
    </target>
    <target name="doc" depends="compile">
      <tstamp>
        <format property="timestamp" pattern="d.M.yyyy" locale="en"/>
      </tstamp>
      <mkdir dir="doc"/>
      <javadoc sourcepath="src" destdir="doc" windowtitle="Thesis Generator">
        <header>Thesis Generator</header>
        <footer>Javadocs compiled ${timestamp}</footer>
        <fileset dir="src/" includes="**/*.java" />
      </javadoc>
    </target>
</project>
```

Build systems - Ant

NetBeans

- By default Netbeans uses Ant as a build system.
  - *build.xml* is located in the project root
  - references *nbproject/build-impl.xml*, which is generated by Netbeans and shouldn't be modified
  - Targets as hooks can be definied in *build.xml*, which will be called by Netbeans's build process automatically:
    - *-pre-init*, *-post-init*, *-pre-compile*, *-post-compile*, *-pre-jar*, *-post-jar*, *-post-clean*, etc.

Build systems - Maven

Features

- Software project management tool
- Building project, running tests, managing dependencies, documentation
- Packages: automatic download of dependencies
- Declarative specification of the build process
- Fix, predefined directory structure, conventions
- Typically Java, but it can handle other language via plugins
- XML-based build file
  - Named *pom.xml* by default
- Official website and a recommended tutorial:
  - http://maven.apache.org/
  - https://www.baeldung.com/maven

Build systems - Maven

Installation

- Standalone installation
  - Binaries are available on official website for download
  - Simply extract it to a preferred location
  - Set the `MAVEN_HOME` env. variable to point to this location
  - Also ensure that the `JAVA_HOME` env. variable points to your JDK installation folder
  - Add the `MAVEN_HOME\bin` folder to your `PATH`.
- UNIX package repository installation
  - Usually available
  - Debian/Ubuntu: `apt-get install maven`
- Typically ships bundled with IDEs (e.g. NetBeans, IntelliJ)

Build systems - Maven

Project

- Project Object Model (POM)
- Project uniquely identified by project's group, artifact Id, version, the 3 abbreviated as GAV together
- Project can divided to multiple modules that can be handled independently

Build systems - Maven

Project Object Model (pom.xml)

- The Project Object Model (pom.xml) is a specification of the project's all important information:
  - identifiers: groupId, artifactId, version
  - how the project is built
  - result of the build
  - test cases for the project
  - dependencies of the project

Build systems - Maven

Project Object Model (pom.xml)

- The root element of the pom.xml file is also a project element.
- The following elements must be defined inside the project:
  - modelVersion: version of the POM specification
  - groupId: unique base name of the company or group that created the project. Group ID should follow Java's package name rules. This means it starts with a reversed domain name, e.g. *hu.elte.inf*
  - artifactId: unique name of the project
  - version: version of the project
  - packaging: applied packaging method(default is jar, other options: pom, maven-plugin, ejb, war, ear, rar, par)

Build systems - Maven

Project Object Model (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>com.mycompany.software</groupId>
  <artifactId>app</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  ...
</project>
```

Build systems - Maven

Directory structure

- A Maven project has a directory structure based on defined conventions.
  - The default directory layout can be overridden using project descriptors, but this is uncommon and discouraged.

```
my-app
|-- pom.xml
`-- src
    |-- main
    |   |-- java
    |   |   `-- com
    |   |       `-- mycompany
    |   |           `-- app
    |   |               `-- App.java
    |   `-- resources
    |       `-- META-INF
    |           |-- application.properties
    `-- test
        |-- java
        |   `-- com
        |       `-- mycompany
        |           `-- app
        |               `-- AppTest.java
        `-- resources
            `-- test.properties
```

Build systems - Maven

Directory structure override

```
<project>
   ...
   <build>
     <directory>target</directory>
     <outputDirectory>classes</outputDirectory>
     <finalName>${project.artifactId}-${project.version}</finalName>
     <testOutputDirectory>test-classes</testOutputDirectory>
     <sourceDirectory>src/main/java</sourceDirectory>
     <scriptSourceDirectory>src/main/scripts
     </scriptSourceDirectory>
     <testSourceDirectory>/src/test/java</testSourceDirectory>
     ...
   </build>
</project>
```

Build systems - Maven

Lifecycle phases

- Maven build system follows a specified *lifecycle*, consisted of phases. The most important phases of the default lifecycle:
  - **validate:** validate the project is correct and all necessary information is available
  - **compile:** compile the source code of the project
  - **test:** test the compiled source code using a suitable unit testing framework. These tests should not require the code be packaged or deployed
  - **package:** take the compiled code and package it in its distributable format, such as a JAR.
  - **integration-test:** process and deploy the package if necessary into a testing environment where additional integration tests can be run

Build systems - Maven

Lifecycle phases

  - **verify:** run any checks to verify the package is valid and meets quality criteria
  - **install:** install the package into the local repository, for use as a dependency in other projects locally
  - **deploy:** done in an integration or release environment, copies the final package to the remote repository for sharing with other developers and projects.
- Further details: http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html
- Command line: `mvn install`
  - Performs the phases until *install* in the default lifecycle, but not the *deploy* phase.

Build systems - Maven

Lifecycle phases

- Beside the default lifecycle, there are other lifecycles, e.g. the clean lifecycle, which can be used to purge previously built binaries from a project.
  - This lifecycle has 3 phases: pre-clean, clean, post-clean.
- Command line: `mvn clean install`
  - Performs the *clean* and then the *install* phases and all phases before them.
  - Ultimately this will remove and rebuild all binaries.

Build systems - Maven

Goals

- Compilation phases consist of one or multiple *goals*
- The goal is a task that is related to the project's compilation or management
  - The order of these goals depends on the phase's binding
  - Many phases contain only one goal
  - E.g. compile phase consists of the `compiler:compile` goal
- Not only phases, but goals can also be passed to the Maven command, peforming only that goal without the previous phases and their goals.
  - E.g. `mvn compiler:compile`
- Custom phases and goals can be defined (in the pom.xml)

Build systems - Maven

Repositories

- A repository holds build artifacts and dependencies.
  - The default local repository in the developer's home folder: *~/.m2/repository*
- If an artifact is available in the local repository, Maven uses it.
- Otherwise, it is downloaded from a central repository and stored in the local repository.
  - Network traffic and build time can be significantly increased for the first build of a project.
  - The default central repository is the Maven Central: https://repo.maven.apache.org
  - Maven can be configured to which repositories and mirrors to use in the *~/.m2/settings.xml* file.
    - Companies often have internal central repositories.

Build systems - Maven

Result of build process

- *target* directory is created during compilation which stores the new files that were generated at compilation time
  - output, e.g. *my-app-1.0-SNAPSHOT.jar*
  - *classes* directory: class files that were created during compilation but not test classes
  - *test-classes*: classes created from test sources
  - *maven-archiver/pom.properties* file that defines the project’s GAV
  - *surefire-reports:* reports of the tests

Build systems - Maven

Project hierarchies

- Maven supports submodule projects:

```
<project ...>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>parent-app</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>pom</packaging>
  <!-- subprojects -->
  <modules>
    <module>first-child-app</module>
    <module>second-child-app</module>
  </modules>
</project>
```

Build systems - Maven

Project hierarchies

- Submodule projects also reference their parents:

```
<project ...>
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>com.mycompany.app</groupId>
    <artifactId>parent-app</artifactId>
    <version>1.0-SNAPSHOT</version>
  </parent>
  <groupId>com.mycompany.app</groupId>
  <artifactId>first-child-app</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>war</packaging>
  ...
</project>
```

Build systems - Maven

Plugins

- Maven’s functionality itself is limited to the basic, but it is a pluginable framework.
- Many different plugins are available
  - e.g. C++, LaTeX, ant build, javadoc, etc.
  - The official plugins are listed on their website: https://maven.apache.org/plugins/
- There are also 3rd party plugins and one can write own plugin

Build systems - Maven

Plugin example: Javadoc

```
<project ...>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-javadoc-plugin</artifactId>
        <version>3.2.0</version>
        <configuration>
          ...
        </configuration>
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

- Generate documentation: `mvn javadoc:javadoc` or `mvn:site`

Build systems - Maven

Dependencies

- The external libraries that a project uses are called dependencies.
- The dependency management feature in Maven ensures automatic download of those libraries from a central repository.

```
<project ...>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.13</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

- GAV uniquely specifies the required artifact
- scope defines how we use the dependency

Build systems - Maven

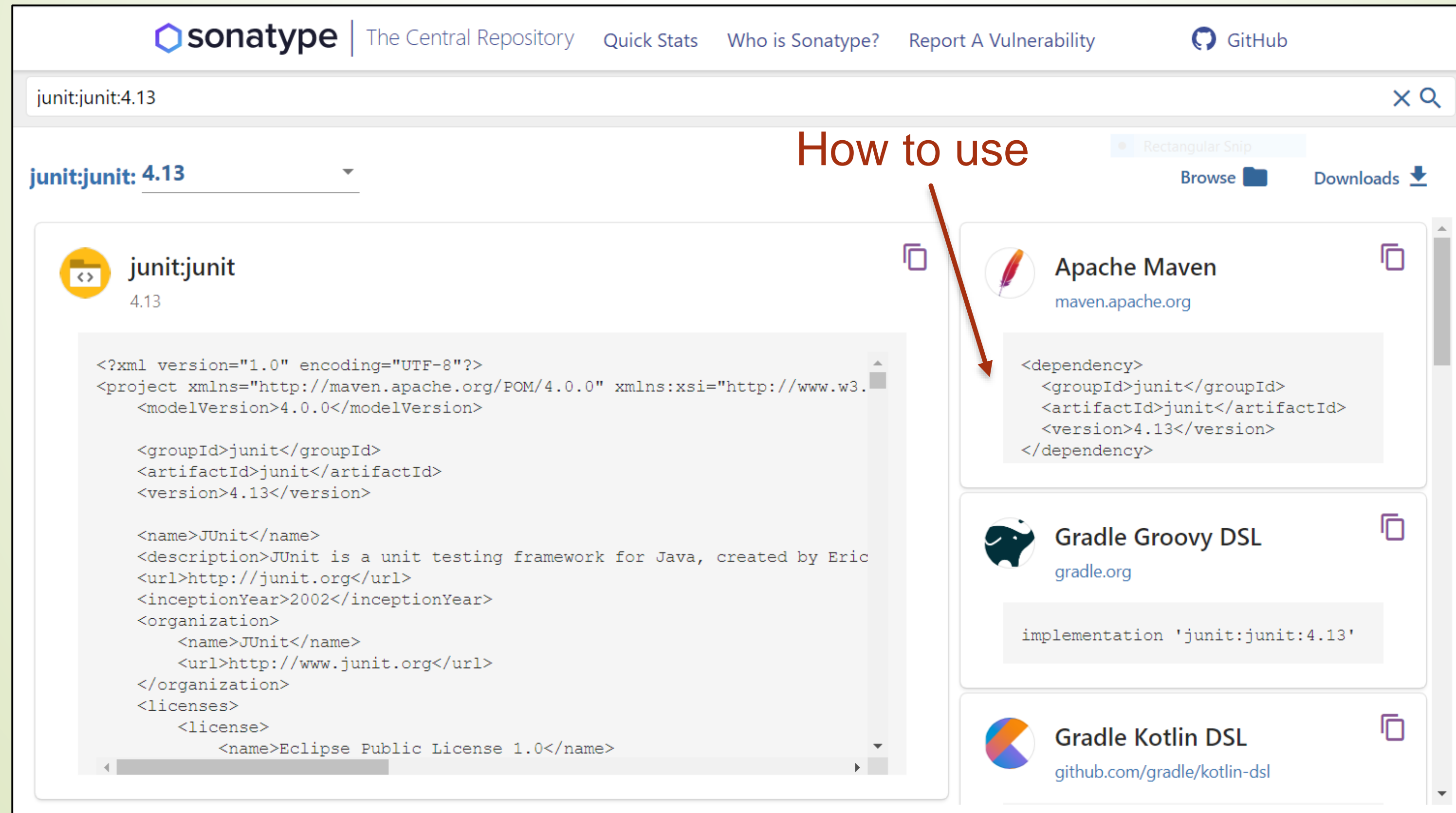
Dependencies' scope

- The most important scopes:
  - **compile:** This is the default if unspecified. Dependencies that required by the compilation
  - **runtime:** Dependency required at runtime, but not required at compilation time.
  - **test:** Dependency is not required in production but it is required for the compilation and execution of testcases.

Build systems - Maven

Search dependencies

- One can browse and search the available libraries in the Maven Central repository:
  - https://search.maven.org/



Build systems - Maven

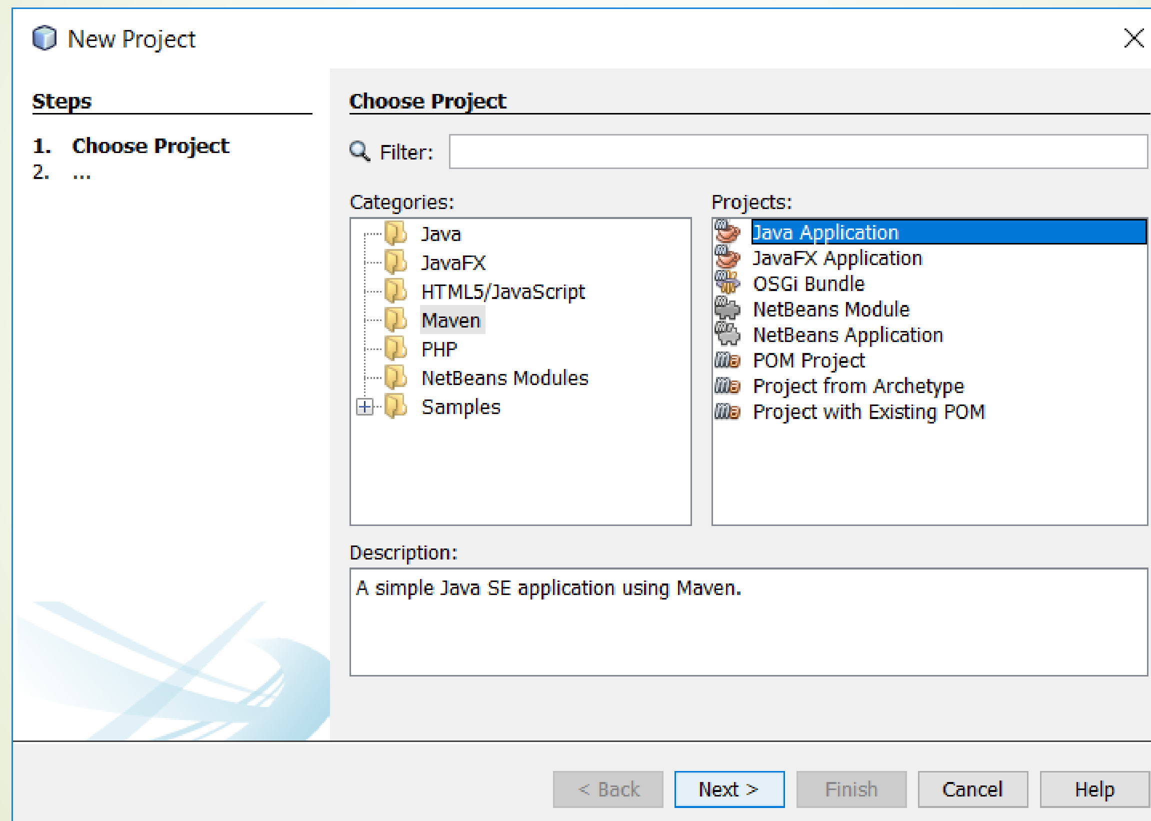
Complete pom.xml file for the ThesisGenerator app

```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>hu.elte.inf</groupId>
    <artifactId>thesis-generator</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>
    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.compiler.source>8</maven.compiler.source>
        <maven.compiler.target>8</maven.compiler.target>
    </properties>
    <build>
        <sourceDirectory>src</sourceDirectory>
        <testSourceDirectory>test</testSourceDirectory>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-jar-plugin</artifactId>
                <version>3.2.0</version>
                <configuration>
                    <archive>
                        <manifest>
                            <addClasspath>true</addClasspath>
                            <mainClass>thesisGenerator.ThesisGenerator</mainClass>
                        </manifest>
                    </archive>
                </configuration>
            </plugin>
        </plugins>
    </build>
    <reporting>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-project-info-reports-plugin</artifactId>
                <version>3.0.0</version>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-javadoc-plugin</artifactId>
                <version>3.2.0</version>
                <configuration>
                    <doctitle>My API for ${project.name} ${project.version}</doctitle>
                    <windowtitle>My API for ${project.name} ${project.version}</windowtitle>
                    <show>public</show>
                </configuration>
            </plugin>
        </plugins>
    </reporting>
</project>
```

Build systems - Maven

NetBeans

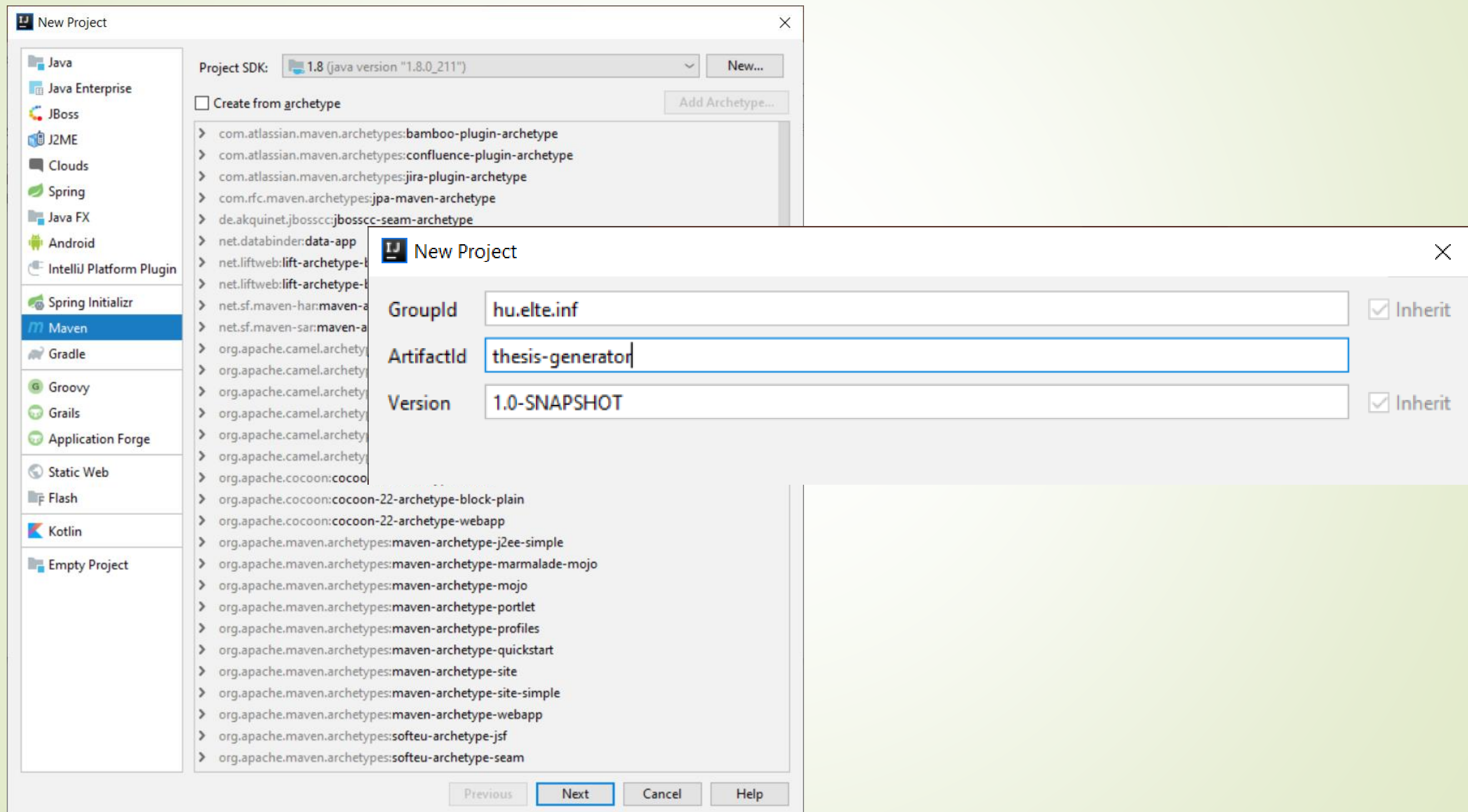
- NetBeans supports Maven-based projects.



Build systems - Maven

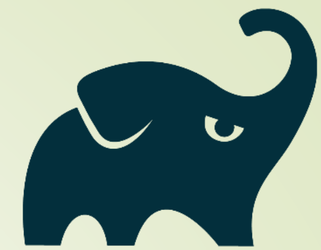
IntelliJ

- IntelliJ IDEA supports a fully-functional integration with Maven.



- *Generate Ant build file is also supported: Build -> Generate Ant Build*

Build systems - Gradle



Features

- Build automation system with increasing popularity [1] [2]
- Aims to merge the best concepts from Ant and Maven
- Supports incremental builds
  - Major performance boost for larger enterprise projects
- Configuration through a Groovy-based domain-specific language (DSL) instead of XML
- Official website: https://gradle.org/
- Tutorial: https://docs.gradle.org/current/userguide/getting_started.html

[1] https://www.baeldung.com/java-in-2019

[2] https://www.jetbrains.com/lp/devecosystem-2019/java/