COMP0104 Software Development Practice:
MAVEN – An Extensible Build Infrastructure

Jens Krinke

Centre for Research on Evolution, Search & Testing
Software Systems Engineering Group
Department of Computer Science
University College London

MAKE

- MAKE is a tool for building programs, in which existing derived files are reused as much as possible.
- MAKE uses a **Makefile**, a system description that lists the components of the system as well as their dependencies and construction steps.
- MAKE uses the change date of components in order to detect changes.

ANT

- Rather than specifying commands, an ANT user specifies **tasks** that realise a specific **target**.
- Each task knows which tools and commands to use to realise the target.
- Every target has a name and optional **dependencies**. The dependencies list targets that must be realised before the actual target.
- Locations of source files and output have to be specified.

Disadvantages of ANT and MAKE

- ANT and MAKE are very flexible and can be adapted to any project structure.
- Projects have different structures which makes it hard to understand an unknown project.
- Standards and conventions have emerged over time (‘best practices’).
- Tools established standards:
  - AUTOCONF generates Makefiles
  - IDEs create default structures without MAKE/ANT

MAVEN:

- Most users consider MAVEN a “build tool”
- It is marketed as project management tool
- Neither is (fully) true:
  - MAVEN is a build management framework
  - It adds support to build artefacts beyond compilation, testing, and deploying.
- It is actually an **extensible build infrastructure** that implements a build lifecycle over a set of **standards** and **conventions** for project structure.

Convention over Configuration

- MAVEN's power lies in underlying standards and conventions.
- MAVEN assumes a specific project structure:
  - a project lifecycle
  - a project object model
  - a dependency management system
  - a set of standards
  - logic for executing plugins
- Plugins know how to build and assemble software

Features

- Declarative Builds
  - Central Project Object Model (POM)
- Dependency Management
  - Projects have version, group and artefact identifier
- Remote Repositories
  - Identifiers allow storage and retrieval in repositories
- Universal Reuse of Build Logic
  - Logic is contained in plugins
- Tool Portability and Integration
  - All data is stored in the Project Object Model

Project Object Model (POM)

- A project is defined by a POM:
  - It names the project,
  - provides a set of unique identifiers, and
  - defines the relationships to other projects through dependencies, parents, and prerequisites.
- A POM can also customise the plugin behaviour
- It can also contain additional information

Repositories

Plugin Repository

- Needed plugins are updated from the central repository
- Easy to keep the infrastructure up to date
- Easy to add new build elements

Project Repositories

- Dependencies on external projects are specified
- External projects are automatically retrieved and built from central project repositories

Dependency Resolution

- Dependencies on (external) artefacts are specified through the unique identifiers.
- Dependencies are resolved by
  - Retrieval of the artefact from the repository
  - Resolving of transitive dependencies
  - Setup in the local project
- Retrieved artefacts are stored in local repository (`$HOME/.m2/repository`)
- The local repository contains plugins, too.

```
$ mvn archetype:generate -DgroupId=uk.ac.ucl.cs.gs04.mvnexp \
-DartifactId=simple
[INFO] Scanning for projects...
...
Choose a number or apply filter (format: [groupId:]artifactId, case
sensitive contains): 253:
Choose org.apache.maven.archetypes:maven-archetype-quickstart
version:
1: 1.0-alpha-1
...
Choose a number: 6:
...
Confirm properties configuration:
groupId: uk.ac.ucl.cs.gs04.mvnexp
artifactId: simple
version: 1.0-SNAPSHOT
package: uk.ac.ucl.cs.gs04.mvnexp
 Y: :
...
$ _
```

Demo

Project Object Model

- The Project Object Model basically contains the project's identifiers and its dependencies
- POMs look small
- A project's POM extend its super-POM and contains all all parent POMs, user settings etc.
- `mvn help:effective-pom` prints complete POM:

```
$ mvn help:effective-pom
...
```

~ 270 lines output

MAVEN goals

- Plugins are collections of goals
- A goal is a “unit” of work
- Examples:
  - archetype:generate
  - maven-compiler-plugin:3.1:compile
  - maven-compiler-plugin:3.1:testCompile
  - maven-surefire-plugin:2.12.4:test
- Goals can be attached to lifecycle phases

Lifecycle Phases

- The build lifecycle is an ordered sequence of phases involved in building a project.
- MAVEN can support different lifecycles.
- In each phase the plugins attached to it will be executed.
- Executing a phase will first execute all preceding phases in order until the specified phase is reached.

Main Phases

validate: validate the project is correct
compile: compile the source code of the project
test: test the code using a unit testing framework.
package: package code in a distributable format.
integration-test: deploy the package into an environment where integration tests can be run
verify: run any checks to verify the package
install: install the package into the local repository
deploy: copies final package to remote repository.

Disadvantages

- MAVEN is a “black box” and it is hard to understand what is actually happening.
- Bound to the decisions of the plugin’s developer.
- Hard to adapt to other project structures, conflicts with project structures in IDEs.
- Conflicting dependencies occur and are hard to resolve.
- Some consider MAVEN to be the perfect example for an over-engineered product.

Concepts

- MAVEN is an **extensible build infrastructure** that implements a build lifecycle over a set of standards and conventions for project structure.
- A **Project Object Model** specifies the project's identifiers and its dependencies.
- MAVEN assumes a default project structure and follows a **convention over configuration** principle.
- It has a **defined lifecycle** and its phases are executed in order.
- **Plugins** attach goals to lifecycle phases; they contain the logic to build the software.