Software Design and Construction

159.251

Process Automation

Amjed Tahir
a.tahir@massey.ac.nz

Original author: Jens Dietrich

References

[CC] Robert Martin:
Clean Code: A Handbook of Agile Software Craftsmanship.
Prentice Hall 2009.

[EJ] Joshua Bloch: Effective Java Second Edition.
Sun Micro 2008.

Additional Readings

The ANT User Manual
http://ant.apache.org/manual/index.html

Summary

- from IDE to customer
- build scripts
- ANT
  - tasks and targets
  - dependencies
  - adding 3rd party tasks
  - variables
  - advanced ANT: conditionals, listeners
- ANT alternatives
- continuous build

Agile and continuous iteration

Agile development will be covered in more details in part 2 of this course.

- software projects run in iterations
- at the end of an iterations, working (tested) code is produced
- this code can be used by the user
- **facilitates communication**
- **Continuous development**: build things small and in iterations

Enabling Short Iterations

- each iteration includes a set of activities with an approximately constant (and significant) cost C:
  - compile
  - package
  - test
  - document
  - deliver
  - ...
- having many (N) short iterations is expensive: N*C
- **solution: automate these activities, reduce overhead (C) to close to zero**

Build Tools

- build tools are used to automate common tasks
- early build tool for C (1977): MAKE
- Java (and other similar languages): ANT, Maven, Gradle
- others: PyBuilder (Python), NANT (.NET), rake (Ruby),

When to Build

- **on demand**: build is triggered explicitly (e.g., by running ANT)
- **triggered**: build starts when a certain event happens (e.g., a commit to the revision control system)
- **scheduled**: build is performanced periodically (e.g., nightly build)

- we will focus **on on demand builds**

Apache ANT

- popular open source build tool for Java
- managed by the Apache foundation.
- created around 2000 by James Duncan Davidson from Sun Microsystems while working on Apache Tomcat
- good integration into IDEs

ANT (ctd)

- can be extended and customised
- large library of extensions available (free + commercial).
- but: build scripts can quickly become **very complex**
- common issues:
  - dependencies
  - classpath

ANT Scripts

- ANT scripts are written in XML
- the root element is a **project** that has a name
- the next level consists of **targets**
- targets describe tasks that have to be performed (compile, test, jar, ...)
- targets are implemented using **tasks**
- tasks are reusable modules to build scripts
- ANT has a large number of **predefined tasks** on board, but it is also possible to get and install additional tasks, or to write your own tasks (using Java)

Targets and Dependencies

- targets are the main steps in the workflow
- targets may **depend on** other targets
- if a target is executed, the targets it depends on must be executed first
- *it is therefore possible (and common) to write a target that (indirectly) depends on all other targets but does not do anything by itself - this is a "run all"*
- a project has a **default target**
- if ANT is executed, it tries to execute the default target:

```
ant [build-script]
```

- the build-script can be skipped if there is a **build.xml** in the current folder
- ANT can also be executed on a particular target:

```
ant [build-script] target
```

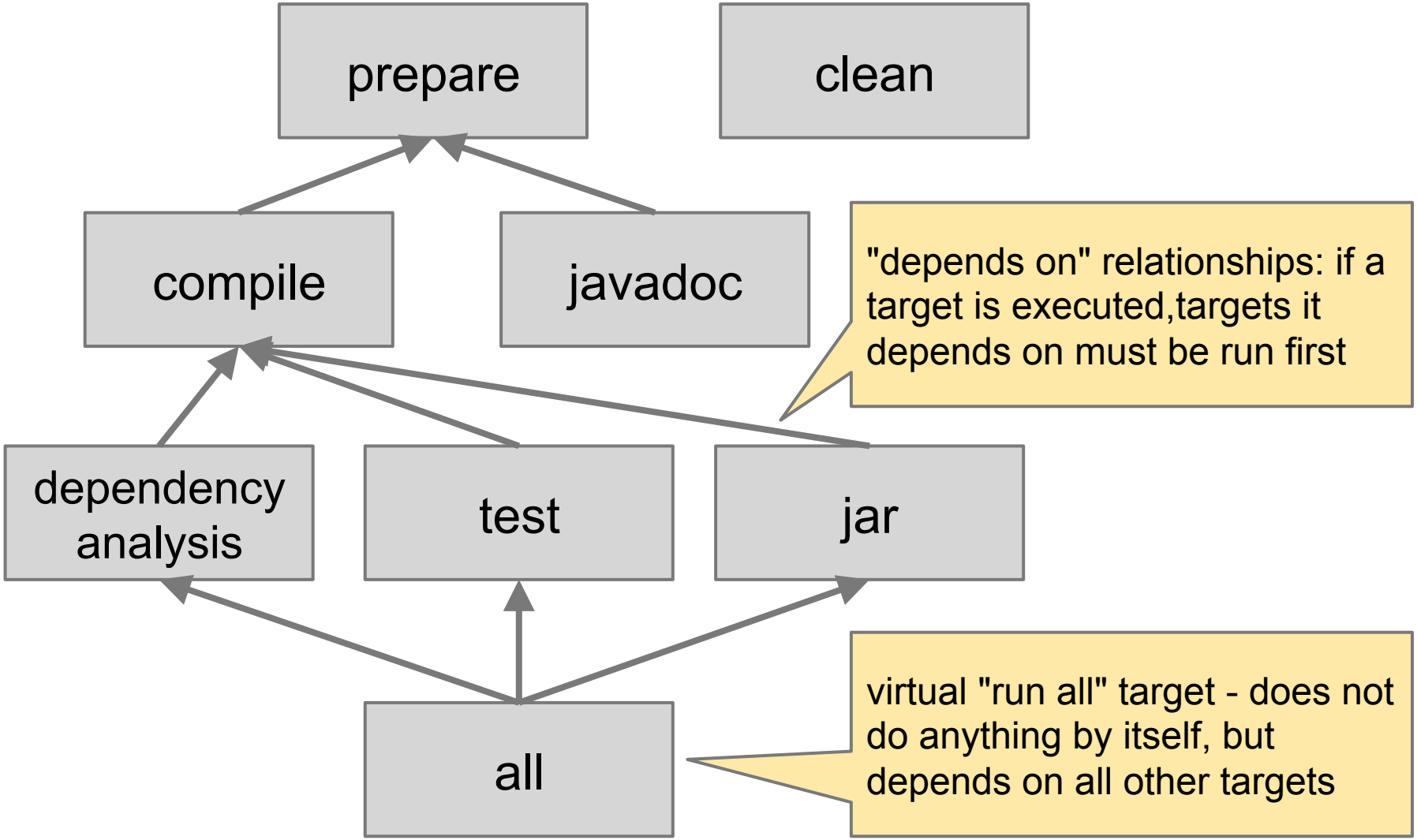
Example: The TaxCalculator

- class to calculate income tax according to Inland Revenue Department, NZ
- simple user interface
- tests
- source code: https://bitbucket.org/jensdietrich/oop-examples/src/1.0/taxcalculator/

TaxCalculator Builds

- compile
- run dependency analysis - calculator should not depend on user interface
- run tests
- generate test reports
- generate documentation
- build an executable jars for tax calculator
- build a jar for test cases

TaxCalculator Targets and Their Dependencies



ANT Targets And Dependencies

```
<target
   name="all"
   description="Main target - compile, jar,
   docs and test"
   depends="jar,tests,dependencyanalysis"
>
 </target>
...
<target
   name="jar"
   depends="compile"
   description="Creates the jar file"
>
 </target>
```

Prepare And Clean

- clean:
  - clean build folder
- prepare:
  - create target folders
  - set variable values
- both use **tasks** for file system manipulation
- tasks are the basic "command" in ANT

Prepare and Clean (ctd)

```
<target name="clean">
    <delete dir="${build.dir}"/>
</target>

<target name="prepare">
    <mkdir dir="${build.dir}"/>
    <mkdir dir="${build.classes}"/>
    <mkdir dir="${build.lib}"/>
    <mkdir dir="${qa.dir}"/>
</target>
```

delete and **mkdir** are tasks - they delete files and directories, and create directories respectively

folders are referenced using **variables**

Properties

- write **DRY scripts** that are easy to maintain
- use only relative path names to make scripts portable (=can be executed on other computers)
- define paths once as a property (=variable), then reference them
- syntax: ${property-name}
- use references even when defining properties!
- unix conventions are used in paths:
  - . (DOT) - current folder
  - .. (DOT DOT) - parent folder
  - / (SLASH) - path separator

Defining and Referencing Properties

```
<project default="all" basedir=".">
   <property name="name" value="taxcalculator"/>
   <property name="version" value="1.2"/>
   <property name="version_suffix" value="1_2"/>
   <property name="build.dir" value="build"/>
   <property name="build.lib" value="${build.dir}/lib"/>
   <property name="build.jar"
       value="${build.lib}/${name}-${version}.jar"/>
```

all folders are relative to . (current project folder)!

define variable name

reference variable name

- build.jar is the path of the main jar file that is created,
- e.g., **build/lib/taxcalculator-1.2.jar**

Setting Paths

- compilation and build requires the configuration of paths
- the build path contains all libraries that are referenced in the code
- by convention, they are located in the lib folder
- in ANT, these referenced must be resolved manually
- PATH is a **data type** in ANT

Setting Paths (ctd)

```
<path id="build.classpath">
   <fileset dir="${lib.dir}">
      <include name="**/*.jar"/>
   </fileset>
</path>

<target name="compile" ..>
   <javac destdir="${build.classes}"
          ..
          classpathref="build.classpath"
          ..
   />
```

define path by set of jar files in lib folder

this is the actual java compilation task (javac is the Java compiler)

instruct the compiler to resolve referenced classes using libs in path

Filters

- often, tasks are performed on file sets
- examples:
  - copy to folder
  - package in archive (jar, zip, ..)

  this should often be done only for certain files in the source folder
- solution: use exclude or include filters (aka blacklists or whitelists)

Filters (ctd)

```
<target name="jar" ...>
    ...
   <jar jarfile="${build.jar}"
      manifest="${build.tmp}/manifest.mf"
      basedir="${build.classes}"
      includes="nz/ac/massey/cs/**/*.*"
   />
    <jar jarfile="${build.test.jar}"
       manifest="${build.tmp}/manifest4tests.mf"
       basedir="${build.classes}"
       includes="test/**/*.*"
    />
```

filters are used as attributes here. Note that filters can also be used as nested tags

Filters (ctd)

- to define filter patterns, wildcards can be used
  ? matches any character
  * matches any text that does not contain a path separator (\ and /)
  ** matches text that may include path separators

- Examples:

```
<include name="lib/*.jar"/>
```

include all files with jar extension in lib folder

```
includes="test/**/*.java"
```

include all java source file in some package with a name starting with test (test is the top folder)

Adding Metadata

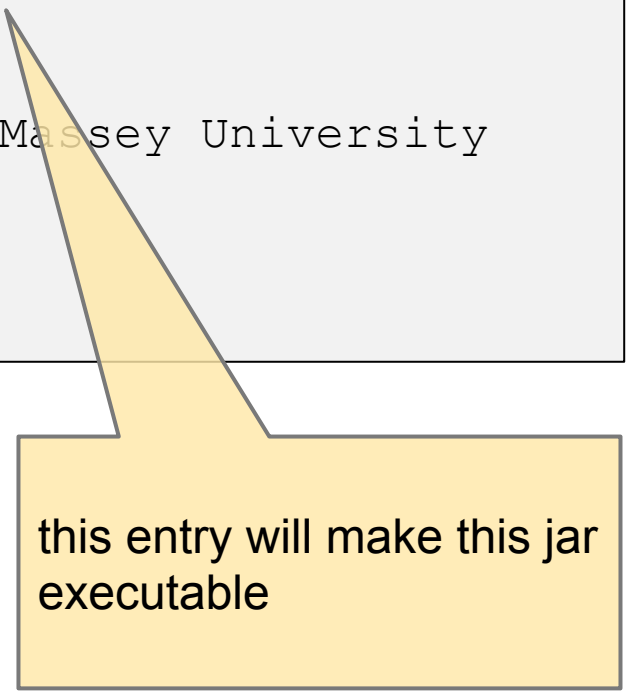
- as part of building executable jars, meta data must be added to the libraries
- the metadata describe the jar
- the metadata also make the jar executable
- the format of metadata is defined in the jar specification
- metadata are stored with the jar file in META-INF/MANIFEST.MF
- this is a text file with a simple **key-value** format
- metadata can be used at runtime to reason about the program

Making Jars Executable

- the Main-class entry can be used to make the jar **executable**
- then the jar can be executed using **java -jar app.jar**
- the JVM will look for the Main-class in the manifest of app.jar, and will execute the main method of this class
- the is called the application entry point
- in many (graphical) operating system, these jars are "executable on (double) click"
- alternatives:
  - build installers using (commercial) products like InstallShield or InstallAnywhere
  - build WebStart distributions for a low TCO solution

Metadata Example

```
Manifest-version: 1.0
Main-class:
    nz.ac.massey.cs.sdc.taxcalculator.ui.TaxCalculatorUI
Name: taxcalculator
Implementation-Title: taxcalculator
Implementation-Version: 1.2
Implementation-Vendor: Jens Dietrich, Massey University
```



Templating

- metadata should be generated dynamically based on properties
- this can be achieved through templating
- a metadata template is defined
- the variables in this template can be bound when a file is copied (to a temporary folder)

The Metadata Template

```
Manifest-version: 1.0
Main-class: @MAIN_CLASS@
Name: @NAME@
Implementation-Title: @TITLE@
Implementation-Version: @VERSION@
Implementation-Vendor: @VENDOR@
```

variable that need to be bound

https://bitbucket.org/jensdietrich/oop-examples/src/1.0/taxcalculator/config/taxcalculator.mf

Metadata Template Variable Binding

```
<target name="jar" ... >
    <filter token="NAME" value="${name}"/>
    ...
    <copy file="${config.dir}/${name}.mf"
        tofile="${build.tmp}/manifest.mf"
        filtering="yes"/>
    <jar jarfile="${build.jar}"
        manifest="${build.tmp}/manifest.mf"
        basedir="${build.classes}"
        includes="nz/ac/massey/cs/**/*.*"
    />
    ...
```

define bindings

apply bindings during copy - create temporary file

use this file as manifest when jar is built

ANT Advanced Features

- targets can be **conditional** (use if and unless attributes)
- scripts can be made **modular** by importing other scripts
- **listeners and loggers** can be used to integrate ANT with other tools
- **custom tasks** can be added - must implement simple interface `org.apache.tools.ant.Task`

Continuous Integration

- scheduled execution of builds
- reduces the risk that programs become inconsistent when multiple developers work on it
- usually running on a server, and builds are triggered by commits to the repository
- tools: AntHill, Jenkins, Hudson
- free hosting: http://www.cloudbees.com/

Maven

- Maven is a modern build tool managed by the Apache foundation (like ANT)
- Maven focuses on two aspects:
  - convention over configuration
  - symbolic dependencies with repository-based dependency resolution

Core concepts

- the **POM** is an xml file (`pom.xml`) that has the project configuration
- Maven creates **artifacts** - usually jar files of executable (incl. library) code
- artifacts have a composite name consisting of group id, artefact name and a version
- an artifact may **depend on** other artifacts
- artifacts are resolved over the network against a central **Maven repository** when maven runs, e.g. https://mvnrepository.com/
- the repository is searchable to locate artefacts

Example 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>nz.ac.massey.sdc</groupId>
             <artifactId>taxcalculator</artifactId>
             <version>1.0-SNAPSHOT</version>
</project>
```

https://bitbucket.org/jensdietrich/oop-examples/src/1.1/taxcalculator2/

Phases

- Maven builds are executed in **lifecycle phase**
- there are dependencies between phases
- syntax (from terminal): **`mvn test`**
- build outputs are stored by default in a **`/target`** folder Maven generates

Maven plugins index:
https://maven.apache.org/plugins/index.html

Standard Phases (Selection):

1. **validate**: validate the project is correct and all necessary information is available
2. **compile**: compile the source code of the project
3. **test**: test the compiled source code using a suitable unit testing framework. These tests should not require the code be packaged or deployed
4. **package**: take the compiled code and package it in its distributable format, such as a JAR.
5. **verify**: run any checks to verify the package is valid and meets quality criteria
6. **install**: install the package into the local repository, for use as a dependency in other projects locally
7. **deploy**: done in an integration or release environment, copies the final package to the remote repository for sharing with other developers and projects.
8. **clean**: cleans up artifacts created by prior builds
9. **site**: generates site documentation for this project

Convention over Configuration

- it is extremely simple to start a basic Maven project
- when files are put in standard locations and follow standard naming conventions, “everything works”
- in particular, Java projects should use the **standard Maven project layout**
- tests (junit classes) have to follow certain naming patterns to be recognised as tests (for instance **`*Test`**, but not **`*Tests`**)
- all conventions can be overruled with configurations (in **`pom.xml`**)

The Maven Project Layout

```
src/main/java                  Application/Library sources
src/main/resources             Application/Library resources
src/main/filters               Resource filter files
src/main/assembly              Assembly descriptors
src/main/config                Configuration files
src/main/scripts               Application/Library scripts
src/main/webapp                Web application sources
src/test/java                  Test sources
src/test/resources             Test resources
src/test/filters               Test resource filter files
src/site                              Site
LICENSE.txt                           Project's license
NOTICE.txt                            Notices and attributions ..
README.txt                            Project's readme
```

https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html

Customising Maven

- conventions can be overridden and redefined using archetypes
- an archetype is essentially a project template
- plugins can be used to further customise Maven, in particular to add build functionality (phases) or custom reporting
- plugins are registered in the **`<plugins>`** sections of the pom

Dependency Resolution

- Maven manages (transitive) dependencies
- instead of copying libraries into the project, it is sufficient to declare them (using group+name+version)
- references are resolved against a repository
- Maven will fetch the respective artifacts during a build, and further artifacts they might depend on
- This means that an initial build can take a long time, but Maven will try to cache artifacts in a local cache
- the repository search function (such as http://search.maven.org/) is used to locate the references (in case of Maven, XML snippets)

TaxCalculator2 Dependencies
(in pom.xml)

```
<dependencies>
   <dependency>
       <groupId>junit</groupId>
       <artifactId>junit</artifactId>
       <version>4.11</version>
       <scope>test</scope>
   </dependency>
   <dependency>
       <groupId>org.json</groupId>
       <artifactId>json</artifactId>
       <version>20160810</version>
   </dependency>
   ..
</dependencies>
```

JUnit is only needed during test phase, not at runtime

block copied from Maven repository search

https://bitbucket.org/jensdietrich/oop-examples/src/1.1/taxcalculator2/

Dependencies and Versioning

- Maven also supports flexible dependencies with features like references to version ranges, latest and release versions
- while this sounds like a good idea as it enables *auto-upgrades* when new versions become available in the repository, care must be taken as the new versions often violate API stability
- semantic versioning is a possible solution, but is not widely used (http://semver.org/)

Maven Integration

- all major IDEs have built-in Maven support, or plugins providing it
- usually, Maven is a project type that can be selected when creating new projects)
- Maven then takes over classpath / buildpath management
- note that Maven needs the network connection, i.e. proxies need to be configured as required

Maven Alternatives

- there are several other build tools based on the same ideas, and also using artefacts from the Maven repository
- **ivy** is an ANT extension that integrated Maven's dependency management into ANT
- **gradle** is a popular build tool, the main difference to Maven is that it uses Groovy instead of XML to write build scripts that are shorter and more concise

Recap: Design Principles Used in ANT

- DRY - use of ANT properties
- Templating - use of filters when copying files
- a domain specific language (DSL) - used to filter files (includes and excludes)