AARHUS UNIVERSITET

Software Engineering and Architecture

Build Management

AARHUS UNIVERSITET

Motivation

- Oracle/Sun provides Java SDK free of charge
  - provides standard command line tools: javac, java, ...
- These are sufficient only for *very* small systems
  - javac only compile one directory at a time
  - javac recompiles everything every time
- Large systems require many tasks
  - manage resources (graphics, sound, config files)
  - deployment (making jars, copying files, upload to repos)
  - management (javadoc, coverage, version control)
- *Reliability require reliable processing*
  - Creating the system exactly the same way every time

AARHUS UNIVERSITET

Build-Management

- This problem is denoted:

> Definition: **Build management**
>
> The process of managing and constructing an executable software system from its parts in a reliable and cost-efficient way.

- Computer Scientists' standard solution: **a tool...**
- The tool read a *build-description*

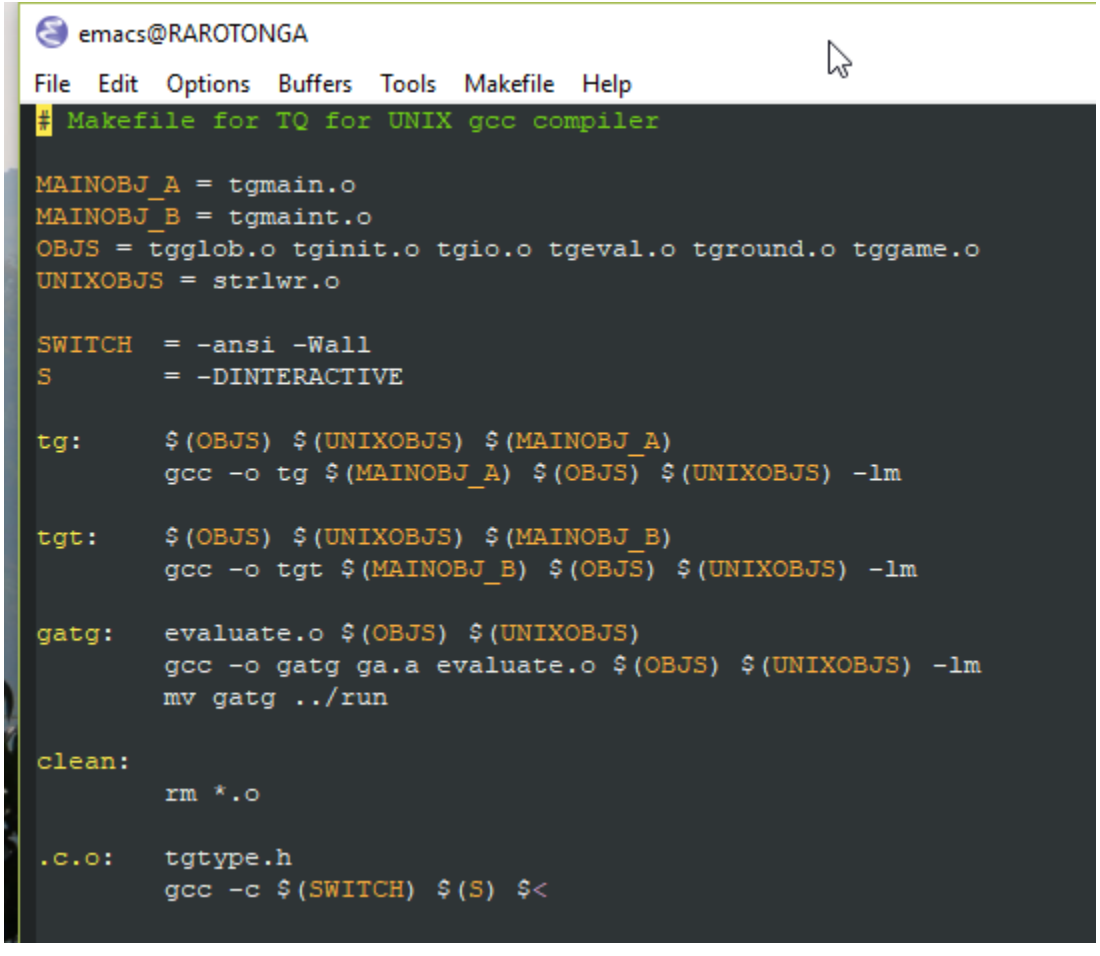
> Definition: **Build description**
>
> A description of the goals and means for managing and constructing an executable software system. A build description states *targets*, *dependencies*, *procedures*, and *properties*.

- Example: Make (Feldmann, 1979)

AARHUS UNIVERSITET

History…

- Make



AARHUS UNIVERSITET

The Two Ways

- Programming languages comes in flavors
  - Procedural languages
    - ‘express **how** to achieve a goal’
  - Declarative languages
    - ‘express **what** goal you want to achieve’

- Old time build languages were *procedural* (ex: make)
  - *Write code to compile all source files*
- Modern build languages are *declarative (almost)*
  - *‘compileJava’* is built into the system

Script Parts

- A **target**. This is the goal that I want, like "compile all source code files."
- A list of **dependencies**. Targets depend upon each other: in order to execute you must first have compiled all source code. Thus the execution target depends upon the compilation target. The build description must provide a way to state such dependencies.
- **Procedures**. The procedures are associated with the targets and describe how to meet the goal of the target, like how to compile the system. For instance the compile goal must have an associated procedure that describes the steps necessary to compile all source files—in this case call *javac* on all files.
- A set of **properties**. Variables and constants are important to improve readability in programming languages, and build descriptions are no different. Properties are variables that you can assign a value in a single place and use it in your procedures.

AARHUS UNIVERSITET

Apache Ant

- Ant is a build-management tool geared towards Java
  - ☺ has some strong build-in behavior
    - javac on source *trees* and does smart recompile
  - ☺ independent of large IDEs
  - ☹ was created on the XML buzzword wave so it is verbose
  - ☹ is very evolutionary in its design
    - *Do the same thing* ***in a zillion different ways***
      - ***No 'conceptual integrity'***

Example

AARHUS UNIVERSITET

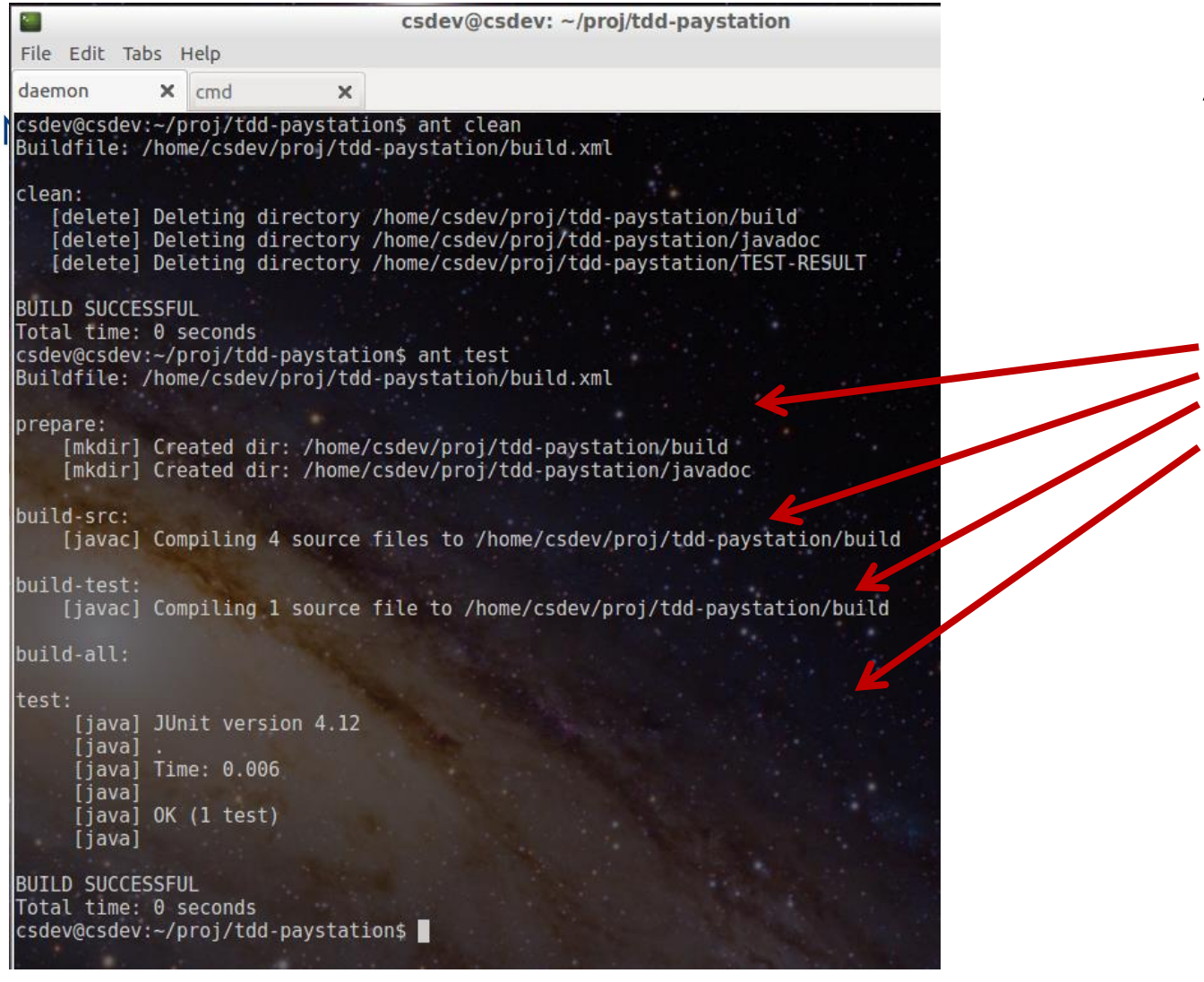
- Targets
  - ‘test’ …
- Dependencies
  - ‘test’ d.o. ‘build-all’ d.o. …
- Procedures
  - <javac ….>
- Properties
  - {$build-directory}

```
<property name="source-directory" value="src"/>
<property name="test-source-directory" value="test"/>
<property name="build-directory" value="build"/>
<property name="javadoc-directory" value="javadoc"/>

<property name="junit-jar" value="lib/junit-4.12.jar"/>
<property name="hamcrest-jar" value="lib/hamcrest-core-1.3.jar"/>
```

```
<target name="clean">
  <delete dir="${build-directory}"/>
  <delete dir="${javadoc-directory}"/>
  <delete dir="${test-output-directory}"/>
</target>

<target name="prepare">
  <mkdir dir="${build-directory}"/>
  <mkdir dir="${javadoc-directory}"/>
</target>

<target name="build-src" depends="prepare">
  <javac srcdir="${source-directory}"
         includeAntRuntime="false"
         debug="true"
         destdir="${build-directory}">
    <classpath refid="class-path"/>
  </javac>
</target>

<target name="build-test" depends="build-src">
  <javac srcdir="${test-source-directory}"
         includeAntRuntime="false"
         debug="true"
         destdir="${build-directory}">
    <classpath refid="class-path"/>
  </javac>
</target>

<target name="build-all" depends="build-src,build-test"/>

<target name="test"  depends="build-all">
  <java classname="org.junit.runner.JUnitCore">
    <arg value="paystation.domain.TestPayStation"/>
    <classpath refid="class-path"/>
  </java>
</target>
```

AARHUS UNIVERSITET

As In



AARHUS UNIVERSITET

Gradle in SWEA

2018 was the first year after switching from Ant…

Gradle

- Gradle is a *convention-based* build-management tool
  - *Convention over configuration* is the mantra!
- *Which means*
  - You cannot see a damn thing about what it does in its build description (build.gradle) !!! ☹ ☹ ☹
  - You have to *know all the conventions or google your butts off* ☹
- Conventions:
  - A fixed set of targets are defined (compileJava, test, …)
  - The source folder hierarchy and naming are **hard coded!**
  - The 'build description' is in *build.gradle* in the project root
- A bit of help: 'gradle tasks' will display all known targets.

AARHUS UNIVERSITET

Gradle build.gradle

- The simplest *build.gradle* file for java dev, contains one line
  - *apply plugin: 'java'*
- And now you can do all basic BM tasks (except running a program! ☺)
  - gradle test
    - Will *compile all production code, all test code, and execute all Junit code in the 'test' source tree*

```
apply plugin: 'java'

repositories {
    jcenter()
}

dependencies {
    testCompile 'junit:junit:4.12'
}
```

Gradle

- How does it work? Magic???
- By *convention*
  - *You must put your code in the right folders!*
    - *src/main/java/HERE*
    - *src/test/java/HERE*
  - *Predefined targets*
    - *Like ‘test’, ‘compileJava’, …*
- By *plugins*
  - *Which are procedural groovy code to inject into the gradle framework*
    - *(‘framework’ is a central topic of this course)*

Gradle

- Gradle is *also* a *dependency-management tool*

- Ex: we need hamcrest

```
apply plugin: 'java'
apply plugin: 'jacoco'

repositories {
    jcenter()
}

dependencies {
    testCompile 'junit:junit:4.12'
    testCompile 'org.hamcrest:hamcrest-library:1.3'
}
```

- Gradle will download ‘org.hamcrest….:1.3’ from JCenter on the internet, and set the classpath correctly

Gradle

- Gradle combines *both declarative and procedural*
  - If you follow ‘gradle conventions’ it knows what you want to do
    - *gradle test, etc.*
  - And ‘task’ let you write complete *groovy* code, if you need it

```
task hello {
   doLast {
      println 'tutorialspoint'
   }
}
```

```
task jacocoMergeAll(type: JacocoMerge) {
   dependsOn(subprojects.test, subprojects.jacocoTestReport)
   subprojects.each { subproject ->
      // Some subprojects have no coverage data as they have
      // no tests. To avoid null exception, I test if
      // a jacoco report directory has been made, otherwise
      // just add an empty file collection
      if (subproject.jacoco.reportsDir.exists()) {
        executionData subproject.tasks.withType(Test)
      } else {
        executionData = project.files([])
      }
   }
}

task jacocoRootReport(type: JacocoReport, group: 'Coverage reports') {
  description = 'Generates an aggregate report from all subprojects'
  dependsOn(jacocoMergeAll)

  additionalSourceDirs = files(subprojects.sourceSets.main.allSource.srcDirs)
  sourceDirectories = files(subprojects.sourceSets.main.allSource.srcDirs)
  classDirectories = files(subprojects.sourceSets.main.output)
  executionData = files("${buildDir}/jacoco/jacocoMergeAll.exec")

  reports {
    html.enabled = true
    xml.enabled = false
  }
}
```

Analysis

Predefined

- A **target**. This is the goal that I want, like "compile all source code files.

Predefined

- A list of **dependencies**. Targets depend upon eac
you must first have compiled all source code. Thus the execution target depends upon the compilation target. The build description must provide a way to state such dependencies.

Predefined

- **Procedures**. The procedures are associated with th
to meet the goal of the target, like how to compile the system. For instance the compile goal must have an associated procedure that describes the steps necessary to compile all source files—in this case call *javac* on all files.

- A set of **properties**. Variables and constants are important to improve readability in programming languages, and build descriptions are no different. Properties are variables that you can assign a value in a single place and use it in your procedures.

AARHUS UNIVERSITET

In SWEA…

- Build management using Gradle is a *postulate* and requirement
  - No exercises in making Gradle targets, groovy code, etc.
- Required for easy hand-in and easing the TA work
- **Hard requirement: ‘gradle test jacocoTestReport’ must always run without errors, and must always execute all your tests!**

Other BM tools

Maven

- Ant is pretty old…
  - It is a Domain Specific Language and *procedural*
    - Tasks define ‘what to do’
- Maven
  - ‘Convention over configuration’
    - You follow the conventions and then Maven knows all the tasks!
  - It is a DSL and it is *declarative*
    - *mvn compile, mvn test, mvn install*
  - *Very very good for publishing libraries on Maven repository*
  - Maven directly supports dependency management (POM)
    - Ant requires help from Ivy to do that

No support for running code ☹

AARHUS UNIVERSITET

Maven POM

- Basic POM just states what the identity of your project is

```
<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.companyname.project-group</groupId>
   <artifactId>project</artifactId>
   <version>1.0</version>

</project>
```

- Note:
  - This simple POM can handle *every* classic aspect (except running a program!)
- Yeah – we all love XML ☺