



AARHUS UNIVERSITET

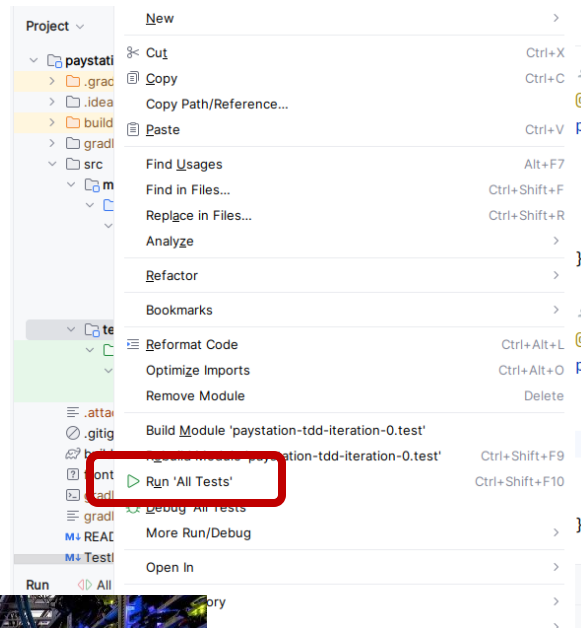
# **Software Engineering and Architecture**

Build Management

- Oracle provides Java SDK free of charge
  - provides standard command line tools: javac, java, ...
- These are sufficient only for 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

# Modern IDEs hide the fact

- IntelliJ, Visual Studio, etc. include build management tools
  - Compile all (touched) files, etc.
- But what do you do when your code is on a server in Helsinki?
  - Only the “shell/commandline” available



```

root@baerbak-e23: ~
root@baerbak-e23:~# docker ls
docker: 'ls' is not a docker command.
See 'docker --help'
root@baerbak-e23:~# docker ps
CONTAINER ID   IMAGE                                COMMAND                  CREATED    STATUS
6602d0c39b65   henrikbaerbak/private:hotstone-server "dumb-init java -jar..." 3 weeks ago Up 3 weeks
tstack_hotstone-server.1.thmd3egrarw3j3sarj2ip8eu8u "docker-entrypoint.s..." 3 weeks ago Up 3 weeks
e31d202304fc   mariadb:10.5.8                      "docker-entrypoint.s..." 3 weeks ago Up 3 weeks
tstack_mariadbserver.1.96yiqnva6w5n0ab4gb3unqdx
root@baerbak-e23:~#
  
```



# 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)

- Make

# History...

```
emacs@RAROTONGA
File Edit Options Buffers Tools Makefile Help
## Makefile for TQ for UNIX gcc compiler

MAINOBJ_A = tgmain.o
MAINOBJ_B = tgmaint.o
OBJS = tgglob.o tginit.o tgio.o tgeval.o tgground.o tggame.o
UNIXOBJS = strlwr.o

SWITCH = -ansi -Wall
S      = -DINTERACTIVE

tg:    $(OBJS) $(UNIXOBJS) $(MAINOBJ_A)
       gcc -o tg $(MAINOBJ_A) $(OBJS) $(UNIXOBJS) -lm

tgt:    $(OBJS) $(UNIXOBJS) $(MAINOBJ_B)
       gcc -o tgt $(MAINOBJ_B) $(OBJS) $(UNIXOBJS) -lm

gatg:   evaluate.o $(OBJS) $(UNIXOBJS)
       gcc -o gatg ga.a evaluate.o $(OBJS) $(UNIXOBJS) -lm
       mv gatg ../run

clean:
       rm *.o

.c.o:   tgtype.h
       gcc -c $(SWITCH) $(S) $<
```

# 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, Ant)
  - *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.

- 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’*



- 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>
```

```
csdev@csdev: ~/proj/tdd-paystation
File Edit Tabs Help
daemon x cmd x
csdev@csdev:~/proj/tdd-paystation$ ant clean
Buildfile: /home/csdev/proj/tdd-paystation/build.xml

clean:
    [delete] Deleting directory /home/csdev/proj/tdd-paystation/build
    [delete] Deleting directory /home/csdev/proj/tdd-paystation/javadoc
    [delete] Deleting directory /home/csdev/proj/tdd-paystation/TEST-RESULT

BUILD SUCCESSFUL
Total time: 0 seconds
csdev@csdev:~/proj/tdd-paystation$ ant test
Buildfile: /home/csdev/proj/tdd-paystation/build.xml

prepare:
    [mkdir] Created dir: /home/csdev/proj/tdd-paystation/build
    [mkdir] Created dir: /home/csdev/proj/tdd-paystation/javadoc

build-src:
    [javac] Compiling 4 source files to /home/csdev/proj/tdd-paystation/build

build-test:
    [javac] Compiling 1 source file to /home/csdev/proj/tdd-paystation/build

build-all:

test:
    [java] JUnit version 4.12
    [java] .
    [java] Time: 0.006
    [java]
    [java] OK (1 test)
    [java]

BUILD SUCCESSFUL
Total time: 0 seconds
csdev@csdev:~/proj/tdd-paystation$
```



AARHUS UNIVERSITET

# Gradle in SWEA

In 2018 Ant was put to rest...

- 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.

# And One BIG Nuisance

- Gradle developers are *not afraid of none-backward-compatible* changes ☹️ ☹️ ☹️
  - Indeed they syntax and semantics have slightly or in **code-breaking ways** changed 30 times since I started using Gradle ☠️ ☠️ 💣 ☠️
- So many hours wasted on absolutely nothing...
  - Beware of slides that may contain old-style gradle ☹️

# Gradle build.gradle

- The simplest *build.gradle* file for java dev, contains one line

– ~~*apply plugin: 'java'*~~

```
plugins {
    id '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'
}
```

```
plugins {
    id 'java'
}

repositories {
    mavenCentral()
}

dependencies {
    testImplementation group: 'org.junit.jupiter',
        name: 'junit-jupiter-api', version: '5.6.2'
    testRuntimeOnly group: 'org.junit.jupiter',
        name: 'junit-jupiter-engine', version: '5.6.2'

    testImplementation group: 'org.hamcrest',
        name: 'hamcrest-library', version: '2.2'
}

test {
    useJUnitPlatform()
}
```

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

- Ex: we need hamcrest

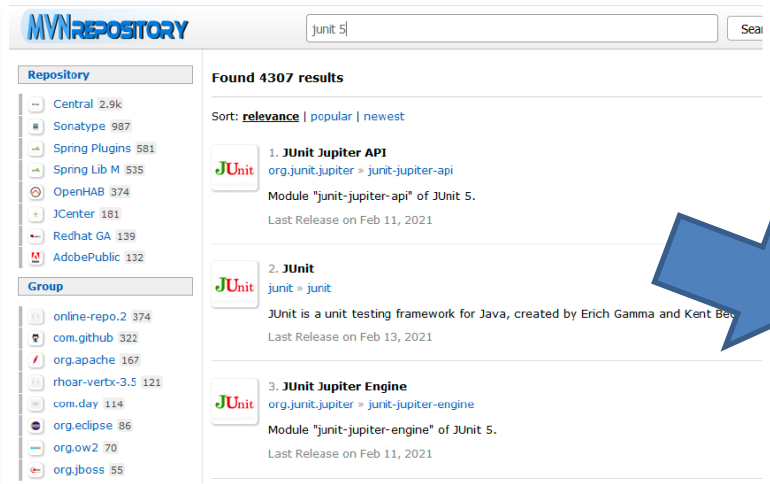


```
plugins {  
    id 'java'  
}  
  
repositories {  
    mavenCentral()  
}  
  
dependencies {  
    testImplementation group: 'org.junit.jupiter',  
        name: 'junit-jupiter-api', version: '5.6.2'  
    testRuntimeOnly group: 'org.junit.jupiter',  
        name: 'junit-jupiter-engine', version: '5.6.2'  
  
    testImplementation group: 'org.hamcrest',  
        name: 'hamcrest-library', version: '2.2'  
}  
  
test {  
    useJUnitPlatform()  
}
```

- Gradle will download 'org.hamcrest....:2.2' from MavenCentral on the internet, and set the classpath correctly



- Find libraries on 'www.mvnrepository.com'



The screenshot shows the Maven Repository search results for 'junit 5'. The search bar contains 'junit 5' and the results are sorted by relevance. The first result is 'JUnit Jupiter API' (org.junit.jupiter:junit-jupiter-api) version 5.7.1, which is highlighted with a blue arrow pointing to the detailed view on the right. The left sidebar shows a list of repositories and groups.

Home » org.junit.jupiter » junit-jupiter-api » 5.7.1



## JUnit Jupiter API » 5.7.1

Module "junit-jupiter-api" of JUnit 5.

License	EPL 2.0
Categories	Testing Frameworks
HomePage	<a href="https://junit.org/junit5/">https://junit.org/junit5/</a>
Date	(Feb 04, 2021)
Files	<a href="#">jar (171 KB)</a> <a href="#">View All</a>
Repositories	Central
Used By	8,001 artifacts

**Note:** There is a new version for this artifact

New Version

5.8.0-M1

Maven Gradle SBT Ivy Grape Leiningen Buildr

```
testImplementation group: 'org.junit.jupiter', name: 'junit-jupiter-api', version: '5.7.1'
```

# Update from time to time!

- Lists security issues

Home » [org.apache.logging.log4j](#) » log4j-core



## Apache Log4j Core

Implementation for Apache Log4J, a highly configurable logging tool that focuses on performance and low garbage generation. It has a plugin architecture that makes it extensible and supports asynchronous logging based on LMAX Disruptor.

License	Apache 2.0
Categories	Logging Frameworks
Tags	logging log4j apache
Ranking	#54 in MvnRepository (See Top Artifacts) #6 in Logging Frameworks
Used By	8,444 artifacts




<a href="#">Central (54)</a> <a href="#">Redhat GA (23)</a> <a href="#">Redhat EA (3)</a> <a href="#">Gael (1)</a> <a href="#">ICM (3)</a>					
Version	Vulnerabilities	Repository	Usages	Date	
2.18.x					
2.18.0		Central	444	Jul 02, 2022	
2.17.2		Central	1,155	Feb 27, 2022	
2.17.x					
2.17.1		Central	2,146	Dec 27, 2021	
2.17.0	2 vulnerabilities	Central	954	Dec 18, 2021	
2.16.x					
2.16.0	3 vulnerabilities	Central	979	Dec 13, 2021	
2.15.x					
2.15.0	5 vulnerabilities	Central	1,170	Dec 10, 2021	
2.14.1	8 vulnerabilities	Central	1,089	Mar 12, 2021	
2.14.x					

- 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
    - But – normally you don’t!

```
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

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

- A list of **dependencies**. Targets depend upon each other; 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.

# 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!**

- Why not just use IntelliJ?

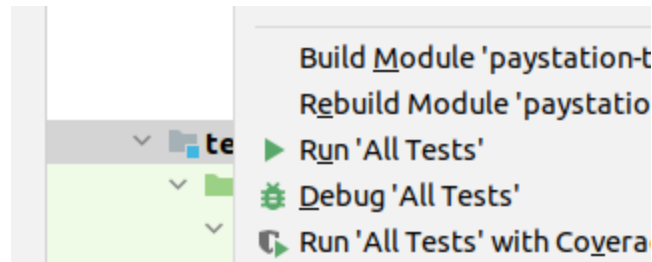
- It does it all, right?

- Answer

- True but...

- Gradle can do stuff that IntelliJ can not.

- Servers do not have a graphical interface!



# Other BM tools

- 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 ☹️



# 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 😊