BUILDMANAGEMENT

(C) Prof. Dr. Stefan Edlich

LEARNING GOALS

This learning unit describes the function of build management. The aim is to be able to use build management practically and assess its importance within the software life cycle.

- Get to know the historical origins and the make tool as well as some make derivatives
- Classification in the development cycle Introduction and practical use of some basic tools

> Note that this unit is always seen in confunction with the CI/CD unit.

BUILD MANAGEMENT AND CONTINUOUS INTEGRATION

In addition to error, configuration, version and release management, the management tasks in the software development process also include build management.

The first build management tools emerged in the 1980s, which often only contained a few files to be compiled. A decade later, software systems became significantly **larger**.

In addition, the **dependencies** of the software to be developed grew. During the development process, a wide variety of components are created by many departments, which then have to be brought together. In addition, e.g. B. Application server required to run the application.

From a technical perspective, there are a variety of **protocols** and **services** that need to be developed and set up. For example, database connections must be established, web services must be accessible, etc. The application is often located in a complete SOA network and can never be viewed in isolation.

> Dependencies make building by hand almost impossible (nevertheless we can still see it in companies: "I works on my computer!")

All of this means that creating an executable application becomes more and more complex: downloading files (from the version management system such as CVS or Subversion), translating, testing, archives such as e.g. B. pack jar, copy, apply, establish connections, start services, etc.

Only then is the application ready and ready to run. It becomes clear very quickly that these processes **cannot be done by hand** every time.

To save time and avoid errors, this process needs to be automated.

> **NOTICE: Configuration management**

Configuration management plays a role in this process, which according to ISO 10007 consists of the following parts:

- Identification
- Surveillance
- Accounting
- Auditing The simplest example is creating different configurations for hardware platforms. It will e.g. B. a program for LINUX and one for Windows are required.

Creation of a prototype

Another point that arises from agile development and the XP principles is the creation of a prototype that can be created at any time - with a mouse click, so to speak. This is an important component that is increasingly required by clients in order to receive continuous feedback about the current status.

For this reason, modern IDEs such as Visual Studio, IntelliJ, JBuilder have **integrated** many build management systems. However, a build cannot always be done from the IDE. In large industrial companies or banks there are builds that have to be run as a batch process at night because they take many hours.

BUILD MANAGEMENT

How is the term build management defined?

> **DEFINITION**: Build management is a process in which a (software) product is created transparently and repeatably. This requires automation of the process.

Therefore we need:

- There is a stable environment (e.g. naming) under which the process can run.
- This process can be specified/programmed using a build language (usually a scripting language).
- This build can be executed with a tool (make, ant, etc.).
- The build delivers clearly defined results and logs.

The build logs are often empty or non-existent, which, analogous to UNIX conventions, indicates a positive build completion. Colored output and reports, as known from JUnit or corresponding test runners, are even possible.

Build management goals

One goal of build management is to make the build of the software **independent** of the software release (version). If possible, a new version should not result in any changes to the build script.

It is important that the target configurations and the different builds for different editions (Linux, Windows, etc.) or formats (jar, ear Enterprise Archive, Webstart, etc.) are **documented** in the same way (also in the build

file) as in the tools used (e.g. jdk 21).

Build management has now become so important and complex that it has become a separate profession. Build managers are therefore busy "daily" with (re)defining and carrying out this process.

A life without build management?

What would happen if build management wasn't used in a larger company?

- The final product would not run and no one would know the cause.
- Details would simply be forgotten (e.g. running the RMI compiler manually, which would result in the stubs missing).
- The knowledge about the software creation process would not be available and documented centrally, but would be, for example, in the heads of the employees (who are guaranteed to get sick or go on vacation at the "wrong" time), on notes or in various files.
- Assigning the version number (e.g. 1.2.24) would be done manually.
- The processes would take too much time. (Tests or generation of documentation would have to be initiated manually, etc.)

CONTINUOUS INTEGRATION

MARTIN FOWLER and **KENT BECK** coined the terms **continuous integration**. They are almost always mentioned in connection with Extreme Programming (see chapter "Extreme Programming (*XP*)" in the learning unit "VOR – Process Models / Agile Models").

> WEBSOURCE: The most well-known document on continuous integration comes from MARTIN FOWLER and was updated on May 1, 2006:
> http://martinfowler.com/articles/continuousIntegration.html

The above document contains the following statement:

> **DEFINITION**: "Continuous integration is a way of developing software. The (daily) work of every developer contributes to the fact that integration can be carried out several times a day. These integration builds are checked by the automated build (e.g. through testing) to catch errors as early as possible. Many teams have reported that this approach significantly reduces integration issues. This allows cohesive software to be created more quickly." Martin Fowler

There are many ways to represent this approach. The most important elements are therefore shown as follows:

BUILD STAGES

The basic idea of a fast automated build is to identify and fix integration problems as quickly as possible. An XP guideline states that a build can be initiated easily and by any developer; **best after checking in new code**. This manually initiated build also serves as a test for this new code, as these are executed in build tests. These builds should **not take longer than 10 minutes**, which is usually achievable (with compiling, testing, packaging, copying, etc.).

Build in stages

The build process is usually divided into **stages**, so-called stages. A **stage build** is therefore a process (a "build pipeline") in which the stages of the build process are passed through one after the other. This is also referred to as a **stage server** on which "committing" and "building" takes place.

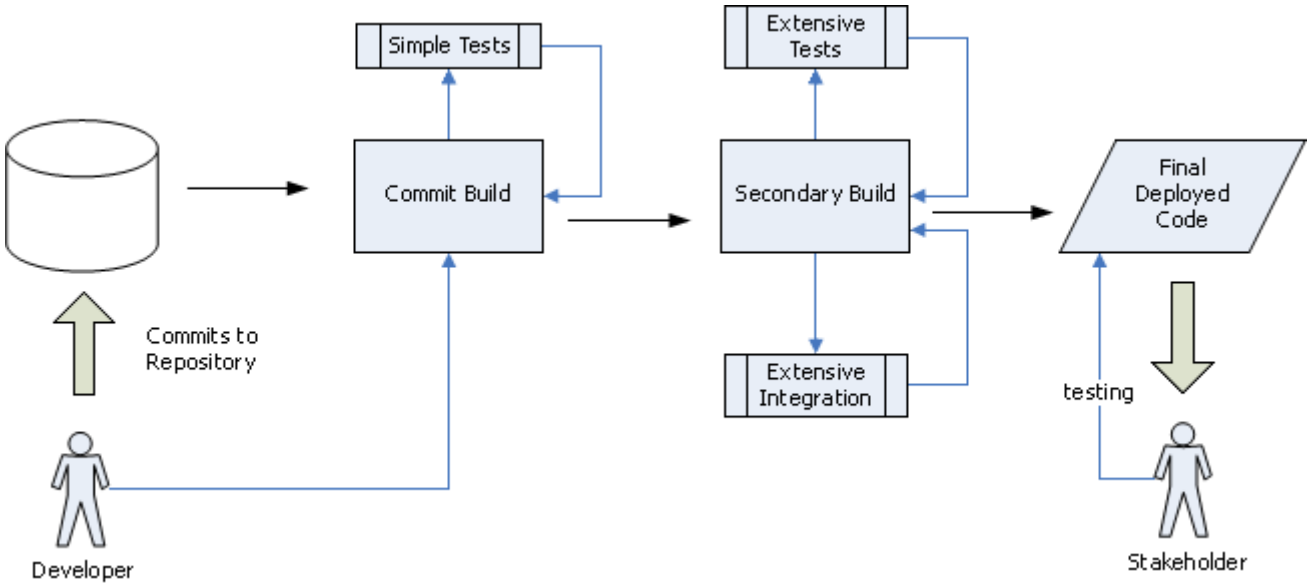
1. Commit Build

When new code is integrated, the first build is called **commit-build** and is executed. This ensures that, if necessary, tests are run for this code and the build for the main program (the so-called **mainline**) can be created.

2. Secondary Build

Further builds such as **secondary builds** will be added later. These are able to carry out more intensive tests and complete the product in a more targeted manner, i.e. H. for example, to create a large war archive ("web archive" from J2EE), which can then be executed on a server such as JBoss or Weblogic. If this process - which is further back in the process - can also be carried out without any problems, these builds can be considered successfully completed.

This process is shown in the following graphic:



Pic: Stages of the build process

CONFIGURATION MANAGEMENT

> **DEFINITION**: ANSI defines configuration management as follows: "Configuration Management [...] is a management process for establishing and maintaining consistency of a product's performance, its functional and physical attributes, with its requirements, design and operational information, throughout its life."

Configuration management refers to a management approach in which the configuration units/attributes of the product are managed and defined throughout the entire product life cycle. This refers to hardware as well as software and services. For example, certain products can only work in conjunction with a specific service.

Configuration parts ISO 10007

ISO 10007 therefore recognizes the following aspects of configuration management:

- KMO - configuration management organization and planning
- AI - configuration identification
- KÜ - configuration monitoring
- KB - configuration accounting
- KA - configuration audit

Build management is therefore in a sense a part of configuration management that relates to the software. How the software is to be configured is centrally defined and documented in the build management script. i.e. for example, for which platform or requirements, which code (or tests or documentation) should be created and installed on the server.

Sometimes build management is (incorrectly) equated with configuration management. This may be appropriate in small projects, but is no longer appropriate for larger concepts (e.g. Toll Collect).

HISTORY AND TOOLS

The history of build management is closely linked to the **make** tool, which still plays an important role today.

This tool has its origins in **UNIX** systems on which code had to be compiled. Compiling it by hand was a tedious task. Often only a few files were changed, which had to be laboriously identified and then translated by hand. Even with commands like "cc *.c" (cc is the compiler and *.c references all C files in this directory) that are easy to type, the compilation process can take too long if many files have been collected but only a few be changed. And what if the source code is distributed across multiple directories? There are 1000 reasons why manual labor is pointless.

STEWARD FELDMAN recognized this deficiency and invented a tool that automates these processes. The processes are simply recorded in a build file that looks like a script file. However, other features such as dependencies were crucial back then.

> DEFINITION: Dependencies are dependencies between code or modules. For example, file A may have been changed and not only A but also B needs to be recompiled.

These dependencies can be recognized by modern tools or noted/specified directly in build files.

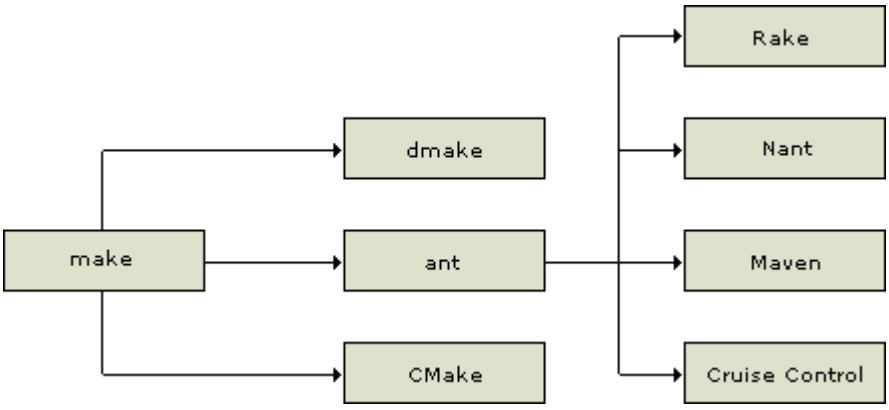
The classic example of a Makefile is shown below:

makefile

```
1 all: hello
2
3 hello: main.o factorial.o hello.o
4 g++ main.o factorial.o hello.o -o hello
5
6 main.o: main.cpp
7 g++ -c main.cpp
8th
9 factorial.o: factorial.cpp
10 g++ -c factorial.cpp
11
12 hello.o: hello.cpp
13 g++ -c hello.cpp
14
15 clean:
16 rm -rf *o hello
```

Make is still being maintained today and has found its place in the GNU Foundation. Thanks to the good support of this tool, GNU make (http://www.gnu.org) is still very important today.

Some of the main make derivatives are shown in the following figure:



Pic: Important make derivatives

After the success of make, a few variants emerged that either rely on platform independence like CMake (http://www.cmake.org) or are specifically tailored for distributed execution like dmake (distributed make) and were developed by SUN for Solaris and OpenOffice became.

APACHE ANT

However, one of the pioneers of modern build management is Ant http://ant.apache.org. Ant's breakthrough lies in XML. The Extensible Markup Language was standardized between 1996 and 1999. XML only provided a description that was difficult for humans to read, but particularly easy for machines to read, and was generated by generators. Unlike make, this involves a DTD (Document Type Definition) or a schema that precisely checks the structure of an XML document. Developing editors/accessibility aids in the IDE is now easy.

The developments of XML and Ant have made a quantum leap in build management. Ant itself has been stable in version 1.6 for a long time and is used in almost all relevant and larger Java projects.

Following this success, other companies and communities also developed similar tools. Nant (http://nant.sourceforge.net) for the .Net environment and Rake (http://rake.rubyforge.org) for Ruby are just two examples here.

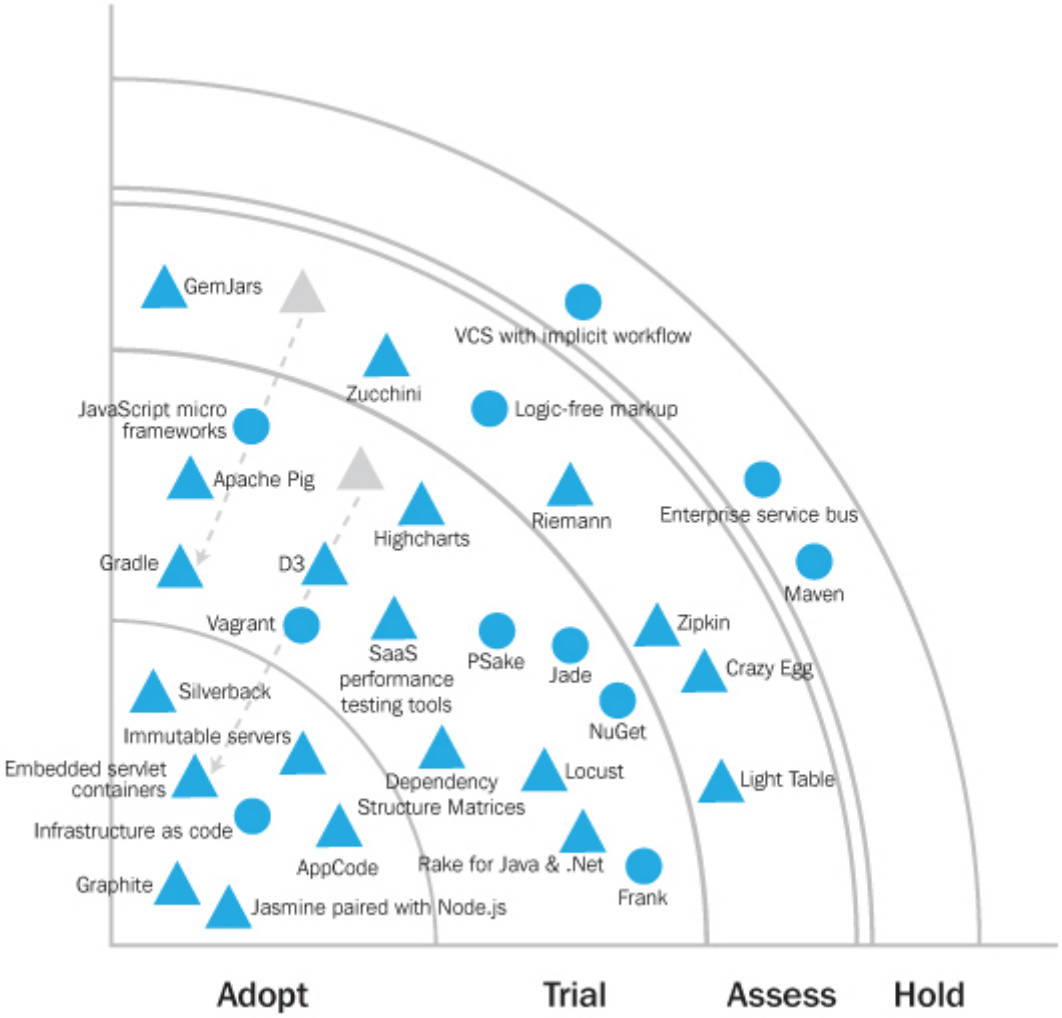
Finally, there are advanced tools that build on top of (or replace) Ant and, in particular, promote the continuous integration cycle. Maven (http://maven.apache.org) and Cruise Control (http://cruisecontrol.sourceforge.net) are the most popular tools. More detailed descriptions can also be found in [Ba03].

PRINZIPIEN DES BUILDMANAGEMENTS

If you look at the history of build management, one of the questions that arises is: What principles can be derived from it?

Tools and APIs are subject to a continuous process of change. They can hardly always be presented here up to date. In the following chapters, the most important principles are outlined and a selection of tools are briefly presented and evaluated.

Note: Which build management systems are relevant?



As you can see from an old (!) Tech Radar, Ant is no longer included as an industry standard. Maven is mentioned but is on hold. Gradle moves from assess to trial.

> So please find the newest TechRadar from Thoughtworks!

In hindsight we see that Maven and Gradle did it. But for performance reasons, many companies like Google and Facebook have developed their own new build management systems. Please google **Bazel** and **Buck**!

BUILD MANAGEMENT IN THE XML/ANT AGE

With the adoption of XML since 1998, ANT has established itself as the leading platform for almost a decade.

> WEBSOURCE: The most important thing is the reference to the manual:
> ant.apache.org/manual/index.html

Here is a simple example from the manual:

```
1 <project name="MyProject" default="dist" basedir=".">
2 <description>
3 simple example build file
4 </description>
5 <!-- set global properties for this build -->
6 <property name="src" location="src"/>
7 <property name="build" location="build"/>
8 <property name="dist" location="dist"/>
9
10 <target name="init">
11 <!-- Create the time stamp -->
12 <tstamp/>
13 <!-- Create the build directory structure used by compile -->
14 <mkdir dir="${build}"/>
15 </target>
```

16 17 <target name="compile" depends="init" 18 description="compile the source " > 19 20 21 22 23 <target name="dist" depends="compile" 24 description="generate the distribution" > 25 26 27 28 29 30 31 32 <target name="clean" 33 description="clean up" > 34 35 36 37 38

Let's analyze the advantages and disadvantages here.

Build management in the XML/ANT age With the adoption of XML since 1998, ANT has established itself as the leading platform for a decade.

WEBSOURCE The most important thing is the reference to the manual:

ant.apache.org/manual/index.html

Here is a simple example from the manual: Code: © ant.apache.org

```
1 <project name="MyProject" default="dist" basedir=".">
2 <description>
3 simple example build file
4 </description>
5 <!-- set global properties for this build -->
6 <property name="src" location="src"/>
7 <property name="build" location="build"/>
8 <property name="dist" location="dist"/>
9
10 <target name="init">
11 <!-- Create the time stamp -->
12 <tstamp/>
13 <!-- Create the build directory structure used by compile -->
14 <mkdir dir="${build}"/>
15 </target>
16
```

```
17 <target name="compile" depends="init"
18 description="compile the source " >
19 <!-- Compile the java code from ${src} into ${build} -->
20 <javac srcdir="${src}" destdir="${build}"/>
21 </target>
22
23 <target name="dist" depends="compile"
24 description="generate the distribution" >
25 <!-- Create the distribution directory -->
26 <mkdir dir="${dist}/lib"/>
27
28 <!-- Put everything in ${build} into the MyProject-${DSTAMP}.jar file -->
29 <jar jarfile="${dist}/lib/MyProject-${DSTAMP}.jar" basedir="${build}"/>
30 </target>
31
32 <target name="clean"
33 description="clean up" >
34 <!-- Delete the ${build} and ${dist} directory trees -->
35 <delete dir="${build}"/>
36 <delete dir="${dist}"/>
37 </target>
38 </project>
```

Let's analyze the advantages and disadvantages here too.

Advantages:

- Easily machine readable
- Easily expandable

But there are also enough **disadvantages**:

- It is **chatty** and therefore not easy for people to read.
- Even with the help of assistants, you will always find a forgotten / or " somewhere.
- Ant is not powerful; d. H. e.g. B. **not Turing-complete**!

Ant is of course powerful in that arbitrary new tasks can be easily defined. This is done in Java and is wrapped in XML. This makes the tasks themselves Java-powered, which isn't a bad thing.

However, the language that Ant defines is limited to XML Schema. You can use the “AntStructure Task” to generate a DTD (Document Type Definition), but this is also incomplete because attributes often depend on parameters.

This makes it clear that the power of this DTD is far from the power of a programming language, i.e. H. not even Turing-complete. There are the `if/else` constructs, but no jumps or loops. And properties are not variables, but `“final”` constants.

Still, Ant is a powerful industry standard that has lasted for a long time.

The chapter “Other build management systems” explains which build languages had the power to win Ant’s chair.

PRINCIPLES: CONVENTIONS, DEPENDENCIES AND PLUG-IN

Maven is often presented as a much more powerful tool than a bare build tool. In addition to the standard tasks clone, compile, package, test, deployment, the focus is on other project-supporting tasks, such as: B. the work structure, bringing the team together, reporting, etc.

The unique selling points have already been defined:

- Standard frame
- Dependency Management
- Project life cycle
- Plug-in architecture
- All of this is controlled by the Project Object Model (POM).

Principle: convention before configuration

The principle says: **Use standard settings, then you don't have to do any configuration work**. These should only be stated if there are justified differences. However, deviations are always possible.

Ruby on Rails by David Heinemeier-Hansson has taken this principle to the extreme many years ago: An MVC environment is automatically generated here, to which the developer has to adapt. This is standard for many tools today; especially with web frameworks.

Maven also has tests, resources, program code and the result in defined folders.

This is sometimes also referred to as standard frames. Not just the places, but also the processes - such as: B. the tests – are standardized.

Maven Plugins

Maven itself is an XML parser that only includes a few built-in plugins. All functionality is integrated via the plug-ins. All of this plug-in functionality, as well as the dependencies, are downloaded from central repositories (see e.g. http://search.maven.org). The plug-in knowledge, structure, dependencies and location of the libraries are stored there.

Dependency Management

With unique identifiers for the dependency group, artifact, and version number, the current version and corresponding dependencies can be automatically discovered and downloaded from the repository.

Archetypes

Archetypes are templates for familiar tasks such as J2EE or web development.

> WEBSOURCE: Introduction to Archetypes https://maven.apache.org/guides/introduction/introduction-to-archetypes.html

They help to establish a correct and uniform project structure in companies and can be customized.

> Conclusion: Many projects rely on Maven instead of Ant to keep the configuration effort low. Maven POM files are typically significantly smaller than ANT files.

PRINCIPLE: TOP DSL

Gradle is based on a '**Groovy DSL**' and was created by HANS DOCKTER.

It also includes some of the principles of Maven:

- extensive dependency management
- defaults = convention over configuration

Let's take a look at a Hello World:

Hello World:

```
3 task hello << {
4 println 'Hello'}}
5
6 task world(dependsOn: hello) << { print " World"}
```

- You can already see here that it is quite flexible and expandable thanks to a syntax similar to that of a programming language.
- It offers many plugins for Java, Groovy, Scala, web projects and similar.
- It integrates well with Maven, Ant or Ivy.

There are also other interesting features:

- Incremental build management
- Parallel execution of UNIT tests
- A demon
- Tasks are dynamic
- a GUI
- Ant tasks are first class objects in Gradle. A complete Ant script can be imported and the tasks can be called transparently from Gradle.

A few examples of the last point.

Gradle and Ant:

```
7 task hello << {
8 String hi = "Ant says Hi!"
9 ant.echo(message: hi)
11
10 }
12 task zip << {
13 ant.zip(destfile: all.zip) {
14 fileset(dir: src) {
15 exclude(name: **.class)
16 include(name: **.java)
17 }
18 }
```

```
19 }
20
21 ant.importBuild 'build.xml'
```

Additionally, many source directory sets can be defined, for which different dependencies, documents, and even different JDKs can be defined.

Under Ant, defining and referencing files was not always trivial. This is not shorter under Gradle, but it is clearer to understand:

```
22 File configFile = file(project1/coredata.json')
23
24 FileCollection collection =
25 files('res/fileA.txt',new File('res/fileB.txt'),['res/fileC.txt',
26 'res/file.txt'])
```

Finally, a comparison with Ant:

```
27 ant.copy(todir: 'build/img') {
28   fileset(dir: 'resources/img')
29 }
30 task copyTask(type: Copy) {
31   from 'src/de/edlich/swt/build'
32   into 'build/src/de/edlich/swt/‘
33   include '**/*.rb'
34   exclude { details ->
35   details.file.name.endsWith('.yaml') &&
36   details.file.text.contains('beta')
37   }
38 }
```

Which projects use Gradle?

Tons of as: Groovy, Grails, Hibernate, Spring (security, integration), Canoo, Griffon, Gant, Aluminum, etc.

Therefore they are in good company here.

What plug-ins are there?

- Java, Groovy, Scala, Antlr
- ear, jetty, maven, osgi, war
- checkstyle, codenarc, eclipse, findbugs, jdepend, pmd, sonar
- and many more...

OTHER BUILD MANAGEMENT SYSTEMS

What other build management systems are there?

Please research online. There are several hundred and almost every programming language has its own system!

Rake

Rake was built by JIM WEINRICH. Rake is also an interesting DSL in Ruby that relies heavily on dependencies. Unfortunately there are no rake books yet, but there are some rake tutorials.

An example:

```
1 task :turn_off_alarm do
2 puts "Turned off alarm. Would have liked 5 more minutes, though."
3 end
4 task :groom_myself do
5 puts "Brushed teeth."
6 puts "Showered."
7 puts "Shaved."
8 end
9 desc "I am describing this task"
10 task :make_coffee do
11 cups = ENV["COFFEE_CUPS"] || 2
12 puts "Made #{cups} cups of coffee. Shakes are gone."
13 end
```

14 task :walk_dog do 15 puts "Dog walked." 16 end 17 task :ready_for_the_day 18 => [:turn_off_alarm, :groom_myself, :make_coffee, :walk_dog] do 19 puts "Ready for the day!" 20 end

> QUOTE: “This means you have the full power of the language at hand.” Jason Seifer

> Nowadays almost **every language** has its own build language such as Scala with SBT.

However, the future lies in other systems such as “Google Bazle” or “Facebooks Buck” or integrated CI/CD systems!

WEBSOURCE

- https://bazel.build/
- Buck https://github.com/facebook/buck
- https://please.build

Please take a look and try to understand these concepts too.

14 ANT CHAPTERS HERE (TODO)

14 Ant Chapters are waiting to be inserted here.

MAVEN PRACTICE

Install and run Maven

Of course, Maven requires Java. There should be at least a 1.5 installation 😉

Test the installation:

```
C:\Users\Edlich> java -version
java version "1.7.0_03"
Java(TM) SE runtime environment (Build 1.7.0_03-b05)
Java HotSpot(TM) 64-Bit Server VM (build 22.1-b02, mixed mode)
C:\Users\Edlich>
```

Downloading the Maven version (checking for the correct version) is done from this address:

http://maven.apache.org/download.html

The installation details for all operating systems are easy to google. The package is unpacked and then the correct paths are set.

For Windows:

```
CODE
F:\CODING> set M2_HOME=F:\PROGRAMS\apache-maven-3.0.4
F:\CODING> set PATH=%PATH%;%M2_HOME%\bin
```

Then Maven should be good to go.

Test:

```
Apache Maven 3.0.4 (r1232337; 2012-01-17 09:44:56+0100)
Maven home: F:\PROGRAMS\apache-maven-3.0.4
Java version: 1.7.0_03, vendor: Oracle Corporation
Java home: C:\Program Files\Java\jdk1.7.0_03\jre
Default locale: de_DE, platform encoding: Cp1252
OS name: "windows 7", version: "6.1", arch: "amd64", family: "windows"
```

Now you can generate an archetype when connected to the internet!

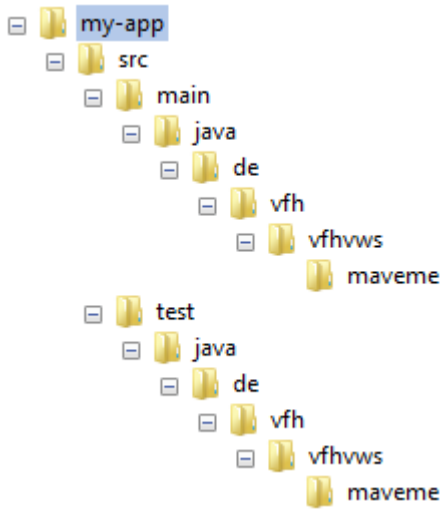
```
F:\CODING> mvn archetype:generate -DgroupId=com.mycompany.app -DartifactId=my-app
-DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

Now Maven will download countless libraries, which may take a few minutes.

One suspects that Maven is a frequent point of criticism here: Maven takes forever to load something from the internet and can sometimes “hang” when resources are not available. Here you have to help a little in order to run Maven in offline mode.

The following structure is then created:

BUI Maven-Struct.gif



PIC: File structure

A Java template is also created:

App.java

```java
package de.vfh.vfhvws.maveme;

/**
 * Hello world!
 *
 */
public class App
{
    public static void main( String[] args )
    {
        System.out.println( "Hello World!" );
    }
}
```

Pic: Java template

A corresponding test file is also created.

Now you can just go with it

```
> mvn compile // (approx. 32 sec)
> mvn package // (approx. 29 sec)
```

compile and create a jar. A target directory is created that contains the classes and other folders.

The whole thing can then be carried out:

```
F:\CODING\my-app\target>java -cp my-app-1.0-SNAPSHOT.jar
de.vfh.vfhvws.maveme.App
Hello World!
F:\CODING\my-app\target>
```

Finally, you can clean up with the following line:

```
> mvn clean
```

The target directory is then deleted.

MAVEN QUICK-REFERNCE

WEBSOURCE

- Maven: The complete Reference http://de.sonatype.com/resources/ebooks
- E-Book: Better Builds with Maven http://de.scribd.com/document/238927/Better-Builds-With-Maven
- Video-Tutorial Maven http://youtube.com/watch?v=al7bRZzz4oU

... end endless more references on the internet.

GRADLE PRACTICE

Gradle “up and away”

The first step is to make sure that Java is installed and can be found in the path.

After downloading `gradle-VERSION-all.zip` from http://www.gradle.org, Gradle must be unpacked.

Then GRADLE_HOME/bin needs to be added to the PATH environment variable. (There are path editors for Windows such as those on softpedia.com)

If PATH has been expanded accordingly, it is necessary to test whether the installation was successful and Gradle is found:

```
F:\CODING>gradle -v
------------------------------------------------------------ ----------


Gradle 1.0-rc-1
```

```
------------------------------------------------------------ ----------
Gradle build time: Wednesday April 11, 2012 11:13 am UTC
Groovy: 1.8.6
Ant: Apache Ant(TM) version 1.8.2 compiled on December 20 2010
Ivy: 2.2.0
JVM: 1.7.0_03 (Oracle Corporation 22.1-b02)
OS: Windows 7 6.1 amd64
```

In contrast to Ant, Gradle only knows projects and tasks, but not the term “targets”, which correspond to the Gradle tasks.

Now let's edit a "Hello World":

```
1 File name: build.gradle
2
3 task hello { // task
4   doLast {
5     println 'Hello SWT master!'
6   }
7 }
```

build.gradle is the file name analogous to Ant's build.xml.

We run this with gradle hello or with `gradle -q` hello. `-q` (quiet) suppresses Gradle's system messages.

GRADLE QUICK REFERENCE

Source: (C) Gradle User Guide

Simplest definition

```
1 task hello << {
2 println 'Hello world!'}
```

The println is added dynamically here with the following code:

```
1 hello.doFirst { // add
2 prints 'Hello Venus'}
3 hello.doLast { // add
4 println 'Hello Mars'}
5 hello << { // add
6 println 'Hello Jupiter'}
```

would yield

```
Hello Venus,
Hello world!,
Hello Mars
```

and in the end

Hello Jupiter

will be executed.

List of all tasks

You can get a list of all self-defined (and built-in) tasks by calling gradle tasks.

Example:

```
F:\CODING\GrTest>gradle tasks
:tasks

------------------------------------------------------------
All tasks runnable from root project
------------------------------------------------------------

Help tasks
----------

dependencies - Displays the dependencies of root project 'GrTest'.
help - Displays a help message
projects - Displays the sub-projects of root project 'GrTest'.
properties - Displays the properties of root project 'GrTest'.
tasks - Displays the tasks runnable from rot project 'GrTest' (some of the
displayed tasks may belong to subprojects).

Other tasks

-----------
copyfile
delfile

To see all tasks and more detail, run with --all.

BUILD SUCCESSFUL

Total time: 2,574 seconds
F:\CODING\GrTest>
```

Use Groovy code

```
1 task upper << {
2   String someString = 'mY_nAmE'
3   println "Original: " + someString
4   println "Upper case: " + someString.toUpperCase()
5 }
1 task count << {
2   4.times { print "$it " }
3 }
```

Obviously all variables can be defined somewhere and reused. This would have killed global and local properties. System and command line properties are covered later.

Dependencies

```
1 task hello << {
2 println 'Hello world!'}
3 task intro(dependsOn: hello) << {
4 println "I'm Gradle"}
```

Dynamic tasks

```
1 4.times { counter ->
2 task "task$counter" << {
3 println "I'm task number $counter" }
4 }
```

Is called with:

```
> gradle -q task1 // or task0, task2, task 3, but not task 4 !
```

Tasks can be created **at runtime**!

Each task is an object and has additional properties!

Task properties can be accessed as follows:

```
1 task hello << {
2 println 'Hello SWT!'}
1 hello.ext.taste = "Banana" // my properties
2 hello.doLast {
3 println "Greetings from the $hello.name task with flavor $hello.ext.taste."}
```

Ant integration

Ant is wonderfully integrated. All tasks are easily available. Here are two examples:

The Ant task Loadfile loads a file. Ant often has characteristics in this. But it could also be anything else. We read the file and output the file name and content:

```
1 task loadfile << {
2   def file = file('Textfile.txt')
3   ant.loadfile(srcFile: file, property: file.name)
4   println "Filename = $file"
5   println "Properties = ${ant.properties[file.name]}"}
```

What is interesting here?

We call the Ant Task with `ant.open`. The parameter attributes are present with colons. For example srcFile:: and after that comes the value. Just like in the Ant documentation.

Now something new. E.g. copy:

```
1 task copyfile << {
2   ant.copy(file: 'Textfile.txt', tofile: 'IchKopie.txt')
3   println "Done!"
4 }
5
6 task delfile << {
7   ant.delete(file: 'IchKopie.txt')
8   println 'IchKopie.txt killed'
9 }
```

So if the file actually exists, it can be copied using gradle copyfile and deleted using gradle delfile.

Interestingly, you can define the file in tofile value in different ways:

```
tofile: tofile // with def tofile = file('ICopy.txt')
tofile: tofile.name // with the definition above
tofile: 'ICopy.txt'
```

More power through methods

```
1 task methoddemo << {
2   println "Result: " + doubleMe("Cream")}
3
4   String doubleMe(String arg) {
```

```
5   arg + "+" + arg
6 }
```

So I can use any method anywhere in any task.

Default tasks

How to define default tasks:

```
1 defaultTasks 'clean', 'run'
2
3 task clean << {
4 println 'Default Cleaning!' // will run}
5
6 task run << {
7 println 'Default Running!' // will run}
8th
9 tasks other << {
10 println "I'm not a default task!" // wants to cry}
```

Gradle GUI > gradle --gui

Features:

```
Tree view
Launcher
Favorites (for long build files)
Pass parameters
Browse new build files, log levels, filters, etc.
```

Eclipse

Unfortunately there doesn't seem to be a nice Eclipse plugin. Or? So you can only use GUI or Gradle as external tools under Eclipse: Weblink: Run Gradle from Eclipse.

In this configuration, the task is passed via prompt. A tree view is not available in Eclipse, but it could be.

But IntelliJ and VSCode Idea may have had Gradle support well before 2017.

ADVANTAGES AND DISADVANTAGES OF THESE SYSTEMS

It's time to look at the pros and cons of the new systems:

Maven:

- If you don't deviate from the standard, your life is easy
- Good ecosystem

- At least two free books

- Reading and understanding XML POMs is no fun
- Expressiveness only via XML parameters
- Better with the internet running (even if it goes offline...)

Gradle:

- Very expressive language
- Great docs
- Many additional features (Deamons, GUI, parallel, dynamic tasks, etc.) + At least 5 books

- Weak Eclipse integration (2013)

In one sentence: (and strikingly simplified)

- XML haters will be happy with Gradle.
- XML lovers will feel at home in Maven.

And of course it makes sense to deal with the specific build systems of a language if you stay completely in that language.

SUMMARY

- The importance of the build management process in connection with continuous integration and what types of builds exist were shown.
- The term configuration management and the build management history as well as common tools on the market were introduced.
- The important basic terms of Ant, such as Target, Tasks, Dependencies, have been defined. You should be able to create Ant scripts yourself. The IDE integration and property handling were covered.
- You should know the most important tasks for build management and the importance of patterns and filesets.
- Advanced concepts such as build number assignment, structuring with Ant or Include or parallel execution under Ant are no longer foreign to you.
- You should have the necessary tools to create your own tasks.
- You have received a brief overview of the history of build management and know the basic principles of build management systems.
- You got to know Ant, Maven and Gradle in practical exercises.
- You now know the advantages and disadvantages of all systems.

> NOTICE: You have reached the end of this learning unit.

EXERCISES

Gain experience with **two** (!) build management systems:

1. In Ant you build tasks like: `CLEAN, GET, COMPILE, JAR, TEST, RUN, DOC`

First build the skeleton with dependencies and echos. Some tasks like compile, test or doc often mess around. The compile task, for example, needs the compiler. But he doesn't know it. Therefore you have to give the Ant environment the lib rt.jar in the environment!

2. Maven

3. Gradle

or any other general purpose build management system (for your language).

No pure package or deployment managers! They are often too special.

> Show how you got your scripts **running**, what the **outputs** are and what **experiences** you have had! I am not interested in internet `copy & paste` or `LLM` output! Please take this issue serious!

Processing time: 60 minutes