Build Tools

Software Engineering

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Agenda for Today

Build Tools

1. Motivation
2. Key Concepts
3. Tools Available
4. Presentation
5. Discussion

Objectives

- Use modern build systems for software

Build Tools

1. **Motivation**
2. Key Concepts
3. Tools
4. Exercise
5. Discussion

Software Complexity

- Software moved from a monolithic implementation to a highly dynamic and modular implementation
- Software grew in size and complexity
  - Different languages
  - Different stages of file transformation
  - Many dependencies to keep track
- Portability of the software became a problem
- Building of the software also got complex
  - Shell scripts got extremely fragile

A tool was needed

- Quality Assurance became a necessity
  - Such that every build was reproducible on different platforms
- Portability between different platforms
  - Let the tool master the environment, instead of the developer
- Incremental builds to speed up the development time
  - Ensuring the build produces the same results as a full clean build

Build Tools

1. Motivation
2. **Key Concepts**
3. Tools
4. Exercise
5. Discussion

File Transformation

- The building of software consists of transforming files from one format to another

  hello_world.c ➔ hello_world

- Must instruct the appropriate tool to perform the transformation

  cc –c –o hello_world.o hello_world.c

- Must check that all dependencies of the target have been also successfully transformed

  cc –o hello_world hello_world.o

Dependency Tracking

- Files may depend on other files
- For a successful transformation, the dependencies are required to be available and transformed
- Usually done by a time stamp on the file
- Before we compile hello_world, we must first check that hello_world.o is newer than our source file, hello_world.c
- If it is not, we must recompile hello_world.o

External Instrumentation

- It is the purpose of the build tool to instruct external tools in the appropriate order to perform the file transformations
- However, build tools may not know all the possible tools it may be used with
- Most build tools use either a shell scripting language or allow for a plug-in framework
- Tools contain some predefined rules for common tasks, like compilation
  - Ant and Java
  - Make and Fortran, C

Build Tools

1. Motivation
2. Key Concepts
3. **Tools**
4. Exercise
5. Discussion

General Idea

- Allow the developer to define targets, dependencies, and rules.
- Dependencies may either be files, or other targets
- When a target is performed, then the dependencies are checked to ensure a correct transformation
- Some tools support incremental builds
  - If the dependencies have not changed since the last transformation, then we do not need to re-transform them

Make

- Created by Stuart Feldman in 1977 at Bell Labs
- 2003, he received the ACM Software System Award for the Make tool
- Arguably, the single most important step in the direction of modern build environments
- Still the de-facto standard for software builds on Unix systems
- Easy to compile and install Unix based applications
  - make
  - make install

Modern Versions

- BSD Make
  - Derived from Adam de Boor's work on a version of make capable of building targets in parallel.
  - Includes conditionals and iterative loops applied to parsing stage.
  - Generation of targets at runtime
- GNU Make
  - Used in GNU/Linux installations
  - Allows for pattern-matching in dependency graphs and build targets
  - Heavy use of external macros

Make example

```
helloworld: helloworld.o
  $(CC) $(CPPFLAGS) $(LDFLAGS) –o $@ $<

helloworld.o: helloword.c
  $(CC) –c $(CPPFLAGS) $(CFLAGS) –o $@ $<

.PHONY: clean
clean:
  $(RM) helloworld helloworld.o
```

Extensions of Make

- Make files also have machine dependencies
  - Compiler options, alternate command names
- This became extremely hard for developers to support various platforms
- IMake
  - Generates Make files from templates and macro functions
- GNU Automake
  - Successor of IMake
  - Generates Make files from a higher level language
  - Dynamic dependency tracking

Apache Ant

- Similar as Make, but Java language specific
- Uses an XML file for build description
- Primary goal was to solve Make's portability problem
  - Make targets depend on a Unix shell
  - Ant's built-in functionality will usually behave identical on all platforms

Apache Ant

- Created by James Duncan Davidson from Sun Microsystems
- Officially released as a stand alone tool July 2000
- Easy to integrate JUnit tests and other external processes
- However, Ant has limitations
  - XML format does not allow for complex build tasks, requires a Java plug-in that encodes the task
  - Limited fault handling rules
  - Developer must make explicit incremental builds

Ant Example

```
<project default="compile">
<target name="compile">
    <mkdir dir="build" />
    <javac srcdir="source" destdir="build"/>
   </target>
   <target name="package" depends="compile">
    <jar jarfile="helloworld.jar" basedir="build"/>
   </target>
    <target name = "clean">
    <delete dir="build"/>
   </target>
</project>
```

Apache Maven

- Dependencies are hard to maintain in Java programs
- Apache ant could not address the problem
- Maven was created with dependency control as the prominent feature
- Uses concept of a Project Object Model
  - Description of software project and external dependencies
  - External dependencies not available, will be downloaded automatically

Maven Example

- Maven 2 will be recommended in this course
- A simple hello world example

  mvn archetype:create \

  -DgroupId=ch.ethz.HelloWorld \

  -DartifactId=hello-world
- And Maven creates a skeleton for our project
  - Project Object Model file
  - Main source directory
  - Test source directory
- Adheres to best practices

Extensions of Apache Maven

- Maven has a rich plug-in framework
- Default installation includes many powerful plug-ins
  - Build statistics exported to a website
  - Integration with continuous testing
  - Deployment of web applications through .war files
  - SCM integration for deployment of systems
  - Software metrics
  - TODO list generation
  - Export to Netbeans or Eclipse
  - And many, many more

Build Tools

1. Motivation
2. Key Concepts
3. Tools
4. **Exercise**
5. Discussion

Ant Exercise

- Still the de facto for industrial builds
- After the initial set up of Ant we will
  - Set some standard project properties
  - Configure the appropriate class path
  - Write the targets to make and deploy our system
  - Produce javadoc from our source code

Maven Exercise

- For this course Maven is a good choice building our project
- From the initial setup of Maven we will
  - Set some standard tags in the POM file
  - Add our dependencies to our project
  - Make and deploy our project
  - And if time permits, use our maven build system within the Netbeans platform

Build Tools

1. Motivation
2. Key Concepts
3. Tools
4. Exercise
5. **Discussion**

Concluding Remarks

- Necessary to include build tools in Configuration Management
  - Automake 1.4 does not produce the same results as 1.9
- Change of machine dependencies will usually not trigger a re-transformation
  - May lead to deployment of software with the wrong compile flags
- Change of machine dependencies may also produce different results
  - Compilation on Java 1.5 may produce different run-time behaviors than Java 1.4

References

- Make
  - http://www.gnu.org/
  - http://www.bsd.org
- Ant
  - http://ant.apache.org
- Maven
  - http://maven.apache.org
  - http://codehaus.org