Build automation

CSE260, Computer Science B: Honors

Stony Brook University

http://www.cs.stonybrook.edu/~cse260

Build Automation

- *Build automation* is the act of scripting or automating a wide variety of tasks that software developers do in their day-to-day activities. Includes tasks to:
  - compile computer source code into binary code
  - package binary code
  - check-out from version control
  - run automated tests
  - deploy to production systems
  - create documentation and/or release notes

make, GNU **make**, **nmake**

- **make** is a classic Unix build tool created by Stuart Feldman in April 1976 at Bell Labs (2003 ACM Software System Award for **make**)
- GNU **make** is the standard implementation of **make** for Linux and OS X
- Microsoft **nmake**, a command-line tool which normally is part of Visual Studio

`make` and GNU **`make`**

- **`make`** is typically used to build executable programs and libraries from source code:

  `make [TARGET ...]`

  - **`make`** searches the current directory for the makefile to use: **`GNUmakefile`**, **`makefile`**, **`Makefile`**
  - without arguments, **`make`** builds the first target that appears in its makefile, which is traditionally a symbolic "phony" target named **`all`**

`make` and GNU `make`

- A makefile consists of *rules*.
  - E.g., GNU Make syntax:

```
target : prerequisites ; command
```

  - For example:

```
hello: ; @echo "hello"
```

- A makefile can also contain definitions of macros
  - usually referred to as *variables* when they hold simple string definitions:

```
CC = clang
```

  - A macro is used by expanding it: **`$()`** or **`${}`**

```
NEW_MACRO = $(CC)
```

- Line continuation is indicated with a backslash **** character at the end of a line

```
target: component \
           component
```

`make` and GNU `make`

- Macros can be composed of shell commands by using the command substitution operator `'`:

```
YYYYMMDD = 'date'
```

  - Lazy evaluation: macros are normally expanded only when their expansions are **actually** required:

```
PACKAGE    = package
VERSION    = 'date +"%Y.%m%d"'
ARCHIVE    = $(PACKAGE)-$(VERSION)
dist:
    # Notice that only now macros are expanded
    #   for shell to interpret:
    # tar -cf package-'date +"%Y%m%d"'.tar
    tar -cf $(ARCHIVE).tar
```

make and GNU **make**

- Overriding macros on the command line:

```
make [MACRO="value" ...] [TARGET ...]
```

- Suffix rules also have "*file targets*" with names in the form **.FROM.TO** and are used to launch actions based on file extension: the internal macro **$<** refers to the first prerequisite and **$@** refers to the target
  - For example, convert **any HTML file** to **txt**:

```
.SUFFIXES: .txt .html
     # From .html to .txt
     .html.txt:
          lynx -dump $<    >    $@
```

- Another way is to use pattern rules:

```
%.txt : %.html
          lynx -dump $< > $@
```

Expanded example 1

```
PACKAGE      = package
VERSION      = ` date "+%Y.%m%d%" `
RELEASE_DIR  = ..
RELEASE_FILE = $(PACKAGE)-$(VERSION)
# Notice that the variable LOGNAME comes from the environment in
# POSIX shells.
# target: all - Default target. Does nothing.
all:
	echo "Hello $(LOGNAME), nothing to do by default"
	 # sometimes: echo "Hello ${LOGNAME}, nothing to do by default"
	echo "Try 'make help'"
# target: help - Display callable targets.
help:
	egrep "^# target:" [Mm]akefile
# target: list - List source files
list:
	# Won't work. Each command is in separate shell
	cd src
	ls
	# Correct, continuation of the same shell
	cd src; \
	ls
# target: dist - Make a release.
dist:
	tar -cf  $(RELEASE_DIR)/$(RELEASE_FILE) && \
	gzip -9  $(RELEASE_DIR)/$(RELEASE_FILE).tar
```

Example 2

main.cpp

```
#include <iostream.h>
#include "functions.h"
int main(){
    print_hello();
    cout << endl;
    cout << "The factorial of 5 is " << factorial(5) << endl;
    return 0;
}
```

hello.cpp

```
#include <iostream.h>
#include "functions.h"
void print_hello(){
   cout << "Hello World!";
}
```

factorial.cpp

```
#include "functions.h"
int factorial(int n){
    if(n!=1){
        return(n * factorial(n-1));
    }
    else return 1;
}
```

functions.h

```
void print_hello();
int factorial(int n);
```

Example 2

Obtain an executable

```
g++ main.cpp hello.cpp factorial.cpp -o hello
```

A basic Makefile

```
all:
	g++ main.cpp hello.cpp factorial.cpp -o hello
```

make -f Makefile

More: Using dependencies

```
all: hello

hello: main.o factorial.o hello.o
	g++ main.o factorial.o hello.o -o hello

main.o: main.cpp
	g++ -c main.cpp

factorial.o: factorial.cpp
	g++ -c factorial.cpp

hello.o: hello.cpp
	g++ -c hello.cpp

ean:
	rm -rf *o hello
```

Using variables and comments

```
# A comment: the variable CC will be the compiler to use.
CC=g++

CFLAGS=-c -Wall

all: hello

hello: main.o factorial.o hello.o
	$(CC) main.o factorial.o hello.o -o hello

main.o: main.cpp
	$(CC) $(CFLAGS) main.cpp

factorial.o: factorial.cpp
	$(CC) $(CFLAGS) factorial.cpp

hello.o: hello.cpp
	$(CC) $(CFLAGS) hello.cpp

clean:
	rm -rf *o hello
```

Example 3

```
PROGRAM = foo
C_FILES := $(wildcard *.c)
OBJS := $(patsubst %.c, %.o, $(C_FILES))
CC = cc
CFLAGS = -Wall -pedantic
LDFLAGS =
all: $(PROGRAM)
$(PROGRAM): .depend $(OBJS)
    $(CC) $(CFLAGS) $(OBJS) $(LDFLAGS) -o $(PROGRAM)
depend: .depend
.depend: cmd = gcc -MM -MF depend $(var); cat depend >> .depend;
.depend:
    @echo "Generating dependencies..."
    @$(foreach var, $(C_FILES), $(cmd))
    @rm -f depend
-include .depend
# These are the pattern matching rules. In addition to the automatic
# variables used here, the variable $* that matches whatever % stands for
# can be useful in special cases.
%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@
%: %.c
    $(CC) $(CFLAGS) -o $@ $<
clean:
    rm -f .depend *.o
```

configure script

- Configure script is an executable script designed to aid in developing a program to be run on a wide number of different computers
- It matches the libraries on the user's computer (i.e., the operating system), with those required by the program, just before compiling it from its source code
- Example usage:

```
./configure
make
make install
```

  - Other:

```
./configure --help
./configure --libs="-lmpfr -lgmp"
./configure --prefix=/home/myname/apps
```

GNU build system (Autotools)

- A suite of programming tools designed to assist in making source code packages portable to many Unix-like systems.
- Parts: Autoconf, Autoheader, Automake, Libtool.
- It is part of GNU toolchain:
  - GNU make: Automation tool for compilation and build;
  - GNU Compiler Collection (GCC): Suite of compilers for several programming languages;
  - GNU Binutils: Suite of tools including linker, assembler and other tools;
  - GNU Bison: Parser generator
  - GNU m4: m4 macro processor
  - GNU Debugger (GDB): Code debugging tool
  - GNU build system (autotools)

GNU build system (Autotools)

- Autoconf generates a configure script based on the contents of a **`configure.ac`** file in GNU **`m4`** macro preprocessor
  - https://www.gnu.org/software/autoconf/
  - Example **`configure.ac`**:

```
AC_INIT(myconfig, version-0.1)
AC_MSG_NOTICE([Hello, world.])
```

  - Now do:

```
autoconf configure.ac > configure
chmod +x configure
./configure
```

  - and you get:

```
configure: Hello, world.
```

http://www.edwardrosten.com/code/autoconf/

GNU build system (Autotools)

```
AC_INIT(myconfig, version-0.1)
echo "Testing for a C compiler"
AC_PROG_CC
echo "Testing for a C++ compiler"
AC_PROG_CXX
echo "Testing for a FORTRAN compiler"
AC_PROG_F77
AC_LANG(C++)
AC_CHECK_LIB(m, cos)
```

Apache Ant

- Apache Ant is a popular for Java platform development and uses an XML file format: by default the XML file is named build.xml

```
<?xml version="1.0"?>
<project name="Hello" default="compile">
    <target name="clean" description="remove intermediate files">
        <delete dir="classes"/>
    </target>
    <target name="clobber" depends="clean" description="remove all artifact files">
        <delete file="hello.jar"/>
    </target>
    <target name="compile" description="compile the Java source code to class files">
        <mkdir dir="classes"/>
        <javac srcdir="." destdir="classes"/>
    </target>
    <target name="jar" depends="compile" description="create a Jar file for the application">
        <jar destfile="hello.jar">
            <fileset dir="classes" includes="**/*.class"/>
            <manifest>
                <attribute name="Main-Class" value="HelloProgram"/>
            </manifest>
        </jar>
    </target>
</project>
```

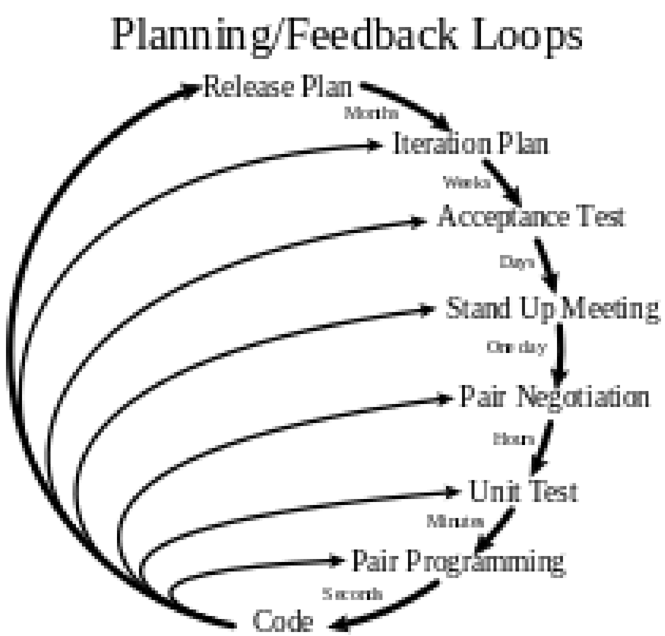
Apache Maven

- A build automation tool used primarily for Java projects, but also other languages: C#, Ruby, Scala, and other languages.
- Maven projects are configured using a Project Object Model, which is stored in a pom.xml-file:

```
<project>
  <!-- model version is always 4.0.0 for Maven 2.x POMs -->
  <modelVersion>4.0.0</modelVersion>
  <!-- project coordinates, i.e. a group of values which uniquely identify this project -->
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1.0</version>
  <!-- library dependencies -->
  <dependencies>
    <dependency>
      <!-- coordinates of the required library -->
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <!-- this dependency is only used for running and compiling tests -->
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

- Then the command: mvn package

Extreme programming (XP)



Planning and feedback loops in extreme programming.

Responsiveness to changing customer requirements
Advocates frequent "releases" in short development cycles.

Agile software development

- The *Agile Manifesto*
  - promotes adaptive planning, evolutionary development, early delivery, continuous improvement and encourages rapid and flexible response to change.

1. Customer satisfaction by rapid delivery of useful software
2. Welcome changing requirements, even late in development
3. Working software is delivered frequently (weeks rather than months)
4. Close, daily cooperation between business people and developers
5. Projects are built around motivated individuals, who should be trusted
6. Face-to-face conversation is the best form of communication (co-location)
7. Working software is the principal measure of progress
8. Sustainable development, able to maintain a constant pace
9. Continuous attention to technical excellence and good design
10. Simplicity—the art of maximizing the amount of work not done—is essential
11. Self-organizing teams
12. Regular adaptation to changing circumstances