

# File Dependencies: make

Steven J Zeil

February 21, 2013

## Contents

<b>1</b>	<b>The make Command</b>	<b>3</b>
<b>2</b>	<b>makefiles</b>	<b>5</b>
2.1	Rules . . . . .	5
2.2	Variables . . . . .	9
2.3	Implicit Rules and Patterns . . . . .	12
<b>3</b>	<b>Working with Make</b>	<b>15</b>
3.1	Touching Files . . . . .	15
3.2	Artificial Targets . . . . .	19

3.3	Dependency Analysis . . . . .	21
3.4	Managing Subproject Builds . . . . .	24
<b>4</b>	<b>Case Studies:</b>	<b>24</b>
4.1	C++ Spreadsheet Model . . . . .	24
4.2	Assignments . . . . .	29



## make

**make** is a command/program that enacts builds according to a dependency graph expressed in a *makefile*.

- **make** devised by Dr. Stuart Feldman of Bell Labs in 1977
- Long a standard component of \*nix systems
  - GNU make is a popular modern variant

.....

## 1 The make Command

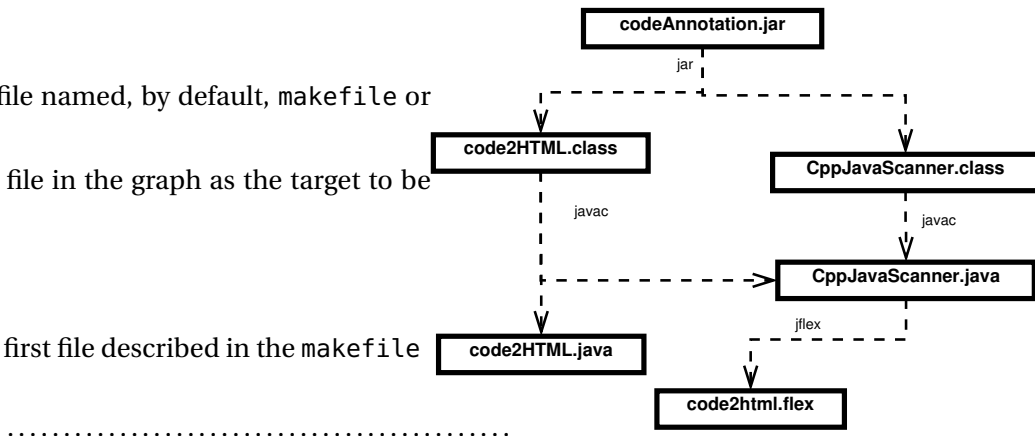
### make



- **make** looks for its instructions in a file named, by default, `makefile` or `Makefile`
- The **make** command can name any file in the graph as the target to be built, e.g.,

```
make CppJavaScanner.class
```

- If no target is given, **make** builds the first file described in the `makefile`



## make Options

Some useful options:

- n Print the commands that **make** would issue to rebuild the target, but don't actually perform the commands.
- k "Keep going." Don't stop the build at the first failure, but continue building any required targets that do not depend on the one whose construction has failed.
- f *filename* Use *filename* instead of the default `makefile`/`Makefile`



.....

## 2 makefiles

### makefiles

At its heart, a makefile is a collection of rules.

.....

### 2.1 Rules

#### Rules

- A *rule* describes how to build a single file of the project.

Each rule indicates

- The *target* file to be constructed
- The *dependencies*: the other files in this project from which the target is constructed.
- The *commands* that must be executed to construct the target from its dependencies.

- Rules may appear in any order

- Except that the first rule's target is the default built by **make** when no explicit target is specified in the command line.



.....

## Rules (cont.)

- A rule has the form

```
target: dependencies  
        commands
```

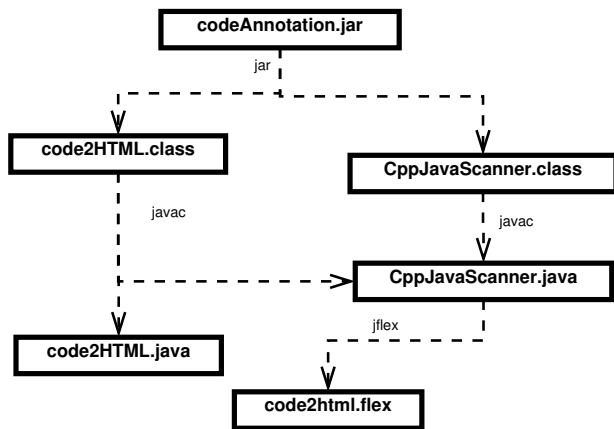
where

- *target* is the target file,
- *dependencies* is a space-separated list of files on which the target is dependent
- *commands* is a set of zero or more commands, one per line, each preceded by a Tab character.

.....

## Rule Examples





codeAnnotation.jar: code2HTML.class CppJavaScanner.class  
jar tvf codeAnnotation.jar code2HTML.class CppJavaScanner.class

CppJavaScanner.class: CppJavaScanner.java  
javac CppJavaScanner.java

code2HTML.class: code2HTML.java CppJavaScanner.java  
javac code2HTML.java

```
CppJavaScanner.java: code2html.flex  
    java -cp JFlex.jar JFlex.Main code2html.flex
```

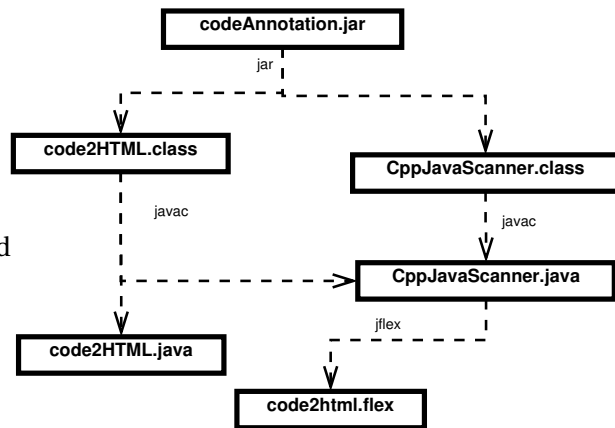
.....

## Why is This Better than Scripting?

Suppose that we edit code2html.java and then invoke **make**

- Only one **javac** will be issued, after which the **jar** command is run.
- **make** has determined the minimum number of steps required to rebuild after a change.

.....



## How make Works

- Construct the dependency graph from the target and dependency entries in the makefile





- Do a topological sort to determine an order in which to construct targets.
- For each target visited, invoke the commands if the target file does not exist or if any dependency file is newer
  - Relies on file modification dates

.....

## 2.2 Variables

### Variables

A makefile can use variables to simplify the rules or to add flexibility in configuring the makefile.

- All variables hold strings.
- Variables are initialized by a simple assignment

*variable = value*

- Variables are immutable (constants)
- Assignments may appear within the makefile or in the command line, e.g.:

make JOPTIONS=-g codeAnnotation.jar

.....



## Referencing Variables

- Variables are referenced as `$(variable)` or `${variable}`, e.g.,

```
CppJavaScanner.class: CppJavaScanner.java
javac $(JOPTIONS) CppJavaScanner.java
```

```
code2HTML.class: code2HTML.java CppJavaScanner.java
javac $(JOPTIONS) code2HTML.java
```

.....

## Adding Power to Variables

GNU make adds some special extensions useful in setting up variables.

- Globbing:

```
SOURCEFILES=$(wildcard src/*.cpp)
```

collects a list of all C++ compilation units in the `filenamesrc` directory

- Substitutions:



```
OBJFILES=$(SOURCEFILES:%.cpp=%.o)
```

collects a list of all object code files expected by compiling those compilation units.

.....

### Example: Using variables

This allows us to write a “generic” rule for compiling C++ programs:

```
PROGRAM=myProgramName
SOURCEFILES=$(wildcard src/*.cpp)
OBJFILES=$(SOURCEFILES:%.cpp=%.o)

$(PROGRAM): $(OBJFILES)
    g++ -o $(PROGRAM) $(OBJFILES)
```

- This is technically, incomplete.
  - We have not explained how to produce a .o file from a .cpp
- Nonetheless, it would work on some systems for the initial build, because they have an “implicit” rule for working with C++

- Still not a good solution by itself - dependencies on .h files have not been captured.

.....

## 2.3 Implicit Rules and Patterns

### Implicit Rules and Patterns

- Implicit rules describe how to produce a single “kind” (extension) of file from another.
  - All make implementations will have some common implicit rules.
  - You can modify the list of implicit rules.
- Pattern rules are a GNU extension for writing “generic” rules
  - Implicit rules could, for the most part, be written as patterns
  - But patterns offer some additional flexibility

.....



## Implicit Rules

An implicit rule looks like

```
.ext_1.ext_2:  
    commands
```

where *ext<sub>1</sub>* and *ext<sub>2</sub>* are file extensions, and *commands* are the commands used to convert a file with the first extension into a file with the second.

Example:

```
.cpp.o:  
    g++ -g -c $<
```

- the *implicit variable* \$< holds the dependency file
- Also commonly used, \$@ denotes the target file.

.....

## Using Implicit Rules

The extensions used in implicit rules must be declared:

```
.SUFFIXES: .cpp .o
```



An implicit rule will be used when a target ends in one of these suffixes and

- there is no rule listing that file as a target, or
- the rule listing that file as a target has no commands

.....

### Implicit Rule Example

```
PROGRAM=myProgramName
SOURCEFILES=src/main.cpp src/adt.cpp
OBJFILES=$(SOURCEFILES:%.cpp=%.o)
.SUFFIXES: .cpp .o

.cpp.o:
    g++ -g -c $<

$(PROGRAM): $(OBJFILES)
    g++ -o $(PROGRAM) $(OBJFILES)

src/adt.o: adt.cpp adt.h
```



- Both `main.cpp` and `adt.cpp` will be compiled on the initial build.
- If `adt.h` is subsequently modified, then `adt.cpp` would be re-compiled.

.....

### Pattern Rules

A pattern rule looks like a regular rule, but uses '%' as a wildcard in the target and one of their dependencies:

```
src/test/java/%.class: src/test/java/%.java junit4.jar
    javac -cp junit4.jar -g src/test/java/%.java
```

- Another implicit variable, `$(*)` contains the string matched by the `%` wildcard.
- One advantage of pattern rules, is that we can add dependencies on other files e.g., `junit.jar`

.....

## 3 Working with Make

### 3.1 Touching Files

#### Modification Dates

`make`



- compares the modification dates of targets and dependencies to determine if the target is out of date.
- uses the success/fail status value returned by commands to determine if construction of a target was successful.

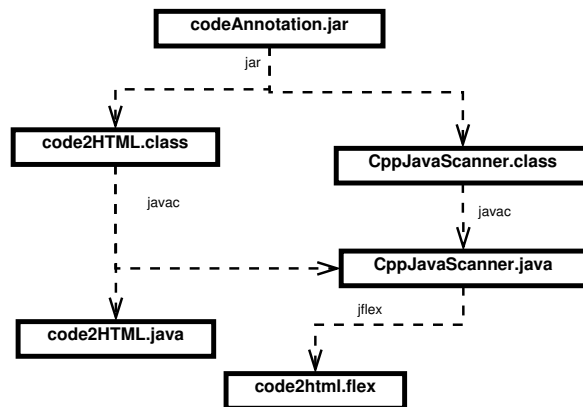
Although this is fairly robust, there are ways to fool **make**

.....

### Touching a File

- The **touch** command in \*nix sets a file's modification date to the current time, without affecting the contents of the file.

What would happen if we touched `code2html.flex`?  
Sometimes this is a useful thing to do on purpose.





.....

### Inadvertant Touches

Suppose we had our code annotation project in a directory `project1` and did the following:

```
> cd project1
> make
> cd ..
> cp -rf project1 project2
> cd project2
> make
```

What would be re-built by the second **make**?

- Almost impossible to tell. All of the files in `project2` would have create/modify dates within a second of each other. Ordering, if any, would be arbitrary.
  - (better to have done `cp -arf project1 project2`)

.....

### Inadvertant Touches

Suppose we had our code annotation project in a directory `project1` and did the following:



```
> cd project1
> make
> cd ..
> cp -rf project1 project2
> cd project2
> make
```

- Similarly, successive calls to **make** can sometimes be confused if the time between creation of some intermediate targets is within a single clock “tick”
- Clock drift between different machines (particular a command server and a file server) can be particularly troublesome.

.....

### Created != Success

- Some commands we might give to create a target will create no file if the ocmmand fails.
  - e.g., **g++** does not create a **.o** file if compilation errors occur
- Others will create some kind of file anyway.
  - E.g., any command that is invoked with output redirection,



command > target

- which could cause **make** to assume that the target need not be re-constructed the next time around.
  - \* Some **make** programs explicitly delete targets if the command fails.

.....

## 3.2 Artificial Targets

### Fooling make Again

A creative way to fool **make**:

What happens if we give a rule whose commands never actually create the target?

```
target: dependency1 dependency2
    echo Nope. Not going make that target!
```

- The first time we run **make**, the dependencies will be created and the **echo** performed.
- Each subsequent time we run **make**, the dependencies will be re-created if necessary and the **echo** performed.

.....

## Artificial Targets

We can take advantage of this trick by adding *artificial targets* that serve as the names for tasks to be performed.

```
build: codeAnnotation.jar
```

```
install: build
        cp codeAnnotation.jar $(INSTALLDIR)
```

```
clean:
        rm *.class CppJavaScanner.java
```

```
codeAnnotation.jar: code2HTML.class CppJavaScanner.class
        jar tvf codeAnnotation.jar code2HTML.class CppJavaScanner.class
```

```
CppJavaScanner.class: CppJavaScanner.java
        javac CppJavaScanner.java
```

```
code2HTML.class: code2HTML.java CppJavaScanner.java
        javac code2HTML.java
```

```
CppJavaScanner.java: code2html.flex
        java -cp JFlex.jar JFlex.Main code2html.flex
```



.....

### Common Artificial Targets

**all** Often made the first rule in the makefile so that it becomes the default. Builds everything. May also run tests.

**build** Build everything.

**install** Build, then install

**test** Build, then run tests

**clean** Delete everything that would have been produced by the makefile in a `build` or `test` run.

.....

## 3.3 Dependency Analysis

### Dependency Analysis

Coming up with a list of dependencies (and keeping it current) can be troublesome.

- Various tools exist for this purpose for programming languages



- The **gcc** and **g++** compilers have a compile-time option, `-MMd`, which emits a `.d` file containing a target and dependency line.
  - Use this with an implicit rule to give the actual command

.....

### Self-Building Makefile

```
MAINPROG=testpicture
CPPS:=$(wildcard *.cpp)
%
CPPFLAGS=-g -D$(DISTR)
CPP=g++
%
OBSJ=$(CPPS:%.cpp=%.o)
DEPENDENCIES = $(CPPS:%.cpp=%.d)

%.d: %.cpp
    touch $@

%.o: %.cpp
```



```
$(CPP) $(CPPFLAGS) -MMD -o $@ -c $*.cpp  
make.dep: $(DEPENDENCIES)  
-cat $(DEPENDENCIES) > $@  
include make.dep
```

- On the first **make**,
  - for each `.cpp` file, an empty `.d` file is created by **touch**
  - All `*.d` files are concatenated to form a file `make.dep`
  - The file `make.dep` is included as part of the makefile.
  - As the `.cpp` files are compiled, the `.d` files are replaced by a rule making the `.o` file dependent on that `.cpp` file and on any `.h` files that it included.
- On subsequent **make** runs,
  - the `.d` files contain the dependencies for each `.cpp` file.
  - All `*.d` files are concatenated to form a file `make.dep`
  - The file `make.dep` is included as part of the makefile.
  - If any `.h` or `.cpp` file has been changed, the `.o` files dependent on it will be regenerated.

.....



## 3.4 Managing Subproject Builds

### Managing Subproject Builds

Subprojects are generally handled by giving each subproject its own makefile and using a master makefile to invoke the artificial targets:

```
all:
```

```
    cd model; make
    cd vncurses; make
    cd vcjava; make
```

```
clean:
```

```
    cd model; make clean
    cd vncurses; make clean
    cd vcjava; make clean
```

.....

## 4 Case Studies:

### 4.1 C++ Spreadsheet Model

#### C++ Spreadsheet Model





```
MAINPROG=testsheet
CPPS=exprparser.cpp expr.cpp lexical.cpp exprfactory.cpp
      expression.cpp cellrange.cpp clipboard.cpp
      cellname.cpp numericnode.cpp stringnode.cpp cellrefnode.cpp
      negatenode.cpp
      plusnode.cpp
      subtractnode.cpp timesnode.cpp dividesnode.cpp ifnode.cpp
      numvalue.cpp strvalue.cpp errvalue.cpp spreadsheet.cpp cell.cpp
      observable.cpp
      timenode.cpp timevalue.cpp
DIR=${PWD}
ASST=$(notdir ${DIR})
DISTR=Unix
EXE=.exe
LFLAGS=-L. -lsheet
#
#
#####
# Macro definitions for "standard" C and C++ compilations
#
CPPFLAGS=-g -fPIC -Wall -c
```



```
CFLAGS=-g
TARGET=libssheet.a
LINK=g++ $(LFLAGS)
#
CC=gcc
CPP=g++
#
#
# In most cases, you should not change anything below this line.
#
# The following is "boilerplate" to set up the standard compilation
# commands:
#

OBSJ=$(CPPS:%.cpp=%.o)
DEPENDENCIES = $(CPPS:%.cpp=%.d)

%.d: %.cpp
```



```
touch $@

%.o: %.cpp
$(CPP) $(CPPFLAGS) -MMD -o $@ -c $*.cpp

#
# Targets:
#
all: $(TARGET) testssheet${EXE}

$(TARGET): $(OBJJS)
    -rm $@
    ar -cvq $@ ${OBJJS}

# g++ -shared -Wl,-soname,$@ -o $@ ${OBJJS}

tests: testcellname.out testssheet.out
```



```
testssheet.out: testssheet${EXE}
    ./testssheet${EXE} ss1.dat > testssheet.out
    ./testssheet${EXE} ss2.dat >> testssheet.out
    ./testssheet${EXE} ss3.dat >> testssheet.out
    ./testssheet${EXE} ss4.dat >> testssheet.out
    ./testssheet${EXE} ss5.dat >> testssheet.out
    ./testssheet${EXE} ss6.dat >> testssheet.out
    diff testssheet.out testsheet.expected
```

```
test%.out: test%$(EXE)
    ./test%${EXE} < test%.dat > test%.out
    diff test%.out test%.expected
```

```
test%$(EXE): test%.o $(TARGET) unittest.o
    g++ -o $@ test%.o unittest.o -L. -lssheet
```



```
clean:
    /bin/rm -f *.d *.a *.o *.exe $(TARGET)

lexical.cpp: lexical.l
    flex -olexical.cpp lexical.l

expr.cpp: expr.y
    bison -d -o expr.cpp expr.y

make.dep: $(DEPENDENCIES)
    -cat $(DEPENDENCIES) > $@

include make.dep
```

.....

## 4.2 Assignments

### Assignments

Setting up an assignment for a course:



```
include ../make.base
MAINPROG=testpicture
#
include ../cppMake.head

Tests/test%.out: Tests/test%.dat Work/${MAINPROG}
    cd $(WORKDIR); /bin/sh ../Tests/test$.dat
    mv $(WORKDIR)/test$.out $@

include ../cppMake.tail
```

- Note the heavy use of included files that standardize this process for multiple assignments over the same semester

.....

### **make.base**

```
DIR=$PWD
ASST=$(notdir $DIR)
WINEXE=.exe
UNIXEXE=
ifneq (,$(findstring MinGW,$(PATH)))
```



```
DISTR=MinGW
EXE=$(WINEXE)
WORKDIR=winwork
else
DISTR=Linux
EXE=$(UNIXEXE)
WORKDIR=Work
endif
HTMLDIR=/home/zeil/cs330/webcourse/Assts/
INSTALLDIR=/home/zeil/cs330/Assignments/$(ASST)
```

.....

### cppmake.head

```
#
PUBLICFILESx=$(wildcard Public/*)
PUBLICFILES=$(filter-out %~, ${PUBLICFILESx})
PUBLIC=$(notdir ${PUBLICFILES})
SOLUTIONFILES=$(wildcard Solution/*.h) $(wildcard Solution/*.cpp)
TESTDATFILES=$(wildcard Tests/test*.dat)
TESTDAT=$(notdir ${TESTDATFILES})
```



```
TESTOUTFILES=$(TESTDATFILES:%.dat=%.out)
SOLUTIONTESTDAT=$(TESTDAT:%.dat=Solution/%.dat)
SOLUTIONTESTOUT=$(TESTDAT:%.dat=Solution/%.out)
INSTALLEDFILES=$(PUBLIC:%=$(INSTALLDIR)/%)
WORKMAIN=Work/$(MAINPROG)
#
all: Work/$(MAINPROG) ${TESTOUTFILES} ${SOLUTIONTESTDAT} ${SOLUTIONTESTOUT}
```

.....

### **cppmake.tail**

```
#
Solution/%.dat: Tests/%.dat
    -rm Solution/$*.dat
    cd Solution; ln ../Tests/$*.dat
Solution/%.out: Tests/%.out
    -rm Solution/$*.out
    cd Solution; ln ../Tests/$*.out
```





```
Work/$(MAINPROG): ${PUBLICFILES} ${SOLUTIONFILES}
    mkdir -p Work
    cp ${PUBLICFILES} Work
    cp ${SOLUTIONFILES} Work
    touch Work/make.dep
    cd Work; make

WinWork/$(MAINPROG).exe: ${PUBLICFILES} ${SOLUTIONFILES}
    mkdir -p WinWork
    cp ${PUBLICFILES} WinWork
    cp ${SOLUTIONFILES} WinWork
    touch WinWork/make.dep
    cd WinWork; make CPP=i586-mingw32msvc-g++ LINK=i586-mingw32msvc-g++ LFLAGS=-lm
    /bin/mv WinWork/$(MAINPROG) $@

install: ${HTMLDIR}/${ASST}.html
    ${INSTALLDIR}/bin/Linux/$(MAINPROG)
    ${INSTALLDIR}/bin/Windows/$(MAINPROG).exe
    ${INSTALLEDFILES}
    ${SOLUTIONTESTDAT} ${SOLUTIONTESTOUT}
    find ${INSTALLDIR} -type f -exec chmod 664 {} ;
```



```
find ${INSTALLDIR} -type d -exec chmod 775 {} ;  
-chmod 775 ${INSTALLDIR}/bin/*/*  
  
${HTMLEDIR}/${ASST}.html: ${ASST}.html  
cp ${ASST}.html ${HTMLEDIR}  
  
${INSTALLDIR}/bin/Linux/${MAINPROG}: Work/${MAINPROG} ${INSTALLDIR}  
mkdir -p ${INSTALLDIR}/bin/Linux  
-cp Work/${MAINPROG} ${INSTALLDIR}/bin/Linux/  
  
${INSTALLDIR}/bin/Windows/${MAINPROG}.exe: WinWork/${MAINPROG}.exe ${INSTALLDIR}  
mkdir -p ${INSTALLDIR}/bin/Windows  
-cp WinWork/${MAINPROG}.exe ${INSTALLDIR}/bin/Windows/  
  
${INSTALLDIR}/%: Public/% ${INSTALLDIR}  
cp Public/$* ${INSTALLDIR}  
  
${INSTALLDIR}:  
-mkdir -p $@
```

```
clean:
```

```
-rm -rf Work/* ${SOLUTIONTESTDAT} ${SOLUTIONTESTOUT} ${TESTOUTFILES}
```

```
cleaner:
```

```
-rm Work/* ${SOLUTIONTESTDAT} ${SOLUTIONTESTOUT} ${TESTOUTFILES}
```

```
-rm ${HTMLDIR}/${ASST}.html
```

```
-rm -rf ${INSTALLDIR}
```

.....

