CSE306 Software Quality in Practice

Dr. Carl Alphonce
alphonc@buffalo.edu
343 Davis Hall
REVIEW
Modeled development process

1. Start
2. Read & understand specifications
3. Write tests
4. Implement
5. Run tests
6. Fail
7. Pass
8. Implement
9. Return to read & understand specifications

Flow:
- Purple arrow from start to read specifications
- Green arrow from read specifications to write tests
- Yellow arrow from write tests to run tests
- Red arrow from run tests to fail
- Green arrow from fail to pass
- Yellow arrow from pass back to read specifications
Modeled development process

start

read & understand specifications

write tests

run tests

fail

implement

pass

LEX feedback question: why commit so often?
Modeled development process

- start
- read & understand specifications
- write tests
- implement
- run tests
- fail
- pass

To document your process.
MERGING IN MORE DETAIL
STARTING POINT
git checkout b
edit F2 to create conflict with F2'
git add F2''
git commit -m "..."
git checkout master
Blob created with conflict markings.
edit F''' to resolve conflict, creating F2^4
add to staging
git commit -m "..."
Question

Is there a difference between
- merging a into b
- merging b into a
Is there a difference between
- merging a into b
- merging b into a

YES!
vs
STARTING POINT
merging master into b
b moves, master does not
also:

parent order differs
The parents of a commit are ordered.

A = A^0
B = A^1 = A~1
C = A^2 = A^2
D = A^1^1 = A~2
E = B^2 = A^2^2
F = B^3 = A^3^3
G = A^1^1^1^1 = A~3
H = D^2 = B^2^2 = A^2^2^2 = A~2^2
I = F^ = B^3^ = A^3^3
J = F^2 = B^3^2 = A^3^3^2

merging b into master
HEAD^1 → C, HEAD^2 → B
merging master into b
HEAD^1 → B, HEAD^2 → C
GNU make

- Who has invoked 'make'?
- Who has modified a make file?
- Who has written a make file from scratch?
"You can use [make] to describe any task where some files must be updated automatically from others whenever the others change."

make and makefiles

- makefile contains rules that describe update dependencies
rules

target : prerequisites
recipe
rules

target : prerequisites

recipe

Must be a tab!
A target is usually the name of a file that needs to be generated/updated during the 'make' process.

The rule will be used by 'make' when the target is out-of-date, and so should say how to update the target.

A target is out-of-date if it is older than any of its prerequisite files.

For example:

```
foo.o : foo.c foo.h
```

The object file foo.o is considered out-of-date if either its .c or .h file is newer.
A target is usually the name of a file that needs to be generated/updated during the 'make' process.

The rule will be used by 'make' when the target is out-of-date, and so should say how to update the target.

"Bear in mind that make does not know anything about how the recipes work. It is up to you to supply recipes that will update the target file properly. All make does is execute the recipe you have specified when the target file needs to be updated." [p. 5]
target

gcc -c -Wall primOpt.c
A target can be "phony" - an arbitrary label for an action given by the rest of the rule.
clean:

```
rm -f primOpt.o main
```

"...the clean target will not work properly if a file named clean is ever created in this directory. Since it has no prerequisites, clean would always be considered up to date and its recipe would not be executed. To avoid this problem you can explicitly declare the target to be phony by making it a prerequisite of the special target .PHONY" [p. 29]
target

.PHONY: clean
clean:
    rm -f primOpt.o main
variables

"A variable is a name defined in a makefile to represent a string of text, called the variable’s value. These values are substituted by explicit request into targets, prerequisites, recipes, and other parts of the makefile."

[p. 59]
"A variable name may be any sequence of characters not containing `:`, `#`, `=`, or whitespace. However, variable names containing characters other than letters, numbers, and underscores should be considered carefully, as in some shells they cannot be passed through the environment to a sub-make [...]. Variable names beginning with `.` and an uppercase letter may be given special meaning in future versions of make.

Variable names are case-sensitive. The names `foo`, `FOO`, and `Foo` all refer to different variables."
variables

"To substitute a variable’s value, write a dollar sign followed by the name of the variable in parentheses or braces: either ‘$(foo)’ or ‘${foo}’ is a valid reference to the variable foo."

[p. 59]
implicit rules

\texttt{primOpt.o : primOpt.h primOpt.c}

expands to

\texttt{primOpt.o : primOpt.h primOpt.c}

\texttt{$(CC) \$(CPPFLAGS) \$(CFLAGS) -c}$

\texttt{CC has default value cc}
edit: main.o kbd.o command.o display.o \ insert.o search.o files.o utils.o
   cc -o edit main.o kbd.o command.o display.o \ insert.o search.o files.o utils.o
main.o: main.c defs.h
     cc -c main.c
kbd.o: kbd.c defs.h command.h
     cc -c kbd.c
command.o: command.c defs.h command.h
     cc -c command.c
display.o: display.c defs.h buffer.h
     cc -c display.c
insert.o: insert.c defs.h buffer.h
     cc -c insert.c
search.o: search.c defs.h buffer.h
     cc -c search.c
files.o: files.c defs.h buffer.h command.h
     cc -c files.c
utils.o: utils.c defs.h
     cc -c utils.c
clean:
     rm edit main.o kbd.o command.o display.o \ insert.o search.o files.o utils.o
objects = main.o kbd.o command.o display.o \ insert.o search.o files.o utils.o
edit : $(objects)
   cc -o edit $(objects)
main.o : main.c defs.h
   cc -c main.c
kbd.o : kbd.c defs.h command.h
   cc -c kbd.c
command.o : command.c defs.h command.h
   cc -c command.c
display.o : display.c defs.h buffer.h
   cc -c display.c
insert.o : insert.c defs.h buffer.h
   cc -c insert.c
search.o : search.c defs.h buffer.h
   cc -c search.c
files.o : files.c defs.h buffer.h command.h
   cc -c files.c
utils.o : utils.c defs.h
   cc -c utils.c
clean :
   rm edit $(objects)
objects = main.o kbd.o command.o display.o insert.o search.o files.o utils.o

edit : $(objects)
    cc -o edit $(objects)
main.o : defs.h
kbd.o : defs.h command.h
command.o : defs.h command.h
display.o : defs.h buffer.h
insert.o : defs.h buffer.h
search.o : defs.h buffer.h
files.o : defs.h buffer.h command.h
utils.o : defs.h

.PHONY : clean
clean :
    rm edit $(objects)
objects = main.o kbd.o command.o display.o \ insert.o search.o files.o utils.o

edit : $(objects)
    cc -o edit $(objects)

$(objects) : defs.h

kbd.o command.o files.o : command.h

display.o insert.o search.o files.o : buffer.h
# Variables

```bash
objects = main.o kbd.o command.o display.o \
insert.o search.o files.o utils.o
```

# Rules

```bash
edit : ${objects}
    cc -o edit ${objects}

${objects} : defs.h

kbd.o command.o files.o : command.h

display.o insert.o search.o files.o : buffer.h
```