Lecture 4: Software Tools; Find/Grep

Kenneth M. Anderson
Software Methods and Tools
CSCI 3308 - Fall Semester, 2003

Today's Lecture

- Review need for software tools
  - Brooks, chapters 1 and 12
- Discuss two specific software tools
  - find and grep

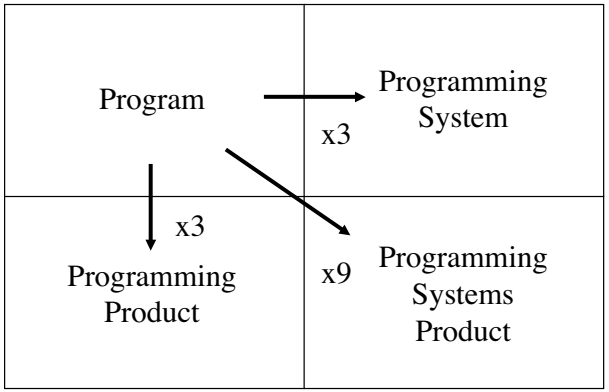
The Tar Pit

- Developing large systems is "sticky"
  - Projects emerge from the tar pit with running systems
    - But most missed goals, schedules, and budgets
    - "No one thing seems to cause the difficulty--any particular paw can be pulled away. But the accumulation of simultaneous and interacting factors brings slower and slower motion."
- CHAOS Report from Standish Group
  - 34% of (reported) software development projects hit their estimates in 2002 (up from 28% in 2001)
    - e.g. many projects fail on some project management dimension

The Tar Pit, continued

- The analogy is meant to convey that
  - It is hard to discern the nature of the problem(s) facing software development
- Brooks begins by examining the basis of software development
  - e.g. system programming

Evolution of a Program



What makes programming fun?

- Sheer joy of creation
- Pleasure of creating something useful to other people
- Creating (and solving) puzzles
- Life-Long Learning
- Working in a tractable medium
  - e.g. Software is malleable

What's not so fun about programming?

- You have to be perfect!
- You are rarely in complete control of the project
- Design is fun; debugging is just work
- Testing takes too long!
- The program may be obsolete when finished!

Software Tools

- Fred Brooks talks about the importance of software tools in Chapter 12 of his book
- Many of the concepts discussed in chapter 12 are "dated"
  - in that modern programming environments provide a standard set of tools that are used across many software projects and development organizations
- However, there are still some lessons to be learned

Lessons Learned

- A balance must be struck between general-purpose tools and customized tools for a project
  - general-purpose tools are maintained by a separate organization and are widely known
    - so you don't have to waste time maintaining such tools or training your people to use them
  - A custom tool, tailored for a specific project, can result in higher productivity, since it can automate a repetitive task that otherwise would need to be done manually

Lessons Learned, continued

- Maintenance of Program Libraries
  - (This section foreshadowed configuration management, which we will discuss later in the semester)
  - Each programmer has a separate workspace
  - Finished components are placed in a system integration library for testing
  - Tested components are incorporated into the "official" release
  - This process should be automated by tools

Lessons Learned, continued

- High-Level Programming Languages and "Interactive Programming" are listed as important tools
  - The important lesson here is that tools we take for granted today, were at one time "new" and "untested" techniques
  - Back in the 60's and 70's
    - most programming was done in assembly; lots of accidental errors occur in these languages
      - like remembering to save registers properly on a context switch
    - most programming was "batch"
      - create punch cards, submit to machine room, get results of run back the next day! (Imagine debugging a program with a 24-hour turn around time!!!)
      - back in 1975, they already had preliminary data on how much more productive interactive programming is on debugging (the data ranged from 2x to 8x better, page 136)

Find and Grep

- Two very helpful tools to have in your tool chest
  - Both help you "find" things in your Unix environment
    - find is used to search for files in the filesystem that match certain criteria
      - why is this useful? it scales to large numbers of files
        - consider having to search for a single file in a filesystem that contains thousands of files
        - find can search the filesystem much faster than you can!
    - grep is used to search for text strings within text files
      - why is this useful? same answer: it scales to large numbers of files
        - consider having to change the name of a procedure in a program with hundreds of source code files
        - grep can find each use of the procedure in seconds

Find

- "Find" searches recursively through directories testing each file against a set of operators
- These operators can compare properties such as a file's name, size, or Unix file permissions against a given query

Examples of the find command

% find ~ -name "*.c" -print

- This command searches for files that end in ".c" starting in your home directory and looking recursively in all of its subdirectories
- the -name operator uses shell **wildcard** syntax for pattern matching. However, the matching is not done by the shell,. Therefore, you must use quoting to prevent the shell from evaluating any special characters

Examples of the find command

% find ~ -size 1000c -perm 700 -print

- This command searches for files that have a file permission set to "700" and a size of 1000 characters

- % find ~ -type f -newer ~/.cshrc -print
  - Find all files (but not directories, symbolic links, etc) that have a timestamp that is newer than the user's .cshrc file

Find operators

- See the "find" man page for a complete list of operators
- Find operators are similar to expressions in an if-then-else statement.
  - Each operator returns true or false.
  - Find evaluates each operator one at a time.
  - If the operator returns true it goes on to the next operator.
  - If the operator returns false, find gives up on this file, and goes on to the next file

Example

./program1 -rwx------ steinker ta

./program2 -rwx------ steinker csci3308

% find . -name "program*" -group ta -print

This prints the file name "./program1"

How would the output change if the command is modified as follows?

%find . -name "program*" -print -group ta -print

Incidentally, -print is an operator that prints the name of the file to stdout and always returns true

More complicated expressions

- You can create more complicated boolean expressions from operators
  - op1 -a op2
    - True if op1 and op2 are true
  - op1 -o op2
    - True if op1 or op2 are true
  - !op1
    - True if op1 is false
  - (op1) Parentheses can be useful for grouping. The parentheses must be quoted to prevent the shell from evaluating them

Example

% find . ! \(-user steinker -o -group ta \) -a -name program1 -print
  - Find all files with the name "program1" that are not owned by the user "steinker" or the group "ta"
- A parenthesis must be quoted to prevent the shell from evaluating it. However, each operator must be a separate command line argument or find will not recognize them.

  % find . ! "(-user steinker -o -group ta)" -a -name …
  - this will not work, since the shell will pass "(-user steinker -o -group ta)" as a single command line argument

  % find . ! "(" -user steinker -o group ta ")" …
  - this will work, as will the first version above

User defined operators

- Find allows you to create your own operators
  - -exec prog1 {} \;
    - executes an external program
    - must be terminated with an escaped semicolon
    - may optionally pass the name of the current file
  - -ok
    - Like exec except the generated command is written to standard out, the user is then prompted; the command is executed if the user inputs "y"

Examples

% find ~/csci3308/src -name "*.c" -exec lpr -Pakira {} \;

- find all files that end in ".c" and prints them on the printer akira

% find ~ -exec test_script {} \; -print

- list all files for which test_script returns "0" as its exit code

% find ~ -name "*.bak" -ok rm {} \;

- find all files that end in ".bak", then prints a string like
  - rm /home/faculty/kena/tmp/prog1.bak
- if the user answers "y", the file is removed
- any other answer and the file is not removed

Grep

- Grep searches through the contents of a file to find lines that match a specified **regular expression**
  - grep architecture /usr/dict/words
- As with the find command, patterns with metacharacters must be quoted to protect them from being evaluated by the shell

Grep, continued

- Grep checks the pattern against each line in the file. If any part of the line matches the pattern, the whole line is printed.
- % grep architect /usr/dict/words
  - architect
  - architectonic
  - architectural
  - architecture

Grep, continued

- To match against the entire line use the ^, begin line, and $, end line, metacharacters in your pattern. You can also use the -x option.
- % grep '^architect$' /usr/dict/words
  - architect
- % grep -x architect /usr/dict/words
  - architect
- Note: the above commands work with the linux version of grep, egrep may be required on other Unix platforms
- Grep has many options for things like displaying lines that do not match, listing only the number of lines that match instead of the lines themselves, etc. For more information see the grep man page.