CSE 15L: Software Tools and Techniques Laboratory

Fall 2018 - http://ieng6.ucsd.edu/~cs15x/

GARY GILLESPIE

Lecture 17

December 5th, 2018

Announcements

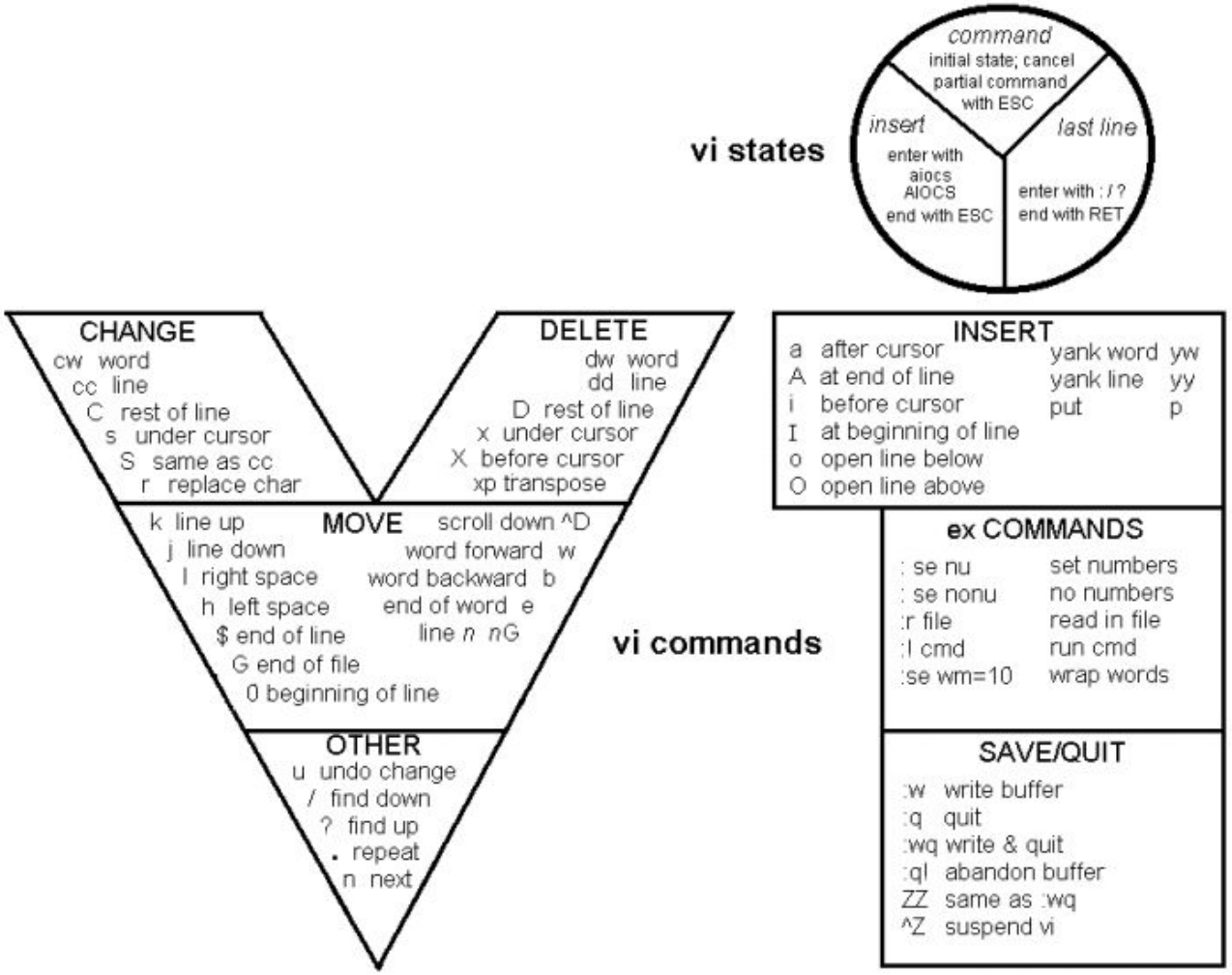
- Final Exam Review on Saturday, 2pm
  - Center 101
- Final Exam next week
  - Mon, Wed, Thurs

- Please schedule your final exam by Saturday at midnight at sections.ucsd.edu
  - Selection won't show up on TritonLink

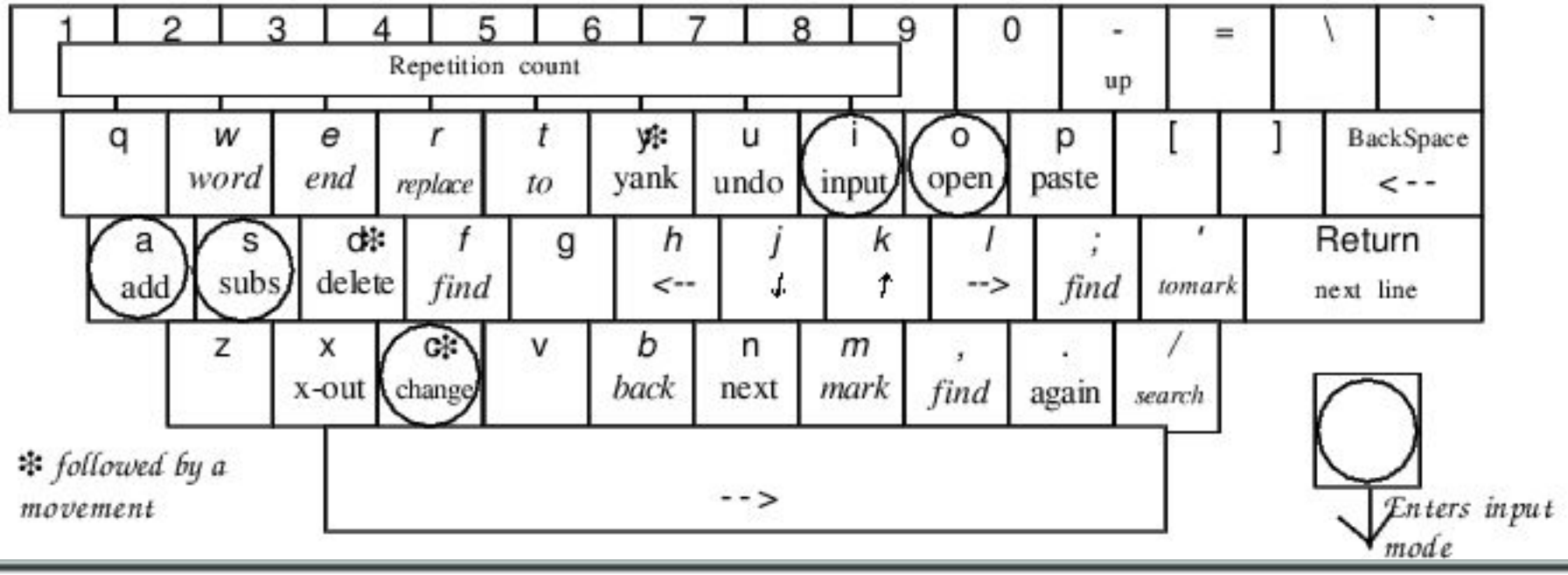
Introduction to vi/vim

Unix vi/vim Editor

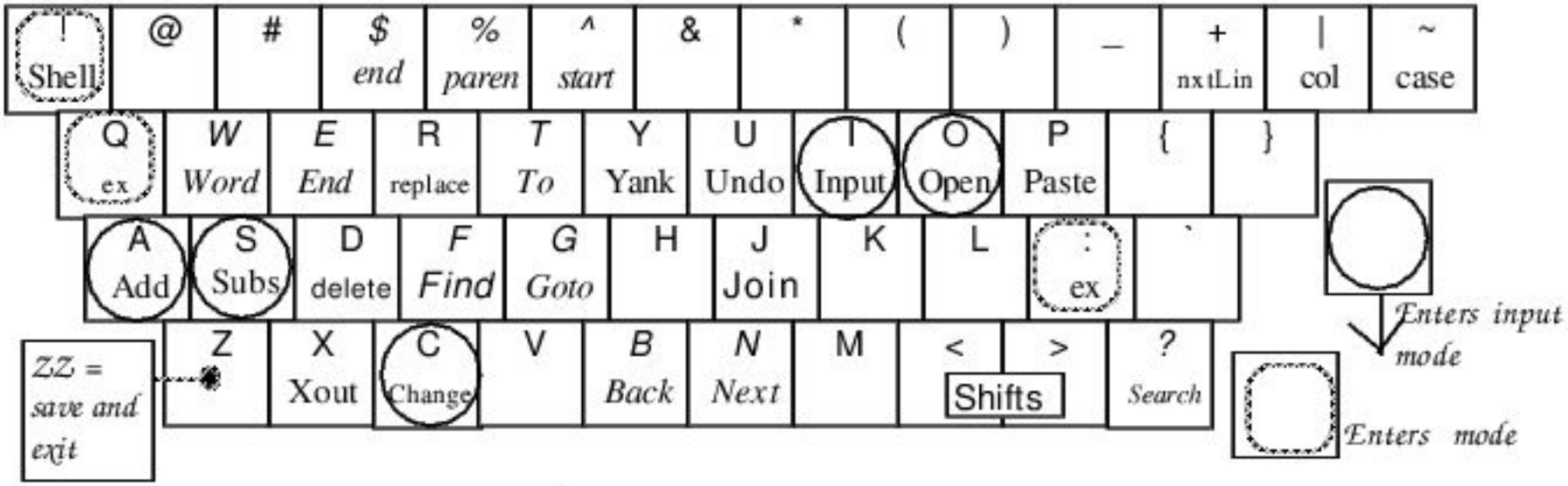
Command line full screen editor



Command Mode Keys



Command Mode Keys with Shift key down



http://www.csci.csusb.edu/dick/samples/vi.notes.html

Edit mode
General Command Syntax

The general format for commands is

ncm

Where:

- n is an optional multiplier value
- c is the command
- m is an optional scale modifier

Examples:

3dw – delete 3 words

More vi/vim

Modes

i
switch to insert mode
(so you can type)

esc
switch to command mode
(normal mode)

Saving and Quitting

:w
write
(save)

:q
quit

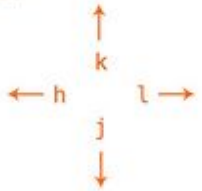
:wq
write and quit
(save and quit)

:q!
quit without saving

Getting Around

use arrow keys

OR:



/text
search for text

gg
go to the first line

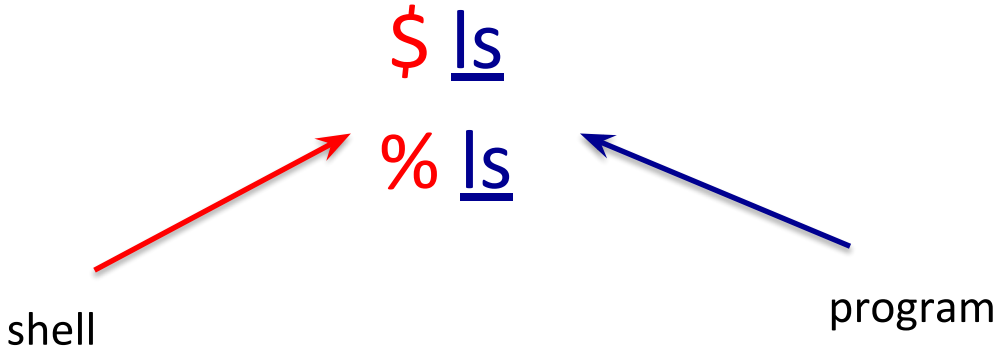
G
go to the last line

33G
go to line 33

Introduction to Unix

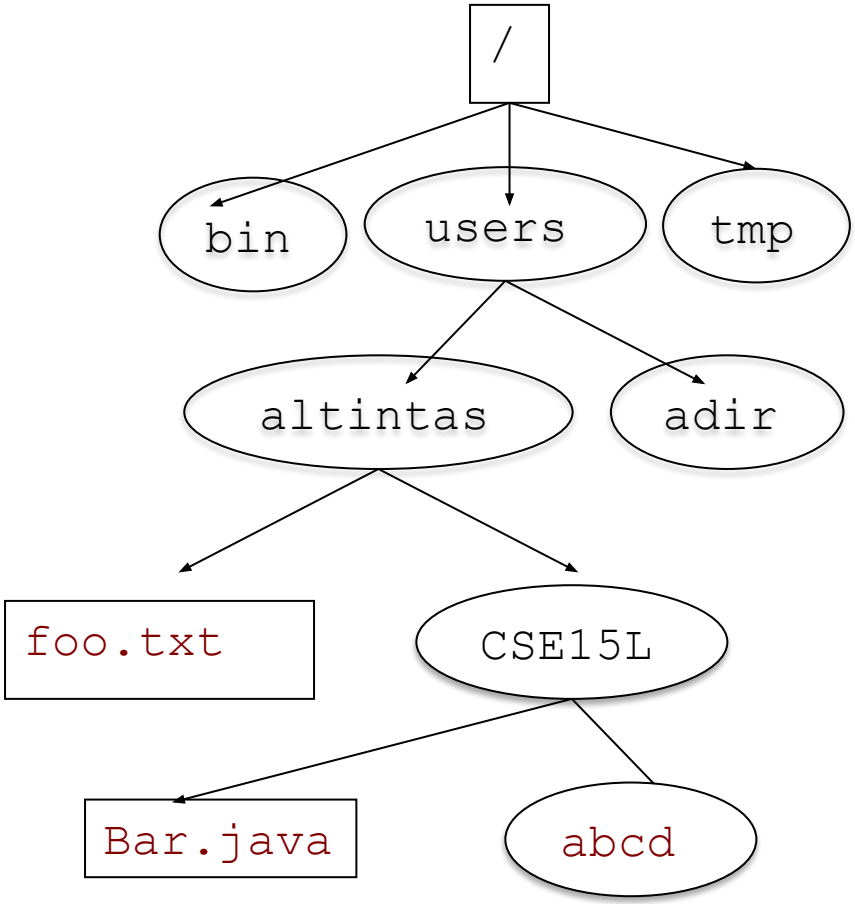
Introduction to Unix

- What is an operating system?
- Unix varieties- Solaris, Linux, MacOSX
- Three parts: kernel, shell and programs



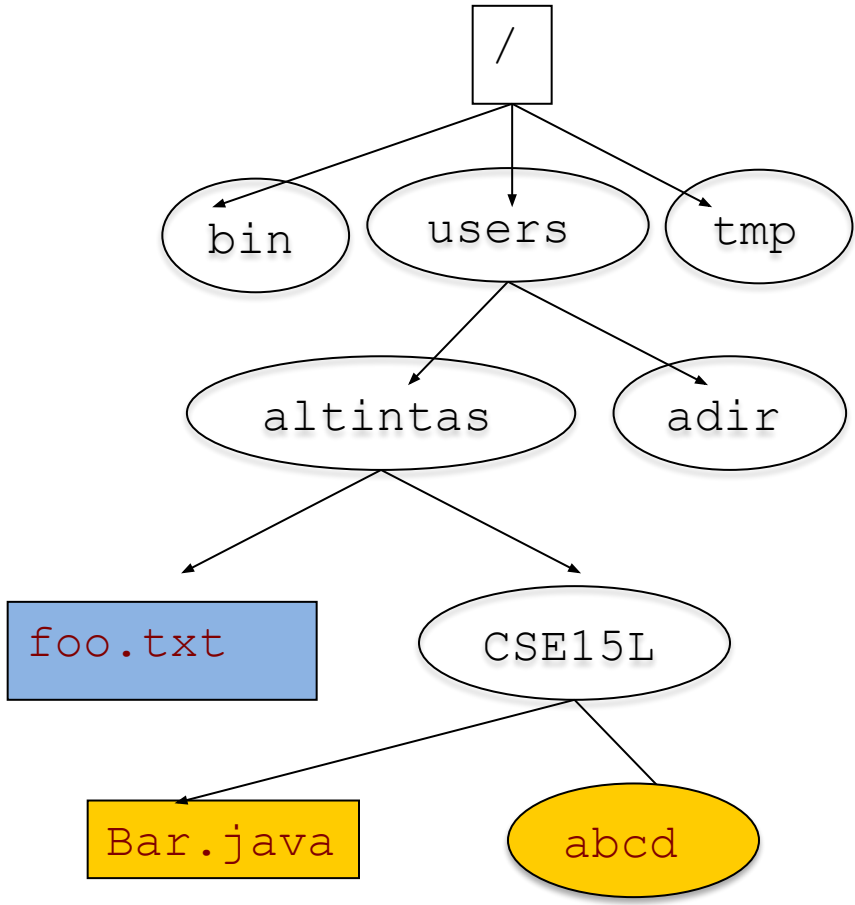
UNIX file hierarchy

- Directories may contain files or other directories
- Leads to a tree structure for the filesystem
- Root directory: /



Path names

- Separate directories by /
- Absolute path names
  - start at root and follow the tree
  - e.g. `/users/altintas/foo.txt`



- Relative path
  - start at working directory
  - `..` refers to level above; `.` refers to working dir.
  - If `/users/altintas/CSE15L` is working dir, all these refer to the same file

    `../foo.txt`

- ~ (TILDE) in path names
  - `~/foo.txt` or `~altintas/foo.txt`

Caution when working with relative paths:

Make sure you check which directory you are in.

stdin, stdout, and stderr

- Each shell (and in fact all programs) automatically open three "files" when they start up
  - Standard input (stdin): Usually from the keyboard
  - Standard output (stdout): Usually to the terminal
  - Standard error (stderr): Usually to the terminal
- Programs use these three files when reading (e.g. `cin` in C++), writing (e.g. `cout` in C++), or reporting errors/diagnostics

Common Unix Commands

- ls -- run with –a **option** to list hidden files (.*)
- mkdir
- cd (“.” is the current directory, “..” is the parent directory, “~” is the home directory)
- * is the wildcard

- pwd
- cp
- mv
- rm

- clear
- grep
- cat
- sort
- man

REDIRECTION

stdout: TRY

- ls > MyFiles.text
- cat > temp1.txt
- cat >> temp1.txt

stdin:

- mail user@domain.com < message

Standard Output and Standard Error

- On Unix and Unix-like operating systems, each process started from the command line has 3 ***file descriptors*** associated with it
  - FD 0: standard input, "stdin"
  - FD 1: standard output, "stdout"
  - FD 2: standard error, "stderr"
- Standard input is normally connected to the keyboard; standard output and standard error are normally connected to the terminal screen from which the application was launched
- However, these file descriptors can be ***redirected*** and connected to files, I/O of other processes, etc.

Redirecting standard I/O

- Redirect standard input to read from a file

  ```
  $  command < somefile
  ```

- Redirect standard output to write to a file

  ```
  $  command > afile1
  ```

- Redirect standard ouput explicitly as FD 1

  ```
  $  command 1> afile2
  ```

- Redirect standard error to write to a file

  ```
  $  command 2> afile3
  ```

- Redirect standard error and standard output to different files

  ```
  $  command 2> afile4 > afile5
  ```

- Redirect everything!

  ```
  $  command > onefile 2> anotherfile < yetanotherfile
  ```

Redirecting standard I/O, cont'd

- Using `>` will truncate the file to zero length (i.e. make it empty) if it exists. If you want to append to the file instead, use `>>` :
  - Redirect standard output to append to a file

    ```
    $  command >> afile1
    ```

  - Redirect standard error to append to a file

    ```
    $  command 2>> afile3
    ```

  - To redirect stderr and stdout to the same file

    ```
    $  command > afile 2>&1
    ```

    (The `2>&1` means: send FD 2 output to the same place as FD 1, namely, the file `afile`)

Pipes and filters

- Pipe: a way to send the output of one command to the input of another
- Filter: a program that takes input and transforms it in some way
  - `wc` - gives a count of words/lines/chars
  - `grep` - searches for lines with a given string
  - `more`
  - `sort` - sorts lines alphabetically or numerically

Examples of filtering

- `ls -la | more`
- `cat file | wc`
- `man ksh | grep history`
- `ls -l | grep dkl | wc`
- `who | sort > current_users`

Intro to Shell Scripting

What is Shell Script?

- A **shell script** is a script written for the shell

- Two key ingredients
  - UNIX/LINUX commands
  - Shell programming syntax

My First Shell Script

$ vi myfirstscript.sh

```
#!/bin/sh

# The first example of a shell script
directory=`pwd`
echo Hello World!
echo The date today is `date`
echo The current directory is $directory
```

$ chmod +x myfirstscript.sh

$ ./myfirstscript.sh

```
Hello World!
The date today is Mon Mar 8 15:20:09 EST 2010
The current directory is /netscr/shubin/test
```

chmod Command

Quote Characters

There are three different quote characters with different behaviour. These are:

“ : double quote, weak quote. If a string is enclosed in “ ” the references to variables (i.e *$variable*) are replaced by their values. Also back-quote and escape \ characters are treated specially.

‘ : single quote, strong quote. Everything inside single quotes are taken literally, nothing is treated as special.

` : back quote. A string enclosed as such is treated as a command and the shell attempts to execute it. If the execution is successful the primary output from the command replaces the string.

Example: `echo “Today is:” `date``

Shell Programming

- Programming features of the UNIX/LINUX shell:
  - < *Shell variables*: Your scripts often need to keep values in memory for later use. Shell variables are symbolic names that can access values stored in memory
  - < *Operators*: Shell scripts support many operators, including those for performing mathematical operations
  - < *Logic structures*: Shell scripts support sequential logic (for performing a series of commands), decision logic (for branching from one point in a script to another), looping logic (for repeating a command several times), and case logic (for choosing an action from several possible alternatives)

Variables

- **Variables are symbolic names that represent values stored in memory**
- **Three different types of variables**
  - Global Variables**: Environment and configuration variables, capitalized, such as HOME, PATH, SHELL, USER, and PWD.**

    When you login, there will be a large number of global System variables that are already defined. These can be freely referenced and used in your shell scripts.
  - Local Variables

    Within a shell script, you can create as many new variables as needed. Any variable created in this manner remains in existence only within that shell.
  - Special Variables

    Such as positional parameters $1, $2 …

    $1 refers to the first string after the name of the script file on the command line, e.g., echo $1

    $2 refers to the second string, and so on.

A few global (environment) variables

SHELL	Current shell
DISPLAY	Used by X-Windows system to identify the display
HOME	Fully qualified name of your login directory
PATH	Search path for commands
MANPATH	Search path for <man> pages
PS1 & PS2	Primary and Secondary prompt strings
USER	Your login name
TERM	terminal type
PWD	Current working directory

Referencing Variables

Variable contents are accessed using '$':

e.g. $ echo $HOME

$ echo $SHELL

To see a list of your environment variables:

```
$ printenv
```

or:

```
$ printenv | more
```

Defining Local Variables

- As in any other programming language, variables can be defined and used in shell scripts.

Unlike other programming languages, variables in Shell Scripts are not typed.

- Examples :

```
a=1234  # a is NOT an integer, a string instead
b=$a+1 # will not perform arithmetic but be the string '1234+1'
b=`expr $a + 1 `  will perform arithmetic so b is 1235 now.
```

Note : +,-,/,*,**, % operators are available.

```
b=abcde # b is string
b='abcde' # same as above but much safer.
b=abc  def # will not work unless 'quoted'
b='abc def' # i.e. this will work.
```

IMPORTANT NOTE: DO NOT LEAVE SPACES AROUND THE =

Positional Parameters

- When a shell script is invoked with a set of command line parameters each of these parameters are copied into special variables that can be accessed.
  - $0 This variable that contains the name of the script
  - $1, $2, ….. $9 1st, 2nd 3rd command line parameter
  - $# Number of command line parameters
  - $$ process ID of the shell
  - $@ same as $* but as a list one at a time
  - $? Return code 'exit code' of the last command
  - `Shift` command: This shell command shifts the positional parameters by one towards the beginning and drops $1 from the list. After a shift $2 becomes $1 , and so on … It is a useful command for processing the input parameters one at a time.

Example:

Invoke : `./myscript one two buckle my shoe`

During the execution of `myscript` variables $1 $2 $3 $4 and $5 will contain the values *one, two, buckle, my, shoe* respectively.

Variables

- vi myinputs.sh

```
#!/bin/sh
echo Total number of inputs: $#
echo First input: $1
echo Second input: $2
```

- chmod u+x myinputs.sh

- myinputs.sh ALTINTAS UCSD CSE

```
Total number of inputs: 3
First input: ALTINTAS
Second input: UCSD
```

Shell Operators

- The Bash/Bourne/ksh shell operators are divided into three groups:
  - defining and evaluating operators,
  - arithmetic operators, and
  - redirecting and piping operators

Arithmetic Operators

- vi math.sh

```
#!/bin/sh
count=5
count=`expr $count + 1`
echo $count
```

- chmod u+x math.sh
- ./math.sh

Shell Logic Structures

The four basic logic structures needed for program development are:

- Sequential logic: to execute commands in the order in which they appear in the program
- Decision logic: to execute commands only if a certain condition is satisfied
- Looping logic: to repeat a series of commands for a given number of times
- Case logic: to replace “if then/else if/else” statements when making numerous comparisons

Conditional Statements (if constructs)

The most general form of the if construct is;

```
if command executes successfully
then
      execute command
elif this command executes successfully
then
      execute this command
      and execute this command
else
      execute default command
fi
```

However- elif and/or else clause can be omitted.

Examples

SIMPLE EXAMPLE:

```
if date | grep "Fri"
then
    echo "It's Friday!"
fi
```

FULL EXAMPLE:

```
if [ "$1"  ==  "Monday“ ]
then
        echo "The typed argument is Monday."
elif [ "$1" == "Tuesday" ]
then
        echo "Typed argument is Tuesday"
 else
        echo "Typed argument is neither Monday nor Tuesday"
fi
```

Note: = or == will both work in the test but == is better for readability.

Tests

String and numeric comparisons used with test or [[ ]] which is an alias for test and also [ ] which is another acceptable syntax

- string1 = string2 True if strings are identical
- string1 == string2 ...ditto....
- string1 != string2 True if strings are not identical
- string Return 0 exit status (=true) if string is not null
- -n string Return 0 exit status (=true) if string is not null
- -z string Return 0 exit status (=true) if string is null

- int1 -eq int2 Test identity
- int1 -ne int2 Test inequality
- int1 -lt int2 Less than
- int1 -gt int2 Greater than
- int1 -le int2 Less than or equal
- int1 -ge int2 Greater than or equal

File inquiry operations

-d file	Test if file is a directory
-f file	Test if file is a regular file
-s file	Test if the file has non zero length
-r file	Test if the file is readable
-w file	Test if the file is writable
-x file	Test if the file is executable
-o file	Test if the file is owned by the user
-e file	Test if the file exists

All these conditions return true if satisfied and false otherwise.

for each loops

- Syntax:

  for *arg* in *list*
  do
  command(s)
  ...
  done

- Where the value of the variable *arg* is set to the values provided in the list one at a time and the block of statements executed. This is repeated until the list is exhausted.
- Example:

```
for i in 3 2 5 7
do
        echo " $i times 5 is  $((  $i  * 5 ))  "
done
```

while loops

- Syntax:

```
while this_command_execute_successfully
    do
        this command
        and this command
    done
```

- EXAMPLE:

```
while test "$i" -gt 0      # can also be  while  [ $i > 0 ]
do
    i=`expr $i - 1`
done
```

Case statements

- The case structure compares a string ‘usually contained in a variable’ to one or more patterns and executes a block of code associated with the matching pattern. Matching-tests start with the first pattern and the subsequent patterns are tested only if no match is not found so far.

```
case argument in
    pattern 1) execute this command
               and this
               and this;;
    pattern 2) execute this command
               and this
               and this;;
    esac
```

The Process ID (PID)

What is the PID?

- PID: is the Process IDentifier
- It is a number used by UNIX to uniquely identify an active process.
- This number may be used as a parameter in various function calls allowing processes to be manipulated, such as killing it.

- Two commands: ps and top

Child, parent and grandparent of processes…

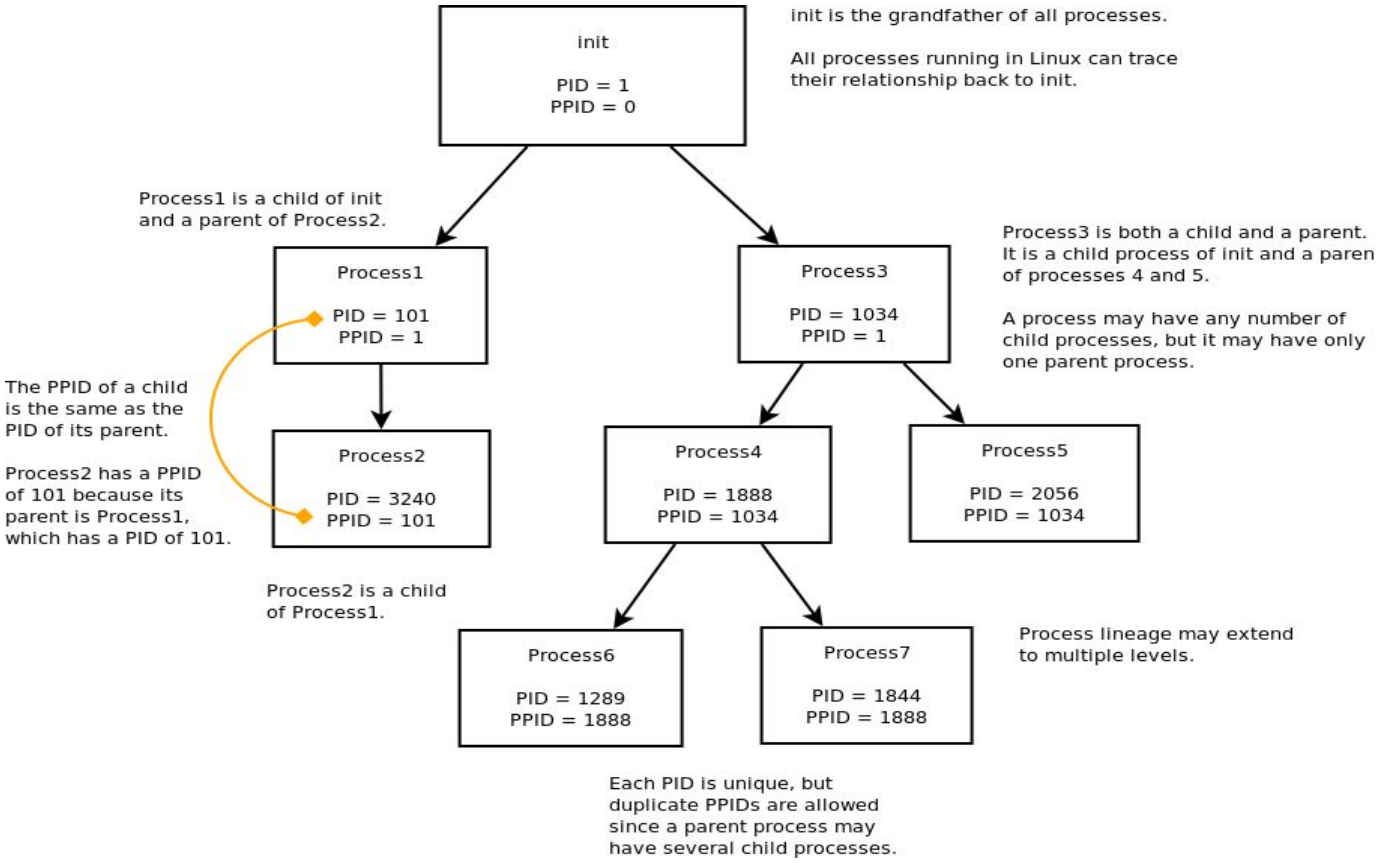
- The grandparent:
  - In every Unix based system, there is a process with PID = 1.
  - This is the grandparent of all processes, e.g., init or launchd.
  - The kernel itself has a PID of 0.

- The parent:

In addition to a unique PID each process has also a PPID (Parent Process ID) that tells which process started it. The PPID is the PID of the process's parent.

Child, parent and grandparent of processes…

pstree: this command will show the relationship of all processes in a tree like structure



Child, parent and grandparent of processes…

- fork(): In Unix a child process is created when the parent process invoke the fork() system call.

- Orphan process: when a parent process dies, the child becomes an orphan process. But don’t worry, It will be immediately adopted by the grandparent: the init process!

How to kill a process?

Step 1: First look at all the processes that are running and filter the ones you want to kill

```
ps [options]
```

This shows all the processes that are currently running

```
ps -aef | grep altintas
```

This shows the running processes limited to those belonging to adam.

Step 2: Then kill the process you selected

```
kill 125
```

This command will kill the process which PID number is 125

Step 3: Then make sure that the process was killed:

```
ps –efl
```

You look again at the process table and make sure that the process which PID is 125 is not listed.

Step 4: Or you can kill it dead:

```
kill -9 125
```

If the process has not stopped, you can kill it dead with the -9 option.

Introduction to Debugging

Debugging is not Algorithmic

In mathematics and computer science, an **algorithm** is a self-contained step-by-step set of operations to be performed.

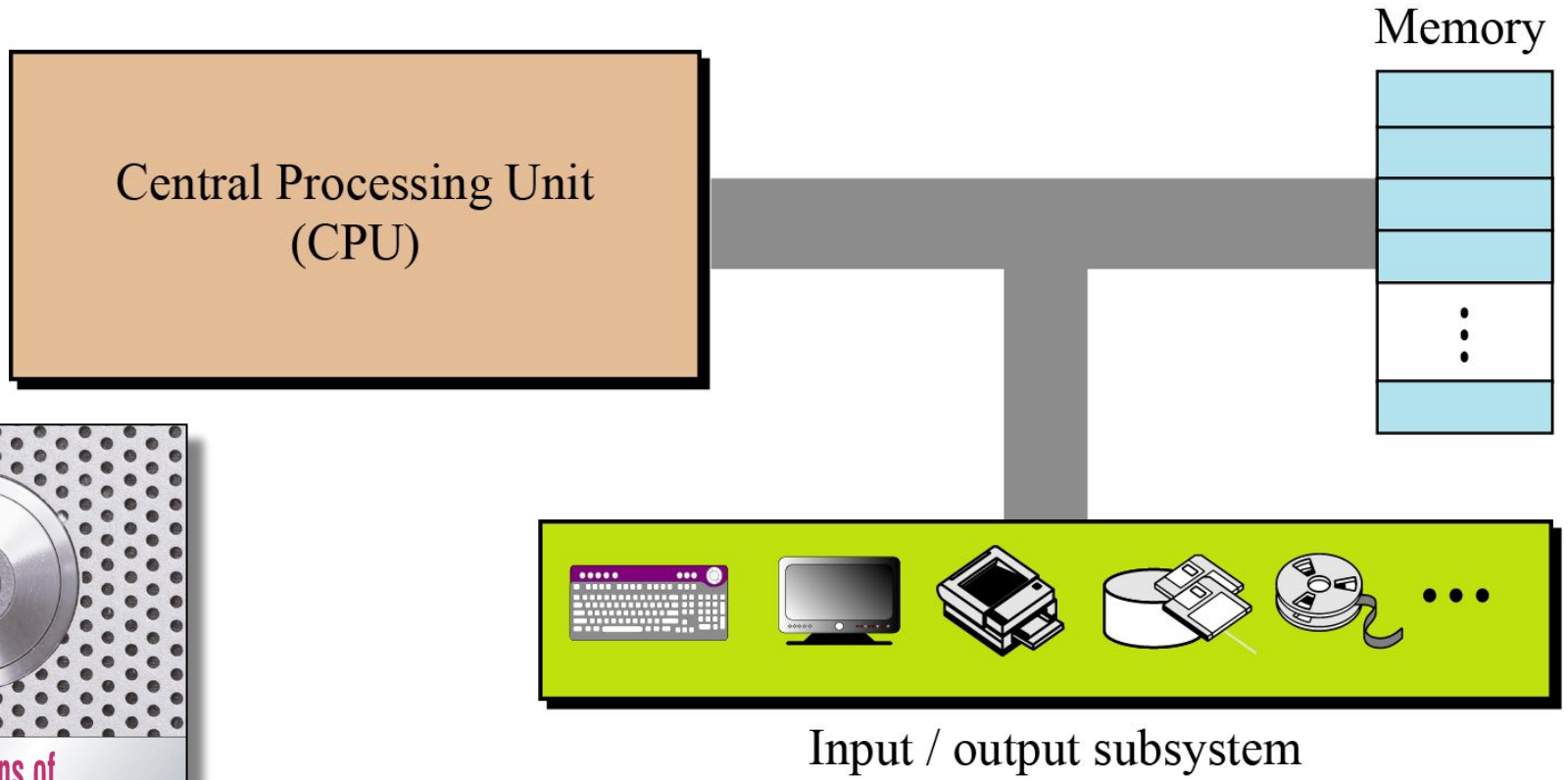
- There is no program that can debug all programs
  - There is not even a program that can just correctly determine whether programs have infinite loop bugs or not!
- Debugging requires the application of good techniques, thinking, and… a little luck

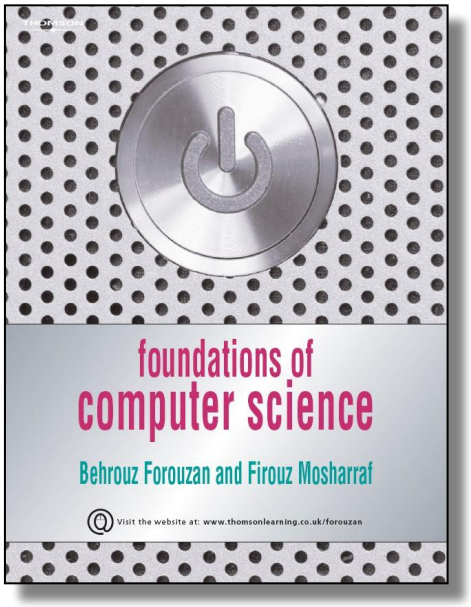
Debugging Steps

1. Understand the system
2. Identify the problem
3. Reproduce the problem in as few steps as possible
4. Diagnose the cause of the problem
5. Fix the problem
6. Reflect and learn from this problem, and this fix

What is memory?

We can divide the parts that make up a computer into three broad categories or subsystem: the central processing unit (CPU), the main memory and the input/output subsystem.





1. What is a memory leak?

2. How does it happen?

3. What are the consequences?

"Memory leaks are like fine wine… they need aging"

1. How can we avoid memory leaks?

2. What tools are available to debug such errors?

- GDB
- JDB
- Valgrind

Software Development/Build Automation

Make and Makefiles

- One simple but powerful scripting framework is make
- When make is run, it looks for a file named Makefile in the current working directory
- The Makefile contains rules, which tell make what to do

make and simple Makefile rules

- make interprets a simple Makefile rule as a script with the given name
- You can have multiple rules, with different names, in one Makefile
- Running "make $name_i$" will execute the actions in the rule that has name $name_i$
- Running "make" will execute the first rule

Rule Interpretation

"to make the target, first make all its dependencies, then perform all the actions"

Makefile timestamps and dependencies

- The dependencies in a rule can be the names of other targets in the Makefile
- In that case, a rule like

  $target_0$: $target_1$ … $target_M$
  $action_1$
  …
  $action_N$

  will execute the actions for **$target_1$** through **$target_M$**, and then the actions for **$target_0$**

- The actions will be performed if any of the dependencies are newer than the target.

The .PHONY target

- A target that is always out of date!

```
clean:
   rm -rf *.o
```

What if there was a file called clean in the same directory?

```
.PHONY: clean
clean:
   rm -rf *.o
```

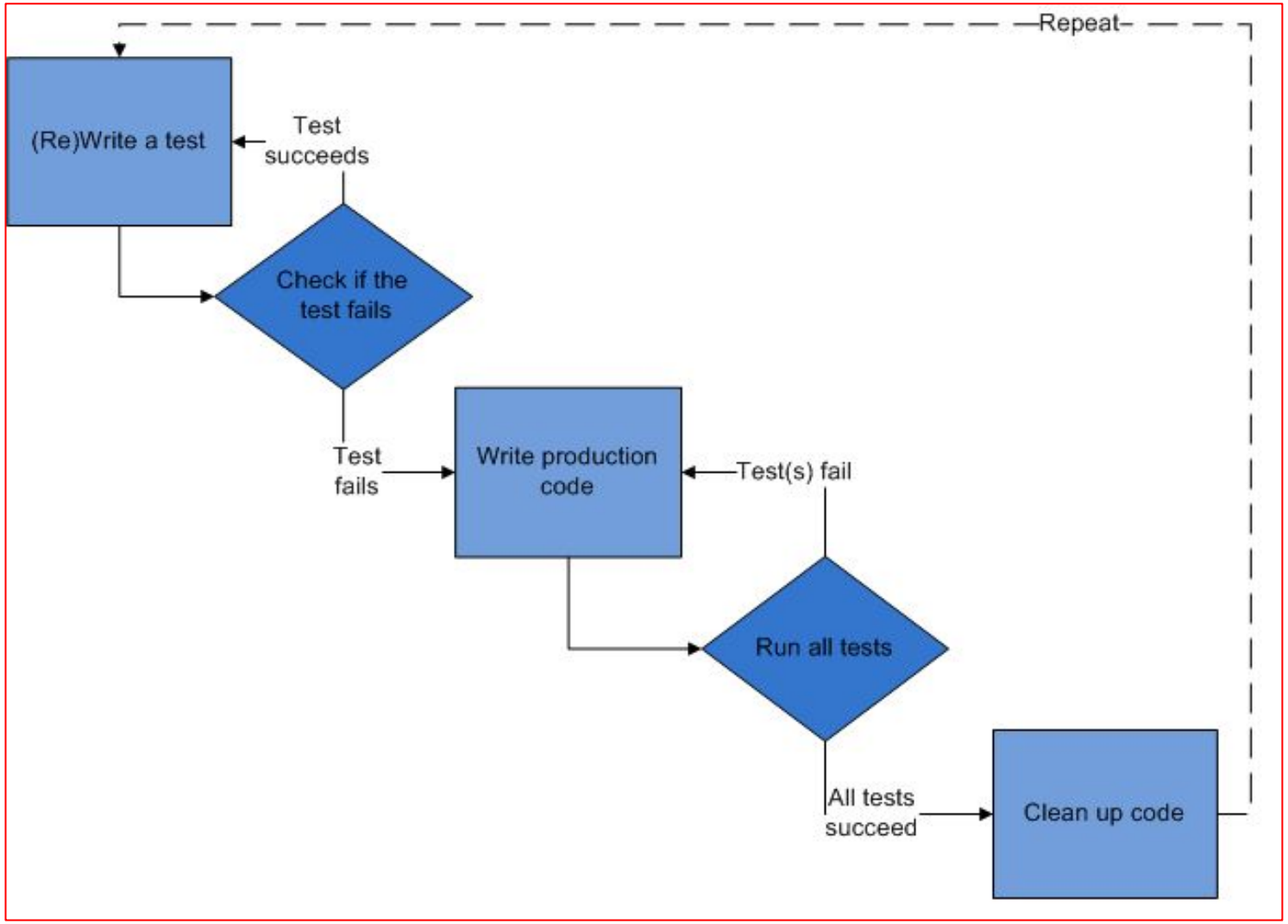
More examples at:
http://makefiletutorial.com/

Test-Driven Development

Test-driven Development (TDD)

- In test-driven development, tests are written before the software that the test is written for

Understand the requirements before writing the code!



Needs to be followed by software integration testing!

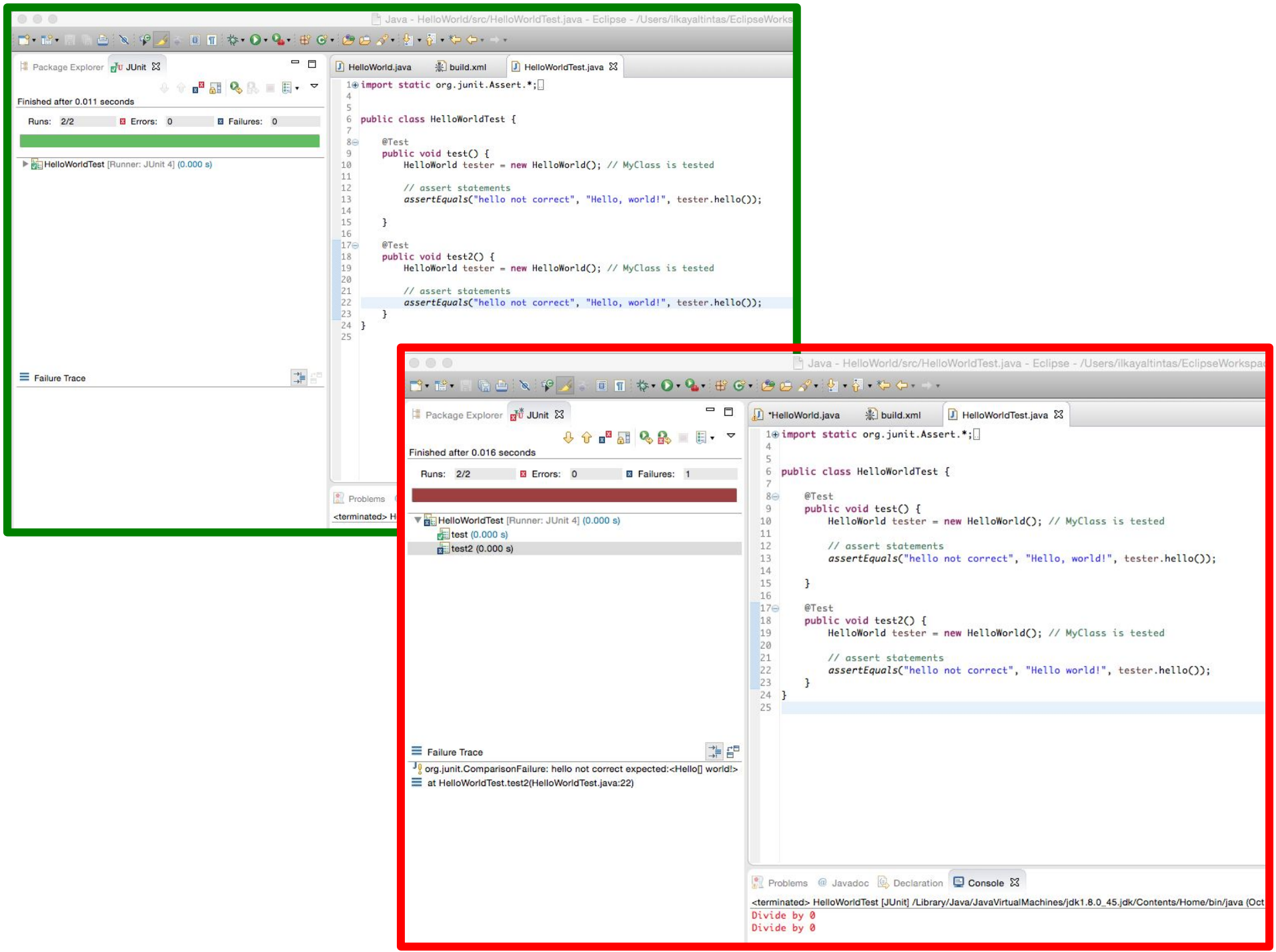
Unit Testing

- You should test every unit of a software system
- What is a *unit*? In object-oriented programming, usually a software unit is taken to be a single method
  - So: we should ideally test every method of every class with every possible input
    - NOT VERY PRACTICAL!

Strategies for Unit Testing

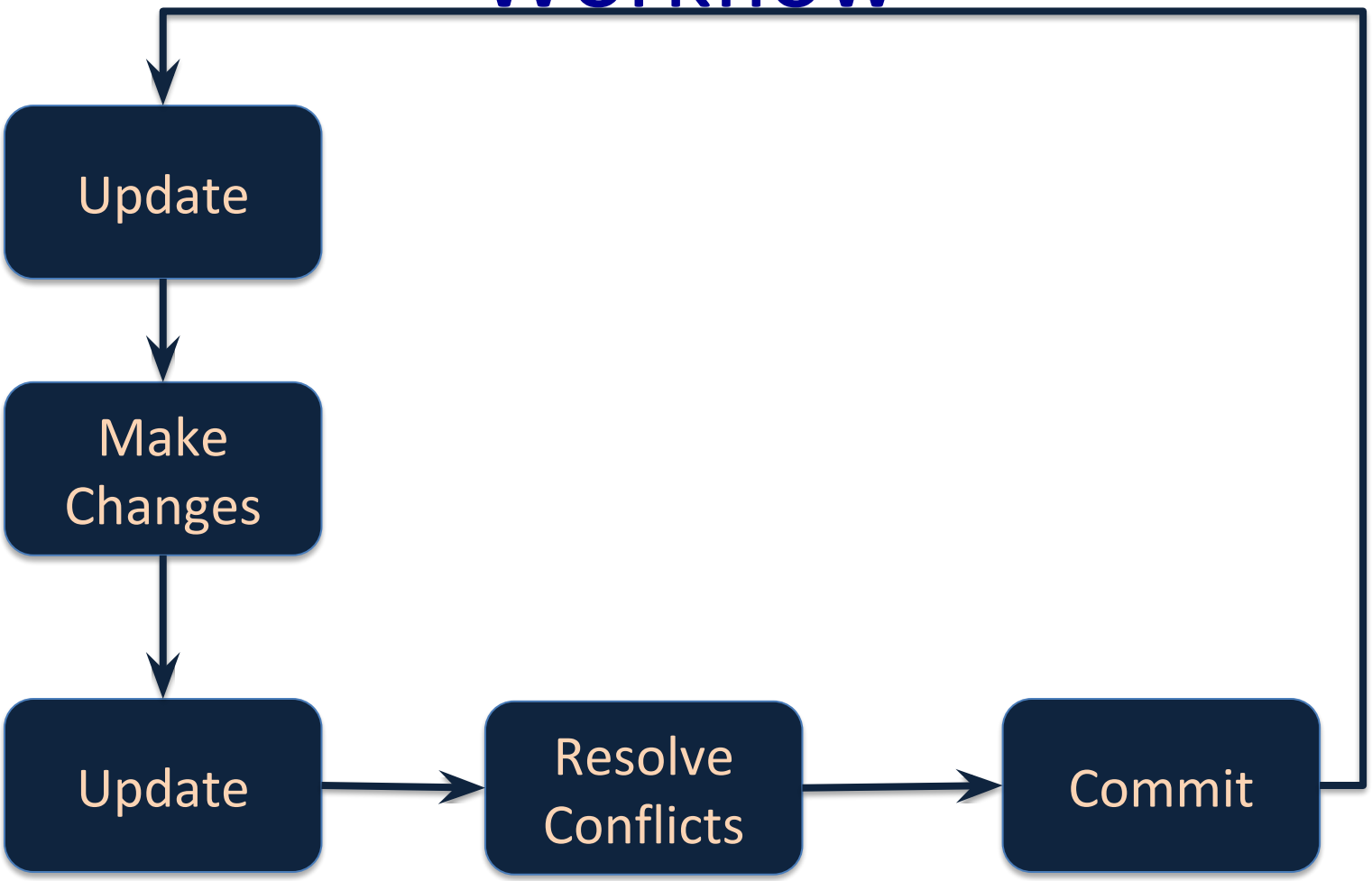
- Identify and prioritize testing of
  - core functionality
  - corner cases for exceptions
  - special input values
  - frequency of a functionality being exercised
- Test related functionality as test suites
- Should test both positive and negative execution paths

Read and understand the specification of what the software you are testing is supposed to do, so you can test whether or not it is doing it!


Java - HelloWorld/src/HelloWorldTest.java - Eclipse - /Users/ilkayaltintas/EclipseWorks
Package Explorer
JUnit
Finished after 0.011 seconds
Runs: 2/2
Errors: 0
Failures: 0
HelloWorldTest [Runner: JUnit 4] (0.000 s)
Failure Trace
HelloWorld.java
build.xml
HelloWorldTest.java
1 import static org.junit.Assert.*;
4
5
6 public class HelloWorldTest {
7
8 @Test
9 public void test() {
10 HelloWorld tester = new HelloWorld(); // MyClass is tested
11
12 // assert statements
13 assertEquals("hello not correct", "Hello, world!", tester.hello());
14
15 }
16
17 @Test
18 public void test2() {
19 HelloWorld tester = new HelloWorld(); // MyClass is tested
20
21 // assert statements
22 assertEquals("hello not correct", "Hello, world!", tester.hello());
23 }
24 }
25
Problems
<terminated> H
Java - HelloWorld/src/HelloWorldTest.java - Eclipse - /Users/ilkayaltintas/EclipseWorkspa
Package Explorer
JUnit
Finished after 0.016 seconds
Runs: 2/2
Errors: 0
Failures: 1
HelloWorldTest [Runner: JUnit 4] (0.000 s)
test (0.000 s)
test2 (0.000 s)
Failure Trace
org.junit.ComparisonFailure: hello not correct expected:<Hello[] world!>
at HelloWorldTest.test2(HelloWorldTest.java:22)
*HelloWorld.java
build.xml
HelloWorldTest.java
1 import static org.junit.Assert.*;
4
5
6 public class HelloWorldTest {
7
8 @Test
9 public void test() {
10 HelloWorld tester = new HelloWorld(); // MyClass is tested
11
12 // assert statements
13 assertEquals("hello not correct", "Hello, world!", tester.hello());
14
15 }
16
17 @Test
18 public void test2() {
19 HelloWorld tester = new HelloWorld(); // MyClass is tested
20
21 // assert statements
22 assertEquals("hello not correct", "Hello world!", tester.hello());
23 }
24 }
25
Problems
Javadoc
Declaration
Console
<terminated> HelloWorldTest [JUnit] /Library/Java/JavaVirtualMachines/jdk1.8.0_45.jdk/Contents/Home/bin/java (Oct
Divide by 0
Divide by 0

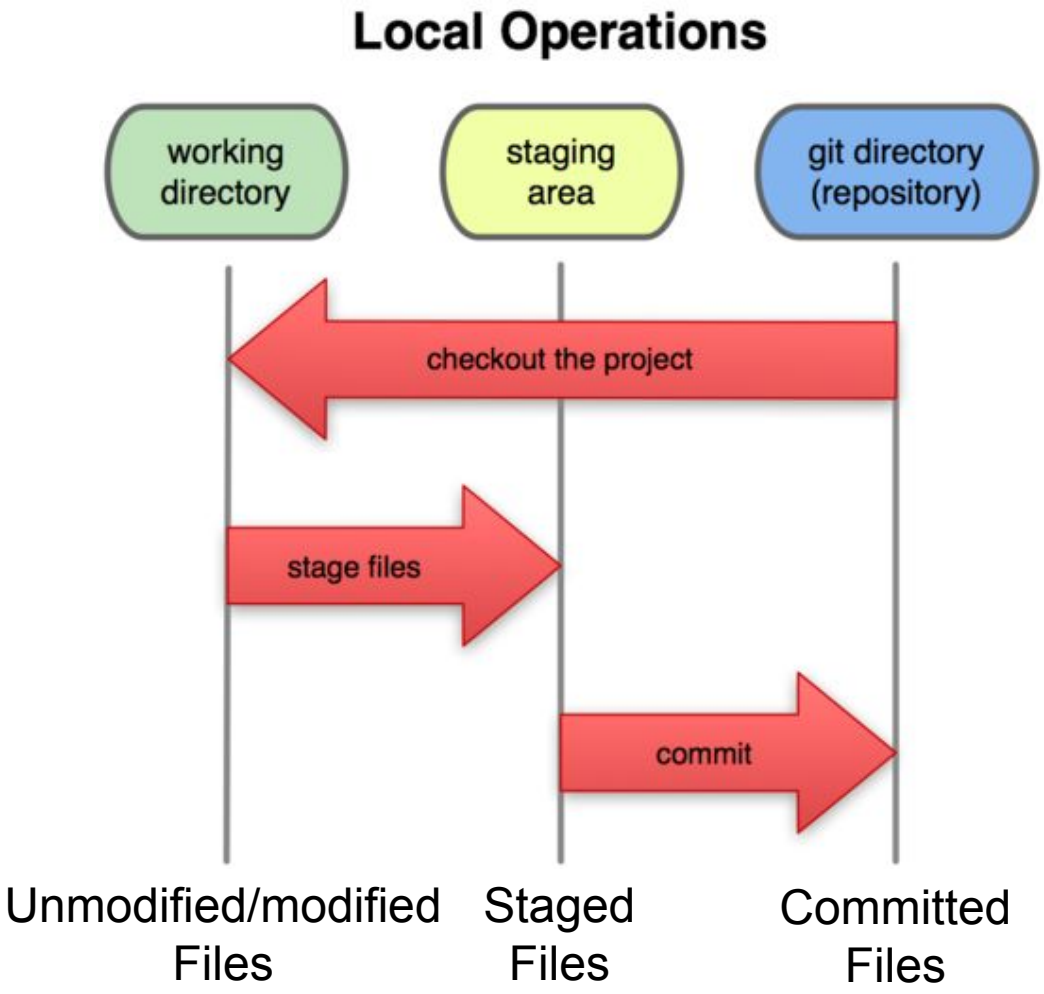
Version Control

Software Version Control Workflow



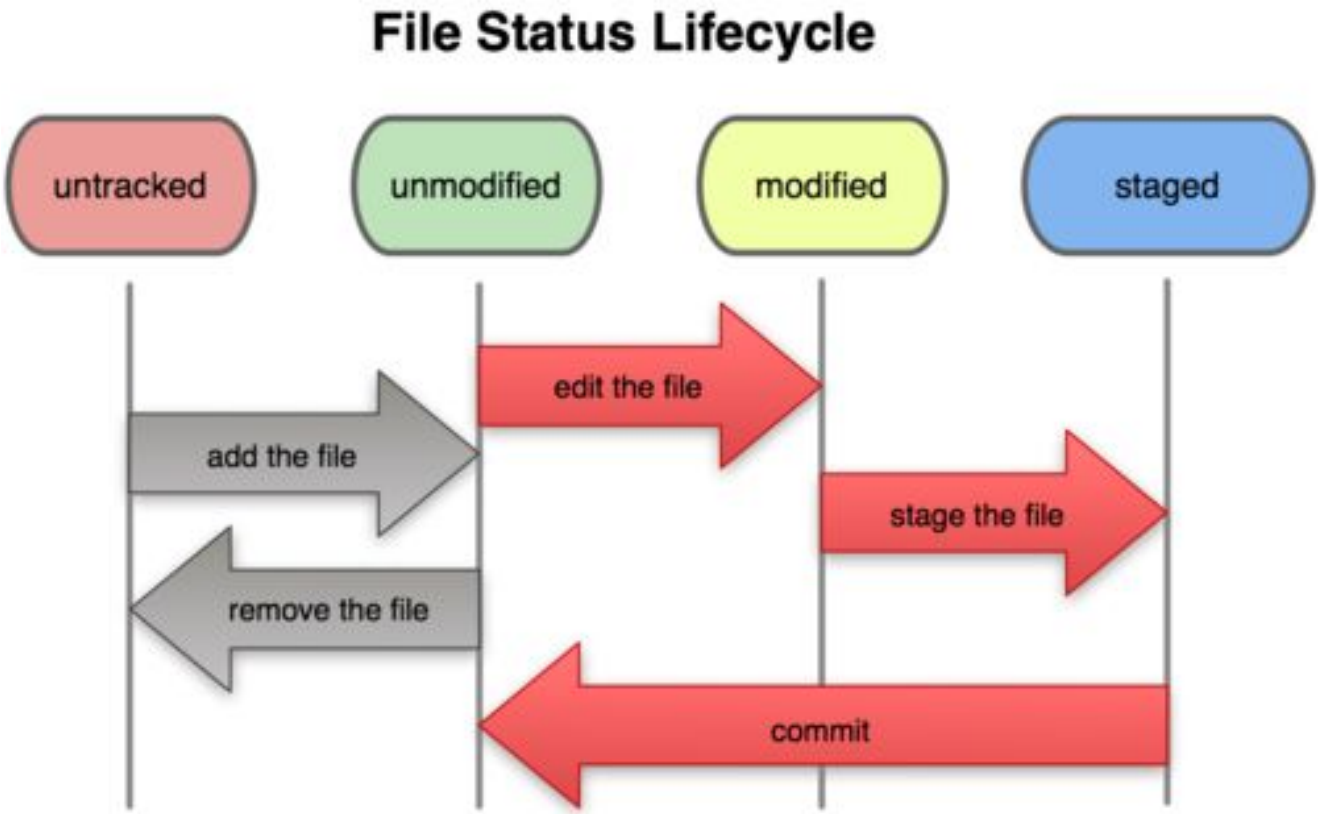
Basic Intro to Git

A Local Git project has three areas



Note: working directory sometimes called the “working tree”, staging area sometimes called the “index”.

A Git file lifecycle



An Introduction to XML

(slides based on "PRINCIPLES OF **DATA INTEGRATION**")

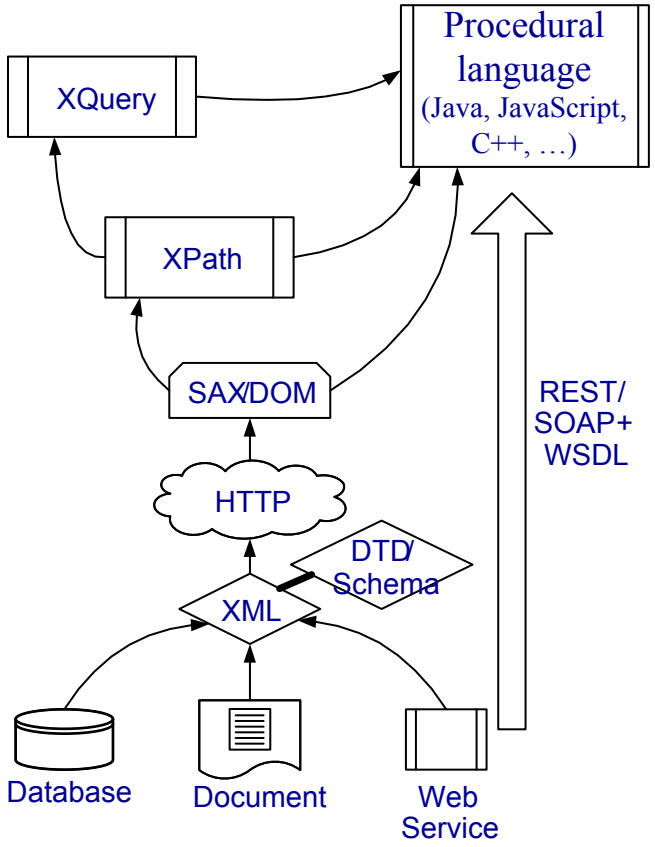
What Is XML?

Hierarchical, human-readable format

- A “sibling” to HTML, always parsable
- “Lingua franca” of data: encodes documents and structured data
- Blends data and schema (structure)

Core of a broader ecosystem

- Data – XML
- Schema – DTD and XML Schema
- Programmatic access – DOM and SAX
- Query – XPath, XSLT, XQuery
- Distributed programs – Web services



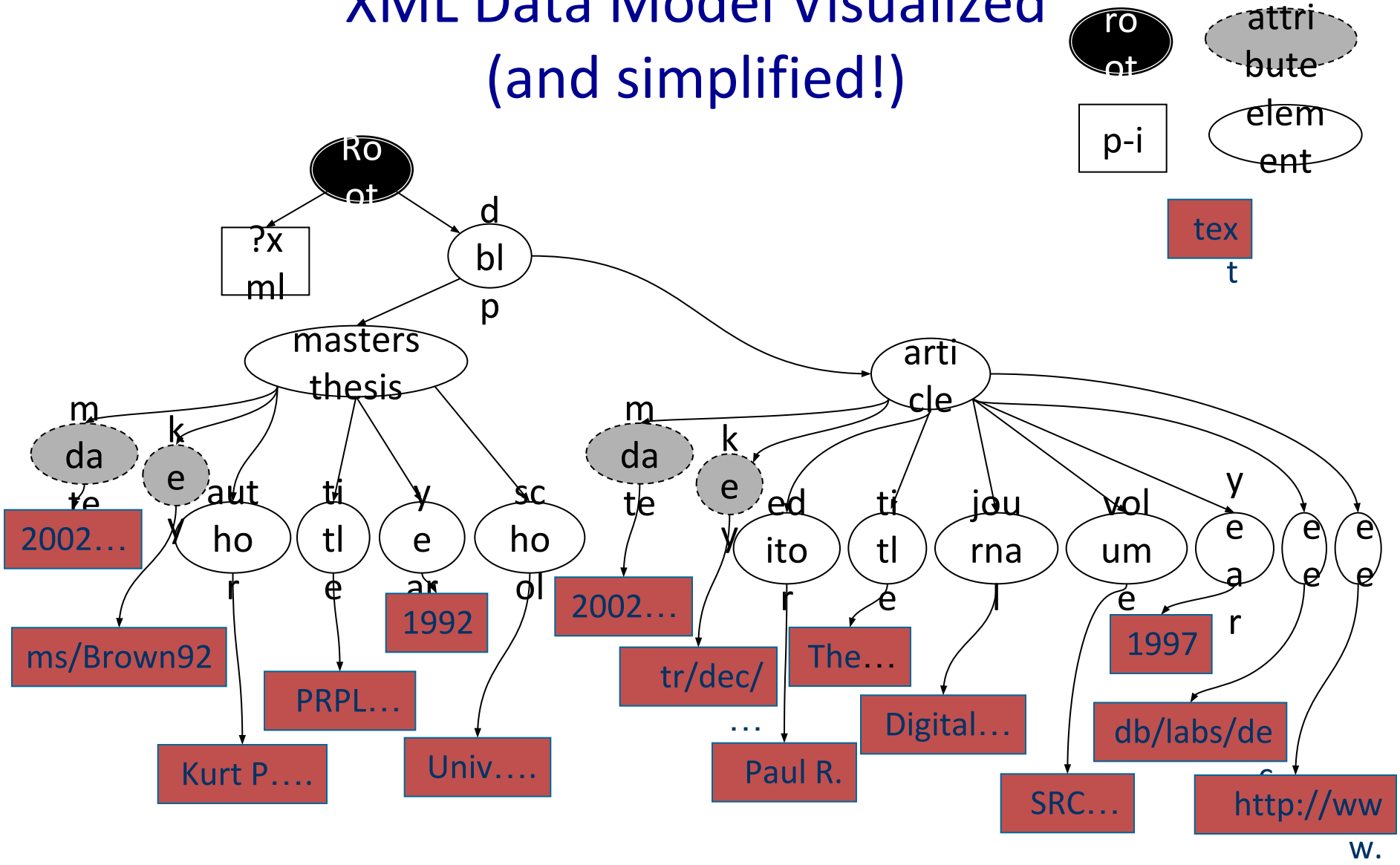
XML as a Data Model

XML "information set" includes 7 types of nodes:

- Document (root)
- **Element**
- **Attribute**
- Processing instruction
- **Text (content)**
- Namespace
- Comment

XML data model includes this, plus typing info, plus order info and a few other things

XML Data Model Visualized (and simplified!)



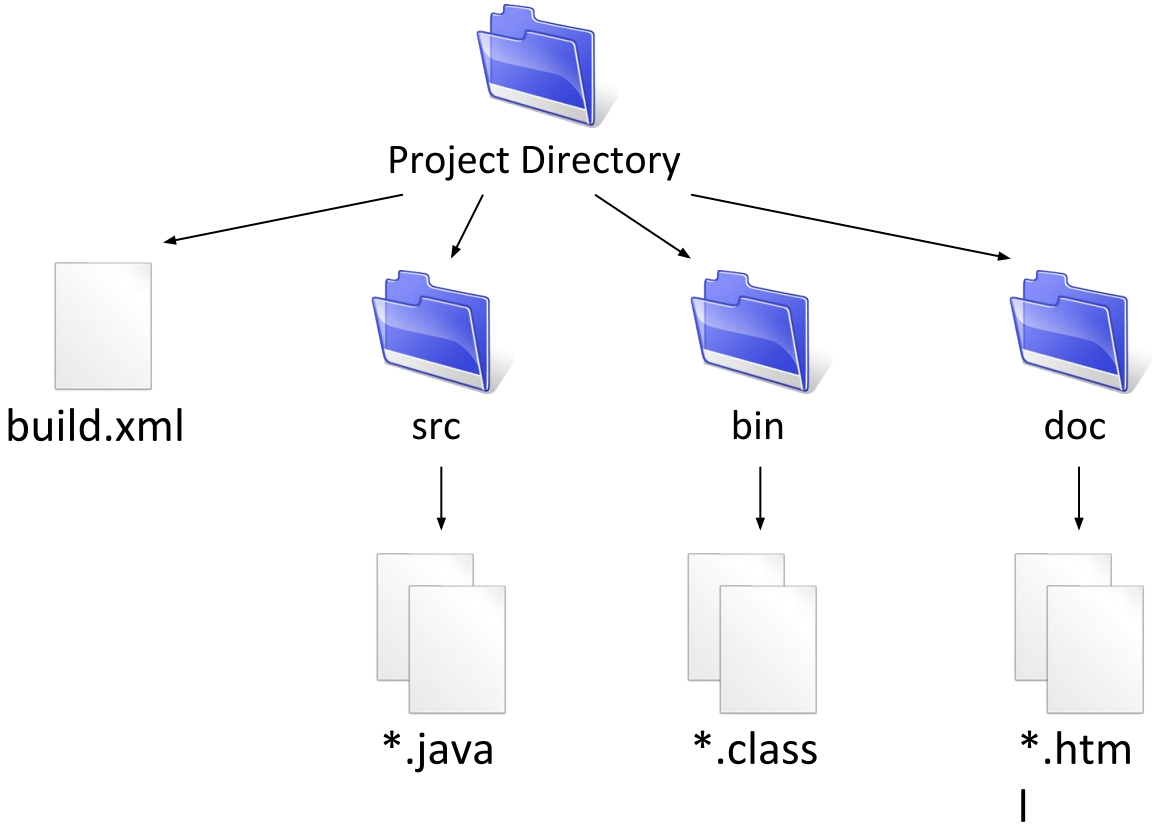
XML is "Semi-Structured"

```
<parents>
 <parent name="Jean" >
    <son>John</son>
    <daughter>Joan</daughter>
    <daughter>Jill</daughter>
 </parent>
 <parent name="Feng">
    <daughter>Ella</daughter>
 </parent>
…
```

An Introduction to Ant

Project Organization

- The following example assumes that your workspace will be organized like so…



Projects

- The project tag is used to define the project you wish to work with
- Projects tags typically contain 3 attributes
  - name – a logical name for the project
  - default – the default target to execute
  - basedir – the base directory for which all operations are done relative to
- Additionally, a description for the project can be specified from within the project tag

Targets

- The target tag has the following required attribute
  - name – the logical name for a target
- Targets may also have optional attributes such as
  - depends – a list of other target names for which this task is dependant upon, the specified task(s) get executed first
  - description – a description of what a target does
- Like make files, targets in Ant can depend on some number of other targets
  - For example, we might have a target to create a jarfile, which first depends upon another target to compile the code
- A build file may additionally specify a default target

Tasks

- A task represents an action that needs execution
- Tasks have a variable number of attributes which are task dependant
- There are a number of build-in tasks, most of which are things which you would typically do as part of a build process
  - Create a directory
  - Compile java source code
  - Run the javadoc tool over some files
  - Create a jar file from a set of files
  - Remove files/directories
  - And many, many others…
    - For a full list see: http://ant.apache.org/manual/coretasklist.html

Completed Build File (1 of 2)

```
<project name="Sample Project" default="compile" basedir=".">

 <description>
  A sample build file for this project
 </description>

 <!-- global properties for this build file -->
 <property name="source.dir" location="src"/>
 <property name="build.dir" location="bin"/>
 <property name="doc.dir" location="doc"/>

 <!-- set up some directories used by this project -->
 <target name="init" description="setup project directories">
  <mkdir dir="${build.dir}"/>
  <mkdir dir="${doc.dir}"/>
 </target>

 <!-- Compile the java code in ${src.dir} into ${build.dir} -->
 <target name="compile" depends="init" description="compile java sources">
  <javac srcdir="${source.dir}" destdir="${build.dir}"/>
 </target>
```

Completed Build File (2 of 2)

```
  <!-- Generate javadocs for current project into ${doc.dir} -->
  <target name="doc" depends="init" description="generate documentation">
    <javadoc sourcepath="${source.dir}" destdir="${doc.dir}"/>
  </target>

  <!-- Delete the build & doc directories and Emacs backup (*~) files -->
  <target name="clean" description="tidy up the workspace">
    <delete dir="${build.dir}"/>
    <delete dir="${doc.dir}"/>
    <delete>
      <fileset defaultexcludes="no" dir="${source.dir}" includes="**/*~"/>
    </delete>
  </target>

</project>
```

Diagnostic Output from Programs

- So far we have concentrated on unit testing of object-oriented software
- Unit testing involves understanding and using the documented interface to the software
- However it can be very useful to inspect the internal operation of the software at finer grain (less abstraction)
  - "**Diagnostic output**" from a running program

Relevance:
What do we need for Diagnostic Output for?

- Tracing
- Timing
- Profiling
- Logging
- Error reporting

The goal in general:

debugging and optimization

Tools for Diagnostic Output

- ✓ Standard output and standard error
- ✓ Debuggers
- ✓ Java assertions
- Java logging framework
- Java profiling framework

We will look at logging today…

Logging

- **Logging** means:
  - *automatically recording* diagnostic output from a program

- Logging output can be useful during development and testing, but also in production code:
  - A web server could log IP address of incoming http requests
  - A mail server could log basic info about each email received
  - … **what else can you think of?**
- Logging using standard error output, or ordinary file output, can be done
- However it is better to make use of a logging framework, which provides lots of useful functionality

Logging Framework Functionality

- A logging framework or API (Application Programmer Interface) usually provides ways to:
  - Specify the origin of a log entry (which application, which class, which method, etc.)
  - Specify the content of the log entry
  - Specify the level of importance of the log entry
  - Control what importance levels are actually being logged
  - Control where the log entries are recorded
  - Analyze resulting log files

Logging Frameworks

- Logging frameworks exist for most application environments
- Several commercial and free frameworks are available for Java

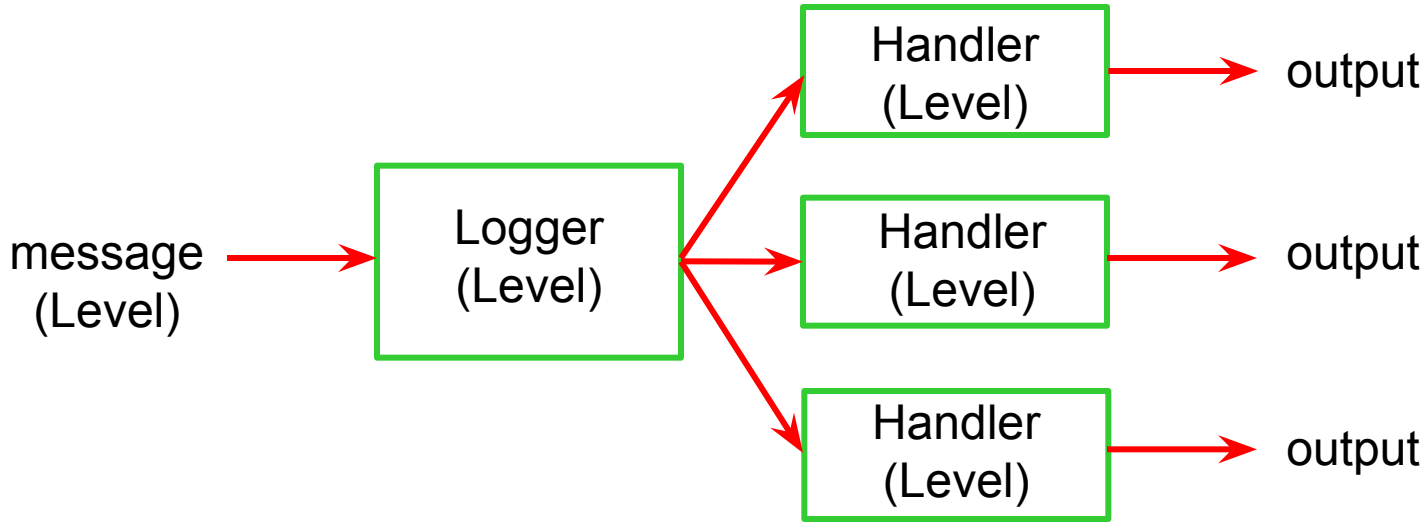
We will concentrate on the framework in the **java.util.logging** package, distributed as part of J2SE since version 1.4.

Classes in java.util.logging

- Important classes in the java.util.logging package:
  - `Logger`
  - `Handler`
    - Subclasses: `ConsoleHandler, FileHandler, SocketHandler`
  - `Formatter`
    - Subclasses: `SimpleFormatter, XMLFormatter`
  - `Level`

Review of Java logging

- Recall the Java logging framework:
- Create a Logger object, and give it one or more Handler objects
- Call the Logger's methods to log messages, as needed



Controlling logging

- The Logger class has methods which permit setting the minimum Level, adding and removing Handlers, etc.

- The Handler classes have methods which permit setting the minimum Level, setting a Formatter, etc.

- You can use those methods in your program, but then if you want to change logging Level for example, you have to edit your program and recompile
  - **Easier and more powerful is using a properties file**

- Then run your program telling it to read from a changed properties file; recompilation not necessary

Logging properties file

- See the javadoc online documentation for the format of a logging properties file

- If the properties file is named for example myprop, and your application is named MyApp, then launch it as follows:

```
java -Djava.util.logging.config.file=myprop  MyApp
```

- That tells the application to read logging configuration from the myprop properties file. No need to edit or recompile the application

Controlling logging

- Logging at a low level is most useful during software development
- In production code, only higher levels should be logged
- Removing or commenting out Logger calls is tedious and error prone…
- So control logging by setting Logger or Handler levels
- But what is the cost?...

Disabling stdout/stderr tracing

- That technique can be applied to controlling debugging and tracing output from simple print statements too

```
protected static boolean DEBUG = true;
```

- Then guard each logging statement with that variable:

```
if(DEBUG) System.err.println("x=" + x);
```

- Edit, set the variable to false, and recompile to disable debugging/tracing

Space cost of code

- When considering space cost of an algorithm, usually we think of how much memory an implementation of the algorithm uses at runtime

- But another aspect of space cost is: How much memory does the compiled program itself take?
  - In space-limited applications (e.g. small mobile devices) or bandwidth-limited downloading situations, this might be important

Size of Java .class files

- Adding logging/tracing/debugging statements to your source code will make compiled bytecode .class files larger
- Editing source code to remove those statements is error-prone
- One solution: guard those statements with a **final** static boolean

Compiling out guarded statements

- If the Java compiler has enough information at compile time to prove that a statement will not be executed, it will not generate byte codes for it

- Declare a global boolean final, and initialize it with a constant:

```
protected final static boolean DEBUG = false;
```

- Then a guarded statement won't be compiled:

```
if(DEBUG) System.err.println("x=" + x);
```

Software Correctness and Efficiency

Relevance:
Why do we need to worry about Software Correctness and Efficiency?

- Software should be correct: it should meet its specifications
  - **Testing helps ensure correctness**
- Software should also be **efficient**:
  - it shouldn't use any more resources than necessary
  - **Resources: time, memory, energy**
- Arguably correctness is more important, but both are important

Software Runtime Costs

- Software efficiency can be stated in terms of runtime costs

- Costs should be stated as functions of the size N of the problem the software is given to solve:
  - **T(N):** time cost function
  - **S(N):** space (memory) cost function
  - **E(N):** energy cost function

Determining Cost Functions

- You can **analyze** the software to determine cost functions: look at the source code
  - ***Time cost:*** *how many instructions will be executed*
  - ***Space cost:*** *how many variables will be created*
  - ***Energy cost:*** *usually determined by CPU and memory usage*
- Or you can **measure** the cost functions: implement the software, instrument it, and run it

Measuring Cost Functions

- **Profiling frameworks** exist for instrumenting running software and collecting time, memory, and energy usage data in detail

- First we will look at a simpler approach for measuring time cost: accessing the system clock, and measuring "**wall clock time**" taken to perform an operation

The Unix time commands

- To time a Java application using the built-in time:

```
time java MyApp
```

- To time a Java application using /usr/bin/time, with verbose output:

```
/usr/bin/time -v java MyApp
```

- Many other options are available for /usr/bin/time; see man time for more info

Introduction to Profiling using VisualVM

Measuring Cost Functions: Profiling

- ✓ We have seen basic techniques for measuring time cost: query system clock, print elapsed wall clock time data
- However more sophisticated profiling frameworks exist for instrumenting running software and collecting time, memory, and energy usage data in detail
- We will look at the **VisualVM** tool, which is an all-in-one tool to profile Java applications

- **VisualVM** is a versatile visual profiling tool for Java.
  - It is the successor to the **hprof** and **jhat** heap dump profiling tools, which were discontinued in Java 9.
- VisualVM can...
  - **create heap dumps** as well as **analyze and visualize** them
    - Upon an OutOfMemoryError, can simply browse the heap to see what is taking up all the space.
  - **create thread dumps** as well as **analyze and visualize** them
    - Can analyze multiple concurrent threads
  - analyze **core dumps**
    - If Java crashes, can extract info about the erroneous process
  - monitor the **garbage collector**
  - profile **CPU usage**
  - track **memory "leaks"** (unnecessarily maintained references)

JVMTI

- VisualVM uses the Java Virtual Machine Tool Interface (JVMTI) to accomplish real-time profiling.
- **JVMTI**
  - allows a program to inspect the state and to control the execution of applications running in the JVM
  - This interface instruments the Java virtual machine at a very low level
  - Other (free or commercial) Java profiling tools also use this interface
  - intended to provide a way to observe the JVM in detail **without interfering with it or slowing it down too much**

Reporting Bugs

- To make a good bug report, a bug reporter should:
  1. **Understand the system**, and what it is supposed to do
  2. **Identify the problem**, and summarize clearly what the problem behavior is
  3. **Reproduce the problem**, and state clearly what conditions permit reproducing the problem, and collect screen shots, error output, stack traces, logging output etc. that document the problem

Bug reporting guidelines: preliminaries

- Make sure your software is up to date
  - Try using latest development version to see whether your bug has already been fixed
- Search the bug repository to see whether your bug has already been reported
- Then get ready to enter a new bug
  - Multiple issues should be filed in separate bug reports

Bug reporting guidelines: steps to reproduce

- Steps to reproduce are the **most important part of any bug report**.
  - If a developer is able to reproduce the bug, the bug is very likely to be fixed.
  - If the steps are unclear, it might not even be possible to know whether the bug has been fixed
- Be precise about the steps to reproduce
  - If the developer can't reproduce the bug with your steps, they have to ask you for more detail.

Happy Holidays!