CSC207H: Software Design
Lecture 2

Wael Aboelsaadat

wael@cs.toronto.edu

http://ccnet.utoronto.ca/20075/csc207h1y/

Office: BA 4261

Office hours: R 5-7

Acknowledgement: These slides are based on material by Prof. Karen Reid

UNIX && the Shell

What is UNIX?

- UNIX is an operating system like Windows or Mac OS X
- From wikipedia:

An operating system (OS) is a set of computer programs that manage the hardware and software resources of a computer. An operating system processes raw system and user input and responds by allocating and managing tasks and internal system resources as a service to users and programs of the system. At the foundation of all system software, an operating system performs basic tasks such as controlling and allocating memory, prioritizing system requests, controlling input and output devices, facilitating networking and managing file systems.

UNIX History

UNIX is a "standard",

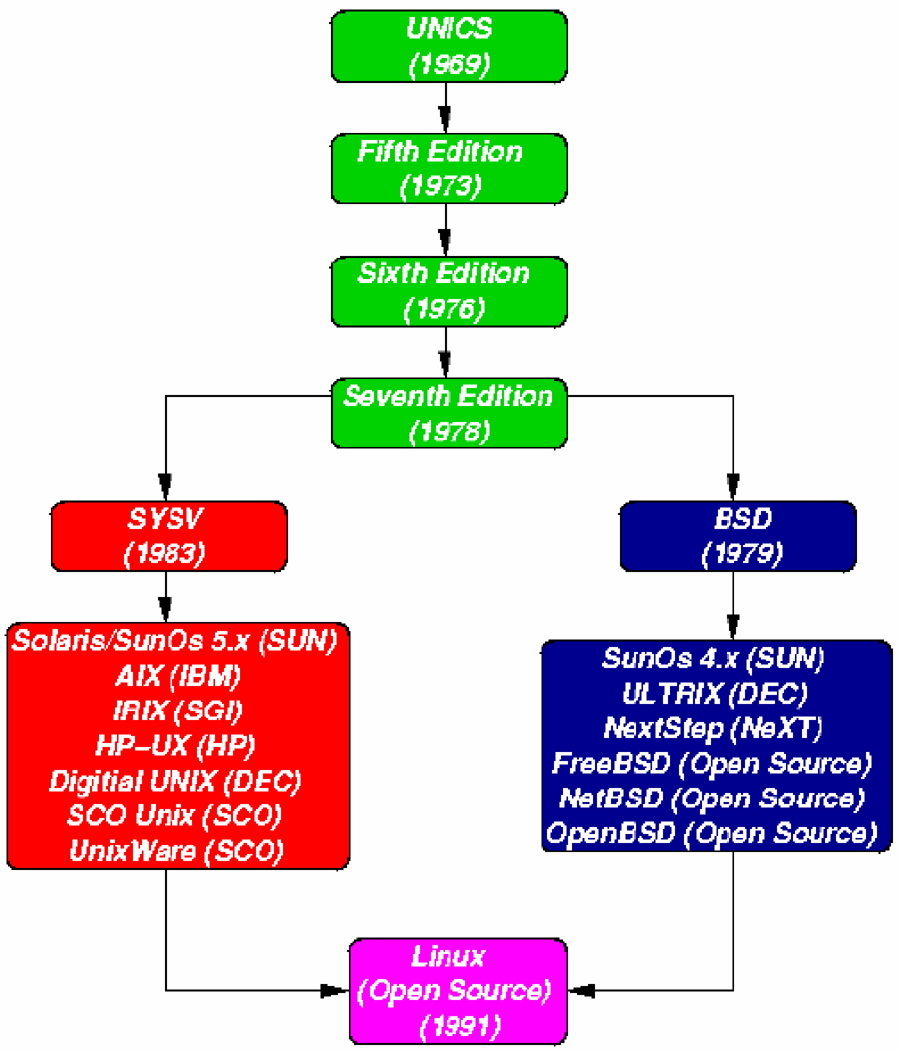
i.e. there isn't ***a*** UNIX,

but multiple implementations

of the UNIX design

(Linux, BSD, Solaris,

AIX, OS X)



Features that made UNIX a hit

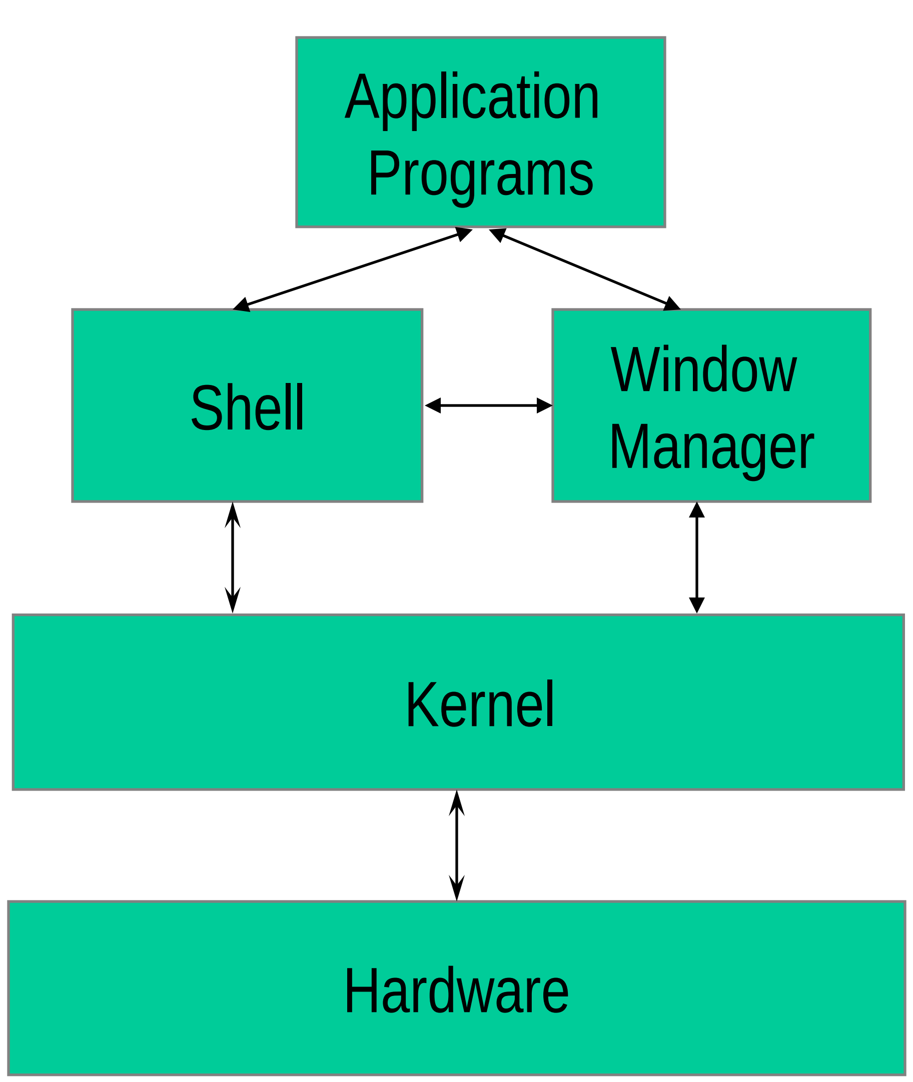
- Multitasking capability
- Multi-user capability
- Portability
- UNIX programs (tools, utilities)
- Library of application software

UNIX philosophies

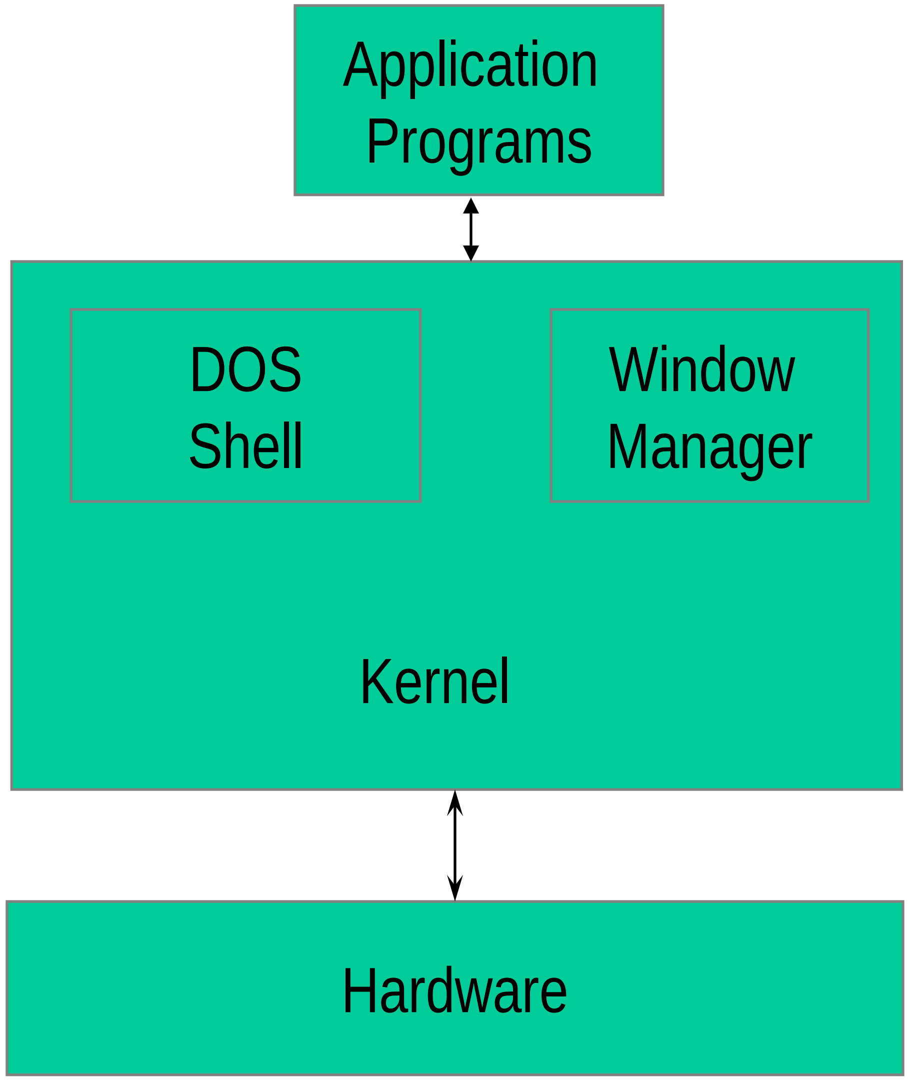
- How software should be written???
  - a program should do one thing only
  - it should do it well
  - complex tasks should be performed by using programs together

    combining small building blocks to make larger one…

UNIX vs. MS Windows architecture

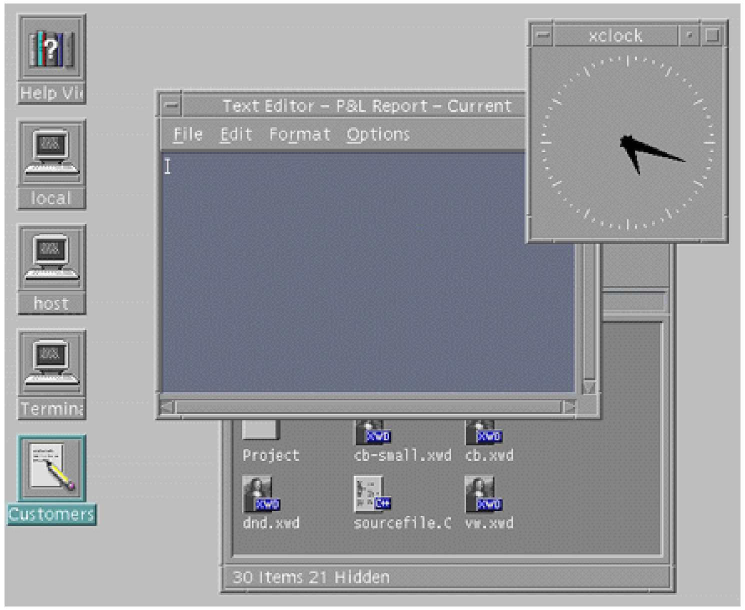


UNIX (minimalist)

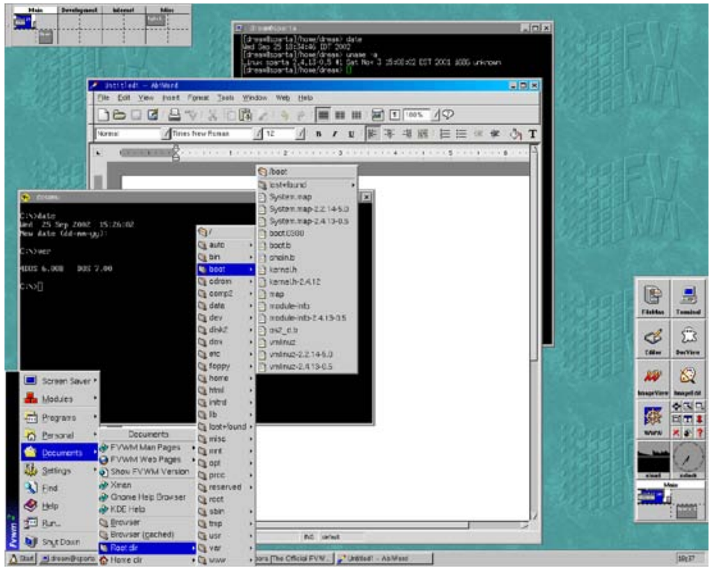


Windows

Why is UNIX better?

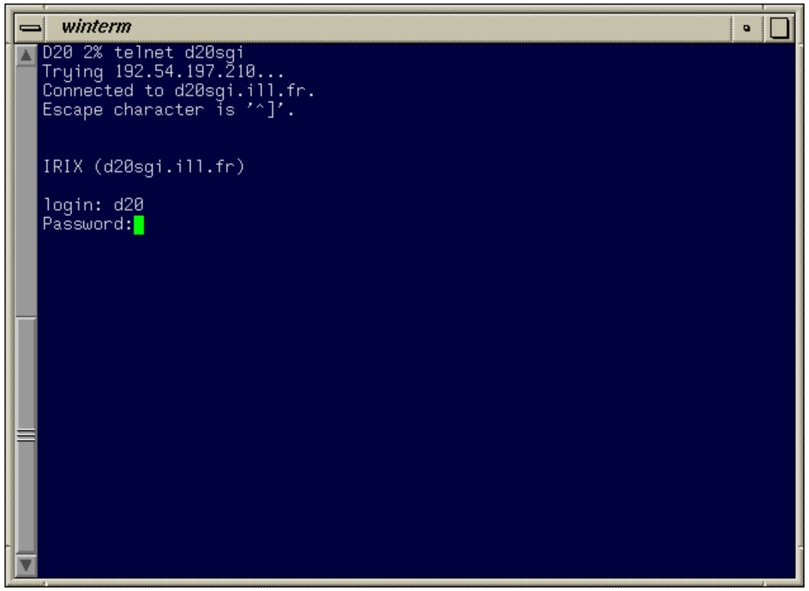


Virtual PC (Red Hat Linux 6.1)
(file:/home/marcrh/)
Netscape: REDHA...
Netscape: REDHAT.COM -- Serving the Linux and Open Source Communities
File Edit View Go Communicator Help
Bookmarks Location: http://www.redhat.com/
What's Related
WebMail Contact People Yellow Pages Download Find Sites Channels
redhat.com
Serving the Linux and open source communities
New... Red Hat Linux 6.2 Available Now! CLICK HERE
Click here for Red Hat Linux 6.2!
Training + Get 6.2! + Product Registration + Downloads + Get Support
support
Sign up for your free redhat.com membership!
Search
redhat.com Linux
sites powered by Google
WIDEOPENNEWS
April 16, 2000
VA Disputes IDC Server Market Findings
Red Hat Store
Support & Downloads
Autostartred_hat_support
Printer
www.redhat.com
Templates
marcrh
Trash
BasiliskII
cdrom
floppy
ed_hat_errata
100%
http://www.redhat.com/commerce
One Three Two Four
11:40PM Apr 16
Full Screen Eject Floppy Eject CD Folders
Floppy Ethernet Hard Drive CD ROM



What is a Shell?

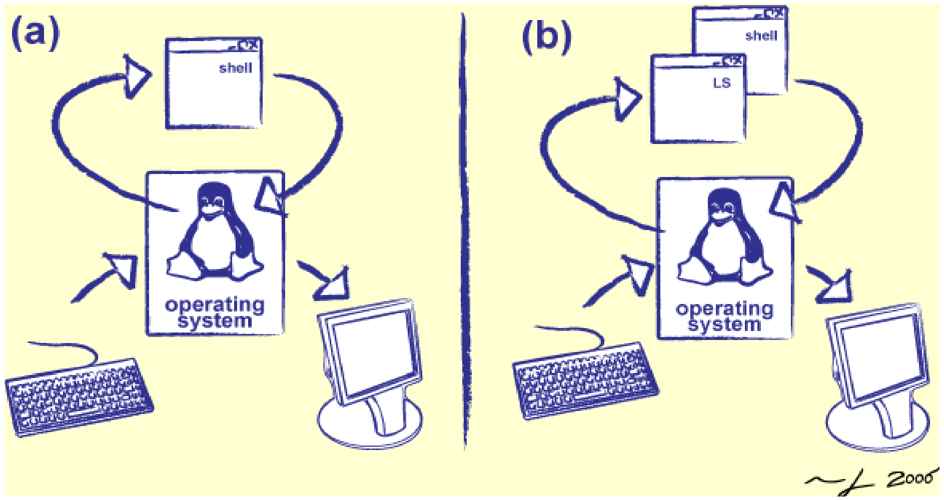
– ... a UNIX shell is a program that accepts and interprets commands and then has the operating system execute them.



```
D20 2% telnet d20sgi
Trying 192.54.197.210...
Connected to d20sgi.ill.fr.
Escape character is '^]'.


IRIX (d20sgi.ill.fr)

login: d20
Password:
```



– It is *not* the operating system

Shell: the Execution Cycle

- When you type a command, the operating system:
  - Reads characters from the keyboard
  - Passes them to the shell
- The shell:
  - Breaks the line of text into words
  - Looks for the program identified by the first word
  - Runs it, passing in the other words as arguments
  - Sends its output to the OS
- The OS displays the output in the current window

Flavors of Unix Shells

- Two main flavors of Unix Shells
  - Bourne (or Standard Shell): sh, ksh, bash, zsh
    - Fast
    - $ for command prompt
  - C shell : csh, tcsh
    - better for user customization and scripting
    - %, > for command prompt

Shell Startup files

- **sh,ksh:**

```
 profile (system defaults)
.profile
```

- **bash:**

```
.bash_profile
.bashrc
.bash_logout
```

- **csh:**

```
 .login:  executed when you logon
 .cshrc:  executed when a new shell is sprawned
 .logout: executed at logout
```

Shell: shell and environment variables

- Shell variable
  - a mechanism that can be used to hold pieces of information for use by system programs or your own use

- Environment variable
  - a variable that is made available to commands as part of the environment that the shell maintains

Shell: creating environment variables

- In the C-shell and its derivatives:
  - **`setenv variable-name value`**

    `e.g.,` **`setenv fruit apple`**

- In the Bourne shell and its derivatives:
  - **`variable-name=value`**

    `export variable-name`

    ```
    e.g.,       fruit=apple
                export fruit
    ```

Shell: creating shell variables

- In the C-shell and its derivatives, the command is:
  - **`set variable-name=value`**
  - `(e.g., `**`set fruit=apple`**`)`

- In the Bourne shell and its derivatives, the command is:
  - **`variable-name=value`**
  - `(e.g., `**`fruit=apple`**`)`

Shell: listing variables

- To list a specific variable
  - (e.g., `echo $fruit`)
- ***set***
  - C-shell - standard shell variables
  - Bourne shell - all shell variables
- ***env***
  - environment variables only
  - C-shell - "***setenv***"

Shell: local commands vs. internal commands

- Local commands
  - such as ***ls, mv*** are part of the operating system
  - run programs as part of that system

- Internal commands
  - such as rehash.
  - Each shell's own set of internal shell commands
  - is what makes each shell different from the rest.

Shell: setting path/PATH

- Example of *adding* to your **PATH**:

`bash`	`export` ***`PATH=$PATH:$HOME/bin:/directory2`***
`tcsh`	`setenv` ***`PATH “${PATH}:${HOME}/bin:/directory2”`***
	`set` ***`path = ( $path $home/bin:/directory2 )`***

- **rehash** lets shell know **PATH** was just updated

Shell: basic editing

keystroke	function on command line
^P	previous line
^N	next line
^A	beginning of line
^E	end of line
^F	forward 1 character
<esc>F	forward 1 word
^B	back 1 character
<esc>B	back 1 word
^D	delete under cursor
^K	delete to end of line
^C	cancel current *CL*, return prompt
^Z	temporary interrupt (can also send to background, slide 38)

Shell: Input/Output Redirection

- Most programs have three I/O streams:
  - *stdin* – standard input
  - *stdout* – standard output
  - *stderr* – standard error.
- They all default to the console ("console" means the keyboard for the input and the screen for the output)

Shell: Input/Output Redirection

- To redirect *stdout* of a program to a file:

  bash:
  ```
  myprogram 1> output.log
  ```
  tcsh:
  ```
  myprogram  > output.log
  ```

- To redirect *stderr* of a program to a file:

  bash:
  ```
  myprogram 2> error.log
  ```

- To redirect both *stdout* and *stderr* to same file (order matters):

  bash:
  ```
  myprogram > combined.log 2>&1
  ```

- To redirect both *stdout* and *stderr* separately:

  ```
  (myprogram >output.log) >&error.log
  ```

Shell: Input/Output Redirection

- Example of “>>” operator to append information to a file:

```
prompt> date > foo

prompt> cat foo
Wed Aug 31 17:27:52 CDT 2005

prompt> date >> foo

prompt> cat foo
Wed Aug 31 17:27:52 CDT 2005
Wed Aug 31 17:27:56 CDT 2005
```

Shell: Input/Output Redirection

function	tcsh	bash
stdout to file	comm >ofile	comm > file
stderr to file		comm 2> file
stdout/err to file	comm >& ofile	comm >ofile 2>&1 comm &> ofile
stdin from file	comm < ifile	comm < ifile
stdout to end of file	comm >> ofile	comm >> ofile
stderr to end of file		comm 2>> ofile
stdout/err to end	comm >>& ofile	comm >> ofile 2>&1
stdin until "c"	comm <<c	comm <<c
redirect stdin/out	comm < ifile > file	comm < ifile > file
stderr to third file		comm < ifile > ofile 2> efile
stdout/err to 2 files	(comm > ofile) >& efile	

Shell: setting variables from a program

- Example of setting a new variable to the output value of a command:

```
> current_date_time=`date`
> echo $current_date_time
Mon Apr 23 14:15:35 CDT 2007
```

- Another example (note **pwd** vs. ***$PWD***):

```
> echo "I am in `pwd` on $HOST"
I am in /home/username on vmps08
> echo "I am in $PWD on $HOST"
I am in /home/username on vmps08
```

I/O Redirection - Pipes/Filters

- Pipelines are a set of processes chained by their standard streams, so that the *stdout* of each process feeds directly as the *stdin* of the next.
- Pipelines are defined using the “|” character.

function	tcsh	bash
pipe stdout to comm2	comm | comm2	comm | comm2
pipe stdout/err to comm2	comm |& comm2	comm 2>&1 | comm2

- E. g., use a pipe and **tee** to direct output of **echo** to both *stdout* and to a file:

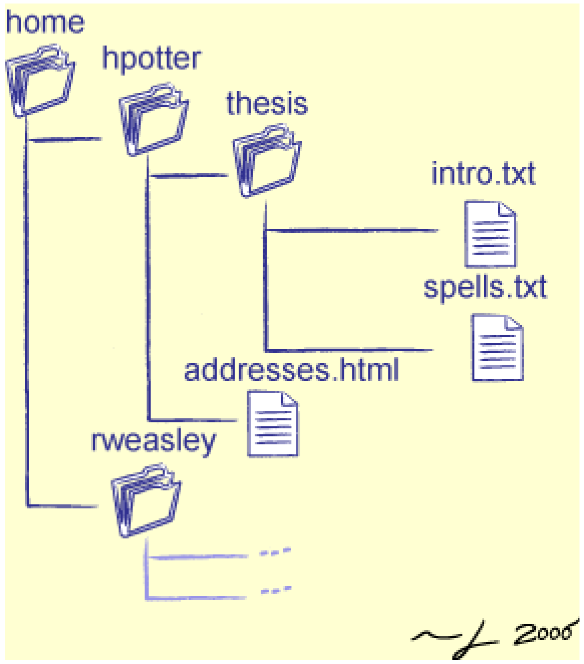
```
prompt> echo "Hello World" | tee output.txt
```

UNIX: files & programs

- file: collection of data
  - UNIX treats everything as a file (including peripherals)
- program:
  - collection of bytes representing code and data that are stored in a file
- process:
  - a program in execution (currently running, loaded from disk into RAM)

UNIX: file system

- The *file system* is the set of files and directories that the computer can access
- "Everything that doesn't go away when the computer is rebooted“
- Data is stored in files
- By convention, filename suffix identifies data type
  - E.g., .txt for text files, .mp3 for sound
  - But it *is* just a convention
  -

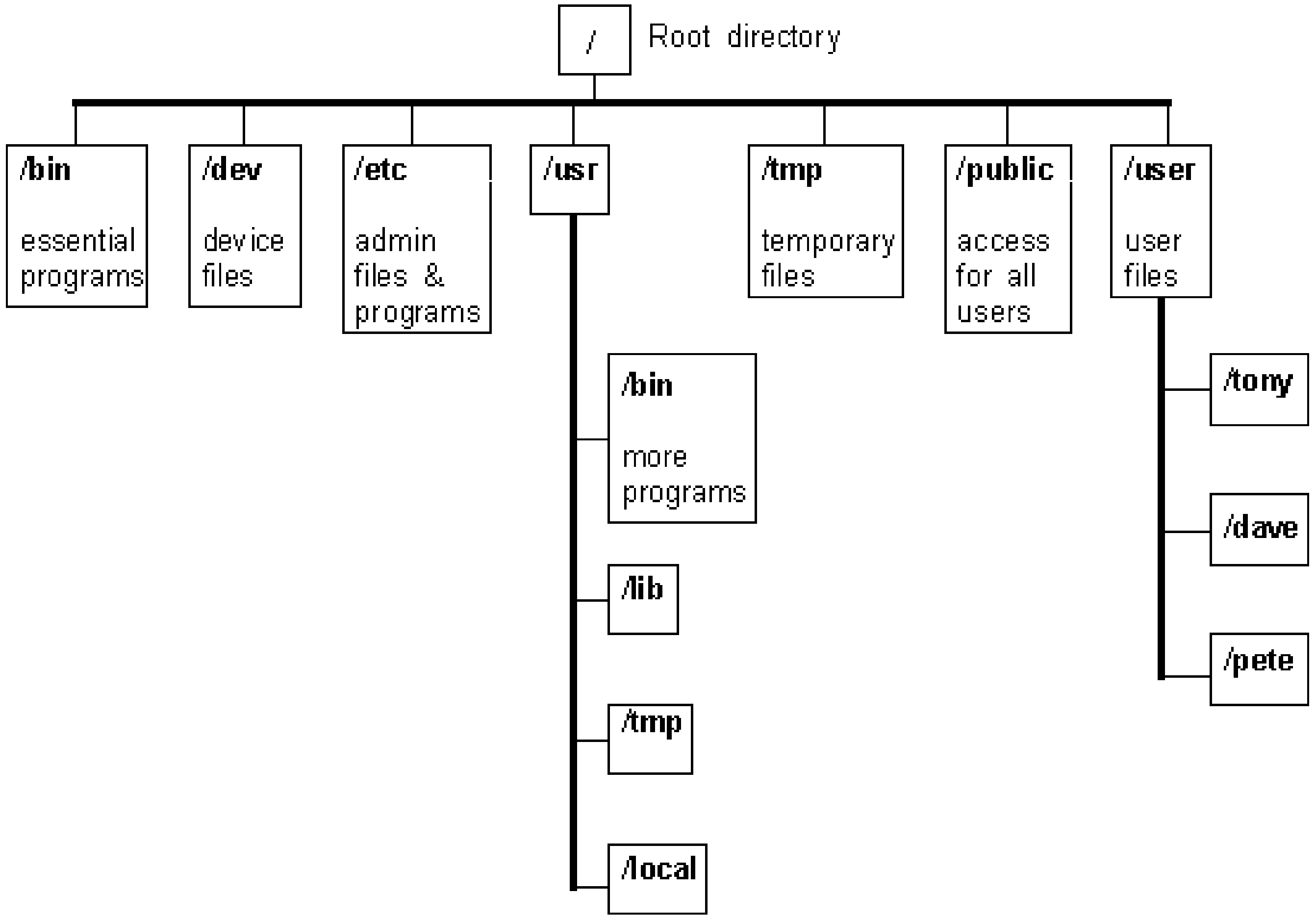


UNIX: file system

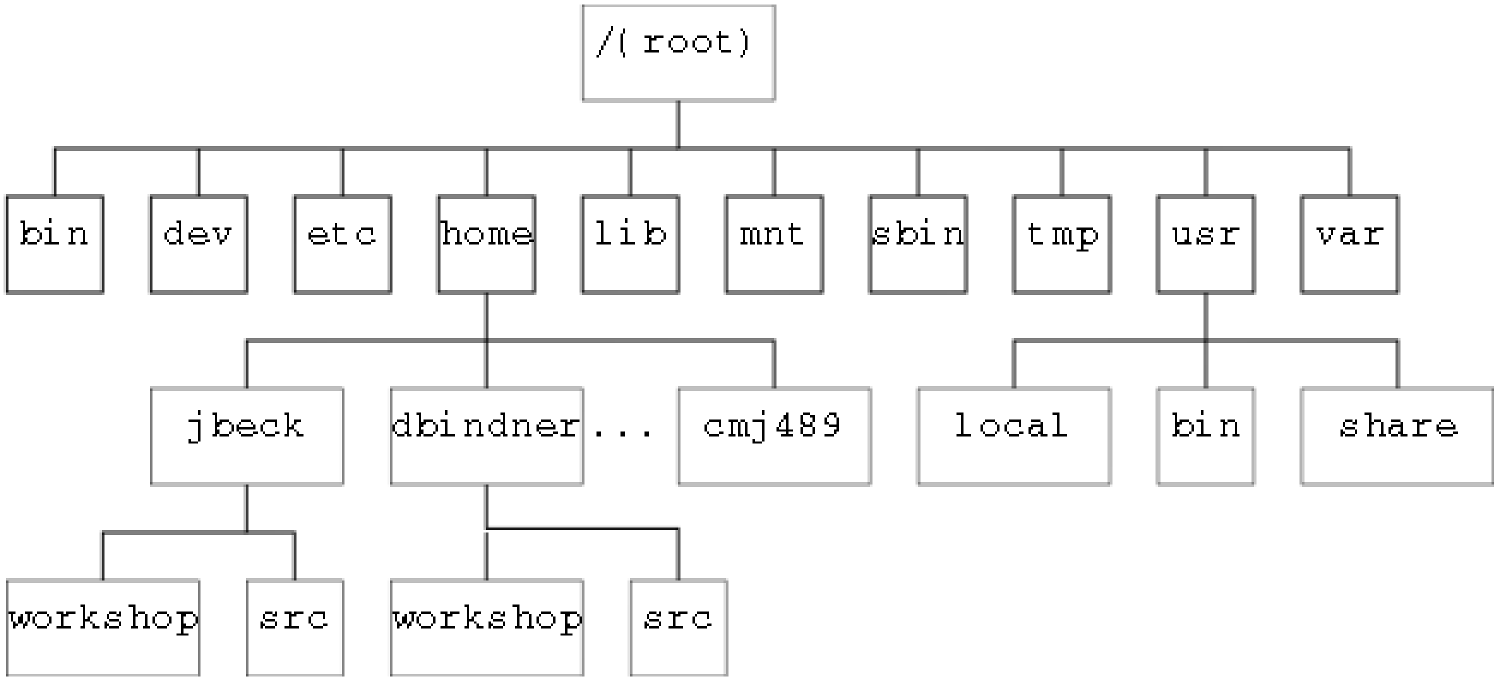
- On Unix, the file system has a unique root directory called / (pronounced "slash")
  - On Windows, every physical drive has its own root directory So C:\home\hpotter\notes.txt is different from D:\home\hpotter\notes.txt

- File and directory names are case-sensitive on most Unix variants, but case-*in*sensitive on Windows
  - E.g., This.txt and this.txt are different files on Unix, but the same file on Windows
  - So you should never rely on casing to distinguish files

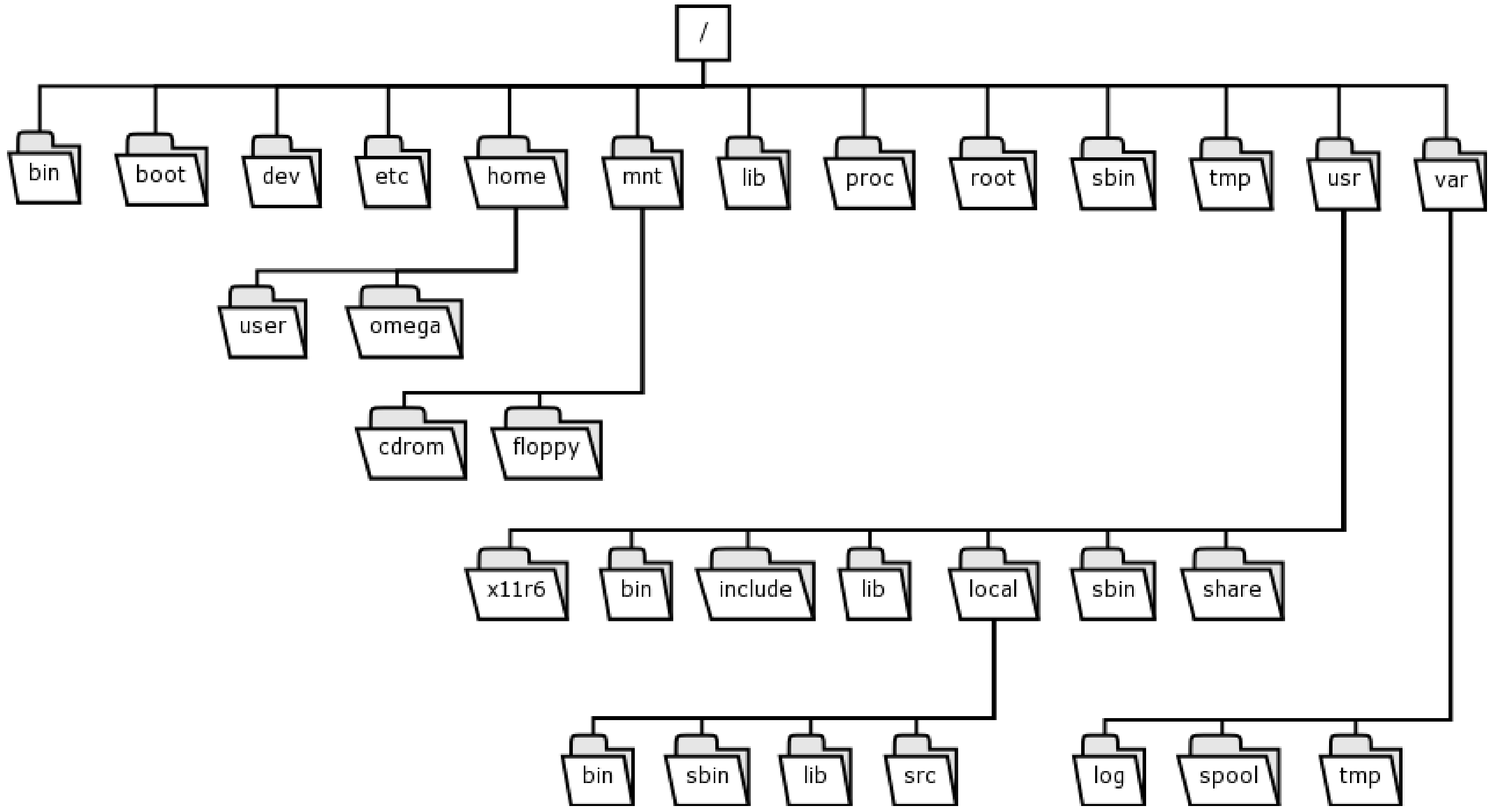
UNIX: file system directories



UNIX: file system directories



UNIX: file system directories



UNIX: file system navigation

- ***pwd*** (**present working directory**) shows the name and location of the directory where you are currently working:

```
> pwd
/u/browns02
```

  - This is a "pathname," the slashes indicate sub-directories
  - The initial slash is the "root" of the whole filesytem

- ***ls*** (**list**) gives you a list of the files in the current directory:

```
> ls
assembin4.fasta  Misc       test2.txt
bin              temp       testfile
```

  - Use the ***ls -l*** (**long**) option to get more information about each file

```
> ls -l
total 1768
drwxr-x---  2 browns02 users   8192 Aug 28 18:26 Opioid
-rw-r-----  1 browns02 users   6205 May 30  2000 af124329.gb_in2
-rw-r-----  1 browns02 users   131944 May 31  2000 af151074.fasta
```

UNIX: file permissions

- Use the ***ls -l*** command to see the permissions for all files in a directory:

```
> ls -l
drwxr-x---  2 browns02   users   8192     Aug 28  18:26      Opioid
-rw-r-----   1 browns02   users   6205      May 30  2000      af124329.gb_in2
-rw-r-----   1 browns02    users   131944 May 31  2000      af151074.fasta
```

  - The username of the owner is shown in the third column. (The owner of the files listed above is browns02)
  - The owner belongs to the group "users"

- The access rights for these files is shown in the first column. This column consists of 10 characters known as the attributes of the file: r, w, x, and **-**

r	indicates read permission
w	indicates write (and delete) permission
x	indicates execute (run) permission
-	indicates no permission for that operation

UNIX: file permissions

```
> ls -l
drwxr-x---  2 browns02   users   8192     Aug 28  18:26     Opioid
-rw-r-----   1 browns02   users   6205     May 30  2000     af124329.gb_in2
-rw-r-----   1 browns02    users   131944 May 31  2000     af151074.fasta
```

- The first character in the attribute string indicates if a file is a directory (d) or a regular file (-).
- The next 3 characters (rwx) give the file permissions for the owner of the file.
- The middle 3 characters give the permissions for other members of the owner's group.
- The last 3 characters give the permissions for everyone else (others)
- The default protections assigned to new files on our system is: -rw-r----- (owner=read and write, group =read, others=nothing)

UNIX: Change Protections

- Only the owner of a file can change its protections
- To change the protections on a file use the ***chmod*** (**change mode**) command.

  [Beware, this is a confusing command.]

  - First you have to decide for whom you will change the access permissions:
    - » **the file owner (u)**
    - » **the members of your group (g)**
    - » **others (o) (ie. anyone with an RCR account)**
  - Next you have to decide if you are adding (**+**), removing (**-**), or setting (**=**) permissions.
- Taken all together, it looks like this:

```
> chmod u=rwx g+r o-x myfile.txt
```

This will set the owner to have read, write, and execute permission; add the permission for the group to read; and remove the permission for others to execute the file named **myfile.txt**.

UNIX: sub-directories

- ***cd*** (**<u>change directory</u>**) moves you to another directory

```
>cd Misc
> pwd
/u/browns02/Misc
```

- ***mkdir*** (**<u>make directory</u>**) creates a new sub-directory inside of the current directory

```
> ls
assembler  phrap     space
> mkdir subdir
> ls
assembler  phrap     space     subdir
```

- ***rmdir*** (**<u>remove directory</u>**) deletes a sub-directory, but the sub-directory must be empty

```
> rmdir subdir
> ls
assembler  phrap     space
```

UNIX: file shortcuts

- There are some important shortcuts in Unix for specifying directories
  - **.** (dot) means "the current directory"
  - **..** means "the parent directory" - the directory one level above the current directory, so ***cd* ..** will move you up one level
  - **~** (tilde) means your Home directory, so ***cd* ~** will move you back to your Home.
    - Just typing a plain ***cd*** will also bring you back to your home directory

Unix: Processes and Multitasking

- To run a program in the background, use the “**&**” character (or “**^Z**” followed by “**bg**”):

```
> myprogram &
[1] 7895
```

- **myprogram** is now running in the background as process id (PID) 7895
- Whenever your process finishes, it will print “Done” to the console.

Unix: Processes and Multitasking

- To check on the status of your jobs running on the system, use the **ps** command

```
> ps -a
 PID TTY          TIME CMD
8095 pts/3    00:00:00 ps
```

- You can get an expanded list by typing **ps agux**, or by using the **top** command
- Use **uptime** to check the load average (how hard system is working) on slowly responding machines

Things You Should Know

cat	cd	clear	cp
date	diff	echo	head
ls	man	mkdir	more
mv	od	passwd	pwd
rm	rmdir	sort	tail
uniq	wc	which	< | >

Text Editors

- Become expert in either one of the following

- *emacs* (very popular)
  - **Has menu bar like MS word along with keyboard shortcuts**
  - **http://www.ucc.ie/doc/editing/emacs.html**

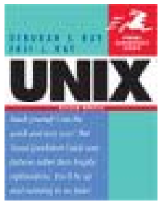
- *vi* (less commonly used) → *vim* is a more robust version
  - **http://www.linux.org/lessons/beginner/l5/lesson5c.html**

UNIX: online help

- man
  - Detailed description of a command
- info
  - More complete descriptions of certain packages
- help
  - Display helpful information about builtin commands
- apropos
  - Search the manual page names and descriptions
- whatis
  - display manual page descriptions

UNIX: Where To Go

- Links from the 207/B07 site – Unix commands
- and a Unix Tutorial for Beginners



Deboray S. Ray and Eric J. Ray: *Unix*. Peachpit Press, 2003, 0321170105.

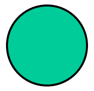
Systematic Testing

Software house: what happens inside?

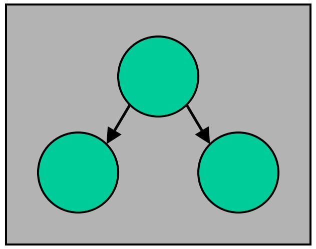
$$\begin{tikzcd}
\boxed{\text{Understand the requirements}} \arrow[d] & \text{Project Managers Group} \\
\boxed{\text{Design the software}} \arrow[d] & \text{Architects Group} \\
\boxed{\text{Write the program}} \arrow[d] & \text{Development/Programmer Group} \\
\boxed{\textbf{Test the program}} \arrow[d] & \textbf{QA Group} \\
\boxed{\text{Write Documentation}} \arrow[d] & \text{Documentation Group} \\
\boxed{\text{Package/Sell/Market}} & \text{Marketing/Sales Group}
\end{tikzcd}$$

Types of Software Testing

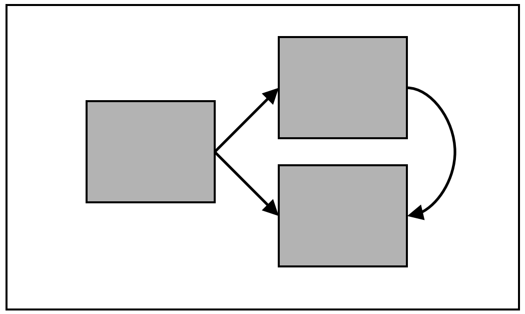
- *Component/unit* testing
  - Individual classes or types



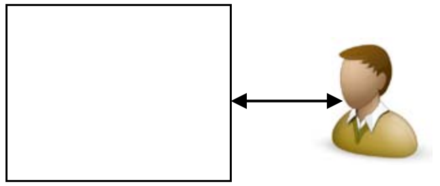
- *Development* testing
  - Also called DIT (development integration testing)
  - Group of related classes or types



- *System* testing
  - Also called SIT (system integration testing)
  - Interaction between classes



- *User Acceptance* testing
  - Also called UAT
  - Interaction between user and program interface



Software Testing Lifecycle

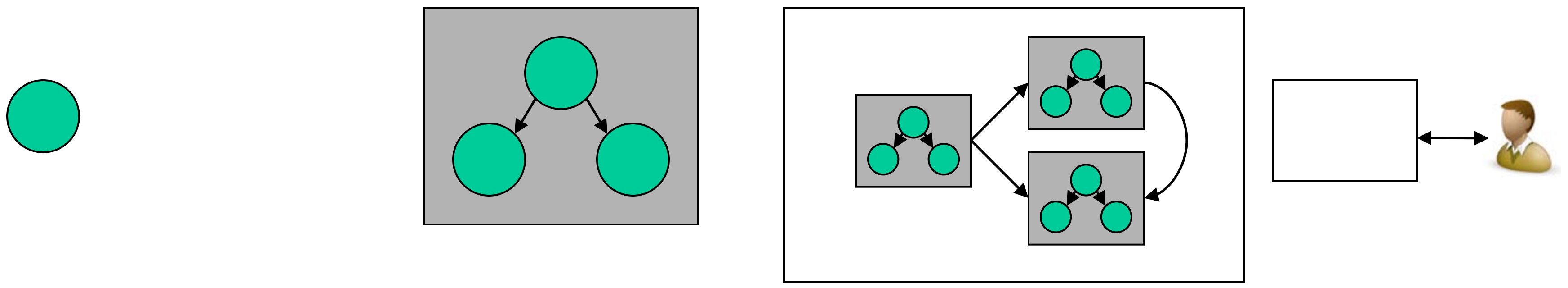
$$\begin{tikzcd}
\boxed{\text{Unit Testing}} \arrow[r, "\text{followed by}"] & \boxed{\text{DIT}} \arrow[r, "\text{followed by}"] & \boxed{\text{SIT}} \arrow[r, "\text{followed by}"] & \boxed{\text{UAT}}
\end{tikzcd}$$

During
coding of a component
By developers

After integration
of various components
By developers/QA

After building the whole
application
By QA mostly

After alpha
release
By QA



Unit testing

- Also called *component testing*
- Testing in isolation *all operations* associated with an object
- Setting and querying *all attributes* (data members) of an object
- Verify a small chunk of code, typically a path through a method or function. Not application level functionality.
- Exercising the object in all possible states
  - boundary conditions
  - both success and failure
  - general functionality

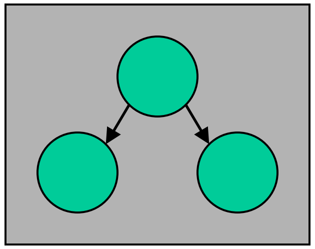
Why is Unit Testing Good?

- Identifies defects early in the development cycle.
- Many small bugs ultimately leads to chaotic system behavior
- Testing affects the design of your code.
- Successful tests breed confidence.
- Testing forces us to read our own code – spend more time reading than writing
- Automated tests support maintainability and extendibility.

Why Don’t We Unit Test?

- “Coding unit tests takes too much time”
- “I’m to busy fixing bugs to write tests”
- “Testing is boring – it stifles my creativity”
- “My code is virtually flawless…”
- “Testing is better done by the testing department”
- “We’ll go back and write unit tests after we get the code working”

Development testing (DIT)



- Testing the interaction of components
- For CSC207H assignments this will usually be the whole program
- For larger systems this may involve sub-components of the system

Regression testing

- This is not a new type of testing; it’s a technique
- Maintain a set of tests
  - For Unit, DIT, SIT and even UAT
- Every time a change is made to the system, run the tests to make sure everything that used to work still does
- When you add a new feature to the system, add new regression tests

Testing Terminology

- Test or test case:
  - A test case describes
    1) an input,
    2) action, or event and
    3) an **expected response**, to determine if a feature of an application is working correctly

Testing terminology

- Fixture:
  - The context of the test
    - *E.g.* a data structure with a set of values
    - In the simplest Java case, a single initialized object. One of the methods of this object will be called.

- Possible results:
  - pass: test produced the expected outcome
  - fail: test ran but produced an incorrect outcome
  - error: test failed to produce an answer: there is something wrong with the test itself

So, how to do Unit testing?

- Unit testing follows a pattern
  - Lots of small, independent tests
  - Reporting passes, failures, and errors
  - Some optional shared setup and teardown (creating the fixture)
  - Aggregation (combine tests into test suites)

JUnit

- JUnit testing framework
  - Written by Erich Gamma in 1997
  - Now hosted at http://www.junit.org
  - Has become the unofficial standard for Java testing
  - Supported by many IDEs
  - Widely imitated: C++, Perl, Python, .NET all have versions
- Once you know one, you can easily use others

Test Hierarchies

- JUnit supports test hierarchies
  - Test Suite-A
    - Test Case1
    - Test Case2
    - Test Suite-B
      - Test Case3
  - Test Suite-C

(and so on …)

Key JUnit Concepts

- assert
- fail
- error

Junit assert/fail methods

- static void assertTrue(boolean ***test***)
- static void assertFalse(boolean ***test***)
- assertEquals(***expected***, ***actual***)
- assertSame(Object ***expected***, Object ***actual***)
- assertNotSame(Object ***expected***, Object ***actual***)
- assertNull(Object ***object***)
- assertNotNull(Object ***object***)
- fail()

Example

```
public class TestAdd extends TestCase {








 public void testPositive() {
  assertEquals(4, 2 + 2);
 }

 public void testNegative() {
  assertEquals((4), (-2) + (-2));
 }
```

Example

```
public class TestAdd extends TestCase {

  public static void main(String[] args) {
    TestSuite suite = new TestSuite();
    suite.addTest(new TestAdd("testPositive"));
    suite.addTest(new TestAdd("testNegative"));
    TestRunner.run(suite);
  }

  public TestAdd(String name) { super(name); }

  public void testPositive() {
    assertEquals(4, 2 + 2);
  }

  public void testNegative() {
    assertEquals((4), (-2) + (-2));
  }
```

Junit testXXX

- A test method doesn’t return a result
- If the tests run correctly, a test method does nothing
- If a test fails, it throws an AssertionFailedError
- The JUnit framework catches the error and deals with it; you don’t have to do anything
- E.g.

```
public void testPositive() {
    assertEquals(4, 2 + 2);
  }
```

Testing for exceptions

- How do you test a case that is supposed to fail by throwing an exception?

- Example 2:

```
public void testFromString() {
   try {
        int x = Integer.parseInt("m") + 3;
       fail();
    } catch (NumberFormatException e) {
        // success
    } catch (Exception e) {
        fail();
    }
}
```

Example 3

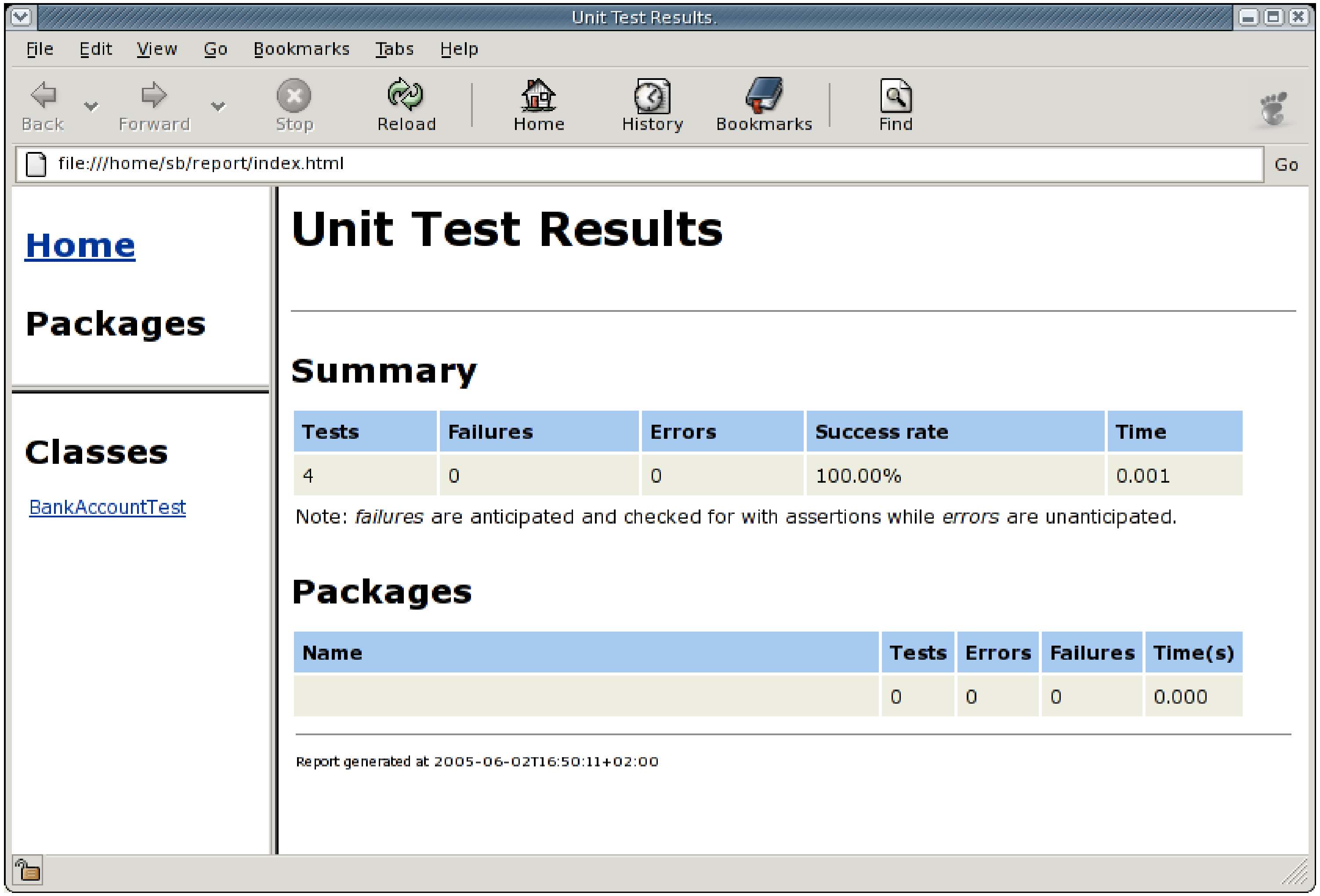
```
import java.util.*;
import junit.framework.*;

public class SimpleTest extends TestCase{

    public void testEmptyCollection() {
            Collection testCollection = new ArrayList();
            assertTrue( testCollection.isEmpty());
    }


    public static void main(String args[]){
            junit.textui.TestRunner.run(SimpleTest.class);
    }
}
```

JUnit Report


Unit Test Results.
File Edit View Go Bookmarks Tabs Help
Back Forward Stop Reload Home History Bookmarks Find
file:///home/sb/report/index.html
Go

Home

Packages

Classes

BankAccountTest

Unit Test Results

Summary

Tests	Failures	Errors	Success rate	Time
4	0	0	100.00%	0.001

Note: *failures* are anticipated and checked for with assertions while *errors* are unanticipated.

Packages

Name	Tests	Errors	Failures	Time(s)
	0	0	0	0.000

Report generated at 2005-06-02T16:50:11+02:00

Junit under the hood..

- The goal of a unit testing framework is to run tests
  - JUnit contains an interface, Test, that defines a method run()
- Two classes implement Test
  - TestCase for a single class
  - TestSuite for a set of tests, possibly in multiple classes
- Test methods are placed in a class that extends TestCase
- TestCases can be added to a TestSuite

Example

```
public class TestAdd extends TestCase {

  public static void main(String[] args) {
    TestSuite suite = new TestSuite();
    suite.addTest(new TestAdd("testPositive"));
    suite.addTest(new TestAdd("testNegative"));
    TestRunner.run(suite);
  }

  public TestAdd(String name) { super(name); }

  public void testPositive() {
    assertEquals(4, 2 + 2);
  }

  public void testNegative() {
    assertEquals((4), (-2) + (-2));
  }
```

Set up and tear down

- The run method has three steps:

  -
    ```
    public void run() {
        setUp();
        runTest();
        tearDown();
    }
    ```
- By default, setUp and tearDown do nothing
- Override setUp and/or tearDown to include test fixture creation and clean-up that is common to a number of tests

Creating a test class in JUnit

1) Define a subclass of TestCase
2) Override the setUp() method to initialize object(s) under test.
3) Override the tearDown() method to release object(s) under test.
4) Define one or more public testXXX() methods that exercise the object(s) under test and assert expected results.
5) Define a static suite() factory method that creates a TestSuite containing all the testXXX() methods of the TestCase.
6) Optionally define a main() method that runs the TestCase in batch mode.

Guidelines for test case creation

- Test for success
  - general cases
  - well-formatted input
  - boundary cases
- Test for failure
  - invalid input
  - will it throw the exceptions it is supposed to?
- Test for sanity
  - if there is redundant information make sure it is maintained
  - data structure consistency

Example: Counter class

- For the sake of example, we will create and test a trivial “counter” class
  - The constructor will create a counter and set it to zero
  - The increment method will add one to the counter and return the new value
  - The decrement method will subtract one from the counter and return the new value

Example: Counter class

- We write the test methods before we write the code
- Don’t be alarmed if, in this simple example, the JUnit tests are more code than the class itself!

JUnit tests for Counter

```
public class CounterTest extends junit.framework.TestCase {
    Counter counter1;

    public CounterTest() { }   // default constructor

    protected void setUp() {   // creates a (simple) test fixture
        counter1 = new Counter();
    }

    protected void tearDown() { } // no resources to release
```

JUnit tests for Counter…

```
    public void testIncrement() {
        assertTrue(counter1.increment() == 1);
        assertTrue(counter1.increment() == 2);
     }
    public void testDecrement() {
        assertTrue(counter1.decrement() == -1);
    }
}       // End from last slide
```

The Counter class itself

```
public class Counter {
        int count = 0;
        public int increment() {
             return ++count;
        }
        public int decrement() {
             return --count;
        }
         public int getCount() {
              return count;
         }
}
```

Read Junit FAQ

http://junit.sourceforge.net/doc/faq/faq.htm