

**Software Engineering**

Computer Science – concerned with theories and methods which underlie computers and software systems.

Software Engineering – concerned with practical problems of producing software.

Software Qualities:

- Correctness
- Reliability
- Robustness
- Performances
- User-friendliness
- Verifiability
- Maintainability
- Reusability
- Portability
- Understandability
- Interoperability
- Productivity
- Timeliness
- Visibility

The Software Crisis

As a user, you want it to do everything, as a customer you don't really want to pay for it, and as a producer you realize how unrealistic the customers are. Requirements will conflict in functionality vs affordability, and in completeness (get everything in) vs timeliness (meet the deadline).

Solving the Software Crisis:

- Well-managed teams
- Clear processes and procedures with documentaion.
- Set deadlines and deliverables
- Communication (client-designer, designer-programmer, between programming teams)
- Use existing work as much as possible.

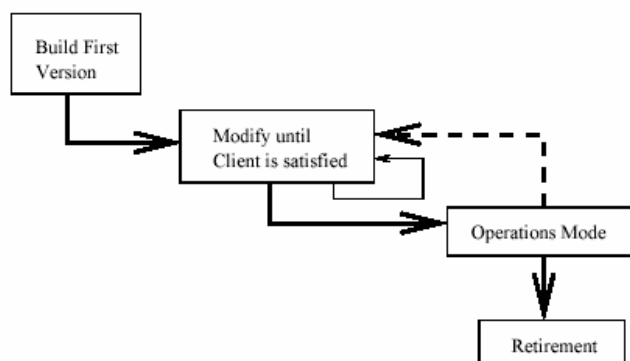
Software Lifecycles

Standard phases:

- Requirements, Analysis, Specification (difficulties include client asking for the wrong product, a computer/software illiterate customer, ambiguous specification)
- Design (system structure – decompose software into modules, module specifications)
- Implementation, Integration
- Validation, Testing
- Operation, Maintenance (difficulties arise with lack of up-to-date documentation, personnel change)
- Change in requirements

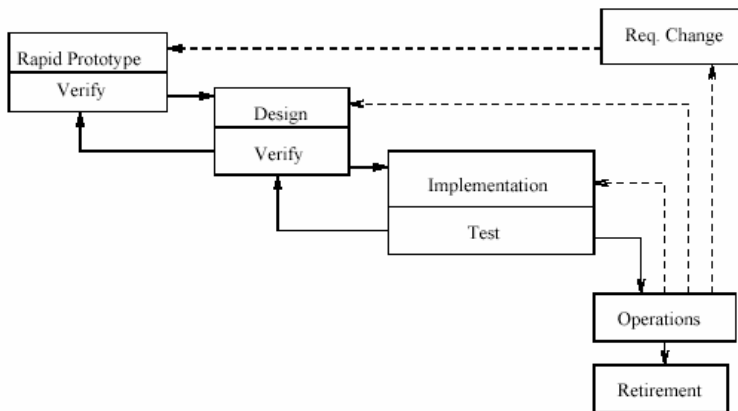
*Code and Fix (NOT a proper life-cycle)*

- Useful for “hacking”
- No process for versioning, testing, etc.
- Difficult to coordinate work of multiple programmers
- Programmers and Users do not understand each other.



Revision Notes

*Waterfall Model*

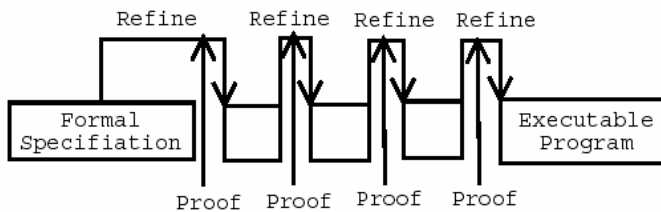


- Measurable progress
- Documentation of each phase
- Commitments are made early and are difficult to change
- Users don't see the system until the end.

*Rapid Prototyping*

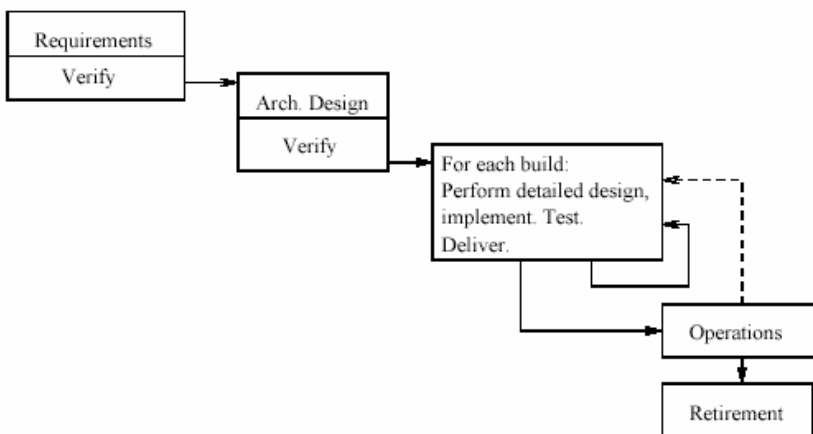
- Prototypes are used to develop requirements
- Prototypes discarded once design begins
- Some criticisms of waterfall model apply.

*Formal Systems Development*



- Mathematical specification
- Fill in details proving each refinement matches specification
- Requires expert mathematics.
- Lengthy
- Hard to apply to complex systems
- Used for safety critical or secure applications

*Incremental Model*

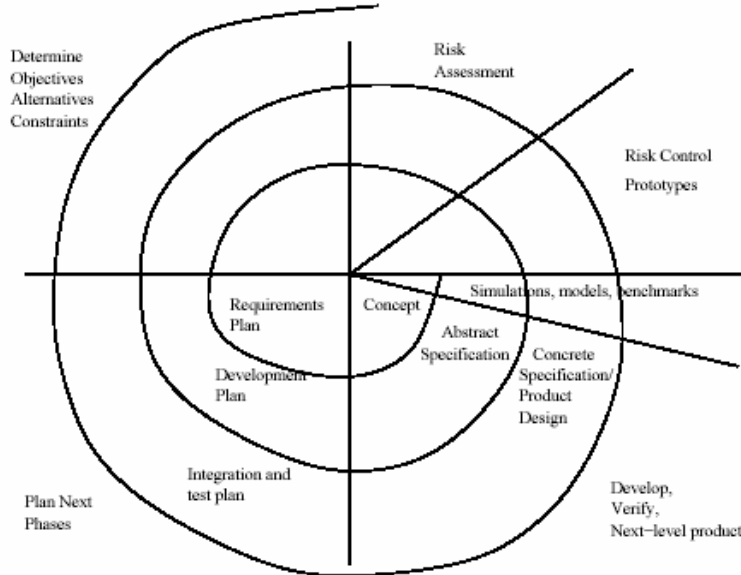


- Used by Microsoft
- Programs built everyday by the 'build manager'
- Last person to check in code that breaks the build becomes build manager.
- Iterations classified according to features

Revision Notes

*The Spiral Model*

- Each iteration driven by 'risk management'
- Depending on risk assessment an appropriate model (waterfall, rapid prototype, etc.) can be used for the next iteration.



Testing

*How to check your program works:*

Proof – mathematics to show specifications match requirements

Refinement – generate code from specification

Testing – run code on test cases

A test suite (collection of test cases) should be complete and precise.

Functional testing (Black box testing) – is not concerned with how the code is written – tests all functions:

- Select categories of input
- Test boundary conditions and typical input in each category.

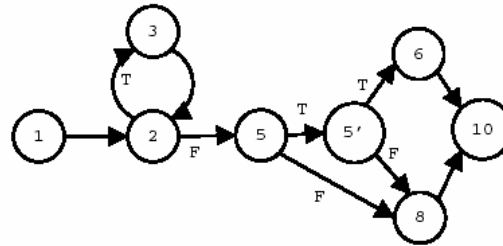
Structural testing (White box testing) – tests all code:

- Based on Control Flow Graphs (CFG): Node is a statement, edge is the ability for a program to flow between statements.
- Example:

```

1 public static int Ptest (int x, int y) {
2     while (x > 9) {
3         x = x - 10;
4     }
5     if (y < 20 && x % 2 == 0) {
6         y = y + 20;
7     } else {
8         y = y - 20;
9     }
10    return x + y;
11 }

```

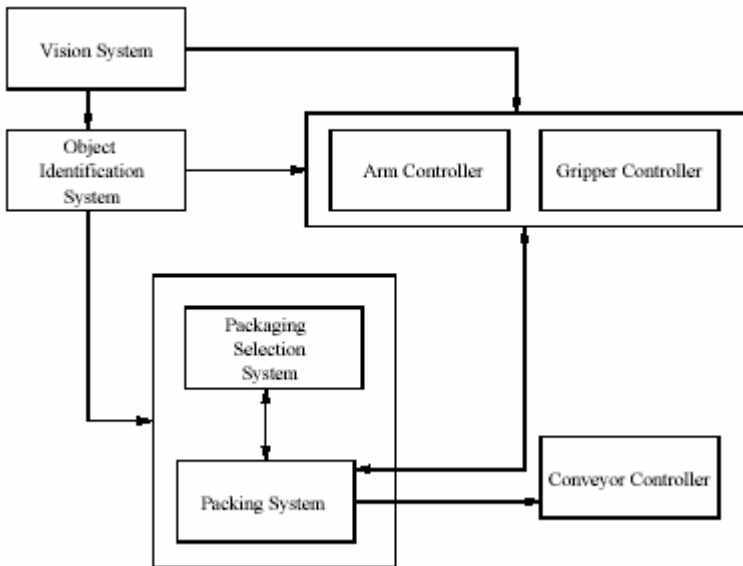


- Statement coverage: every node traversed at least once
- Edge coverage: every edge traversed at least once
- Condition coverage: binary operators evaluated in every possible ways
- Relational coverage: for <, >, <=, >=, the equal condition is treated as a separate branch
- Path coverage: every path traversed at least once

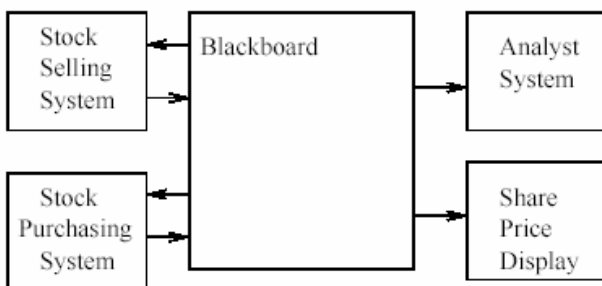
Design

Design documentation: Identification, Type, Purpose, Function, Subcomponents, Dependencies, Interface, Resources, Processing, Data.

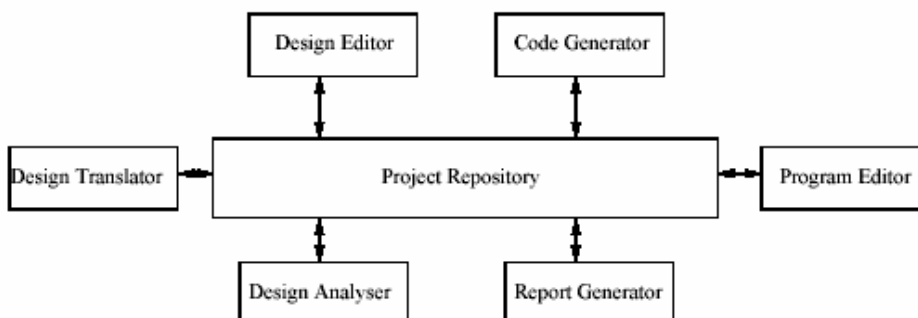
*Pipeline Architecture (e.g. Robot Packing Control System):*



*Blackboard Architecture (e.g. Stock Brokerage System):*

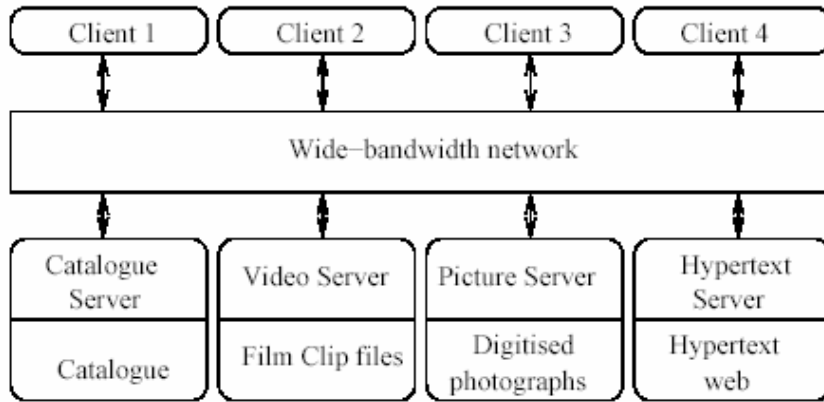


*Repository Architecture (e.g. CASE toolset):*

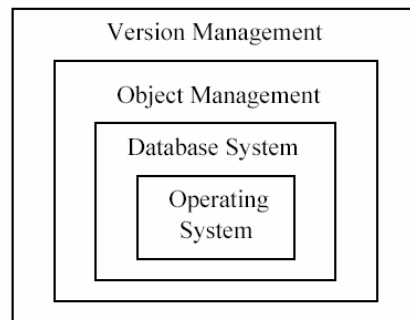


Revision Notes

*Client-Server Architecture (e.g. Film/Picture library system):*



*Abstract Machine Architecture (e.g. Version Control system):*



Other than diagrams, designs can be communicated through Unified Modeling Language, Pseudo-code, etc.

## Revision Notes

### Documentation

Two types: one for users, another for yourself and other programmers.

Common mistakes: being too technical, assuming things are obvious that are not, skipping detail.

Useful tips: include many examples, get someone unfamiliar with the program to read through the manual, keep the manual constantly up-to-date.

#### *Documentation for yourself/other programmers:*

- Code should be easy to read – coding standards
- Code should be well organised
- Comments should be up-to-date.
- Should be possible to find out about a function or variable with analyzing the code.

Code standards: naming conventions, commenting, common method types, indentation, curly braces, access modifiers, etc.

#### *JavaDoc*

API documentation support package. Parses declarations and documentation comments to produce HTML pages. Runnable on packages or individual source files. Produces complete set of HTML pages each time it is run.

Documentation comments start `/**` followed by a return. Subsequent lines start `*` (indented by a space to line up). Insert a blank line between description and any tags. End comment with `*/`

Comments should appear immediately before a class, interface, constructor, method or field declaration. They may include HTML markup. First sentence should be a summary.

To create the HTML, type `javadoc <package> <files>` at the command line (`-d <directory>` specifies destination for the HTML files).

Tags (specially formatted information): begin with `@`. Must begin on new line.

`@author <text>`,

`@param <parameter name> <description>` (only used before a method definition),

`@version <text>` (used before class definition. ignored by javadoc unless `-version` command line argument

is used),

`@return <description>` (only used before method declaration),

`@see <classname>` ('See Also' – used before classes, methods or fields),

`@see <fullclassname#methodname>`.

`@exception <classnameOfException> description` (used before method),

`@deprecated <explanation>`

`@since <version>`



## Revision Notes

### Debugging

- Finding faults in a program after failure has been detected.

Methods: using print statement to print out the state of variables during execution of the program.

### Debuggers:

- Interpretive – reads a program and simulates it's action a line at a time. Easier to program, safer.
- Direct execution – runs the actual program in a special mode where the debugger can read and write the program's memory. Faster, more accurate.

Used either 'line-at-a-time' or with 'breakpoints'.

JDB is the java debugger. Run as 'jdb <classname> <arguments>'. To use the debugger, you must compile the code using the -g option.

- > stop in Hanoi.main – specifies that the debugger should stop at this point.
  - > run – starts the debugger's execution
  - > list – shows the code at the point where the debugger currently is (using line numbers). Can take argument of an integer number of lines to show either side of current line or can take an argument of a method name and will list the 10 lines around that method.
  - > step – steps through one line at a time
  - > locals – shows values of variables at that time
- Some other commands:
- stop in <class>.<method/line> - sets breakpoint
  - clear <class>.<method/line> - removes breakpoint
  - step up – execute until current method returns to its caller
  - print <variable> - prints variable's contents
  - help
  - quit/exit

## Revision Notes

### eXtreme Programming

Lightweight methodology for small-medium teams developing software in the face of vague or rapidly changing requirements.

*Values* - Communication, Simplicity, Feedback, Courage.

*Principles* - Rapid feedback, assume simplicity, incremental changes, embrace change, quality work.

*Activities* - Coding, Testing, Listening, Designing.

#### *Practices*

The Planning Game: User write 'stories' to describe functionality of the system.

Developers estimate time for each story

A story that is too big to estimate is called a 'spike'. Then the developers can plan time to look into the spike.

Users can split, merge and partition their stories

Next release planned

Small releases            Every release as small as possible. Make sense as a whole

Simple designs

Enables rapid feedback, sense of accomplishment, reduced risk, customer confidence and adjustments to changing requirements.

Metaphor            Guide the program with a simple Metaphor (shop, board game, etc)

Gives everyone a coherent viewpoint

Alternative to formal architecture

Simple Design

Runs all test

No duplication

States every intention to the programmers

Fewest possible methods and classes

Testing

Test everything that might possibly break, all the time

The tests *are* the specifications

Tests written before coding

- Functional tests (specified by user, automated, run daily, part of spec)

- Unit tests (written by developers, always run at 100%)

Refactoring

Improve structure of code

Small steps

Supported by unit tests

Pair Programming

One partner uses keyboard/mouse and thinks about implementation

The other thinks about the overall approach, tests, simplicity, etc.

Roles can switch

More productive

Collective Ownership

Anyone can change code (must pass unit tests)

Everyone is responsible for code. Whoever finds a problem solves it.

Continuous Integration

Regular integration

All tests pass

40-hour Week

If it takes more than 40 hours, you're working too hard.

You must be fresh to tackle problems.

On-site customer

Writes functional tests

Makes priority and scope decisions

Answers questions

Coding standards

Code looks uniform

No complicated constructions

No need to reformat code

XP requires Pair programming, Rapid increments, constant testing and an on-site customer. In return you don't have to put comments in the code, write formal requirements and write design documents.



## Revision Notes

### Unix

#### *Some Commands*

apropos <keyword> – locate commands by keyword lookup

find <starting directory> -name <filename> - print

grep <pattern> <files>

cat <filename> - dumps file to screen

more <filename> - shows file one screen at a time

less <filename> - similar to more. Can scroll backwards using 'k' and forwards using 'j'

head <filename> - displays first 10 lines

tail <filename> - displays last 10 lines

#### *Compression*

compress/uncompress - .z extension, old algorithm

gzip/gunzip - .gz extension, recent and free algorithm

zip/unzip - .zip extension, based on PC algorithm

#### *Archiving*

- Tidies up filespace
- Useful for transporting directories to other people
- Often used for distributing Unix software.

tar options:

c – create new tarball

t – retrieves an index of the tarball

x – unpacks a tarball

v – verbose mode

f – next argument is the name of the file

#### *Processes*

Every time a command is run, it creates a process. Putting '&' after a command puts it in the background.

top shows the processes currently running on the CPU.

top commands: q – quit; k <PID> - kill process; U, return, <username>, return – see processes just for one user.

Can also kill from the command line using kill. PID number can be found out using ps.

Typing nice -n <priority> before a command runs it at a specified priority (-20 is the highest, 20 is the lowest).

#### *Input and Output*

STDIN – standard input (keyboard)

STDOUT – standard output (screen)

STERR – standard error output

> - Send output to specified file (overwrite)

>> - Send output to specified file (append)

< - Send command input from file

\$ program1 > temp

\$ program2 < temp

is equivalent to program1 | program2. This is called piping.

## Shell

Location: `#!/usr/bin/sh`

File must be executable to run without specifying 'sh' first.

Arguments are kept in \$1, \$2, \$3 etc...

Print to the screen using 'echo'

Conditionals: `if`  
                  condition  
          `then`  
                  statements  
          `else`  
                  statements  
          `fi`

Conditions: `if test 2 -ge 1` returns true (`2 >= 1`)  
other operators, = (`-eq`), > (`-gt`), >= (`-ge`)

Loops: `for IDEN in ...`  
      `do`  
          statements  
      `done`

e.g., `for FILE in *.sh`  
      `do`  
          `echo "$FILE\n"`  
      `done`

Environment Variables: `PATH`, `TERM` (type of terminal – `xterm`), `HOME`  
`echo $PATH` prints out your path.

To change path in one window: `export PATH = $PATH:new_location`

To change it permanently, modify your `.profile` file.

To fetch input from the keyboard, use `read VAR`, e.g,

```
#!/usr/bin/sh
read VAR
echo $VAR
```

`read` can be used with multiple arguments: `read ONE TWO` will bind the first word to `$ONE` and the rest to `$TWO`.

## *Installation and Make*

The command, 'make', looks for a file called `makefile` or `Makefile`.

Make looks in the `makefile` for a target (typically, 'clean' or 'all')

N.B. Make assumes variables are one letter long, so parentheses are needed `$(VAR)` not `$VAR` (this would be interpreted as `$(V)AR`).

Automatic variables: `$@` (target of the rule), `$<` (the first dependency), `^` (all the dependencies), `$?` (all dependencies newer than the target), `*$*` (stem of a pattern matching rule).



Revision Notes

Version Control and CVS (Concurrent Versions System)

Version control – changes to a single file are tracked via a graph. Nodes are distinct versions, edges denote that a target node was derived from a source node.

Forward deltas – original version stored, subsequent versions stored as sets of changes.

Backwards deltas – most recent version stored, previous versions stored as sets of changes.

The environment variable CVSROOT must be set to the location of the CVS repository.

cv`s checkout <module>` - checks out the files into a directory with the module name in the working directory.

cv`s diff -c` – shows the changes you have made

cv`s commit <filenames>` - commit files to the repository (if they do not already exist in the repository, use

cv`s add <filenames>`).

cv`s update` – checks out the most recent version out of the repository.

## Revision Notes

### Perl

#### Printing and Strings

The following all print 'Hello World!':

```
$1 = "Hello"  
$2 = "World"  
print "Hello World!"  
print "Hello", "World!"  
print $1, $2  
print "$1 $2"
```

```
print ${1}World prints 'HelloWorld'
```

Quotes around strings can be replaced by qq\*...\* where \* can be any character. Single quotes take literal values. Double quotes evaluate their contents. If you want to use a " or ' in a string delimited by that character, use \" or \'

#### Operators

'x' can be used to repeat strings – print "Happy" x 2 prints out "Happy Happy Happy"  
'.' can be used to concatenate strings.

chop(\$string) shortens \$string by one character. *Returns* the removed character.  
chomp(\$string) removes trailing input record separators, such as '/n'. Returns 1 if successful.  
chop and chomp cannot be used on actual strings – only on variables that contain them.

length(string) returns the length of the string.  
substr(string, 0, 3) returns the first three letters of \$string.

```
join ':' "one" "two" "three" returns one:two:three
```

#### *Boolean Operators:*

Numbers:	Strings:
<	lt
>	gt
==	eq
!	not
&&	and
	or

if-then-else can be represented by ? :, e.g., \$b = (\$a < 5) ? 0 : \$a  
(there is also the option of using unless: unless (\$a >= 5) { print "Hello"; }

Single statements can be followed by the condition:            print "Hello" if \$a < 5;  
   \$a = \$a++ while a < 5;

In Perl, false is represented by 0 (or "") and true by anything else.

## Revision Notes

### Loops

while (condition) {	until (condition) {	do {	for (\$i=0; \$i < 10; \$i++)
...	...	...	{
}	}	} while (condition);	...
			}

'last' breaks out of the loop.  
'next' forces the next iteration.  
'redo' repeats current iteration of the loop.

die ("message") sends a message to the Standard Error. System errors are kept in \$!.

### Data Structures

#### *Lists*

Comma-separated sequences of values (usually, but not always, enclosed in parentheses). Do not have to be the same datatype.

'Obvious' shorthands can be used, e.g., 1..8 or 'A'..'H'  
(\$a, \$b, \$c) = (1, 2, 3)

#### *Arrays*

Arrays are lists with names (preceded, not by \$, but by @)  
Individual elements accessed as \$array[2] (n.b. scalar variable)  
Arrays are always 'flattened out', e.g.,  
  @one = (1, 2, 3)  
  @two = (a, b, @one, c) = (a, b, 1, 2, 3, c)

A scalar variable takes the last element of a list assigned to it, so \$one = 3.  
A scalar variable assigned to an array takes the size of this array: \$size = @two = 6  
Array elements can be set as well as accessed using the \$array[2] notation:  
  \$var = \$array[2] (get); \$array[2] = \$var (set)

@foo = (1,2,3,4,5); @foo[0..2] = (4,5,6); so now @foo = (4,5,6,4,5); This is called a *slice*.  
@foo[0,2,4] = ("hop", "skip", "jump"); so now @foo = ("hop", 5, "skip", 4, "jump");  
@foo[0, 1] = @foo[1,0]; so now @foo = (5, "hop", "skip", 4, "jump");

```
foreach $item (list) {  
  ...  
}
```

reverse list – returns a new list – the reverse of it's input.  
sort list – returns a new list – the sorted version of the input.  
x – creates an array of repeated items, e.g. @hello = ("hello") x 5

chop and chomp applied to a list/array of strings will affect every element.

splice – removes a slice from an array. @list = (2,4,6,8,10);

```
splice (@list, 3);  
so @list = (2,4,6);
```

```
splice (@list, 2, 2); # selects the first 2 elements, then removes the next two.
```

## Revision Notes

so @list = (2,4,10);

@new = (7,6);

splice (@list, 2, 2, @new); # selects the first 2 elements, then replaces the next two with @new.

so @list = (2,4,7,6,10);

splice (@list, 2, 1, 7, 6); # selects the first 2 elements, then replaces the next one with 7, 6. @new.

so @list = (2,4,7,6,8,10);

splice *returns* the portion of the list that is removed.

@list = (1,2,3);

push – adds element onto the end of an array – push @list, (4,5,6); so @list = (1,2,3,4,5,6).

*Returns* the length of the new array.

pop – takes the tail value off – pop @list; so @list = (1,2).

*Returns* the removed element.

unshift – adds elements onto the front of an array – unshift @list, (-1, 0); so @list = (-1, 0, 1, 2, 3);

*Return* the new list's length.

shift – takes the head value off.

*Returns* the removed element.

map – performs operation of each element of a list. Returns the new list.

@singular = ("cat", "dog", "rabbit", "hamster");

@plural = map \$\_."s", @singular;

Arguments to a perl script are automatically stored in @ARGV.

### Hashes

A hash is an array in which each element contains a key and an associated value.

The key is a string in curly braces. Quotes not needed if no spaces.

Hash names start with %.

Creating hashes:

- Alternating key and value: %fruit = (banana, yellow, apple, green, orange, orange);

- or %fruit = (  
    banana => yellow,  
    apple => green,  
    orange => orange);

The value of %fruit{apple} is "green".

You can change and create new (key, value) pairs in the same way: %fruit{apple} = "red" and

%fruit{grapes} = "green"

A *slice* of a hash is an array of values specified by a list of keys:

%fruit { "apple", "banana" } is equal to ("green", "yellow")

keys %fruit *returns* a list of all the keys, i.e. ("banana", "apple", "orange")

values %fruit *returns* a list of all the values, i.e. ("yellow", "green", "orange")

each %fruit *returns* successive value pairs. Returns next one each time it's called.

delete \$fruit{key} deletes the corresponding pair. *Returns* the value of the pair it deleted (N.B. scalar variable type).

exists \$fruit{key} *returns* true if it exists (N.B. scalar variable type).

## Revision Notes

reverse %fruit *returns* a new hash with the keys and values swapped around.

N.B. % has a special meaning in print statements, so print "%fruit" will print fruit. print %fruit will print key, value pairs.

### File Handling and Launching Programs

STDIN, STDOUT, STDERR – all file handles.

To create a filehandle, use open(<HANDLENAME>, "<access type><filename>");

Access types: < read  
> write (overwrites)  
>> append  
<+ read and write  
>+ read and write (wipes file first)

print HANDLENAME string prints to the given file.

Catch exceptions with die: open (IN, "<myfile") or die "Can't open myfile\n"; - die kills current process and prints message to STDERR.

Always close files after use with close(HANDLE);

Normally, you read stuff a line at a time, e.g., \$line = <HANDLE>, but the read command can be useful if you don't want to get a line at a time:

read(filehandle, buffer, length[, offset])  
buffer is the scalar variable to contain the read string  
length is how many bytes to read in  
offset is the optional (default zero) starting position.

read *returns* the number of characters it read.

*Tests on files:*

if (flag file) { ... }

Flags: -e file exists  
-s file has a non-zero size  
-z file has a zero size  
-r file is readable  
-w file is writable  
-x file is executable  
-f file is not a directory  
-d file is a directory

### *Navigating the file system*

opendir (HANDLE, "directoryname")

readdir (HANDLE) returns a list of the contents of the directory.

closedir(HANDLE)

'Globbing' – accessing files using the unix syntax. Must be used enclosed in < >.  
E.g., foreach \$file (</stud/ug/pxs02u/cgi-bin/\*.pl>) { print "\$file\n"; }

## Revision Notes

### *Calling the Operating System*

UNIX environment variables are stored in %ENV.

To change an environment variable for only a certain block of code (a block if enclosed in { } ), use the keyword local:

```
{  
local $ENV{PATH} = ".";  
...  
}
```

Other File System commands (all return 1 if successful, 0 is not – often used with 'die'):  
Can be used with NT and Unix.

chdir ("name")  
unlink ("name") – same as rm  
rename ("name", "newname")  
mkdir ("name", 0755) – creates directory with specified access modes.  
rmdir ("name")  
chmod (0644, "name")

other system commands (i.e. those in bin/sh) can be called using the system function:

```
system("finger pxs02u");
```

N.B. Bizarrely, system() returns 0 if successful and 1 if not, so when using with die, you must use && instead of or: system ("finger pxs02u") && die ("Could not retrieve information about pxs02u");

If we wanted to return the output of the system command instead of just using it, we can use 'quoted execution' – put the command in back quotes ( ` ` ): \$info = `finger pxs02u`;

### Pattern Matching

Meta-characters: | ( ) [ ] ^ \$ \* + ? .

These can be matched if preceded by a \

. matches any character

Square brackets containing a list of characters (a character class) will match any single character from the list: h[ae]llo

[0-9] matches any digit. Also \d

[^0-9] – matches anything except a digit. Also \D

[A-Za-z] – also \w (negation with \W)

\s matches a whitespace character – space, tab, return, newline. Again, \S negates this.

Variables can be used as patterns: m/[\$word1 | \$word2]/

Evaluation of these variables can be prevented by using single quotes as the delimiters.

Repetition: \* - zero or more; + - one or more; ? – zero or one.

Repetition counts: enclosed in curly braces. E.g., \d{4} specifies a sequence of four digits.

Also, {n,} – must occur at least n times; {n,m} – must occur between n and m times.

If RE and RX are regular expressions, RE | RX matches RE or RX. Can be grouped together in brackets.

Revision Notes

**Anchors:**     ^ - specifies the pattern must be at the beginning of the line, e.g. /^[A-Z] [0-9]+/  
                   \$ - specifies the pattern must be at the end of the line, e.g. /[0-9]+[A-Z]\$/  
                   Accordingly, /^\$/ matches the empty line (not one containing white space)  
                   \b matches a word boundary (i.e. \bcat\b/ matches cat but not catamaran)  
                   ? matches shortest string (. \* always matches the longest match)

N.B. // matches the most recently used regular expression.

For pattern matching comparisons, use =~

*Modifiers*

m/ ... / i	Ignore case	m/ ... / s	. matches newline as well
m/ ... / g	Find all occurrences	m/ ... / x	ignores whitespace
m/ ... / m	Treat string anchors as line anchors	(except in [] or \)	
		m/ ... / o	only compile expressions once.

Modifiers can be combined m/ ... / ig

*Remembering Matches*

(\$url =~ m!(http|ftp):/(.\*?/)(.\*\$!)

This sets the first bracketed section to \$1, the second to \$2, etc...

Alternatively,

(\$protocol, \$site, \$path) = (\$url =~ m!(http|ftp):/(.\*?/)(.\*\$!)

An alternative to .\*?/ would be [^/]\*/ - this is a common trick. Or .\*?(?=/) – lookahead assertion (forgotten if matched).

A pattern matches the left-most string, so (S\*i) matches SSiSSi but actually ignores the second half.

*Split*

The split function acts as a tokeniser, returning a list.

split /\s+/, \$line – splits line into individual words.

If no arguments, delimiter assumed to be whitespace and the string \$\_ (\$\_ is the default input).

*Substitution*

s/pattern/substitution/

Again, \$1, etc. remember matches, so s/(w\*)(\d\*)/\$2\$1/

the 'e' modifier evaluates expressions at run time, so s/\d+/\$&+1/e – adds one to a number. N.B. \$& is the last successful pattern match.

Translation: tr/original/replacement/ (original and replacement must be same length). E.g., tr/[A-Z]/[a-z].

join (char, list) – opposite of split, i.e. return a list as a string (each element joined with char)



## Revision Notes

### Functions

Subroutines are declared as:

```
sub name {  
    ...  
}
```

Call it using &name(arguments). & can be omitted if subroutine defined earlier than call. Possible to make an empty definition at the top to avoid using &.

Arguments to a subroutine are kept in the anonymous array, @\_. N.B. that to you assign things to these variables, @\_[0], etc., the actual parameters passed to the subroutine change.

E.g.,  
&subR(\$arg);  
subR {  
 @\_[0]++;  
}

This actually increments \$arg.

N.B. Subroutines return the last value of the last expression in the block. It is clearer to use the return keyword (which can be used anywhere).

### *Variable Scope and Lifetime*

Normally, variables are visible to all statements in the program.

Using 'my' before a declaration in a subroutine means the variable is only visible within that subroutine.

Can be used with lists: my (\$arg1, \$arg2, \$arg3) = @\_

'my' can be used in any block of code (not just a subroutine), but should be avoided in loops, as it would be newly created each iteration.

'local' works similarly, but their scope includes any subroutines called from the subroutine in which it is declared.