Lecture 02

Introduction to Software Engineering

COSC110

Introduction to Programming and the UNIX environment

Outline

- Bourne Again Shell (bash)
  - Command Line Expansion
  - Standard Channels
  - Pipes
  - Input/Output
  - Environment variables
  - Sequencing commands

Shell Shortcuts

- Simple editing
  - Backspace and left/right arrow keys
- Command/filename completion
  - Tab
- Command history
  - Up/down arrow keys (or the history command)
- Wildcards
  - ? represents exactly one character
    - E.g. **?.txt** will match a.txt and A.txt, but not aa.txt
  - * represents zero or more characters
    - E.g. ***.txt** will match any file that ends with .txt
  - […] represents any of the enclosed characters
    - E.g. **[aA].txt** will match a.txt and A.txt but not b.txt or aa.txt
  - These can be combined
    - E.g. **?a*b[ab].txt** will match any file whose name has a as the second character in its name, followed by any number of characters, followed by a b, followed by an a or b, followed by .txt

Standard Channels

- Each process has three standard channels:
  - Standard input (0, stdin)
  - Standard output (1, stdout)
  - Standard error (2, stderr)
- By default:
  - stdin is the keyboard
  - stdout is the display
  - stderr is the display

Redirecting Standard Channels

- The standard channels can be overridden by redirecting them to/from files
  - < filename
    - Indicates input should come from the given file
    - < is shorthand for 0<
  - > filename
    - Indicates output should be written to the given file
    - > is shorthand for 1>
  - 2> filename
    - Indicates errors should be written to the given file

Redirection

- Other redirection options
  - >>
    - >> appends to a file rather than overwriting it
  - &> filename
    - Redirects both standard output and standard error
  - Redirecting to /dev/null
    - Suppresses all output
  - Standard input and output can also be redirected through pipes |

Pipes

- Pipes redirect the output from the program to the left of the | to be the standard input of the program on the right
- E.g. **ls –l | less**
  - **ls –l** lists files in the current directory
  - **less** displays input one page at a time
  - **ls –l | less** lists files in the current directory one page at a time

tee

- **tee** reads from standard input and writes to standard output and zero or more files
- E.g. **ls –l | tee list | less**
  - Performs same action as previous example
  - Also creates a file called *list* that contains the output of **ls –l**

Parameter Expansion

- **echo <message>**
  - Prints <message> to the screen
- Parameters can be set and reused in later commands:

```
a=1
b=hello
echo $a $b
```

- Results in:

```
1 hello
```

Shell Variables

- Variables can be created within a shell environment
  - E.g.

    a=1

    message="Hello world"

- When accessing the variable, we need to precede it with the $ sign
  - E.g.

    echo $message $a

Arithmetic Expansion

- $((<expression>))
  - Evaluates an integer expression
- E.g.

```
a=1
b=2
c=$((a+b))
echo $c
```

- Results in:

```
3
```

Input/Output

- We have already seen the **echo** command for basic output
- The **read** command allows basic input
  - E.g.

```
read -p "Please enter an integer" x
echo $x
```

Environment Variables

- Shell variables are only accessible by the shell process that creates them
  - Inaccessible in other processes
- Environment variables are imported to all processes created in that environment
  - **env** lists the current environment variables
- Shell variables can be exported to environment variables
  - E.g. **export message**

Sequencing Commands

- Remember the UNIX philosophy
  - Do one thing, and do it well
- We often need to run multiple commands to meet our requirements
  - Running one command then another
- Multiple commands can be specified on the same line
  - Separated by ;
  - E.g. **date ; who**

Redirecting Combined Output

- Consider the following command:

  date ; who | less

- This will run the date command, then run who with the output of who piped through less
- To pipe the output of both date and who, we need the following

  (date ; who) | less

Exit Status

- When a process completes, it sets an exit status which can be examined by the OS or shell
- A status of 0 indicates success, while any non-zero value indicates failure
  - Usually interpreted as an error code
- The exit status of the last process to complete can be accessed with **$?**

Conditional Execution

- Consider the command:

  cp report.txt newReport.txt ; pico newReport.txt

- If the copy fails, we don’t want pico to run
- To give the required behaviour, replace the ; with &&
  - The command to the right of the && will only run if the command to the left returns a 0 exit status

  cp report.txt newReport.txt && pico newReport.txt

Conditional Execution

- Consider the command:

  cp report.txt newReport.txt ; pico newReport.txt

- If the copy succeeds, we don't want to edit the report
  - i.e. We'll use pico to create the report only if the copy fails
- To give the required behaviour, replace the ; with ||
  - The command to the right of the || will only run if the command to the left returns a nonzero exit status

  cp report.txt newReport.txt || pico newReport.txt

Concurrent Execution

- Sometimes you do not want to wait for a command to complete before starting a new command
  - E.g. a command may take a long time, and you want to do other things while it is processing
- A command can be set to run in the background using &
  - E.g.

cp largeFile.txt largeCopy.txt & ls

[1] 2324

largeFile.txt largeCopy.txt otherFile

Killing a Process

- If a process is running in the foreground, it can be killed by pressing **Ctrl-C**
- Processes can also be killed using the **kill** command
  - E.g.

cp largeFile.txt largeCopy.txt & ls

[1] 2324

largeFile.txt largeCopy.txt otherFile

kill 2324

- The **ps** command reports a snapshot of current processes
  - See **man ps** for more details

Summary

- BASH is a powerful shell with many different options
- Each process has standard channels which can be redirected
- We can sequence commands in various ways

Outline

- Making Scripts
  – Writing script files
  – Running script files
- Scripting
  – Sequencing
  – Selection
  – Iteration

Writing Script Files

- Can be inconvenient to write a long series of commands to the command line
  - Especially when you'll need to use the exact same commands later
- Shell scripts allow you to perform a larger task by combining various commands into a single file
  - Then you can simply run the shell script rather than all of the commands it contains
- A shell script is a text file that lists the commands to run

Example Script Files

```
#!/bin/bash
# The first line specifies which shell to use
# Any other line that begins with a # is ignored

echo "Enter a number"
read a
echo The number you entered was $a
```

Running a Script File

- Suppose the example script was saved in a file named *script.sh* in the current directory
- To run the script, you must first ensure you have executable permission on the file script.sh

  chmod a+x script.sh
- You can then run the script from the command prompt by specifying its path
  - Since it is in the current directory, the path can be represented as ./script.sh

  ./script.sh

Sequencing commands

- **Example**: Write a shell script to calculate the area of a rectangle
- This will require the user to input the width and height of the rectangle
  - We will assume these values are integers
- We will then multiply the width and height together
- And output the result

rectangleArea.sh

```
#!/bin/bash
# read input
echo "Enter width of rectangle"
read width
echo "Enter height of rectangle"
read height
# calculate area
area=$((width * height))
# output result
echo "The area is $area"
```

Selection

- We have seen a way to have commands only run conditionally
  - && or || between commands
- bash also provides an if statement in the following form:

```
if test-command
then
    commands
fi
```

- If **test-command** returns a 0 exit status, the commands after the then will be run
  - Otherwise execution will continue after the fi

Determining if text files exist

- **Example:** Write a shell script that outputs the message “Text files exist” only if the current directory contains at least one file with a “.txt” extension

- **ls** will be useful here
  - Consider the command **ls *.txt**
    - If there are any .txt files, this will list them and exit with a 0 status
    - If there are no .txt files, this will output a “No such file or directory” error and return a nonzero status

txtFilesExist.sh

```
#!/bin/bash

# we need to redirect both standard output and standard error
# because we don't want to see output from the ls command –
# we're only interested in the "Text files exist" message

if ls *.txt &> /dev/null
then
	echo "Text files exist"
fi
```

Performing Alternate Actions

- Sometimes we want to run one set of commands if a command succeeds, and another set if the command fails
- The bash if statement accommodates that as follows:

```
if test-command
then
  commands
else
  commands
fi
```

txtFilesExist.sh

```
#!/bin/bash

# This example expands the previous one by creating a text file
# if none already existed

if ls *.txt &> /dev/null
then
	echo "Text files exist"
else
	echo "No text file exists – creating test.txt"
	echo "Automatically generated file" > test.txt
fi
```

Conditional Expressions

- Rather than using a command as the condition, bash also supports the use of expressions
- ((expression))
  - Evaluates an arithmetic expression
  - Any nonzero value has an exit status of 0 (false)
  - A zero value has an exit status of 1 (true)
- [[expression]]
  - Note the spaces around the expression
  - Returns a status of 0 or 1 depending on the logical expression

Logical Expressions

-a file	True if file exists
-d file	True if file exists and is a directory
-r file	True if file exists and is readable
-s file	True if file exists and has size greater than 0
-w file	True if file exists and is writable
-x file	True if file exists and is executable
file1 –nt file2	True if file1 is newer than file2 (or exists and file2 does not)
file1 –ot file2	True if file1 is older than file2 (or file2 exists and file1 does not)
-v varname	True if the shell variable varname has an assigned value
-z string	True if the length of string is 0
string1 == string2	True if the strings are equal
string1 != string2	True if the strings are not equal
string1 < string2	True if string1 sorts before string2 lexicographically
string1 > string2	True if string2 sorts before string1 lexicographically

Simple Artificial Intelligence

- **Example:** Write a simple bash program which performs the following operations:
- Ask the user if they are happy or sad.
- If the user is sad, ask how many days since they last exercised
- If it has been more than one day since they last exercised, output the message “Go for a walk, you might feel better”
- In all other cases, the program should say “Sorry, I don’t know how to help”

ai.sh

```
#!/bin/bash
echo "How do you feel today? Happy or Sad?"
read happyOrSad
if [[ $happyOrSad == "Sad" ]]
then
	echo "How many days since you last exercised?"
	read lastExercised
	if (($lastExercised>1))
	then
		echo "Go for a walk, you might feel better"
		exit
	fi
fi
echo "Sorry, I don’t know how to help"
```

Iteration

- Sometimes we want to repeat an operation over and over again until a condition is true or false
- Bash supports two constructs for this
  - While repeats commands while ever test-command has an exit value of 0

while test-command

do

commands

done

  - Until repeats commands while ever test-command has a non-zero exit value

until test-command

do

commands

done

While Example

```
#!/bin/bash
stop=0
while ((stop==0))
do
        echo "Option Menu"
        echo "1: print menu"
        echo "2: exit menu"
        read choice
        if (($choice==2))
        then
                stop=1
        fi
done
```

Iterating Over a List

- It can also be useful to iterate over a list of strings
  - For example, iterating over a list of filenames could allow you to perform an operation on each file
- Bash supports such iteration through the for statement:

for name **in** list

do

    commands

done

Simple for Example

```
#!/bin/bash
for day in Monday Tuesday Wednesday Thursday Friday
do
	echo $day
done
```

for Example

```
#!/bin/bash
for file in *
do
	if [[ -d $file ]]
	then
		echo $file is a directory
	fi
done
```

Summary

- Shell scripting allows simple commands to be combined to perform complex operations
- bash supports:
  - Sequencing
  - Selection
  - Iteration
- Many powerful aspects of bash have been skipped in this unit

Outline

- The Software Lifecycle
  – Requirements
  – Analysis
  – Design
  – Implementation
  – Integration
  – Maintenance
- Software Errors
- Ethics
- Software Development Processes
  – Waterfall
  – Agile

Developing Software

- We have now seen everything required to develop software
  - Sequence
  - Selection
  - Iteration
- But complex systems require more than that. They must be:
  - Planned
  - Tested and Optimised
  - Maintained

Systems Development Life Cycle (SDLC)
Life-Cycle Phases

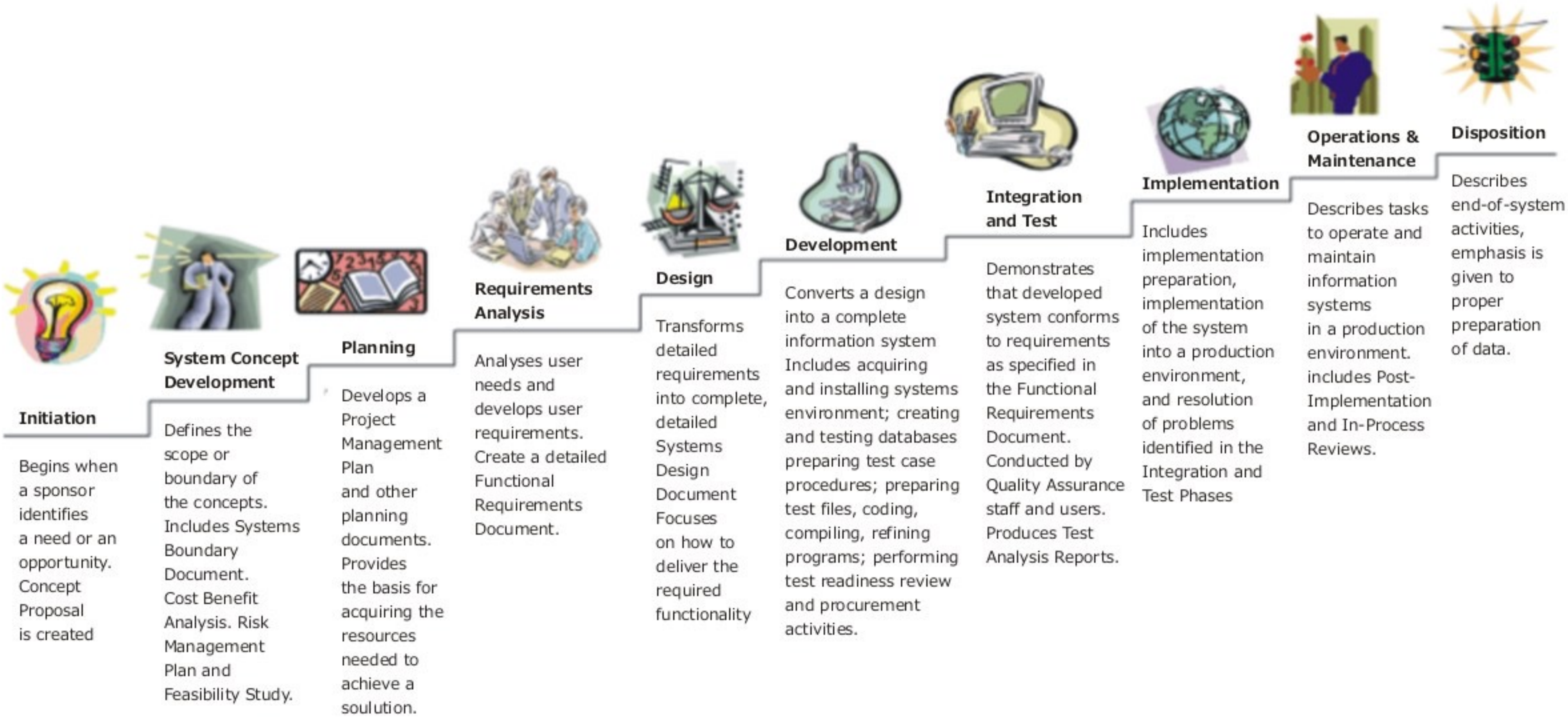


Image source: US Department of Justice, https://commons.wikimedia.org/wiki/File:Systems_Development_Life_Cycle.jpg Public domain

Requirements Phase

- A broad statement of a problem that needs to be solved
  - Calculate the area of a rectangle
  - Create a Web browser to view Web pages
  - Monitor sensors in a factory, ensuring optimal efficiency

Analysis Phase

- Determines exactly what the system needs to do to solve the problem
  - Breaking down the system into different pieces
    - Determining required inputs, outputs, etc.
- Does not describe how the system will solve the problem

Design Phase

- Describes how the system will perform the operations required by the analysis
- Often uses pseudocode to specify what the system will do

Implementation Phase

- Describes the design using a programming language
  - In this unit, we'll use Python
- Creates a solution that can actually be used

Integration Phase

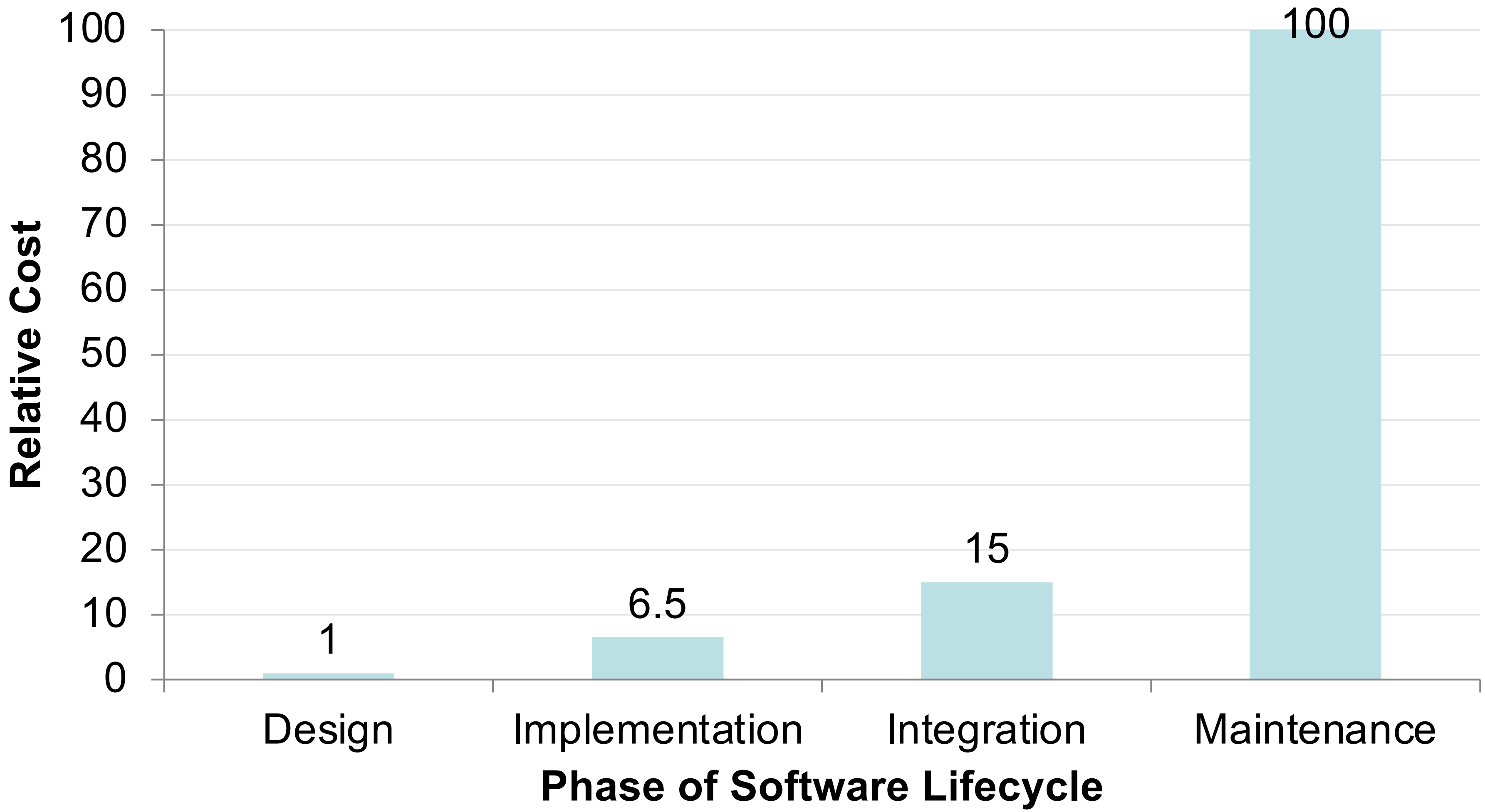
- Deploys the developed solution into the environment where it will actually run
- Ensures all necessary data are available for the new system, and that the new system communicates correctly with other systems that require it

Maintenance Phase

- Ensures the deployed system continues to work correctly
  - Fixing any errors that are discovered
  - Providing new functionality that is required

- A lot of software has a much longer lifetime than initially expected
  - Lifetimes of 20 years or more are not uncommon

The Cost of Correcting a Fault



Data source: IBM System Science Institute Relative Cost of Fixing Defects
Though note some controversy: https://www.theregister.com/2021/07/22/bugs_expense_bs/

Software Errors

- Software faults have very real consequences
  - A woman was charged $6.3 million rather than $63 because of an input error
  - A car insurance system was only designed for people aged up to 100, so a 101 year old had great difficulty getting insurance
  - A woman in Canada could not get her tax refund because the system said she was dead
  - A bank's processing systems were unavailable over a weekend, meaning people could not access their own money

Bugs Can Cost Millions

- NASA's Mariner I rocket was developed to conduct a flyby survey of Venus
  - At a cost of $80 million (over $600 million in today's dollars)
- Within 5 minutes of take-off, the rocket exploded
- The problem was a missing hyphen in one line of the code
  - This caused false information to be sent to the spacecraft control systems, causing the rocket to crash

They can even cost lives

- Between 1985 and 1987, the Therac-25 radiation therapy machine gave massive overdoses of radiation to 6 patients
  - Instead of doses of 100-200 rads, the patients received 13,000-25,000 rads
- Three patients died
- The Therac-25 reused code from the Therac-6 and Therac-20 models without adequate testing
- There were other weaknesses in the design of the operator interface

Reasons for errors

- Complexity of systems
- Insufficient testing
- Failure to handle unexpected circumstances
- Data entry errors
- Inadequate training
- Overconfidence in software
- Compromises to reach deadlines

Ethical Theories

- Relativism
  - No universal right or wrong
- Kantianism
  - Universal moral laws based on reason
- Egoism
  - What’s right matches one’s self-interest
- Utilitarianism
  - Right or wrong based on total happiness

Codes of Ethics

- Australian Computing Society
  - https://www.acs.org.au/content/dam/acs/acs-documents/Code-of-Ethics.pdf
- Association for Computing Machinery
  - http://www.acm.org/about-acm/acm-code-of-ethics-and-professional-conduct
- Institute of Electrical and Electronics Engineers
  - http://www.ieee.org/about/corporate/governance/p7-8.html

Software Engineering

- Many modern software systems are too large and complex to be developed by a single person
  - And software size continues to grow
- Software Engineering is so much more than writing code
  - Applying processes/methodologies to allow teams to develop systems effectively

Software Development Processes

- Splitting software development into distinct stages
  - Allowing better planning and management
- Help ensure software works correctly, on time and within budget
- No particular process necessarily best for all projects

Waterfall Model

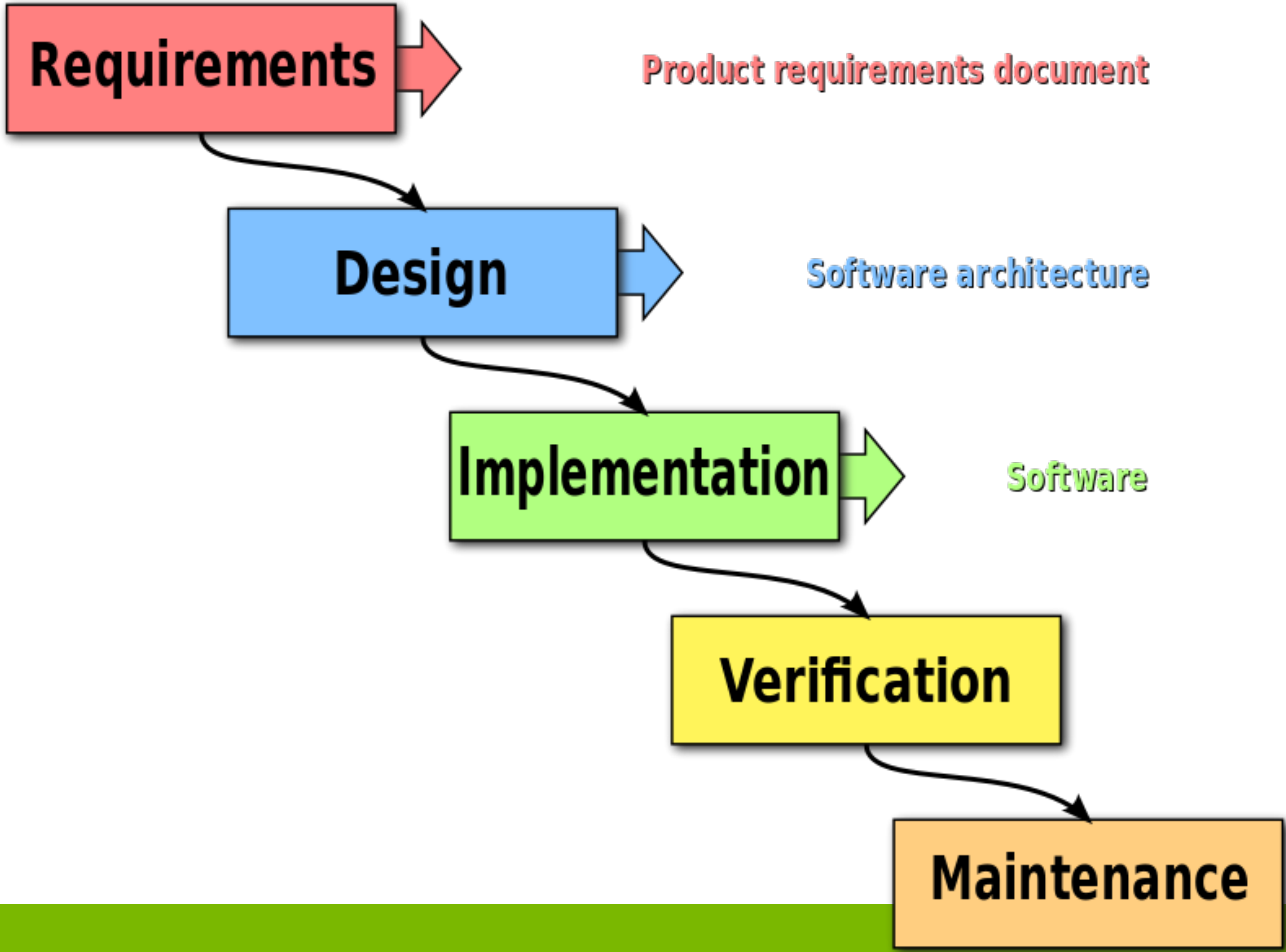


Image source: Peter Kemp / Paul Smith, https://commons.wikimedia.org/wiki/File:Waterfall_model.svg

Agile Model

- Requirements and solutions evolve through collaboration between teams
  - Adaptive planning
  - Evolutionary development
  - Early delivery
  - Continuous improvement
  - Rapid and flexible response to change

Summary

- Software Engineering is more than just programming
- Errors become more costly in later phases of the software lifecycle
- Software development processes aim to allow better planning and management of software systems, while ensuring they are correct, on time, and within budget