COMP61511 (Fall 2017)
Software Engineering Concepts
In Practice
Week 2
Bijan Parsia & Christos Kotselidis
<bijan.parsia, christos.kotselidis@manchester.ac.uk>
(bug reports welcome!)
FizzBuzz In Way Too Much Detail
The Naivest Fizzbuzz

- Any proposals?
- Let's see the obvious!
A Rational FizzBuzz

- Let's consider a "standard" implementation
  - Not silly
  - Not golfy
- I.e., a simple loop oriented implementation
DRY

- "Don't Repeat Yourself"
  - A fundamental principle of SE
  - It is against
    - Cut and Paste reuse
    - Not Invented Here syndrome
- Is our current version DRY?
A Dryer Version

- We repeat $i \% 3 == 0$ and $i \% 5 == 0$
- Let's abstract that out!
EVEN DRIER!!!

- We repeat the _ % _ == 0 pattern!
- We say `print` a lot
- We can fix it!
Parameterization

- Basic software principle: Don't **hard code** stuff!
  - Make your code parameterisable!
- The current version hard codes a lot, e.g.,

  ```
  FIZZ = 'Fizz'
  BUZZ = 'Buzz'
  ```

- We have to **modify the source code** if we want to change this!
  - What else is hard coded?
  - We can **fix it**!
Still Hard Coding!

- The **kind of test** is hard coded
- We can fix that!
The Path To Hell...

- ...is paved with good intentions!
- Each choice was somehow **reasonable**
  - We applied good **SE principles**
  - We made choices **that are often good**
- But we ended up in **nonsense land**
  - **Local** sense led to **global** nonsense
Judgement

- Software engineers can't just follow rules
- Good software engineering requires judgement
  - When to apply which rules
  - When to break rules
  - *How* to apply or break them
  - The reason for each rule
    - And whether it makes sense now
Acknowledgement

This lecture was derived from the excellent blog post FizzBuzz In Too Much Detail by Tom Dallling.

Tom uses Ruby and goes a couple of steps further. Worth a read!
Who Owns Your Code?

- You *wrote* some code!
  - All week!
  - Both systems and tests!
- A key question:
  - Who *owns* that code?
    - Or different bits of it?
  - What *kind* of ownership?
Intellectual Property (IP)

*Intellectual property* is any articulable, tangible production of a mind whose physical realisations are restricted by law (in production, distribution, etc.)

- We don’t control what other people think!
- We can control what they do with certain thoughts.
- Intellectual Property rights give power to certain people to control what other people do
  - For example, whether they can distribute a book, song, or program
## Kinds Of Intellectual Property

<table>
<thead>
<tr>
<th>Name</th>
<th>Establishment</th>
<th>Enforcement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copyright</td>
<td>Automatic, immediate</td>
<td>Civil and Criminal</td>
</tr>
<tr>
<td>Patent</td>
<td>Application; exposure before application destroys it</td>
<td>Mostly civil</td>
</tr>
<tr>
<td>Trademark</td>
<td>Application and vigorous defense</td>
<td>Mostly civil</td>
</tr>
<tr>
<td>Trade Secret</td>
<td>Automatic (by not telling people) and NDAs</td>
<td>Mostly civil</td>
</tr>
</tbody>
</table>
Copyright

**Copyright** is a licensable monopoly of **tangible** expression of an idea with respect to reproduction, derivation, display, distribution, and the like.

- Protects the **expression** not the **idea**
  - Though these **blur** at the limit
    - Some plagiarism is a copyright violation; some is not
- Typically **automatically assigned at creation time**
  - No "notice" or "registration" needed
    - Though these might help with lawsuits
A patent is a licensable monopoly of the use or sale of a "non-obvious" invention (of a process, machine, design (sometimes), mechanism, procedure, etc.

- A patent is an incentive to disclose
  - Many patentable inventions could be exploited "secretly"
  - Goal is to add to our common knowledge
- Prior art destroys a patent
  - Including your own
- Defensive patenting "common"
- Independent invention no defense
Trade Secret

A trade secret is an invention which is not disclosed

- Persists forever
  - Unless leaked
  - Or reinvented
- Typically protected by secrecy
  - Or specific contracts
    - "Non-Disclosure Agreements" (NDAs)
Who Owns Your Code?

- Copyright starts with the creator
  - I.e., you!
  - Cheap! (Even to register)
  - Unless you create it as work-for-hire
    - Or otherwise transfer it
- Patents belong to the patenter
  - Expensive(ish) to secure
- Trade secrets belong to the inventor
Are You Working For Hire?

- Not quite!

3.6. **Student IP Licence to the University**

3.6.1. Each Student grants to the University a licence to use the:

3.6.1.1. IP created by him or her in the course of his/her studies at the University and which they own; and

3.6.1.2. IP in any thesis or dissertation submitted to the University for the award of a degree.

3.6.2. In each case, the licence will take effect upon the creation of the relevant IP.

3.6.3. The licence:

3.6.3.1. is a continuing, non-exclusive, worldwide, irrevocable, royalty-free licence to use the IP in any format (whether existing or future);

3.6.3.2. will last for as long as the relevant IP remains in existence; and

3.6.3.3. is granted so that the University can (i) use such IP for its and its subsidiaries' administrative, promotional, educational and teaching purposes; and (ii) do all such things in relation to such IP which would otherwise be an infringement of such IP.

3.6.4. As part of the licence, the University is also permitted to sub-license to others the rights granted to it by Students.

3.6.5. Any thesis or dissertation submitted to the University for the award of a degree may be placed by the University in its institutional repository in electronic or other format.
What To Keep In Mind (Now)

- Software engineers typically **produce** IP
  - Even if not protected, our **output** is "intellectual"
  - Various forms of IP drive
    - **product** value
    - **employee/entrepreneur** value
- Software engineers typically **use** IP
  - All sorts and in all ways
  - IP considerations a constraint on the design space
Comprehending Product Qualities
Comprehension?

- We can distinguish two forms:
  - Know-that
    - You believe a true claim about the software
    - ...with appropriate evidence
  - Know-how
    - You have a competancy with respect to the software
    - E.g., you know-how to recompile it for a different platform
- They are interrelated
- Both require significant effort!
Quality Levels

- We talked about different kinds of quality
  - But for each kind there can be degrees or levels thereof
  - "Easy" example: High vs. Low performance
- Most qualities in principle are quantifiable
  - Most things are quantifiable in some sense
- But reasonable quantification isn't always possible
  - Or worth it
  - Being clear about your vagueness is essential!
Clarity

Our discussion will be adequate if it has as much clearness as the subject-matter admits of, for precision is not to be sought for alike in all discussions, any more than in all the products of the crafts...for it is the mark of an educated [person] to look for precision in each class of things just so far as the nature of the subject admits...

— Aristotle, Nicomachaen Ethics, Book 1, 3
We demand rigidly defined areas of doubt and uncertainty!
A **defect** in a software system is a **quality level** (for some quality) that is **not acceptable**.

- Quality levels need to be elicited and negotiated
  - All parties must agree on
    - **what** they are,
    - their **operational definition**
    - their **significance**

What counts as a defect is often determined late in the game!
Question

If your program crashes then it

1. definitely has a bug.
2. is highly likely to have a bug.
3. may or may not have a bug.
Question

If your program crashes, and the cause is in your code, then it

1. definitely has a bug.
2. is highly likely to have a bug.
3. may or may not have a bug.
Bug Or Feature?

(Does QA hate you? — scroll for the cartoons as well as the wisdom.)

- Even a **crashing code path** can be a **feature**!
- Contention arises when the stakes are high
  - and sometime the stakes can seem high to some people!
  - defect rectification costs the same
    - whether the defect is **detected**...
    - ...or a feature is **redefined**
- Defects (even redefined features) aren't personal
This is a logical, not temporal, order.
Problem Definition

The penalty for failing to define the problem is that you can waste a lot of time solving the wrong problem. This is a double-barreled penalty because you also don’t solve the right problem.
—McConnell, 3.3
Quality Assurance

- Defect **Avoidance** or **Prevention**
  - "Prerequisite" work can help
    - Requirement negotiation
    - Design
    - Tech choice
  - Methodology
- Defect **Detection** & Rectification
  - If a defect exists,
    - Find it
    - Fix it
The Points Of Quality

1. Defect prevention
   - Design care, code reviews, etc.
2. Defect appraisal
   - Detection, triaging, etc.
3. Internal rectification
   - We fix/mitigate before shipping
4. External rectification
   - We cope after shipping
## Defect Detection Techniques

<table>
<thead>
<tr>
<th>Removal Step</th>
<th>Lowest Rate</th>
<th>Modal Rate</th>
<th>Highest Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Informal design reviews</td>
<td>25%</td>
<td>35%</td>
<td>40%</td>
</tr>
<tr>
<td>Formal design inspections</td>
<td>49%</td>
<td>55%</td>
<td>65%</td>
</tr>
<tr>
<td>Informal code reviews</td>
<td>20%</td>
<td>25%</td>
<td>35%</td>
</tr>
<tr>
<td>Formal code inspections</td>
<td>45%</td>
<td>60%</td>
<td>70%</td>
</tr>
<tr>
<td>Modeling or prototyping</td>
<td>35%</td>
<td>65%</td>
<td>80%</td>
</tr>
<tr>
<td>Personal desk-checking of code</td>
<td>20%</td>
<td>40%</td>
<td>60%</td>
</tr>
<tr>
<td>Unit test</td>
<td>19%</td>
<td>30%</td>
<td>50%</td>
</tr>
<tr>
<td>New function (component) test</td>
<td>20%</td>
<td>30%</td>
<td>35%</td>
</tr>
<tr>
<td>Integration test</td>
<td>25%</td>
<td>35%</td>
<td>40%</td>
</tr>
<tr>
<td>Regression test</td>
<td>19%</td>
<td>25%</td>
<td>30%</td>
</tr>
<tr>
<td>System test</td>
<td>25%</td>
<td>40%</td>
<td>55%</td>
</tr>
<tr>
<td>Low-volume beta test (&lt;10 sites)</td>
<td>25%</td>
<td>35%</td>
<td>40%</td>
</tr>
<tr>
<td>High-volume beta test (&gt;1,000 sites)</td>
<td>60%</td>
<td>75%</td>
<td>85%</td>
</tr>
</tbody>
</table>

Source: Adapted from *Programming Productivity* (Jones 1996a), "Software Defect-Removal Efficiency" (Jones 1996), and "What We Have Learned About Fighting Defects" (Shull et al. 2002).
Defect Detection Techniques
Experiencing Software

- It's one to know that there are bugs
  - **All** software has bugs!
- It's another to be able to **trigger** a bug
  - Not just a specific bug!
  - If you understand the software
    - You know how to break it.
- Similarly, for **making changes**
  - tweaks, extensions, adaptations, etc.
- The more command, the more modalities of mastery
Forms Of Knowledge (Manifestations)

- **Human interpretable**
  - Comments, design docs, user stories, javadoc
- **Source code**
  - Both, a written description and a "live" object
  - Also things like demo code, examples, test suites, etc.
- **Diagrams**
  - "Mere" pictures to semi-formal to formal diagrams: ER docs, UML, etc.
- **Formal specifications**
- **Competencies**
  - I can **make it crash**
Sources Of Knowledge (Modalities)

- **Analytical** knowledge
  - Derived from *inspection* and *reasoning*
  - Can be *automated* using formal methods
- **Experimental** knowledge
  - Derived from the conduct of experiments
  - Typically tests
- **Experiential** knowledge
  - Derived from *personal interaction* with the software
  - Strong "know-how" component
Lab!
Revisiting Rainfall

We're going to look at your rainfalls before discussing it in detail.

We're going to do a code review!

You're going to work in 2-person teams!
Three Tasks

1. Do a code review!
2. Write some tests based on your code review!
3. Do an essay review!

To the lab! Material in the usual place.
Testing Rainfall
Rainfall

- Key point: 0 out of 47 programs passed all 13 tests
  - 1 program passed 8 tests
  - 2 passed 7
  - 5 passed 6
  - 0 passed 5 or 4
  - 5 passed 3
  - 3 passed 2
  - 1 passed 1
  - 15 passed 0
  - 11 "had a problem with submission"
  - 4 "We could not compile your code."

  *The rainfall problem is still a challenge!*
Let's Talk Testing

- You had *limited time*
  - So test generation had to be quick!
  - Typically ad hoc
    - Can we do better?
- How testable is `rainfall.py`?
  - You were responsible *only* for `average_rainfall(input_list)`
    - Only this *unit*! Can ignore all else!
    - *Perfect* for `doctest`
Problem Statement

Design a program called rainfall that consumes a list of numbers representing daily rainfall amounts as entered by a user. The list may contain the number -999 indicating the end of the data of interest. Produce the average of the non-negative values in the list up to the first -999 (if it shows up). There may be negative numbers other than -999 in the list.
Set Up

def average_rainfall(input_list):
    >>>>> average_rainfall('<<FIRST TEST INPUT>>')
    <<FIRST EXPECTED RESULT>>
    
    # Here is where your code should go
    return "Your computed average as a integer" #<< change this!

$ python setup.py
Your computed average as a integer
First Test Run

$ python -m doctest setup.py
**********************************************************************
File "~/Users/bparsia/Documents/2018/Teaching/COMP41511/labs/lab1/followup/setup.py", line 2, in
Failed example:
    average_rainfall(<FIRST TEST INPUT>)}
Exception raised:
    Traceback (most recent call last):
      File "~/anaconda/lib/python3.5/doctest.py", line 1320, in __run
        compileflags, 1), test.globs)
    File "<doctest setup.average_rainfall[0]>", line 1
    average_rainfall(<FIRST TEST INPUT>)}
    SyntaxError: invalid syntax
**********************************************************************
1 items had failures:
  1 of 1 in setup.average_rainfall
***Test Failed*** 1 failures.
First Test

- Where do we get our first real test?
  - Hint: Read the docs:

```
But that's clearly not a correct solution. When fully implemented, we'd expect to see something like:

$ python rainfall.py 2 3 4 67 -999
19.0
```
Convert To Appropriate Doctest

- For a system test, we’d need to use subprocess etc.
  - But we can just test our unit!
    - `average_rainfall(input_list)`
    - But it takes a list not a string as input!
  - '2 3 4 67 -999' ==> [2, 3, 4, 67, -999]
    - We had to massage the input to get our test!
Tested `Average_rainfall` V2

def average_rainfall(input_list):
    """>>> average_rainfall([2,3,4,67, -999])
    19.0
    """
    # Here is where your code should go
    return "Your computed average as a integer" #<-- change this!

$ python ./setup.py
Your computed average as a integer
Second Test Run

```
$ python -m doctest ./firstfull.py
**********************************************************************
File "~/Users/bparsia/Documents/2018/Teaching/COMP61511/labs/lab1/followup/2firstfull.py", line 2
Failed example:
   average_rainfall([2, 7, 4, 67, -999])
Expected:
   19.0
Got: 'Your computed average as a integer'
**********************************************************************
1 items had failures:
  1 of 1 in firstfull.average_rainfall
***Test Failed*** 1 failures.
```
Yay!

- We have a **real** and **reasonable** test!
  - And a clear **format** for subsequent tests
  - And an **infrastructure** that makes it easy to run tests
- We have a **broken implementation**
  - As witnessed by a test!
- We Can Fix It!
Rosie Sez

WE CAN FIX IT!

IT'S A SMALL MATTER OF PROGRAMMING
First Implementation

```python
def average_rainfall(input_list):
    return sum(input_list)/len(input_list)
```

- Will this fail this test?
- Is there a test that it will pass?
First Implementation With Test

```python
def average_rainfall(input_list):
    >>> average_rainfall([2,3,4,67, -999])
    19.0
    >>> average_rainfall([2,3,4,67])
    19.0

    # Here is where your code should go
    return sum(input_list)/len(input_list)
```

Third Test Run

$ python -m doctest 4firstimpl2.py
**********************************************************************
Failed example:
    average_rainfall([2,7,4,67, -999])
Expected:
    19.0
    Got:
    -184.6
**********************************************************************
1 items had failures:
  1 of 2 in 4firstimpl2.average_rainfall
***Test Failed*** 1 failures.
def average_rainfall(input_list):
    # Here is where your code should go
    return sum(input_list[:len(input_list)-1])/len(input_list[:len(input_list)-1])

- Fixes one test but not the other!
- Tests work together
def average_rainfall(input_list):
    rainfall_sum = 0
    count = 0
    for i in input_list:
        if i == -999:
            break
        else:
            rainfall_sum += i
            count += 1
    # Here is where your code should go
    return rainfall_sum/count
Fourth Test Run

$ python -m doctest 5secondimpl.py
**********************************************************************
File "/Users/bparsia/Documents/2018/Teaching/COMP61511/labs/lab1/followup/5secondimpl.py", line
Failed example:
    average_rainfall([2,7,67, -999])
Expected: 19.0
Got: 19.0
**********************************************************************
1 items had failures:
  1 of 2 in 5secondimpl.average_rainfall
***Test Failed*** 1 failures.

Whaaaaaaaaaaaaaaaaaaaaat?!
A Bug!

- There was a bug in our tests
  - All along!

  ```python
def average_rainfall(input_list):
    >>> average_rainfall([2, 3, 4, 67, -999])
    19.0
  ```

  vs.

  ```python
def average_rainfall(input_list):
    ***  >>> average_rainfall([2, 3, 4, 67, -9])
    19.0
  ```

- Earlier tests failed for *two reasons*
- One bug *concealed* the other!!!
$ python -m doctest secondimpl2.py

$ python -m doctest -v secondimpl2.py

Trying:
  average_rainfall([2, 3, 4, 67, -999])
Expecting:
  19.0
ok

Trying:
  average_rainfall([2, 3, 4, 67])
Expecting:
  19.0
ok

1 items had no tests:
  secondimpl2
1 items passed all tests:
  2 tests in secondimpl2.average_rainfall
2 tests in 2 items.
2 passed and 0 failed.
Test passed.
Next Tests?

- These tests clearly aren't enough
- What next?
  - Look for boundary conditions ([-999])
  - Look for "odd equivalents"
    - Is [-999, 1] the same as [-999]?
    - How about [][] and [-999]?
    - How about [-999] and [-999, 0]
  - Look for normal cases you haven't covered
    - [-1 0 10]
    - For each new feature iterate the earlier moves!
      - e.g., is [-1 -2 -3 -999 1] the same as [][]?
A Classification Of Tests
A Classification Of Tests

- Based on a 5W+H approach by Ray Sinnema (archived)
  - **Who** (Programmer vs. customer vs. manager vs...)
  - **What** (Correctness vs. Performance vs. Useability vs...)
  - **When** (Before writing code or after)
    - Or even before architecting!
  - **Where** (Unit vs. Component vs. Integration vs. System)
    - Or lab vs. field
  - **Why** (Verification vs. specification vs. design)
  - **How** (Manual vs. automated)
    - On demand vs. continuous
Who?

- Sinnema: Tests give confidence in the system
  - I.e., they are evidence of a quality
- **Who** is getting the evidence?
  - Users? Tests focus on external qualities
    - Can I accept this software?
  - Programmers? Tests focus on internal qualities
    - Can I check in this code?
  - Managers? Both?
    - Are we ready to release
- But also, **who** is writing the test?
  - A bug report is a (typically partial) test case!
What?

- Which **qualities** am I trying to show?
  - Internal vs. external
  - Functional vs. non-functional?
  - Most **developer testing** is functional (i.e., correctness)
    - And at the unit level
    - Does this class **behave as designed**
When?

- **When** is the test written?
  - **Before** the code is written?
  - **After** the code is written?
- Perhaps a better distinction
  - Tests written with **existing code/design in mind**
  - Test written **without regard** for existing code/design
  - This is related to white vs. black box testing
    - Main difference is whether you **respect the existing API**
Where?

- **Unit**
  - Smallest "chunk" of coherent code
  - Method, routine, sometimes a class
  - McConnell: "the execution of a complete class, routine, or small program that has been written by a single programmer or team of programmers, which is tested in isolation from the more complete system"

- **Component** (McConnell specific, I think)
  - "work of multiple programmers or programming teams" and in isolation
Where? (Ctnd)

- **Integration**
  - Testing the *interaction* of two or more units/components

- **System**
  - Testing the system as a whole
  - In the lab
    - I.e., in a controlled setting
  - In the field
    - I.e., in "natural", uncontrolled settings
Where? (Cndt Encore)

- **Regression**
  - A bit of a funny one
  - **Backward looking** and **change oriented**
    - Ensure a change *hasn't broken anything*
    - Esp previous fixes.
Why?

- Three big reasons
  1. **Verification** (or validation)
     - Does the system possess a quality to a certain degree?
  2. **Design**
     - Impose constraints on the design space
       - Both structure and function
  3. **Comprehension**
     - How does the system work?
       - Reverse engineering
     - How do I work with the system?
How?

- **Manual**
  - Typically interactive
    - Human intervention for more than initiation
  - Expectations **flexible**

- **Automated**
  - The test executes and evaluates on initiation
  - Automatically run (i.e., continuously)
Test Coverage(S)
• Esp. for **fine grained** tests, generality is a problem
• We want a **set** of tests that
  ■ determines some property
  ■ at a reasonable level of confidence
• This typically requires **coverage**
Coverage And Requirements

- Consider **acceptance** testing
  - For a test suite to **support** acceptance
    - It needs to provide information about all the critical requirements
- Consider **test driven development**
  - Where tests drive design
  - What happens without requirements coverage?
Code Coverage

- A test case (or suite) **covers a line of code**
  - if the running of the test executes the LOC
- Code coverage is a minimal sort of completeness
  - See McConnell on "basis" testing
    - Aim for **minimal** test suite with full code coverage
  - See `coverage.py`
  - Tricky bit typically involves **branches**
    - The more branches, the harder to achieve code coverage
Input Coverage

- Input spaces are (typically) too large to cover directly
  - So we need a **sample**
  - Pure **sample** probably inadequate
    - Space too large and uninteresting
  - We want a **biased** sample
    - E.g., **where the bugs are**
      - Hence, attention to boundary cases
    - E.g., **common inputs**
      - That is, what's *likely* to be seen
Situation/Scenario Coverage

- Inputs aren't everything
  - Machine configuration
  - History of use
  - Interaction patterns
- Field testing helps
  - Hence alpha plus narrow and wide beta testing
- **System tests** answer to this!
Limits Of (Developer) Testing

- Testing always has **limits**
  - Tests are **wrong**
  - Tests are **buggy**
  - Tests are **incomplete**
- "Self" Testing subject to cognitive biases
  - **Confirmation bias**: We interpret wrongly
  - **Observer-expectancy effect/Experimenter bias**: We influence others to interpret incorrectly
  - **Congruence bias**: We look in the wrong place
Developing Test Strategies

- Have one! However preliminary
  - Ad hoc testing rarely works out well
- Review it regularly
  - You may need adjustments based on
    - Individual or team psychology
    - Situation
- The McConnell basic strategy (22.2) is a good default
Developer Test Strategies

McConnell: 22.2 Recommended Approach to Developer Testing

- "Test for each relevant requirement" to make sure that the requirements have been implemented."
- "Test for each relevant design concern" to make sure that the design has been implemented... as early as possible"
- "Use "basis testing" ...At a minimum, you should test every line of code."
- "Use a checklist" of the kinds of errors you've made on the project to date or have made on previous projects."
- Design the test cases along with the product.
What About Input Coverage In WC?

- By reverse engineering `wc` we aim for an alternative python implementation
- With a clear `spec` according to CW1
- How can we achieve functional correctness of `miniwc`?
  - By achieving 100% input coverage to satisfy the specification
  - Let’s see some examples...
Empty Text File

```
bash-3.2$ touch empty_file.txt
bash-3.2$ wc empty_file.txt
 0 0 0 empty_file.txt
```
Common Case: 1 Line

bash-3.2$ echo "This is common text in one line" > common_1line_file.txt
bash-3.2$ wc common_1line_file.txt
    1    7   32 common_1line_file.txt
Common Case: 2 Lines

```bash
bash-3.2$ echo -e "This is common text in \n two lines" > common_2line_file.txt
bash-3.2$ wc common_2line_file.txt
 2   7   36 common_2line_file.txt
```
Visualising Potential Errors

- Guard against program **input**
  - What kind of file? Different types, wrong names...
  - Contents of file?
- Provide input coverage for every output **dimension**
  - Number of lines (single, multiple)
  - Number of characters (common case, large, small)
  - Number of words (how are words counted?)
  - Number of bytes (encoding?)
Coursework Recap
Coursework Activities

- Reading
- Q1
  - Mostly related to reading
  - Mostly "Recall"...with some interpretation
    - They will go higher on the Bloom taxonomy!
- SE1
  - Reading and analysing
- CW1
  - Reverse engineered a specification
  - Reengineered miniwc from the spec
    - Program construction
A Note On Marks

- UK marks run from 0-100%
  - <=49 = Failing (<40% serious failure)
  - 50-59 = Pass
  - 60-69 = Merit
  - over 70 = Distinction
  - NOTE THE WIDE BAND AT THE TOP AND BOTTOM
- A 65% is a good mark
- An 85% is exceedingly rare
- Over 70% is fairly rare
Q1

- Mean of 3.57 (71%)
  - Last year: 3.71 (74%)
- We will do some "in exam conditions"
- Let's delve
Simplified Problem

- This was a small problem
  - With clear boundaries
- Even here:
  - We ended up with support programs
    - And corners cut
- Software engineering is (complex) system engineering
  - On both the product and project sides
  - We use a complex infrastructure!
**Challenges**

*What were the challenges you encountered?*

*What challenges were inherent to the problem?*

*What challenges were environmental?*
CW1 Marks

- In Blackboard!
  - **Average**: 4/10 (40%)
  - **Max**: 8
    - >70% 6
    - **60%**s 14
    - **50%**s 5
    - <49% 26

- Not unusual for first assignment!
  - Learning curve
  - Final coursework average tends to be ≈62%
CW1 Feedback

- Feedback is in Blackboard
- Feedback is detailed but abstract

There seems to be a miniwc.py: 0.5/0.5 points.
There seems to be a doctest_miniwc.py and test files: 0.5/0.5 points.
Prohibited libraries have been used: 0/1 points. Formatting was correct: 1/1 points.
The script passed 15.1% of miniwc simple tests: 2/5 points.
The script passed 0.0% of miniwc binary tests: 0/1 points.
The script passed 0.0% of miniwc unicode tests: 0/1 points.
Penalties: none.
Total marks: 4.0/10.0
We're going to do more `wc`
- Job 1 is to **fix your** `miniwc.py` (now called `wc.py`)
  - Fix your tests!
  - Add **more** tests!
    - Unicode! Binary!
- Job 2 is to **add new functionality**
  - Flags! Multiple input files
- Job 3 is to **update your tests**
  - Note that Job 3 isn't *temporarily* last!

You will do a **code review** of `miniwc.py`
- **Discuss** the feedback!
- This is the **only time** you can talk with a classmate about it!
Other Coursework

- SE2!
  - You need to read *No Silver Bullet*
- SE1!
  - TAs are available to discuss