COMP61511 (Fall 2018)
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**!
print("""1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
16
17
Fizz
19
Buzz
Fizz
22
23
Fizz
A Rational FizzBuzz

- Let's consider a "standard" implementation
  - Not silly
  - Not golfy
- I.e., a simple **loop oriented implementation**
A Rational FizzBuzz (Source)

```python
for i in range(1, 101):
    if i % 3 == 0 and i % 5 == 0:
        print('FizzBuzz')
    elif i % 3 == 0:
        print('Fizz')
    elif i % 5 == 0:
        print('Buzz')
    else:
        print(i)
```
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!**
for i in range(1,101):
    fizz = i % 3 == 0
    buzz = i % 5 == 0
    if fizz and buzz:
        print('FizzBuzz')
    elif fizz:
        print('Fizz')
    elif buzz:
        print('Buzz')
    else:
        print(i)
EVEN DRIER!!!

- We repeat the `___ % ___ == 0` pattern!
- We say `print` a lot
- We can `fix it`!
EVEN DRIER!!! (Source)

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

def divisible_by(numerator, denominator):
    return numerator % denominator == 0

def fizzit(num):
    fizz = divisible_by(num, 3)
    buzz = divisible_by(num, 5)
    if fizz and buzz:
        return FIZZ + BUZZ
    elif fizz:
        return FIZZ
    elif buzz:
        return BUZZ
    else:
        return i

for i in range(1, 101):
    print(fizzit(i))
```
Parameterization

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

```python
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**!
Parameterization (Source)

"""We parameterise by:
* The range of integers covered.
* The text that is output.
* The multiples that trigger text to be output

https://www.tomdalling.com/blog/software-design/fizzbuzz-

def fizzbuzz(bounds, triggers):
    for i in bounds:
        result = ''
        for text, divisor in triggers:
            result += text if i % divisor == 0 else ''
        print(result if result else i)

fizzbuzz(range(1, 101), [
    ['Fizz', 3],
    ['Buzz', 5]])
Still Hard Coding!

- The **kind of test** is hard coded
- We can fix **that**!
def fizzbuzz(bounds, triggers):
    for i in bounds:
        result = ''
        for text, predicate in triggers:
            result += text if predicate(i) else ''
        print(result if result else i)

fizzbuzz(range(1, 101), [
    ['Fizz', lambda i: i % 3 == 0],
    ['Buzz', lambda i: i % 5 == 0],
    ['Zazz', lambda i: i < 10]
])
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](https://example.com/post) by Tom Dalling.

Tom uses Ruby and goes a couple of steps further. Worth a read!
Product Qualities

processverified.usda.gov
Qualities (Or "Properties")

- Software has a variety of **characteristics**
  - Size, implementation language, license...
  - User base, user satisfaction, market share...
  - Crashingness, bugginess, performance, functions...
  - Usability, prettiness, slickness...
"Quality" Of Success

- Success is determined by
  - the success criteria
    - i.e., the nature and degree of desired characteristics
  - whether the software fulfils those criteria
    - i.e., possesses the desired characteristics to the desired degree
Inducing Success

- While success is **determined** by qualities
  - the determination isn't **straightforward**
  - the determination isn't **strict**
    - for example, **luck** plays a role!
  - it depends on how you **specify** the critical success factors
## 20.1. Characteristics of Software Quality

<table>
<thead>
<tr>
<th>External Qualities</th>
<th>Non-Functional</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Functional</strong></td>
<td>Usability</td>
</tr>
<tr>
<td>Correctness</td>
<td>Efficiency</td>
</tr>
<tr>
<td>Accuracy</td>
<td>Reliability</td>
</tr>
<tr>
<td>Adaptability</td>
<td>Integrity</td>
</tr>
<tr>
<td></td>
<td>Robustness</td>
</tr>
<tr>
<td><strong>For Modification</strong></td>
<td>Readability</td>
</tr>
<tr>
<td>Maintainability</td>
<td>Understandability</td>
</tr>
<tr>
<td>Flexibility</td>
<td></td>
</tr>
<tr>
<td>Portability</td>
<td></td>
</tr>
<tr>
<td>Reusability</td>
<td></td>
</tr>
<tr>
<td><strong>For Comprehension</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Internal Qualities**

Testability
External Vs. Internal (Rough Thought)

- **External** qualities:
  - McConnell: those "that a user of the software product is aware of"
- **Internal** qualities:
  - "non-external characteristics that a developer directly experiences while working on that software"
- Boundary varies with the kind of user!
External Definition

- **External** qualities:
  - McConnell: those "that a user of the software product is aware of"
  - This isn't quite right!
    - A user might be aware of the implementation language
  - "characteristics of software that a user directly experiences in the normal use of that software"?
Internal Definition

- **Internal** qualities:
  - "non-external characteristics that a developer directly experiences while working on that software"
  - Intuitively, "under the hood"
External: Functional Vs. Non-Functional

- Functional $\approx$ **What** the software does
  - Behavioural
  - What does it accomplish for the user
  - Primary requirements
- Non-functional $\approx$ **How** it does it
  - Quality of service
    - There can be requirements here!
  - Ecological features
Key Functional: Correctness

- **Correctness**
  - Freedom from faults in
    - spec,
    - design,
    - implementation
  - Does the job
  - Fulfills all the use cases or user stories

Implementation and design could be perfect, but if there was a spec misunderstanding, ambiguity, or change, the software will not be correct!
External: "Qualities Of Service"

- **Usability** — can the user make it go
- **Efficiency** — wrt time & space
- **Reliability** — long MTBF
- **Integrity**
  - Corruption/loss free
  - Attack resistance/secure
- **Robustness** — behaves well on strange input

All these contribute to the user experience (UX)!
Internal: Testability

- A critical property!
  - Relative to a **target quality**
    - A system could be
      - highly testable for **correctness**
      - lowly testable for **efficiency**
  - Partly determined by test infrastructure
    - Having **great hooks** for tests pointless without **tests**
Internal: Testability

- Practically speaking
  - Low testability blocks *knowing*
    qualities
  - Test-based evidence is essential
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
- Both require significant effort!
Quality Levels

- We talked about different **kinds** of quality
  - Coming in degrees or **amounts**
  - "Easy" example: Good vs. poor performance
- Most qualities in principle are **quantifiable**
  - Most things are quantifiable
- But reasonable quantification isn't always **possible**
  - Or **worth it**
Defects As Quality Lacks

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

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
Problem Definition

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>45%</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>15%</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>15%</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 1986a), "Software Defect-Removal Efficiency" (Jones 1996), and "What We Have Learned About Fighting Defects" (Shull et al. 2002).
Defect Detection Techniques

- Regression test
- Informal code reviews
- Unit test
- New function (component) test
- Integration test
- Low-volume beta test (< 10 users)
- Informal design reviews
- Personal desk checking of code
- System test
- Formal design inspections
- Formal code inspections
- Modeling or prototyping
- High-volume beta test (> 1000 users)
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
Lab!

KEEP CALM AND CARRY ON
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
Rainfail

- 14 out of 29 students submitted
- Key point: 1 out of 15 programs passed all 13 tests
  - 1 program passed ALL tests
  - 1 passed 9
  - 6 passed 8
  - 3 passed 6
  - 1 passed 4
  - 1 passed 0
  - 2 "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*
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" #<-- change this!

$ python lsetup.py
Your computed average
First Test Run

$ python -m doctest lsetup.py

Failed example:
average_rainfall(<<FIRST TEST INPUT>>)
Exception raised:
SyntaxError: invalid syntax

1 items had failures:
  1 of 1 in lsetup.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, -999]`
    - We had to *massage* the input to our test!
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" #<-- change this!

$ python 1setup.py
Your computed average
Second Test Run

$ python -m doctest 2firstfull.py
*************************************************
File "~/Users/bparsia/Documents/2018/Teaching/COMP61511/la
Failed example:
   average_rainfall([2,3,4,67, -999])
Expected:
   19.0
Got:
   'Your computed average'
*************************************************
1 items had failures:
   1 of  1 in 2firstfull.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):
    """>> average_rainfall([2,3,4,67, -999])
    19.0
    """
    # Here is where your code should go
    return sum(input_list)/len(input_list)
```

- Will this fail this test?
- Is there a test that it will pass?
def average_rainfall(input_list):
    """""">> average_rainfall([2,3,4,67, -999])
    19.0
>>> average_rainfall([2,3,4,67, -999])
    19.0
    """

    # Here is where your code should go
    return sum(input_list)/len(input_list)
Third Test Run

```
$ python -m doctest 4firstimpl2.py
**********************************************************************
File "/Users/bparsia/Documents/2018/Teaching/COMP61511/la
Failed example:
   average_rainfall([2,3,4,67, -999])
Expected:
   19.0
Got:
   -184.6
**********************************************************************
1 items had failures:
  1 of  2 in 4firstimpl2.average_rainfall
***Test Failed*** 1 failures.
```
Second Implementation

```python
def average_rainfall(input_list):
    """>>> average_rainfall([2,3,4,67, -999])
    19.0
    >>> average_rainfall([2,3,4,67 -999])
    19.0
    """
    # Here is where your code should go
    return sum(input_list[:-1])/len(input_list[:-1])
```

- Fixes one test but not the other!
- Tests **work together**
def average_rainfall(input_list):
    """"""""""""""""""""""
    >>> average_rainfall([2, 3, 4, 67, -999])
    19.0
    >>> average_rainfall([2, 3, 4, 67, -999])
    19.0
    """""""""""""""""
    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/la
Failed example:
   average_rainfall([2,3,4,67, -999])
Expected:
   19.0
Got:
   19.0
********************************************************************
1 items had failures:
   1 of   2 in 5secondimpl.average_rainfall
***Test Failed*** 1 failures.

Whaaaaaannnnnnaaaatat?!
A Bug!

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

```
def average_rainfall(input_list):
    """>>> average_rainfall([2, 3, 4, 67, -99])
    19.0
```

vs.

```
def average_rainfall(input_list):
    """    >>> average_rainfall([2, 3, 4, 67, -99]
    19.0
```

- Earlier tests failed for *two reasons*!
- One bug *concealed* the other!!!
Yay!

```
$ python -m doctest 6secondimpl2.py
$
$ python -m doctest -v 6secondimpl2.py
Trying:
    average_rainfall([2,3,4,67, -999])
Expecting:
    19.0
ok
Trying:
    average_rainfall([2,3,4,67, -999])
Expecting:
    19.0
ok
1 items had no tests:
    6secondimpl2
1 items passed all tests:
    2 tests in 6secondimpl2.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, −999]
    - 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? (Ctnd 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)
Coverage


• 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-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  35 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?)