MANCHESTER 1824
The University of Manchester

COMP61511 (Fall 2019)

Software Engineering Concepts In Practice

Week 1

Bijan Parsia & Christos Kotselidis <bijan.parsia, christos.kotselidis@manchester.ac.uk>

(bug reports welcome!)

The Field

- Software engineering aims to
  - produce successful **software systems**
  - via successful **development projects**

Programming is only one aspect!

Topic Overview

- We look at different topics
  - The nature of **complex systems**
  - **System construction** (coding, debugging, testing, problem definition)
  - **Professionalism** and professional life
  - **Social/ethical/legal** issues

(We may adjust the topics as needed or desired)

Course Goals

- This course unit aims to give students a
  1. a strong **conceptual understanding** of software engineering
  2. the ability to relate that conceptual understanding to **practical application**
  3. familiarity with the software engineering **scientific and practioner literature**
     - sufficient to drive ongoing learning

Course Structure

- **Lectures**
  - Active learning
- **Labs & Coursework**
  - Make sure you understand the coursework!
- **Readings**
  - Most readings available online (in one form or another)
  - Lectures go beyond texts
  - Texts go beyond lectures

Texts

- **Practitioner's** text:
  - Steve McConnell, *Code Complete: A Practical Handbook of Software Construction*
  - Physical copies and available as ebook in Safari Online
- **Researcher's** text:
  - Andy Oram and Greg Wilson (eds), *Making software what really works, and why we believe it*
  - Physical copies and as ebook
- Online posts and research papers.

Some Issues

- The Library's e-versions are **rate limited**
  - Limited concurrent users and about 10 physical copies

Mitigations

- You can **purchase** the books
  - Code Complete (£15.92); Making Software (£19.89)
    - Prioritise Code Complete (for this class)
  - Safari Books Online Subscription service
    - 10 day free trial
    - £39 a month (and can cancel)

A Note On The Texts

They are very good, but...

...software engineering is not a **settled** field.

Read critically and look for the most recent evidence.

(The early chapters of Making Software are helpful for this!)

Additional Core Text

- *Guide to the Software Engineering Body of Knowledge (SWEBOK Guide)*
    - Free PDF.
    - Not a textbook, but a good touchstone about what a pro should know
    - Extensive coverage and bibilography. Fairly readable.
    - But a guide, not an embodiment of the Body of Knowledge

Assessment

- Coursework (**50%**)
  - Each week, a mixture (not all every week)
    1. MCQ quizzes
    2. Short essays
    3. Individual engineering assignments
  - Precise mark breakdown varies
- Exam (**50%**)
  - Taken **online**
  - Very like 1 & 2

Materials & Blackboard

- All course materials are available online
  - http://syllabus.cs.manchester.ac.uk/pgt/COMP61511/
- We use Blackboard for
  - Coursework
  - Online forum
    - Use this!
    - Subscribe!
    - **We care** and want to help
  - Exam

Variant Circumstances

- Disability
  - **Equality act definition**: is any condition which has a significant, adverse and long-term effect on a person's ability to carry out normal day-to-day activities.
  - Disability Advisory and Support Service
    - Exam & Study support
    - Great, helpful people

Feel free to consult a course instructor about this! We're **happy** to advise.

Variant Circumstances

- Mitigating circumstances
  - a **temporary** condition or situation which has a significant, adverse effect on a person's ability to carry out normal day-to-day activities
- Help available from SSO & Counselling service & your GP
- Medical (mental or physical) issues are not the only mitigating circumstances!

A Note About Assistance

- **Early intervention** is more effective
  - If you are having **challenges** of any sort
    - the sooner known **by us**, the better handled
- This is very true for **mitigating circumstances**
  - If something is interfering, **document** it!
  - There is a "too late" here!

Again, when in doubt, **ask us**.

ALL late work is handled by the MitCircs committee, not us.

Expected Conduct

- We expect of you (and ourselves)
  - To be fair minded
  - To treat each other well
  - To avoid academic malpractice
  - To take responsibility for course duties
  - To be engaged, curious, and active
- If you have a problem or issue
  - Please raise it with us
  - If that seems unhelpful, contact your course tutor

Academic Malpractice

- We take it very seriously!
  - Most common forms: **Plagiarism & Collusion**
- The (few) points you can get aren't worth losing your degree!
- Guard against them!
  - Don't cut-paste-modify
  - Quote and cite text
  - Cite ideas
  - **Do not discuss assignments with other students**
    - Use the discussion board!

Preliminaries

What Is Software Engineering?

- The production of **software systems** whether
  - "standalone"
  - components of larger systems
- Most software **interacts** with
  - various forms of hardware
  - other software systems
  - people!

Software engineering is increasingly seen as a branch of **systems engineering**

What Is System Engineering?

Systems engineering is a ***methodical, disciplined*** *approach for the design, realization, technical management, operations, and retirement of a system.*
— *NASA System Engineering Handbook*

What Is System Engineering?

A "system" *is a construct or* ***collection of different elements*** *that together produce results* ***not obtainable by the elements alone****. The elements, or parts, can include people, hardware, software, facilities, policies, and documents; that is, all things required to produce system-level results.*
— *NASA System Engineering Handbook*

What Is System Engineering?

- a **methodical**, **disciplined** approach for the
  - design,
  - realization,
  - technical management,
  - operations, and
  - retirement
- of a **system**.

— NASA System Engineering Handbook

What Is System Engineering?

A "system" is a **construct** or collection of
different elements
that together produce
results
not obtainable by the elements alone.

The elements, or parts, [include]...
all things required
to produce system-level results.

— NASA System Engineering Handbook

(Complex) Systems

System **results** are **emergent**

1. The **problem** being tackled is complex, **amorphous**, or evolving
2. There is **heterogeneity in the components** and how they **interact**
3. The system design takes into account the **whole lifecycle**
4. Thus the **design space** is very large and complex

One View Of The Design Space

- Even if we **super simplify** it:

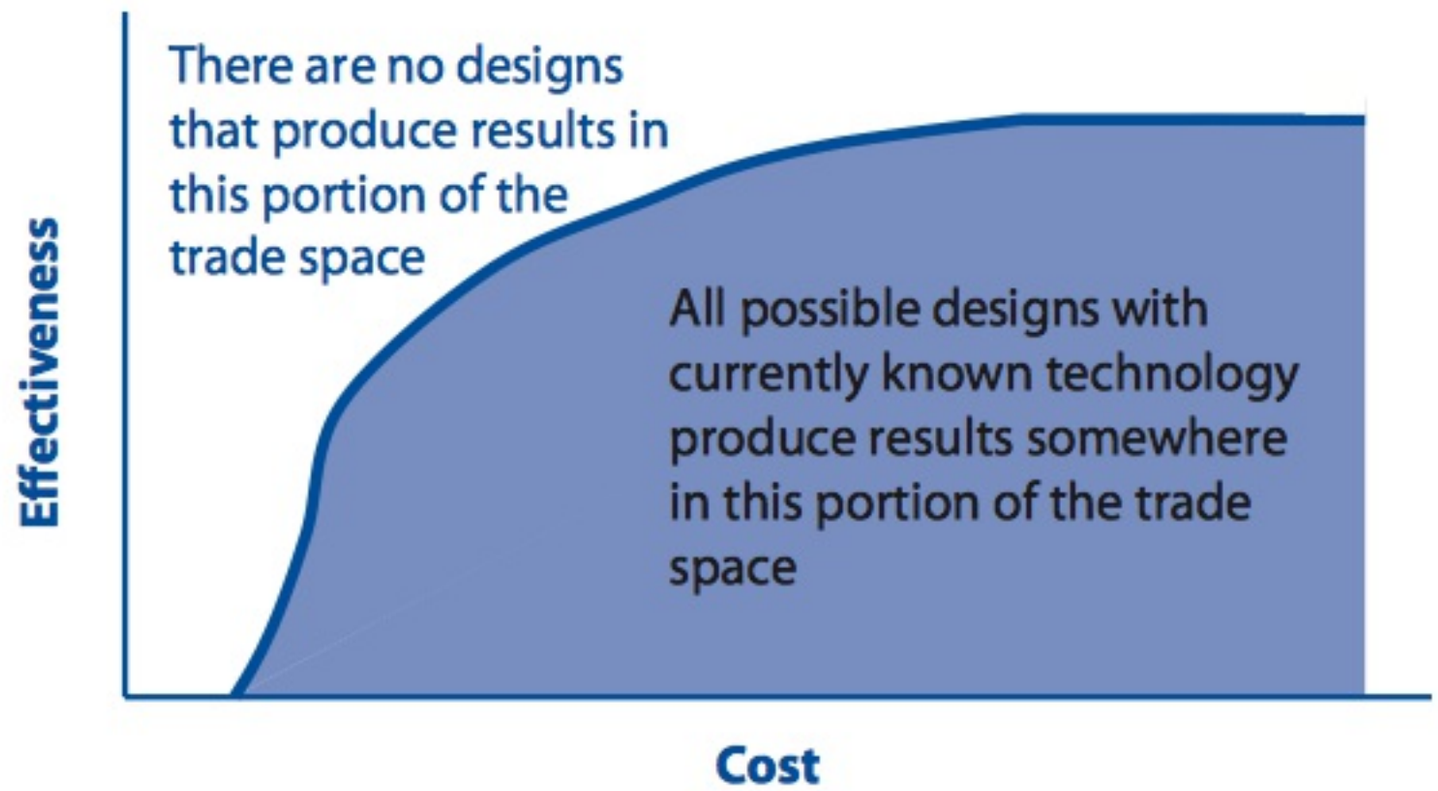
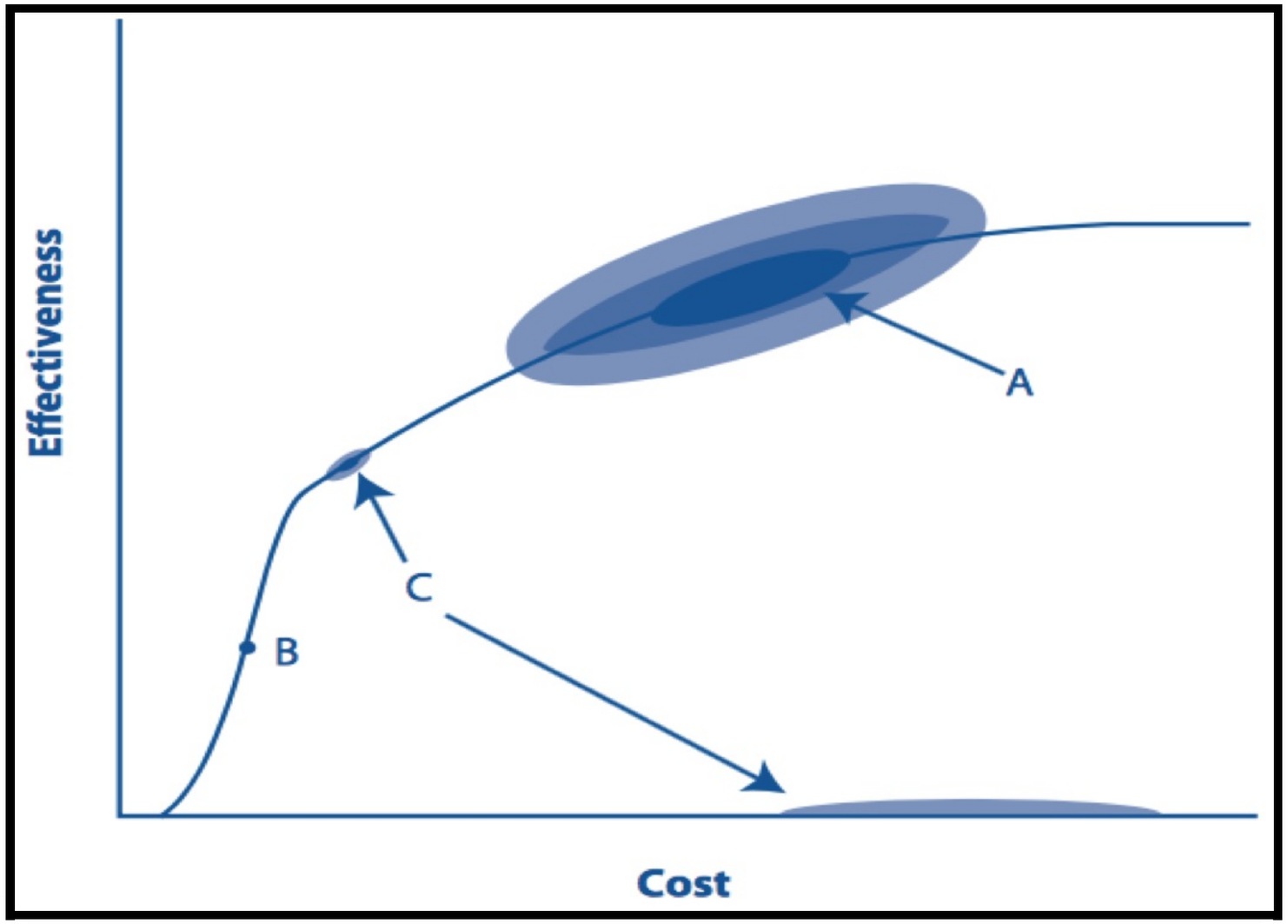


Figure 2.5-1 The enveloping surface of nondominated designs

One View Of The Design Space

- It's not so simple:



Fred Brooks: No Silver Bullet

The complexity of software is in ***essential property****, not an accidental one. Hence descriptions of a software entity that abstract away its complexity often abstract away its essence.*

Fred Brooks: No Silver Bullet

Many of the classical problems of developing software products derived from this essential complexity and ***its [super]linear increase[] with size****.*

Fred Brooks: No Silver Bullet

Much of the complexity [the software engineer] must master is ***arbitrary complexity****, forced without rhyme or reason by the many human institutions and systems to which his interfaces must confirm. These differ from interface to interface, and from time to time,* ***not because of necessity but only because they were designed by different people***

Question Time!!

- The Boeing Dreamliner requires about
    1. 700,000 lines of code.
    2. 1.7 million lines of code.
    3. 5.7 million lines of code.
    4. 6.5 million lines of code.

Software Creep (Planes!)

Software takes over!

The F-35 Joint Strike Fighter, scheduled to become operational in 2010, will require about ***5.7 million lines of code*** *to operate its onboard systems. And Boeing's new 787 Dreamliner, scheduled to be delivered to customers in 2010, requires about* ***6.5 million lines*** *of software code to operate its avionics and onboard support systems.*

Question Time!!

- A premium-class automobile probably contains
    1. 100,000 lines of code.
    2. 1 million lines of code.
    3. 10 million lines of code.
    4. 100 million lines of code.

Software Creep (Cars!)

Software takes over!

These are impressive amounts of software, yet if you bought a premium-class automobile recently, "it probably contains close to ***100 million lines of software code****,"... All that software executes on 70 to 100 microprocessor-based electronic control units (ECUs) networked throughout the body of your car.*

Software Creep (Future Cars!)

Software takes over!

Late last year, the business research firm Frost & Sullivan estimated that cars will require ***200 million to 300 million lines of software code*** *in the near future.*

Software Creep (Future Cars!)

Software takes over!

*Broy estimates that more than **80 percent of car innovations come from computer systems** and that software has become the major contributor of value (as well as sticker price) in cars.*

Question Time!!

- Software creep occurs because
    1. the cost of hardware grows faster than the cost of software.
    2. the capabilities of mechanical systems grows slower than the cost of software.
    3. the capabilities of mechanical systems is hard to improve relative to the capabilities of software.
    4. software can be updated easily.

Millions Of Lines Of Code!

- Let's look at some code base sizes!
  - A visualisation
  - A spreadsheet
  - A discussion

(Can we believe these estimates?)
(How do we interpret them?)
(Was Brooks wrong?)
(Were these slides wrong?)

Non Complex Software Systems?

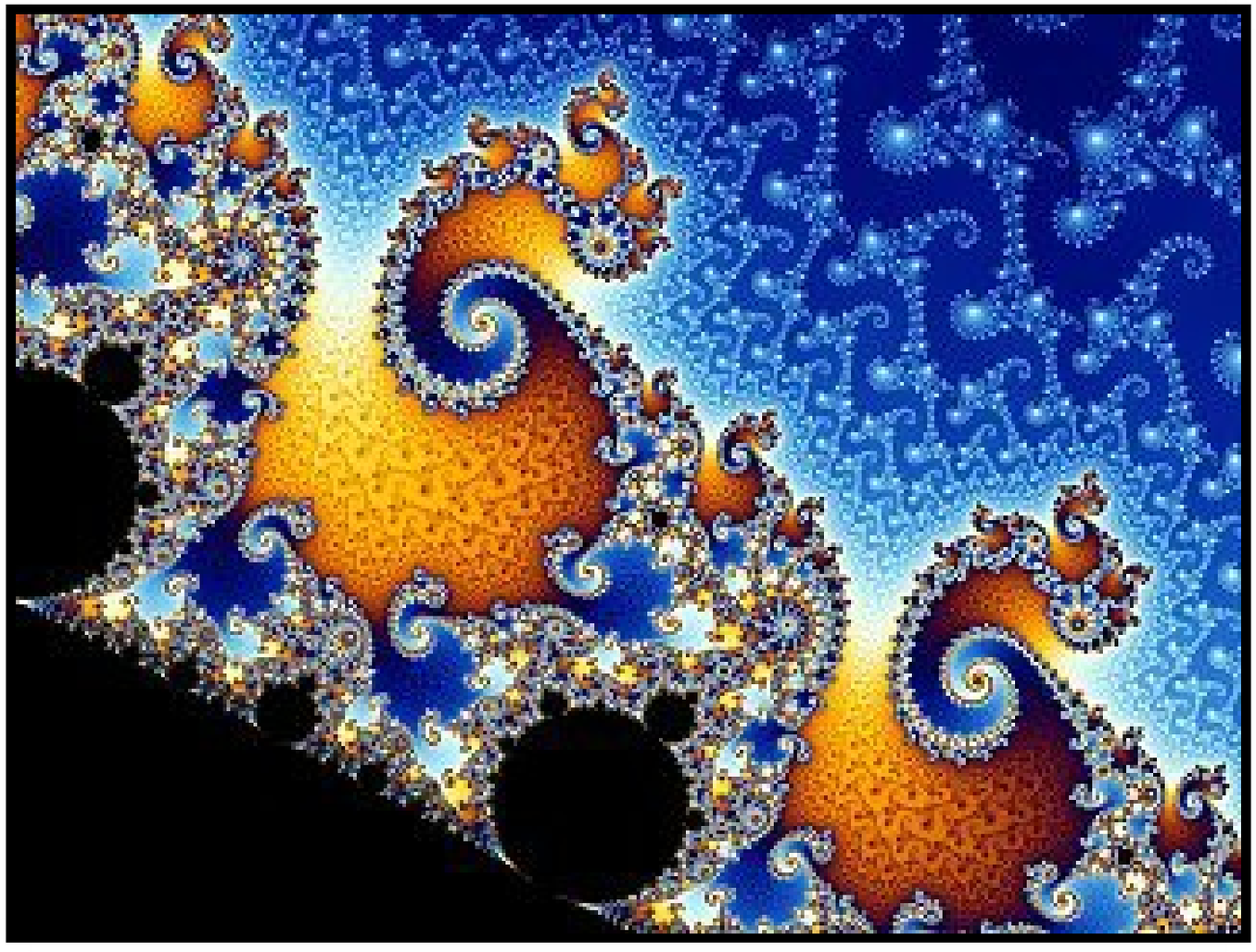
Averaging problem: Write a program that will read in integers and output their average. Stop reading when the value 99999 is input.
— *Soloway*

- Is a program that solves the "rainfall problem"
  1. a complex system?
  2. part of a complex system?
  3. a simple system?

Lab 1

- About **1hr**
  - We'll play it a bit by ear
- Lab materials are **online**
  - Long URL: http://studentnet.cs.manchester.ac.uk/pgt/COMP61511/labs/lab1/
  - Short URL: bit.ly/wk1lab1

Problem Complexity



FizzBuzz Buzz

Most good programmers should be able to write out on paper a program which does this in a under a couple of minutes.

*Want to know something scary ? – **the majority of comp sci graduates can't**. I've also seen self-proclaimed senior programmers take more than 10-15 minutes to write a solution.*

—Imran Ghory

FizzBuzz

But I am disturbed and appalled that any so-called programmer would apply for a job without being able to write the simplest of programs. That's ***a slap in the face*** *to anyone who writes software for a living.*

—Jeff Atwood

Is this a good attitude?

FizzBuzz Us!

- 29 students enrolled
- 24 on time submissions
- 22 `.zip` (1 `.jpg`, 1 `.py`)
  - 4 wrongly named! (e.g., `Bijan_Parsia_Lab0.zip)
  - 3 didn't unzip to a directory (1 also wrong named)
  - 16/24 = 72%!
- 5 were "eyeball" detectably wrong

45% success rate!!!

Let's look!

FizzBuzz Golf!

- We had 5 plasyers (out of 22 submissions)
- Sizes ranged from 83 (the winner!) to 533
- Winner!

```
for i in range(1, 101):
    print((not i % 3) * "Fizz" + (not i % 5) * "Buzz" or i)
```

Last year's:

```
for i in range(1,101): print('Fizz'[i%3*4:]+'Buzz'[i%5*4:]or i)
```

FizzBuzz Critique

Here's a clue for you: ***I don't do well in programming tasks during interviews****, and I've love someone to come into my comments and tell me I can't program based on this event. No, I've only faked it while working for Nike, Intel, Boeing, John Hancock, Lawrence Livermore, and through 14 or so books–not to mention 6 years of online tech weblogging.*

— Shelley Powers

FizzBuzz Critique 2

In fact, you'll find a lot of people who don't necessarily do well when it comes to programming tasks or other complex testing during job interviews. Why? Because the part of your brain that manages complex problem solving tasks is the first that's more or less scrambled in high stress situations. ***The more stress, the more scrambled.*** *The more stressed we are, the more our natural defensive mechanisms take over, and the less energy focused into higher cognitive processes.*

— Shelley Powers

FizzBuzz Complexity

- Of the **question** itself
  - What **constructs** does FizzBuzz require?
  - What kind of errors **can** (reasonably) happen?
- Of the **use** of the question
  - What can we **conclude** about a FizzBuzz failure?
  - Are **environmental factors** significant?
  - Does **widespread awareness** of FizzBuzz questions invalidate them?

The Rainfall Problem (Results)

Source	Cohort	Results							
		Number of Students participated/submitted	Number of students in course/class/total	Fully correct percentage NEGATIVE required	Fully correct percentage NEGATIVE not required *	Correct when DIVZERO ignored	Nearly correct [7] percentage	No correct subplans / No (pertinent) code	Average partial score
This Article	Context 1	151	> 550	45 %	n/a	66 %	+37%	0 %	88 %
	Context 2	192	243	53 %		70 %	+36%	1 %	89 %
	Context 3	165	236	72 %		77 %	+20%	0 %	94 %
Fisler [6]	T1	61	154	54% [1]	n/a	74% [1]	+21-23% [4]	3-6% [4]	?
	T1Acc	44	44	39% [1]		52% [1]			
	T2	63	224	11% [1]		22% [1]			
	T3Non	43	65	2% [1]		14% [1]			
	HS	7	7	0% [1]		5% [1]			

—Do We Know How Difficult the Rainfall Problem is?

Rainfall Complexity

Rainfall requires students to compose code implementing multiple tasks. Our version (in Section 2) requires six tasks:

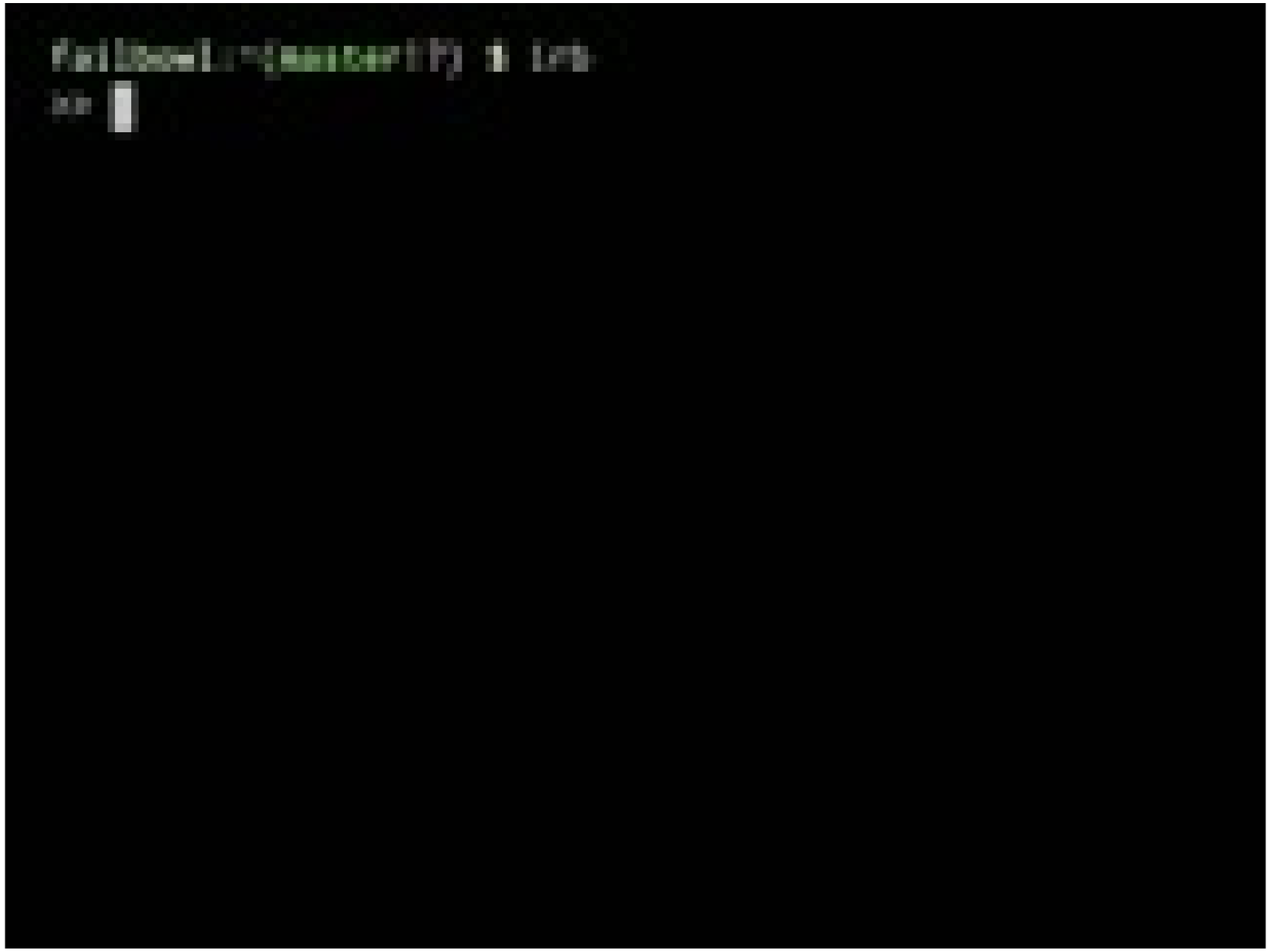
- Sentinel: Ignore inputs after the sentinel value
- Negative: Ignore negative inputs
- Sum: Total the non-negative inputs
- Count: Count the non-negative inputs
- DivZero: Guard against division by zero
- Average: Average the non-negative inputs

— The Recurring Rainfall Problem

Rainfall Solution

```
def average_rainfall(input_list):
    # Here is where your code should go
    total = 0
    count = 0
    for m in input_list:
        if m == -999:
            break
        if m >= 0:
            total += m
            count += 1
        # ignore other negatives
    avg = 0 if count == 0 else total / count
    return avg
```

Wat Or "Hidden Complexity"



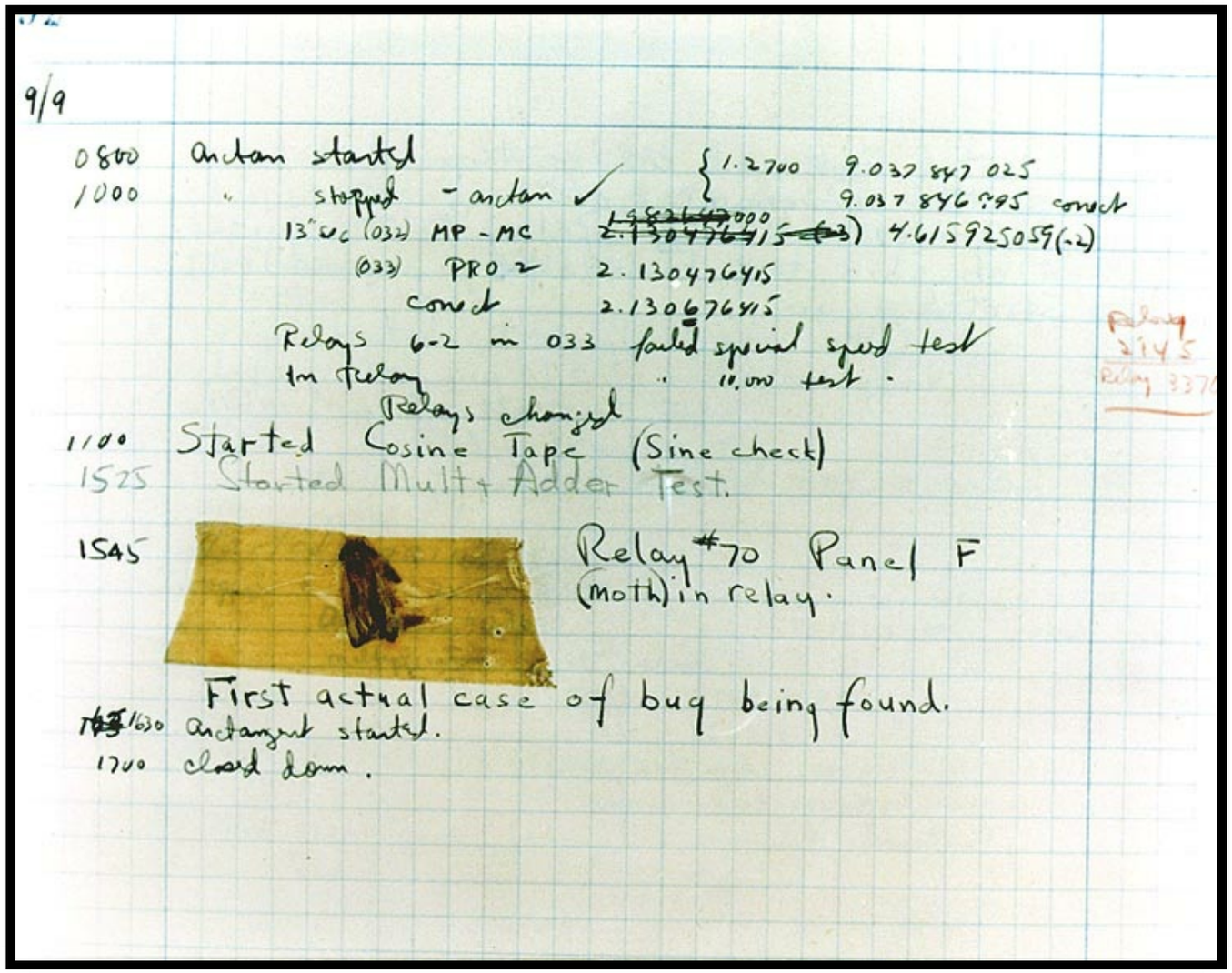
Fred Brooks: No Silver Wat

Much of the complexity [the software engineer] must master is ***arbitrary complexity****, forced without rhyme or reason by the many human institutions and systems to which his interfaces must confirm. These differ from interface to interface, and from time to time,* ***not because of necessity but only because they were designed by different people***

Testing Correctness

Beware of bugs *in the above code; I have only* ***proved*** *it correct, not* ***tried*** *it.*
— Don Knuth, Figure 6: page 5 of classroom note

Really Beware Of Bugs!



—Grace Hopper's Bug Report

Developer Testing

- We can distinguish between
  - Testing done by **non-specialists** (McConnell: "Developer testing")
    - For many projects, the only sort!
  - Testing done by (test) **specialists**
- If you **compile** and **run** your code
  - Then you've done a test! (or maybe two!)
    - If only a **"smoke" test**

*Testing is **inescapable**; **good** testing takes **work***

Question Time!

- If you **compile** your code
    1. you have tested it for syntactic correctness.
    2. you have tested it for semantic correctness.
    3. you have tested it for both.
    4. you haven't tested it at all.

What Is A Test?

*A **test case** is a **repeatable** execution situation of a software **system** that produces recordable outcomes. A **test** is a **particular attempt** of a test case*

What Is A Test?

- The outcomes may be **expected**
  - E.g., we expect passing 1+1 to a calculator to return 2
  - Generally boolean outcomes (**pass** or **fail**)
    - We might have an **error** that prevents completion
- The outcomes may be **measurement** results
  - E.g., we want to find the **time** it takes to compute 1+1

What Is A Test? (3)

- The outcomes should testify to some software **quality**
  - E.g., correctness, but also efficiency, usability, etc.
- A (single) test specifies a **very particular** quality
  - E.g., correct **for a given input**
  - E.g., uses X amount of memory **for this scenario**

*The **fundamental challenge** of testing is **generalisability***

Generalisability Problem (1)

Testing shows the ***presence****, not the* ***absence*** *of bugs.*
—Edsger W. Dijkstra, Nato Software Engineering Conference, pg 16 1969

What Is A Test Case?

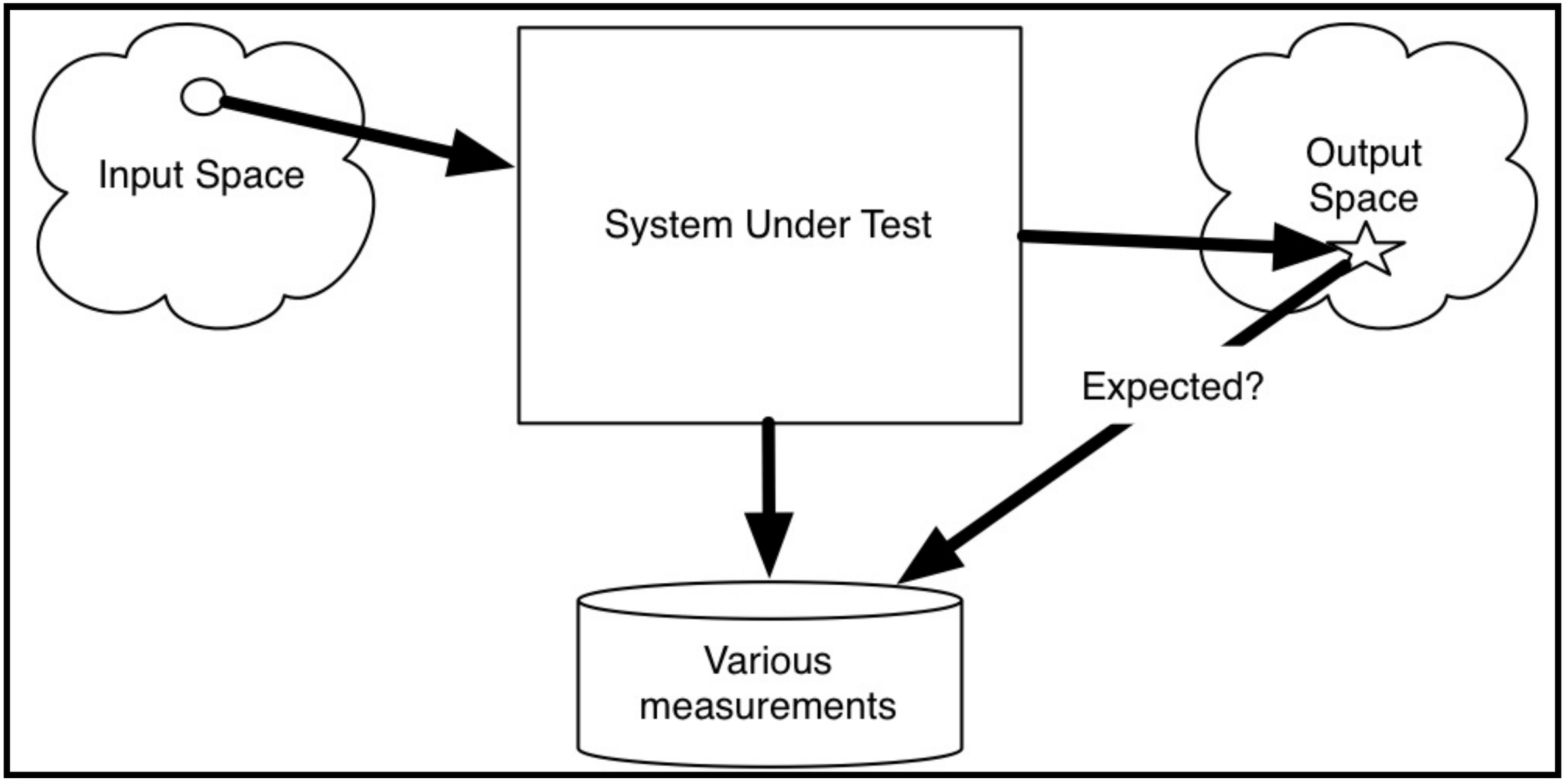
A test case is a ***specified*** *input with an* ***expected output****; if the program being tested gives, for the given input, an output equivalent to the expected one then that test has* ***passed****; otherwise* ***failed****.*

Terminology Note

- **Test** and **test case** are often used interchangeably
  - And in other loose ways
  - Most of the time it doesn't matter because easy to distinguish
- We often talk about a **test suite** or **test set**
  - But this also might be subordinated to a *test*
  - For example,

> *"We used the following **test suite** to **stress test** our application".*

Anatomy Of A Test (1)



Generalisability Threat

- A test case (A):
  - **Goal:** Correctness to the specification
    - **Input:** a pair of integers, `X and Y`
    - **Output:** the integer that is their `sum`
  - **Test Input:** `X=1` and `Y=1`
  - **Expected output:** `2`
- Test result of System S is **pass**
  - What can we conclude?

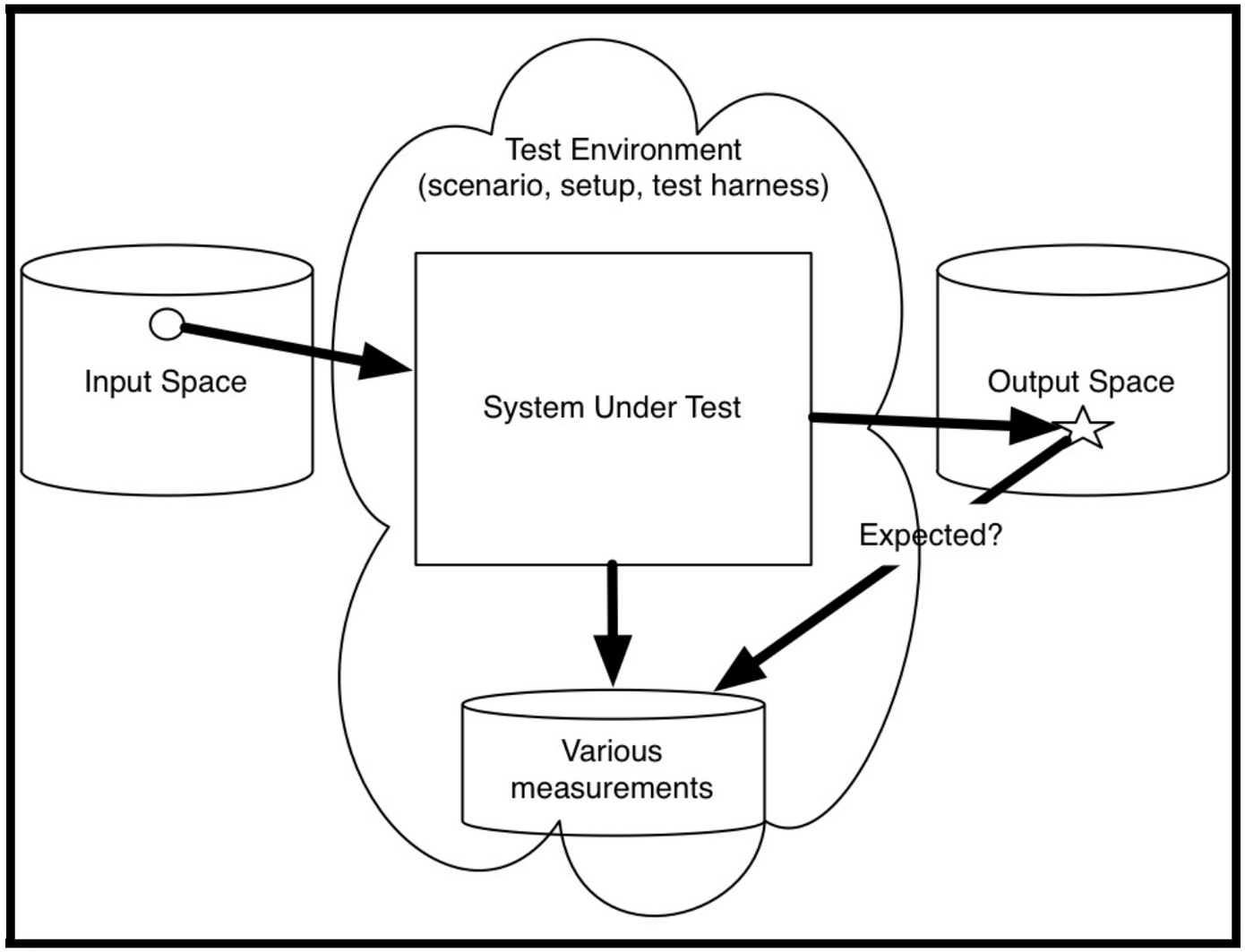
Question Time!

- From the test result **Pass test case A**, we can conclude that:
    1. System S **correctly implements** the specification.
    2. System S **correctly implements** the specification **for this input**
    3. Both 1 and 2
    4. Neither 1 nor 2

Question Time!

- From the test result **Fail test case A**, we can conclude that:
    1. System S **does not correctly implement** the specification.
    2. System S **does not correctly implement** the specification **for this input**
    3. Both 1 and 2
    4. Neither 1 nor 2

Anatomy Of A Test (2)



Environment Matters

The next most significant subset of [Modification Requests (MRs)] were those that ***concern testing*** *(the testing environment and testing categories)—****24.8% of the MRs****. ...it is not surprising that a significant number of problems are encountered in testing a large and complex real-time system...First, the* ***testing environment*** *itself is a* ***large and complex system that must be tested****. Second, as the real-time system evolves, so must the* ***laboratory test environment evolve****.*
Making Software, pg. 459.

A Good Test

- A good test case is
  - **part** of a suite of test cases
  - **understandable**
    - i.e., you can relate it
      - to the spec
      - to the system behavior
  - **fits in** with the test environment
  - is (given the suite) **informative**

Environment Matters

Table 25-1 Summary of faults (modified) *Making Software pg 459*

MR Category	Proportion		
Previous	4.0%		
Requirements	4.9%		
Design	10.6%	**D+C**	**D+C+R**
Coding	18.2%	28.8%	33.7%
Testing Environment	19.1%	**TE+ T**	
Testing	5.7%	24.8%	
Duplicates	13.9%		
No problem	15.9%		
Other	7.8%		

Word Count (wc)

- wc is a ubiquitous small tool
  - Goes back to at least 1971
  - Still in active use
- This will be our go to example for a while!
  - Esp. in the lab!

Word Count

```
WC(1)                    BSD General Commands Manual                    WC(1)

NAME
     wc -- word, line, character, and byte count

SYNOPSIS
     wc [-clmw] [file ...]

DESCRIPTION
     The wc utility displays the number of lines, words, and bytes contained in each input file, or standard input (if no file is specified) to the
     standard output.  A line is defined as a string of characters delimited by a <newline> character.  Characters beyond the final <newline> charac-
     ter will not be included in the line count.

     A word is defined as a string of characters delimited by white space characters.  White space characters are the set of characters for which the
     iswspace(3) function returns true.  If more than one input file is specified, a line of cumulative counts for all the files is displayed on a
     separate line after the output for the last file.

     The following options are available:

     -c      The number of bytes in each input file is written to the standard output.  This will cancel out any prior usage of the -m option.

     -l      The number of lines in each input file is written to the standard output.

     -m      The number of characters in each input file is written to the standard output.  If the current locale does not support multibyte charac-
             ters, this is equivalent to the -c option.  This will cancel out any prior usage of the -c option.

     -w      The number of words in each input file is written to the standard output.

     When an option is specified, wc only reports the information requested by that option.  The order of output always takes the form of line, word,
     byte, and file name.  The default action is equivalent to specifying the -c, -l and -w options.
```

Word Count Testing

- Create your own **Python** version of WC
- With a specification based on the original done
- `man wc`
    - does it cover everything?
    - how can test against the exact spec?
    - Reverse Engineering to the rescue!

WC Reverse Engineering

Reverse engineering *is the process of* ***analyzing*** *a subject system to identify the system's components and their interrelationships and to create representations of the system in another form or at a higher level of abstraction. -IEEE*

Let's Start With Something Small

- What about an empty file?
  - Spec 1: **if** the **input** of wc is an empty file, it **returns** 0 for all outputs
  - Note: **Defined** input, **Expected** output

```
bash-3.2$ touch empty_file
bash-3.2$ wc empty_file
       0       0       0 empty_file
```

What About A Normal Case?

```
bash-3.2$ echo 'My name is Christos' > test_file
bash-3.2$ wc test_file
       1       4      20 test_file
```

Building The Specification

- By reverse engineering `wc` we:
  - understand better the behavior of our program
  - are building the specification
  - design our test cases!
- What if we build **new** software?
  - Customer specification
  - Agile or BUFR requirements gathering

Designing Tests

- Not a trivial process
- Numerous parameters to factor in
- Common case vs Corner cases
- Can we test everything?
    - In principle **No**, hence appropriate test selection is key to success
    - Beware of **Trade-offs**
    - Test coverage vs Execution Time vs Resources

Doctest

- **DocTest** is a unit testing framework for Python
  - Will be used during out lab sessions!
  - Provides an easy-to-use and modular interface for:
  - Isolated Testing
  - Regression Testing
  - Or extended for all kinds of testing (almost!)
- More on the lab this afternoon!

Simple Example

```
 # Everything is in a docstring!
"""
>>> 1+1
2
>>> 1+1 #Note that the supplied expected answer is *wrong*. This test will fail
3
>>> [1, 2, 3][1]
2
"""

 # We add the boilerplate to make this module both executable and importable.
if __name__ == "__main__":
    import doctest
    # The following command extracts all testable docstrings from the current module.
    doctest.testmod()
```

Unit Testing WC

```
>>> import subprocess
>>> subprocess.check_output('wc test1.txt', shell=True)
b'      10      10      20 test1.txt\n'
```

Product Qualities

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

Software Quality Landscape

20.1. Characteristics of Software Quality

External Qualities	
Functional Correctness Accuracy Adaptability	**Non-Functional** Usability Efficiency Reliability Integrity Robustness
For Modification Maintainability Flexibility Portability Reusability	**For Comprehension** Readability Understandability
Testability	
Internal Qualities	

External Vs. Internal (Rough Thought)

- **External** qualities:
  - McConnell: those "that a user of the software product is aware of"
- **Internal** qualities:
  - "non-external characterisitcs 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 langauge
  - "characteristics of software that a **user** directly experiences in the normal use of that software"?

Internal Definition

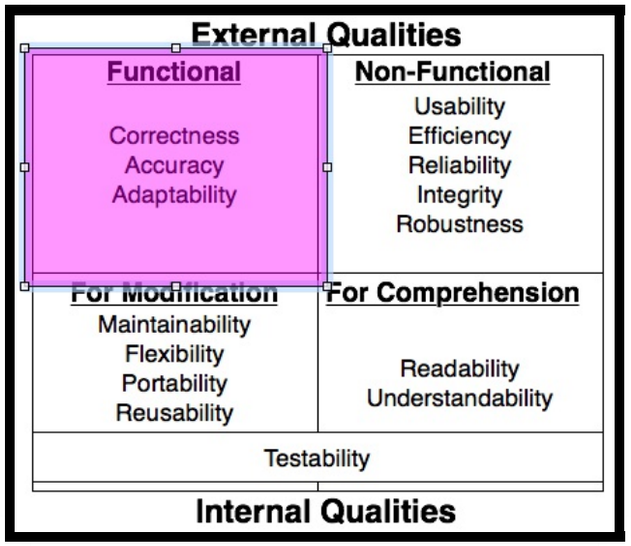
- **Internal** qualities:
  - "non-external characterisitcs that a **developer directly experiences while working on that software**"
  - Intuitively, "under the hood"

External: Functional Vs. Non-Functional

- Functional ≈ **What** the software does
  - Behavioural
  - What does it accomplish for the user
  - Primary requirements
- Non-functional ≈ **How** it does it
  - Quality of service
    - There can be requirements here!
  - Ecological features

Key Functional: Correctness

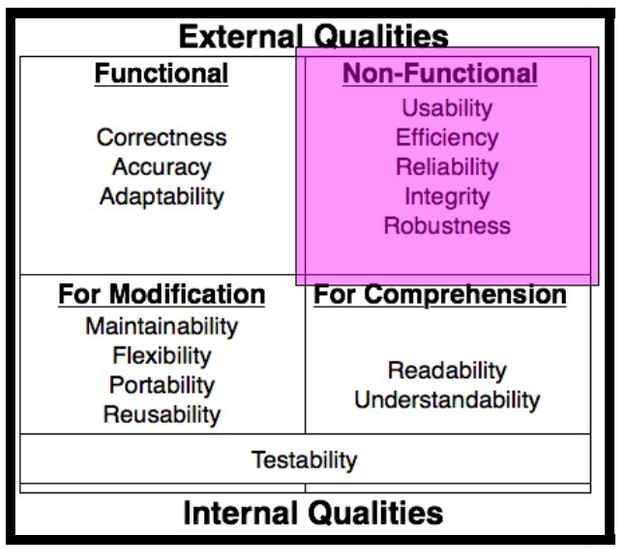
- **Correctness**
  - **Freedom from faults** in
    - spec,
    - design,
    - implementation
  - **Does the job**
    - Fulfills all the **use cases** or **user stories**



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 **correctenss**
      - 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

Quality Interactions: External

20.1, Code Complete

How focusing on the factor below affects the factor to the right	Correctness	Usability	Efficiency	Reliability	Integrity	Adaptability	Accuracy	Robustness
Correctness	↑		↑	↑			↑	↓
Usability		↑				↑	↑	
Efficiency	↓		↑	↓	↓	↓	↓	
Reliability	↑			↑	↑		↑	↓
Integrity			↓	↑	↑			
Adaptability					↓	↑		↑
Accuracy	↑		↓	↑		↓	↑	↓
Robustness	↓	↑	↓	↓	↓	↑	↓	↑

Helps it ↑

Hurts it ↓

Figure 20-1. Focusing on one external characteristic of software quality can affect other characteristics positively, adversely, or not at all

Coursework!

- There are **four bits** of coursework
  - Readings (see materials page) and the lab pages
  - A short Multiple Choice Question (MCQ) **Quiz** (Q1) related to (some of the) readings
  - A short **Essay** (SE1) related to a reading
  - A **programming assignment** (CW1)

Coursework!

- Q1 & SE1 are due at the **start of next class** (Fri at 9:00)
- CW1 is due on **Thurs at 19:00** <-- The day before!!!
- Total of **20 points**
  - 5 for Q1 and 5 for SE1
  - 10 for CW1

Please Don’t Stress!

- It's not a lot of work
  - So if you are learning Python you should have time!
- It's partly for diagnosis
- Partial work counts
- We're here to help!
- Don't leave before being sure you know what you're doing

Please Work Alone!

- It's important to figure out **what you can do**
- These are a **small number** of points and CW1 is fairly simple
- We're here to help!
- Use the Blackboard forum!
    - We will monitor
    - Don't share code there
    - You can share "high level tips"