COMP61511 (Fall 2017)

Software Engineering Concepts *In Practice*

Week 1

Bijan Parsia & Christos Kotselidis

<**bijan.parsia**, **christos.kotselidis**@manchester.ac.uk>
(bug reports welcome!)

Topic Overview

- Software engineering is the discipline that attempts to
  - produce successful **software systems**
  - by means of successful **software development projects**
- We look at different topics
  - The nature and challenges 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** of that understanding
  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
  - You should aspire to read both books
    - Though you don't have to just in the 5 weeks!
  - There will be other readings

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**
  - There's a limited number of concurrent users.
  - There are about 10 physical copies
- Mitigations:
  - **This week** == photocopies for you.
  - You can **purchase** the books (esp eBooks eg Kindle)
    - **Code Complete** (£16.99); **Making Software** (£19.89)
  - **Safari Books Online** Subscription service
    - 10 day free trial
    - £26 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/pg**
- We use **Blackboard** for
  - Coursework
  - Online forum
    - Use this!
  - 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 (among other things!)
    - Great, helpful people
- **Mitigating circumstances**
- Help available from **SSO** & **Counselling service**

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

A Note About Assistance

- **Early intervention** is more effective
  - If you are having **challenges** of any sort
    - the sooner they are know **by us**
    - the more likely we can find a good resolution
- This is very true for **mitigating circumstances**
  - If something is interfering, document it!
  - Fill out the form **when** the interference is happening
  - There is a "too late" here!

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

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

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
    - directly and indirectly
  - 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. *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**.

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

— **NASA System Engineering Handbook**

(Complex) Systems

System **results** are **emergent** and (metaphorically) **nonlinear**

1. The **problem** being tackled is
    - complex,
    - **amorphous**, or
    - evolving
2. Generally there is considerable **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:

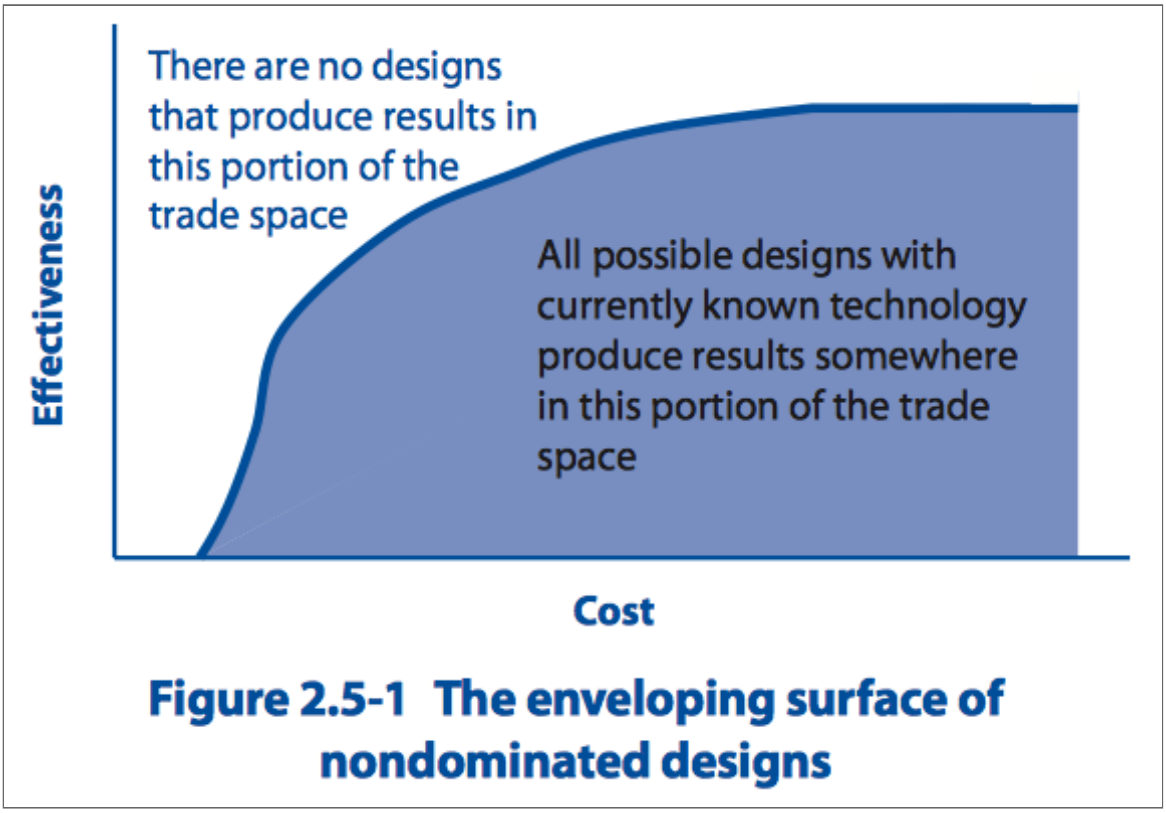


Figure 2.5-1 The enveloping surface of nondominated designs

One View Of The Design Space

- It's not so **simple**:

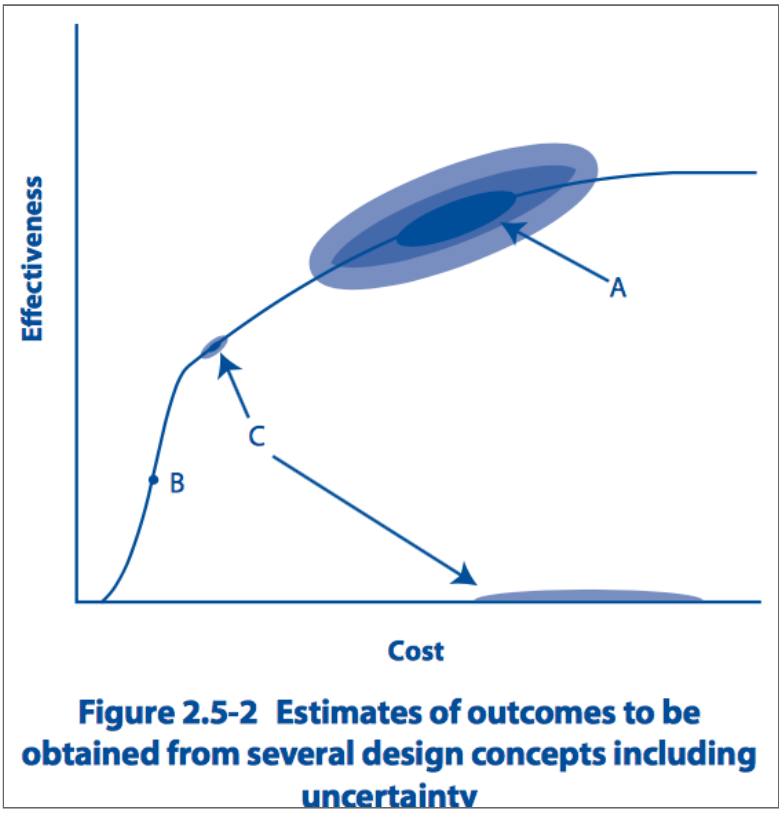


Figure 2.5-2 Estimates of outcomes to be obtained from several design concepts including uncertainty

Complex Vs. Complicated

Tendentious but common view: They are differnet

- Complexity is **intrinsic**
  - A **fundamental** feature of the problem (or solution)
  - Irreducible except by sacrifice
  - A complex solution may be as **simple as possible**
- Complication is **extrinsic**
  - An **accidental** feature of our approach to the problem
  - Lossless reduction of complications possible
  - A complicated system may be **over**complicated
    - Some complications might be acceptible
    - Getting as simple possible might not be best

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.*

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 avionics system in the F-22 Raptor, the current U.S. Air Force frontline jet fighter, consists of about **1.7 million lines** of software code. 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****," says Manfred Broy, a professor of informatics at Technical University, Munich, and a leading expert on software in cars. 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.*

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?)

Why Software Creep (Robots!)

A suggestion why software **takes over**!

> *This **linear growth of mechanics**, coupled with the **exponential growth of processing power** [Moore's law], has led to the inevitable—the ability to overcome poor mechanical performance with complex but cheap (!) processing. Instead of spending significantly more money for better control, you **invest in a better processor and more complex processing** [i.e., software].*
> *James Hendler, **Robots for the Rest of Us***

Less Complex Systems?

- A(n abstract) manufacturing challenge:
  - Producing a **standardized part** with
    - 1mm **tolerances**
    - **defect rate** of 1 per 10,000 items
- **Baking** a cake
  - Baking hundreds of cakes?
- **Painting** a house exterior
- **Mailing** a letter
  - Contrast with **delivering** a letter

Less complex doesn't mean **easy**

Non Complex Software Systems?

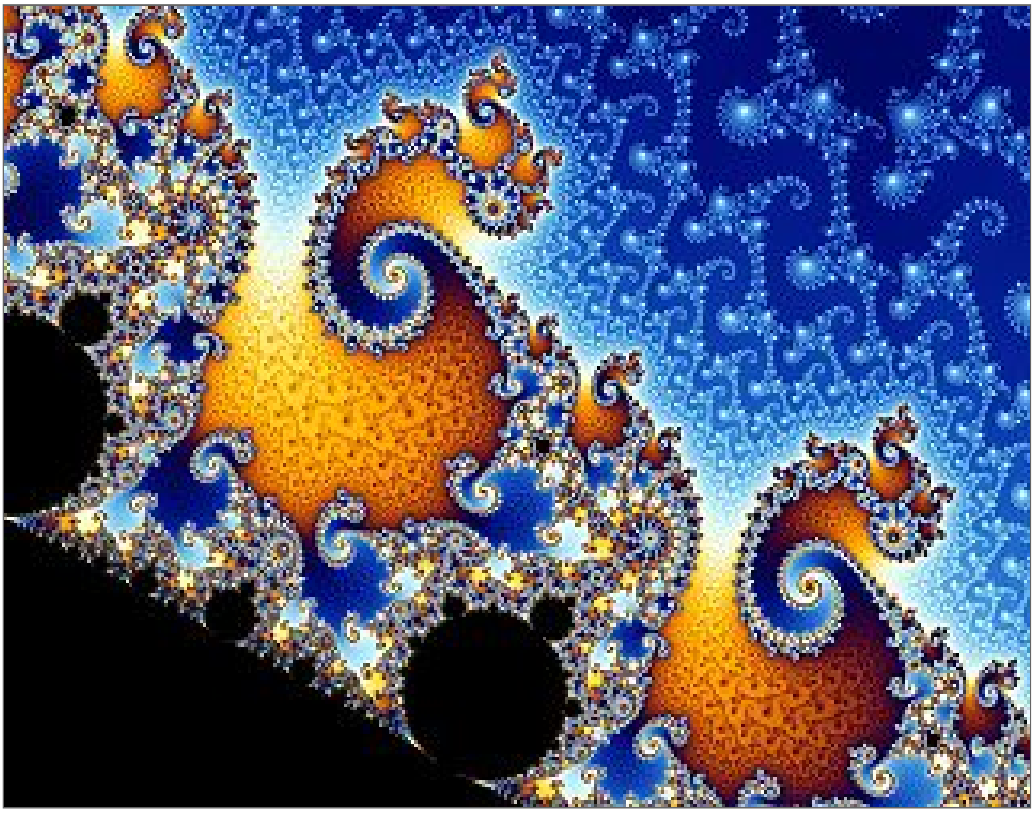
> *Averaging problem: Write a program that will read in integers and output their average. Stop reading when the value 99999 is input.*
> — ***Soloway***

- Is a program that solves the "rainfall problem"
  1. a complex system?
  2. part of a complex system?
  3. complicated?
  4. 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**
  - 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 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"

wat

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***

Wat Is Everywhere!

Keep an eye out for the Wats!

Product Qualities

Qualities (Or "Properties")

- Software has a variety of **characteristics** or aspects
  - Size, implementation language, license…
  - User base, user satisfaction, market share…
  - Crashingness, bugginess, performance, functions…
  - Usability, prettiness, slickness…
- 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

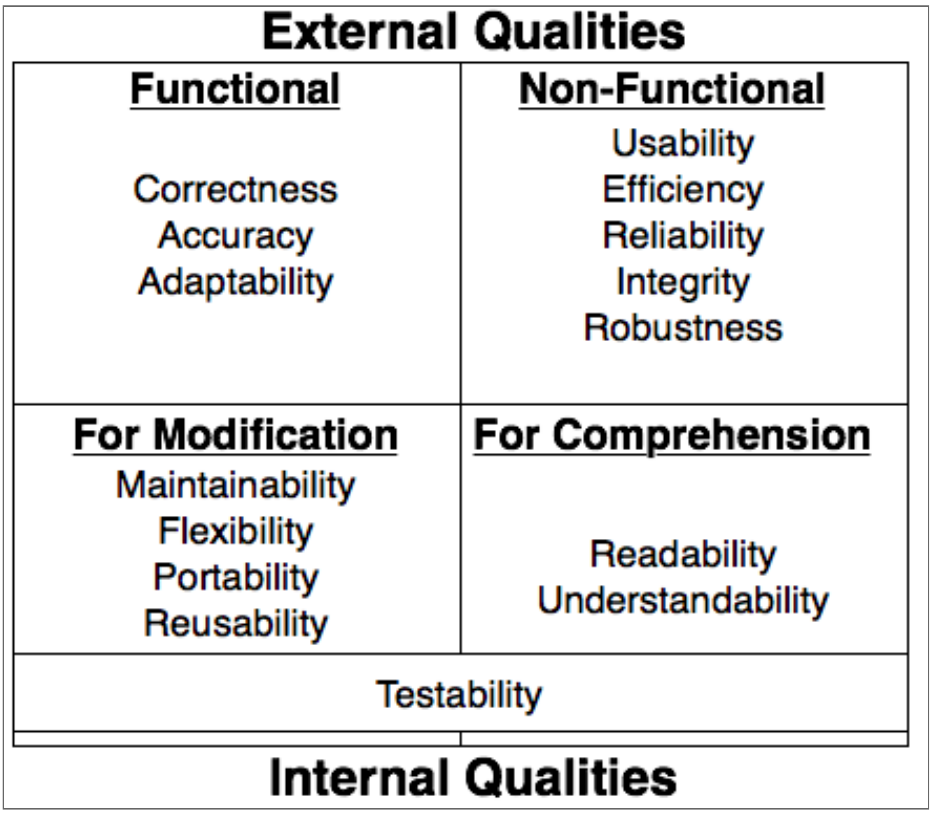
- While success is **determined** by qualities
  - the determination isn't **straightforward**
  - the determination isn't **strict**
    - for example, **luck** plays a role!
  - it depends on how you **specify** the critical success factors

Business Level Success

- Consider a **business goal**:
  - To increase marketshare by 25% in 2017
- This is **quantifiable** and **measurable**
  - So, we know when we've **achieved this goal**
  - The goal might have been **undesirable**!
- How do we do that?
  - Change the **product**
  - Change the **marketing**
  - Change the **price**

Software Quality Landscape

20.1. Characteristics of Software Quality, Code Complete

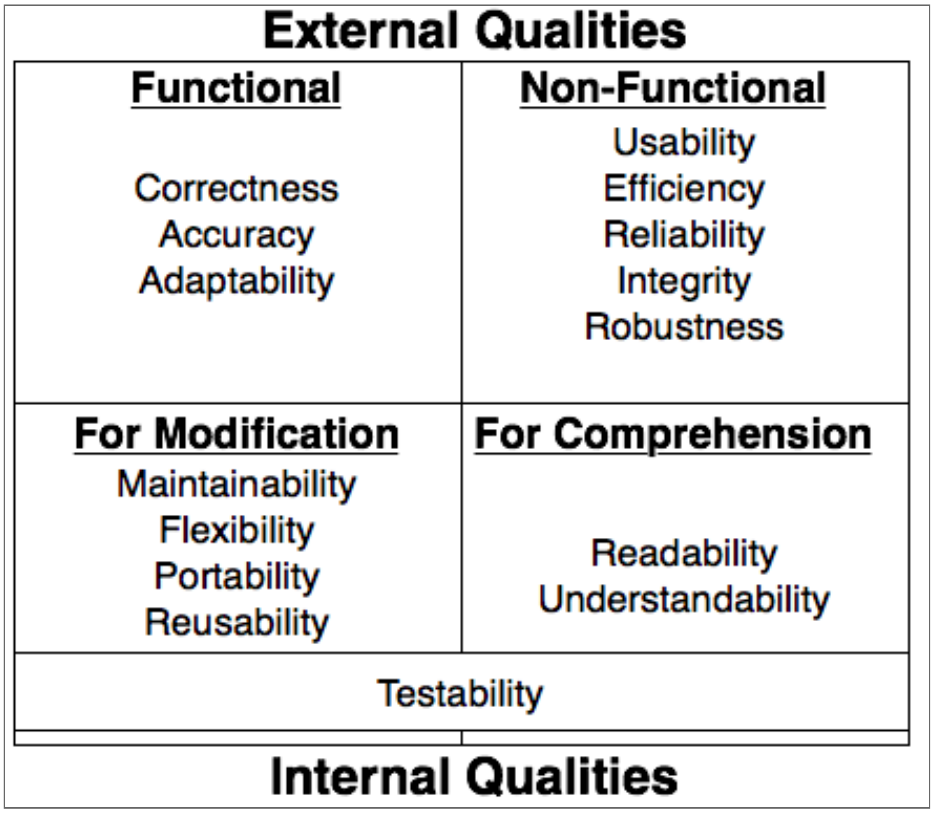


External Vs. Internal (Rough Thought)

- **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!

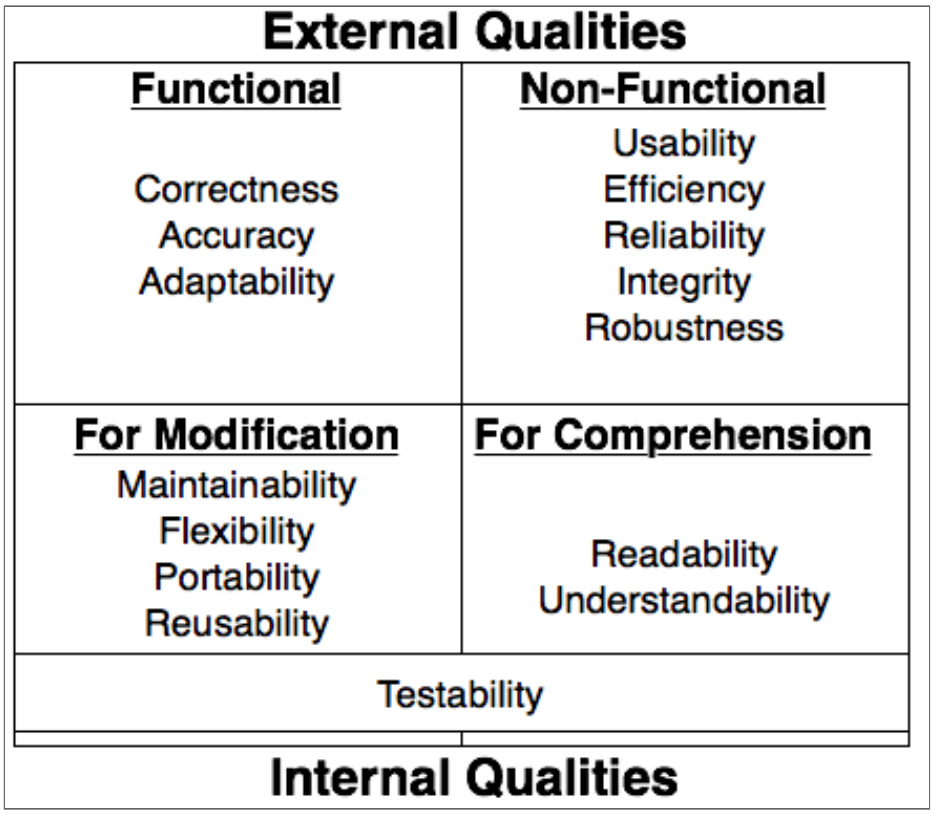
External Definition

- **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"?

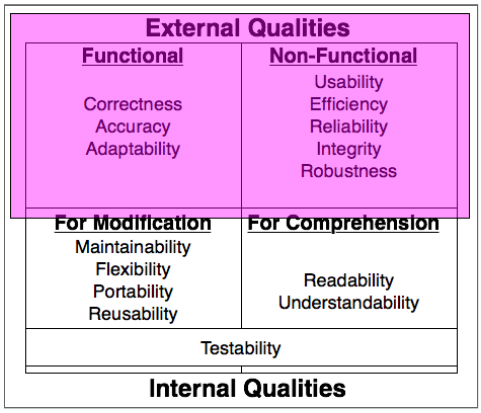


Internal Definition

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



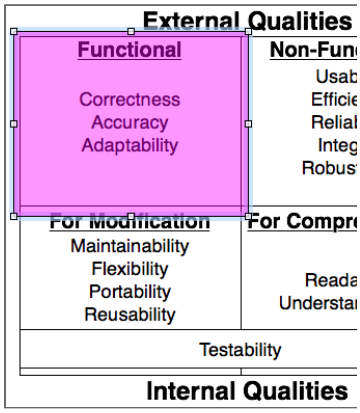
External: Functional Vs. Non-Functional



- 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

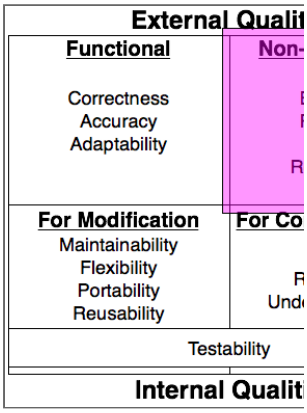
- **Correctness**
  - **Freedom from faults** in
    - spec,
    - design,
    - implementation
  - **Does the job**
  - Fulfills all the **use cases** or **user stories**



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)!

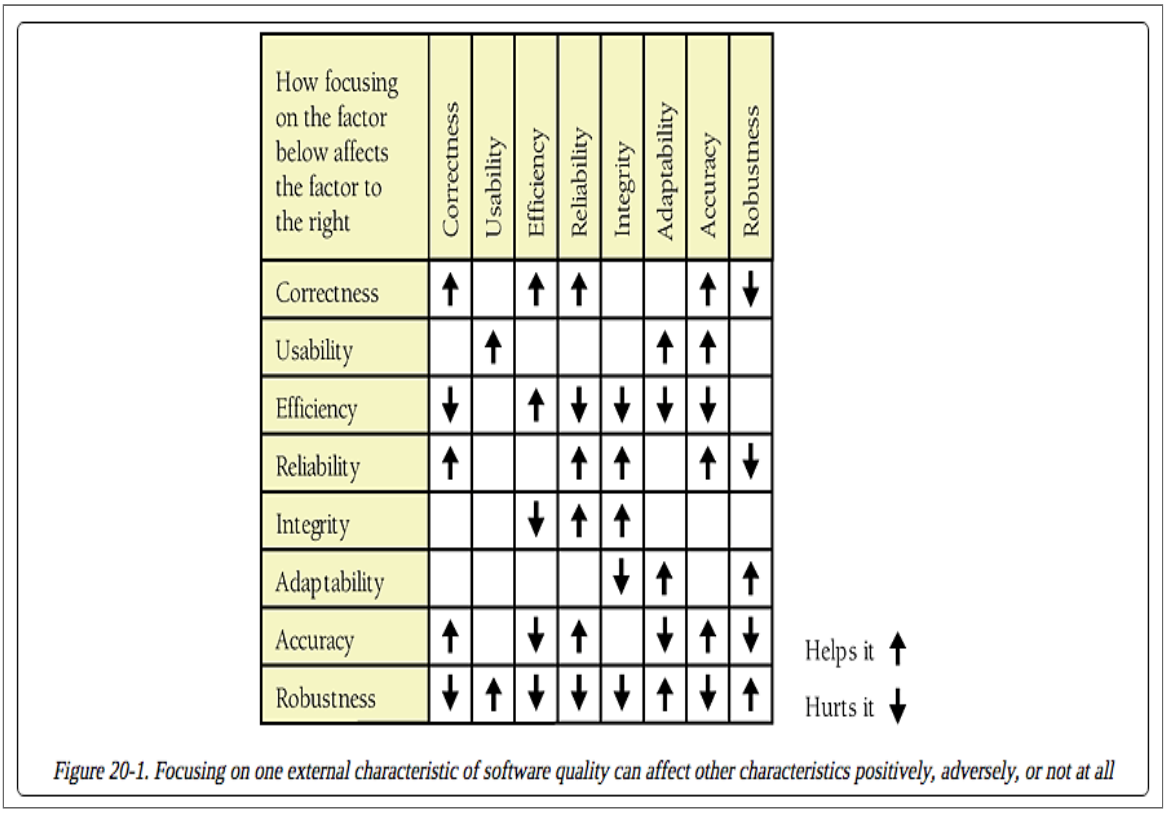
We're going to focus on **efficiency** this week!

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**
- Practically speaking
  - Low testability blocks **knowing** qualities
  - Test-based evidence is essential

Quality Interactions: External

20.1, Code Complete



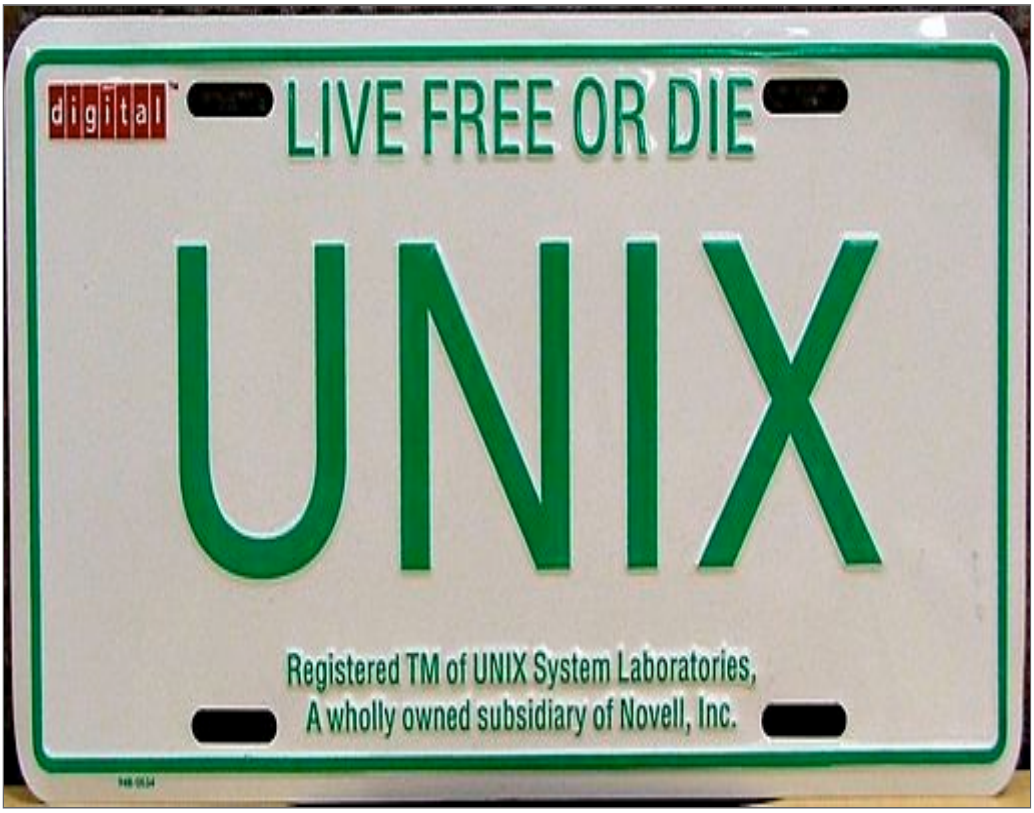
How focusing on the factor below affects the factor to the right	Correctness	Usability	Efficiency	Reliability	Integrity	Adaptability	Accuracy	Robustness
Correctness	↑		↑	↑			↑	↓
Usability		↑				↑	↑	
Efficiency	↓		↑	↓	↓	↓	↓	
Reliability	↑			↑	↑		↑	↓
Integrity			↓	↑	↑			
Adaptability					↓	↑		↑
Accuracy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

The Unix Philosophy



Design Philosophies

- **Design** is the establishment of a **plan** or **set of constraints** on the construction of a system (or object)
  - Think blueprint for a house
  - Designs can occur at different levels of **detail** or **abstraction**
- A **Design Philosophy** is a set of a constraints **on your designs**
  - A "design of designs"
  - Typically the **most abstract** design
  - Typically independent of any particular **domain problem**

Unix

- Unix is an Operating System, set of tools, a UI/UX, and a **philosophy**
  - Developed at AT&T in the 1970s
    - By Ken Thompson, Dennis Ritchie, and a cast of many
  - Multitasking and multiuser
  - It has many descendents, variants, clones, and related systems
  - Closely related to the C programming language
    - Historically; they developed together
- Linux, MacOS, and BSD are popular modern versions

The Unix Philosophy

One formulation by Peter H. Salus:

> *This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface.*

- Simple! (to say; tricky to master)
- Mostly for **programmers** and **power users**
- It got **complicated**
  - See the 17 rules!

Small, Sharp Tools

> *Write programs that do one thing and do it well.*

- Modularity is a key principle
  - Programs (should) have a **single job** or **responsibility**
  - They should **focus** on that
  - When you work on (or us) that tool you don't have to think about other things
    - Much!
- Sharp
  - Focused, pointy...and you can **cut yourself**

Working Together

> *Write programs to work together. Write programs to handle text streams, because that is a universal interface.*

- The **principle**: Programs should "work together"
- The **mechanism**: Consume and produce **text**
  - "a universal interface"
- (We need more than this!)

Standard Streams

- Unix programs (typically) have **three streams** by default:
  - `stdin`: **ST**andar**D IN**put
    - This can be **interactive**!
  - `stdout`: **ST**andar**D Out**put
    - Printed to the shell by default
    - Captures the **expected** output of the program
  - `stderr`: **ST**andar**D ERR**or
    - Often printed to the shell, but sometimes hidden
    - Captures **error messages**
- Many programs require more streams than this!

Let's Look At Some Tools

v · t · e	Unix command-line interface programs and shell builtins	[hide]
File system	cat · chmod · chown · chgrp · cksum · cmp · cp · dd · du · df · file · fsck · fuser · ln · ls · mkdir · mount · mv · pax · pwd · rm · rmdir · split · tee · touch · type · umask	
Processes	at · bg · chroot · cron · fg · kill · killall · nice · pgrep · pkill · ps · pstree · time · top	
User environment	clear · env · exit · finger · history · logname · mesg · passwd · su · sudo · uptime · talk · tput · uname · w · wall · who · whoami · write	
Text processing	awk · banner · basename · comm · csplit · cut · diff · dirname · ed · ex · fmt · fold · head · iconv · join · less · more · nl · paste · printf · sed · sort · spell · strings · tail · tr · uniq · vi · **wc** · xargs · yes	
Shell builtins	alias · cd · echo · test · unset · wait	
Networking	dig · host · ifconfig · inetd · netcat · netstat · nslookup · ping · rdate · rlogin · route · ssh · traceroute	
Searching	find · grep · locate · whatis · whereis	
Documentation	apropos · help · man	
Miscellaneous	bc · dc · cal · expr · lp · od · sleep · true and false	
	List of Unix commands	

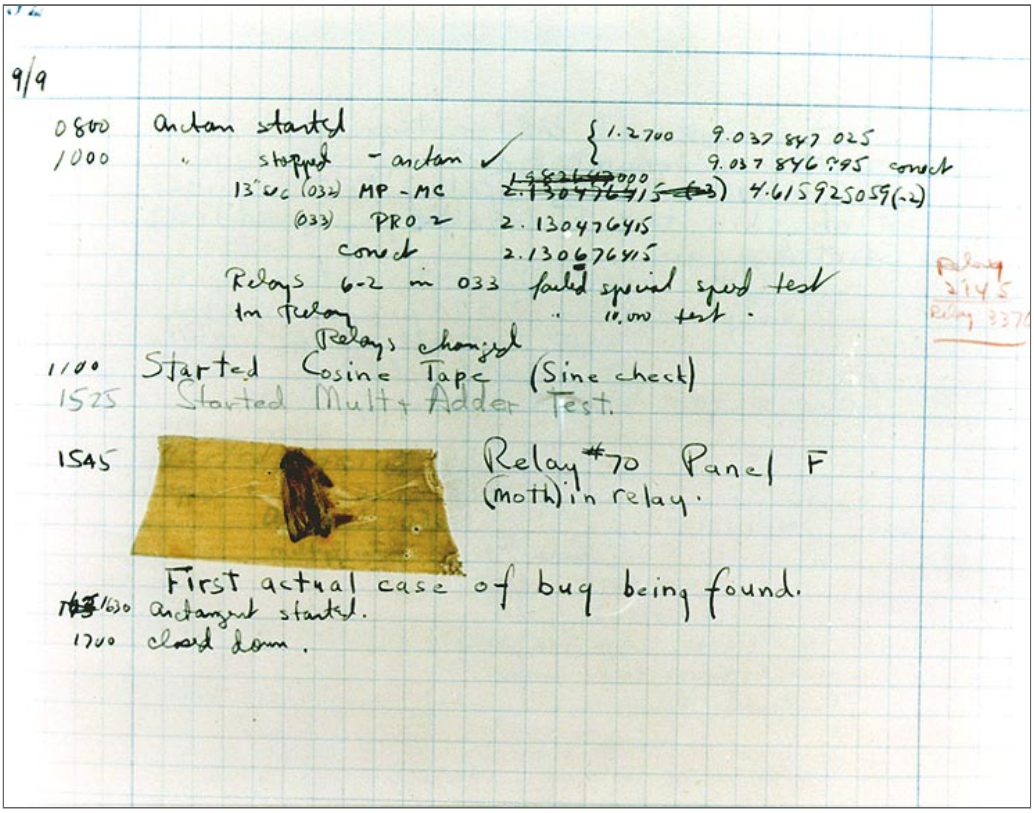
Word Count (`wc`)

- `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!

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
  - you have tested it for syntactic correctness.
  - you have tested it for semantic correctness.
  - you have tested it for both.
  - 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*

- The outcomes may be **expected** (i.e., specified in advance)
  - 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? (2)

*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*

- 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)

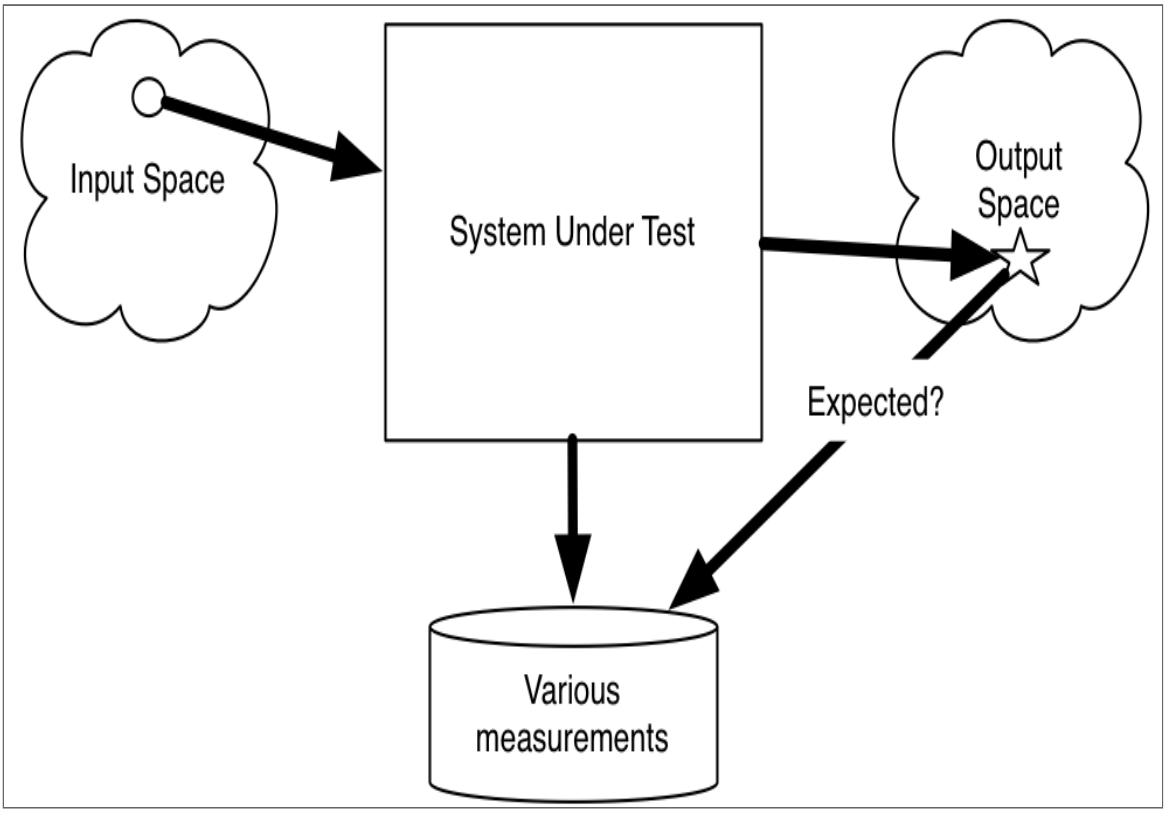
> *Testing shows the **presence**, not the **absence** of bugs.*
> *—Edsger W. Dijkstra, **Nato Software Engineering Conference, pg 16 1969***

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?

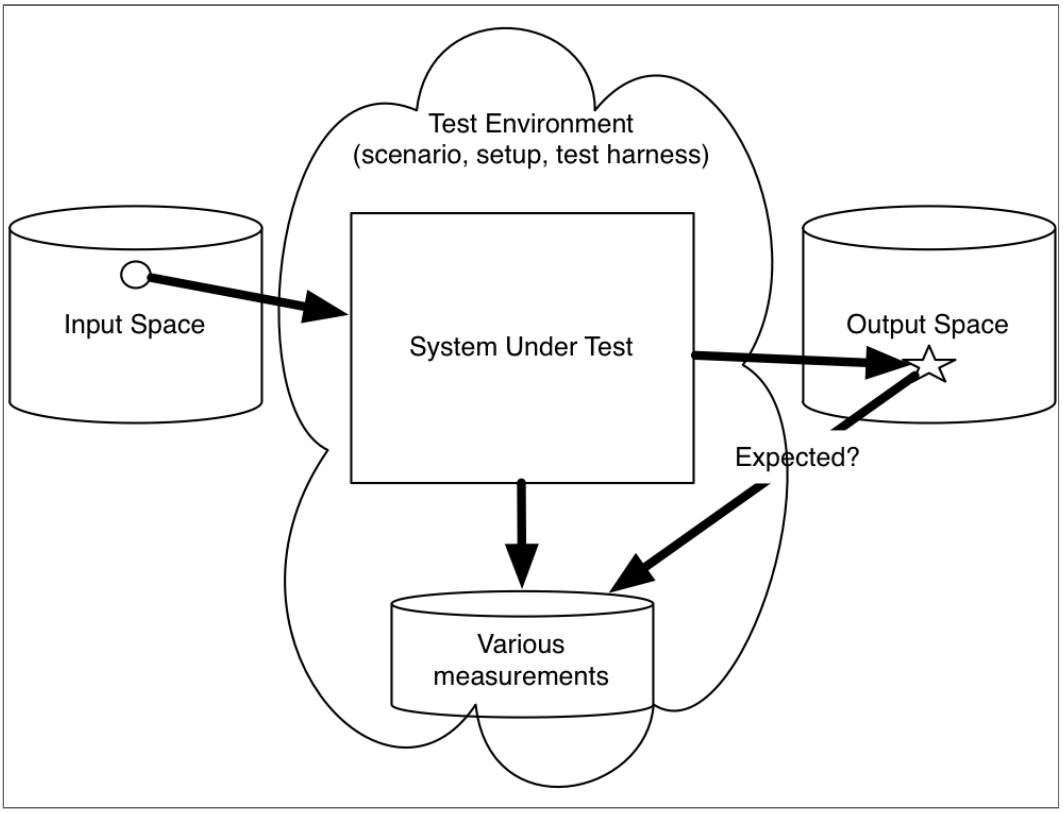
Question Time!

- 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%		

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)
- Q1 & SE1 are due at the **start of next class** (Thurs at 9:00)
- CW1 is due on **Wed at 19:00** <-- The day before!!!
- Total of **30 points**
  - 5 for Q1 and 4 for SE1
  - 20 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 are **aware** of differences in experience and skill
- We're here to help!
- We will adjust things if they seem **out of wack**
- Use the Blackboard forum!
  - We will monitor
  - Don't share code there
  - You can share "high level tips"