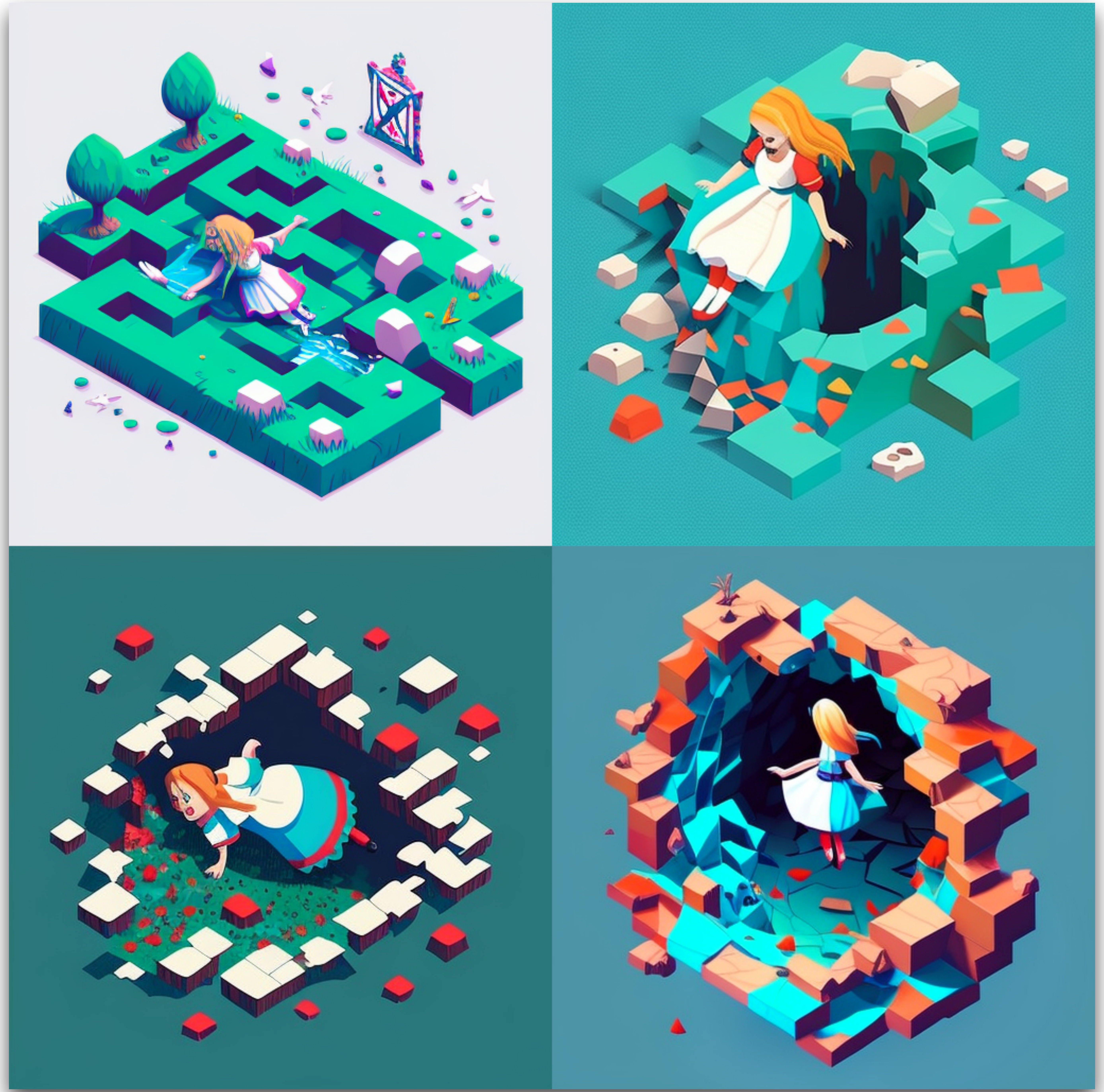
Common Concerns and Principles in SE

CS350 Introduction to Software Engineering

Shin Yoo

Concerns? Principles?

- Instead of talking about concrete techniques or definitions, I would like to enumerate recurring ideas that apply to all SW projects - or even all engineering projects, to some degree.
- These may sound too abstract and fluffy, but they are embedded in everything you will see in this course
  - They can help you understand a problem better
  - They can also help you design a solution better

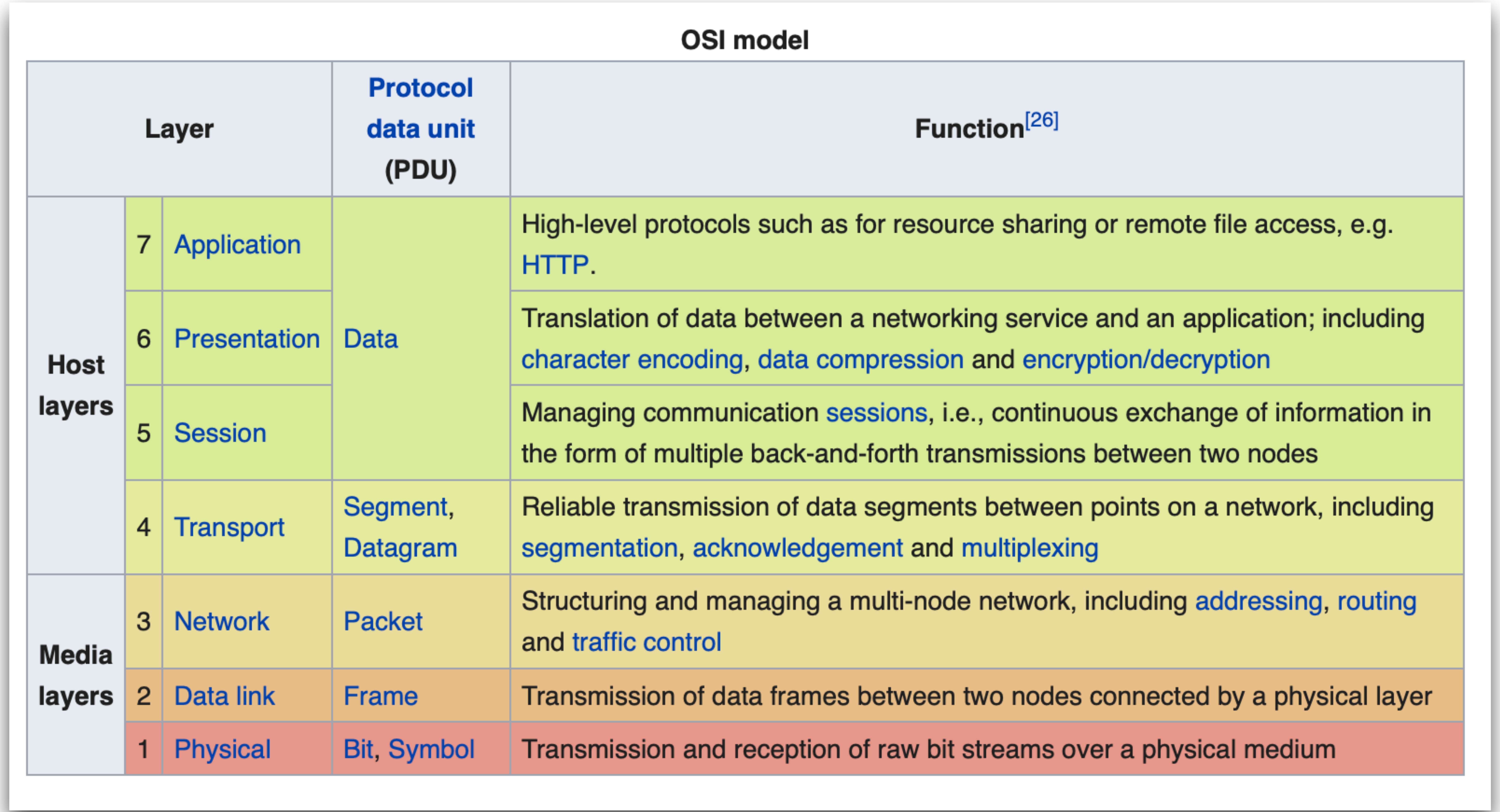


Abstraction

Computer science is essentially the art of abstraction.

- Apparently Donald Knuth said “layers of abstraction” when asked what is the single most important thing that connects everything in computer science … (see https://www.youtube.com/watch?v=bmSAYlu0NcY which is in itself a good lecture, we will come back to this later)
- What is abstraction?
  - The process of removing or hiding various lower level details so that we can generalize and focus on properties of greater importance
- Why is it important?
  - If you reveal the entire internal workings of a system to the outside, the outside world will connect to/use/exploit your internals. This introduces problems when you later want to modify your system.

What are the examples of abstraction in SW?



OSI model

Layer			Protocol data unit (PDU)	Function[26]
Host layers	7	Application	Data	High-level protocols such as for resource sharing or remote file access, e.g. HTTP.
	6	Presentation		Translation of data between a networking service and an application; including character encoding, data compression and encryption/decryption
	5	Session		Managing communication sessions, i.e., continuous exchange of information in the form of multiple back-and-forth transmissions between two nodes
	4	Transport	Segment, Datagram	Reliable transmission of data segments between points on a network, including segmentation, acknowledgement and multiplexing
Media layers	3	Network	Packet	Structuring and managing a multi-node network, including addressing, routing and traffic control
	2	Data link	Frame	Transmission of data frames between two nodes connected by a physical layer
	1	Physical	Bit, Symbol	Transmission and reception of raw bit streams over a physical medium

Separation of Concerns

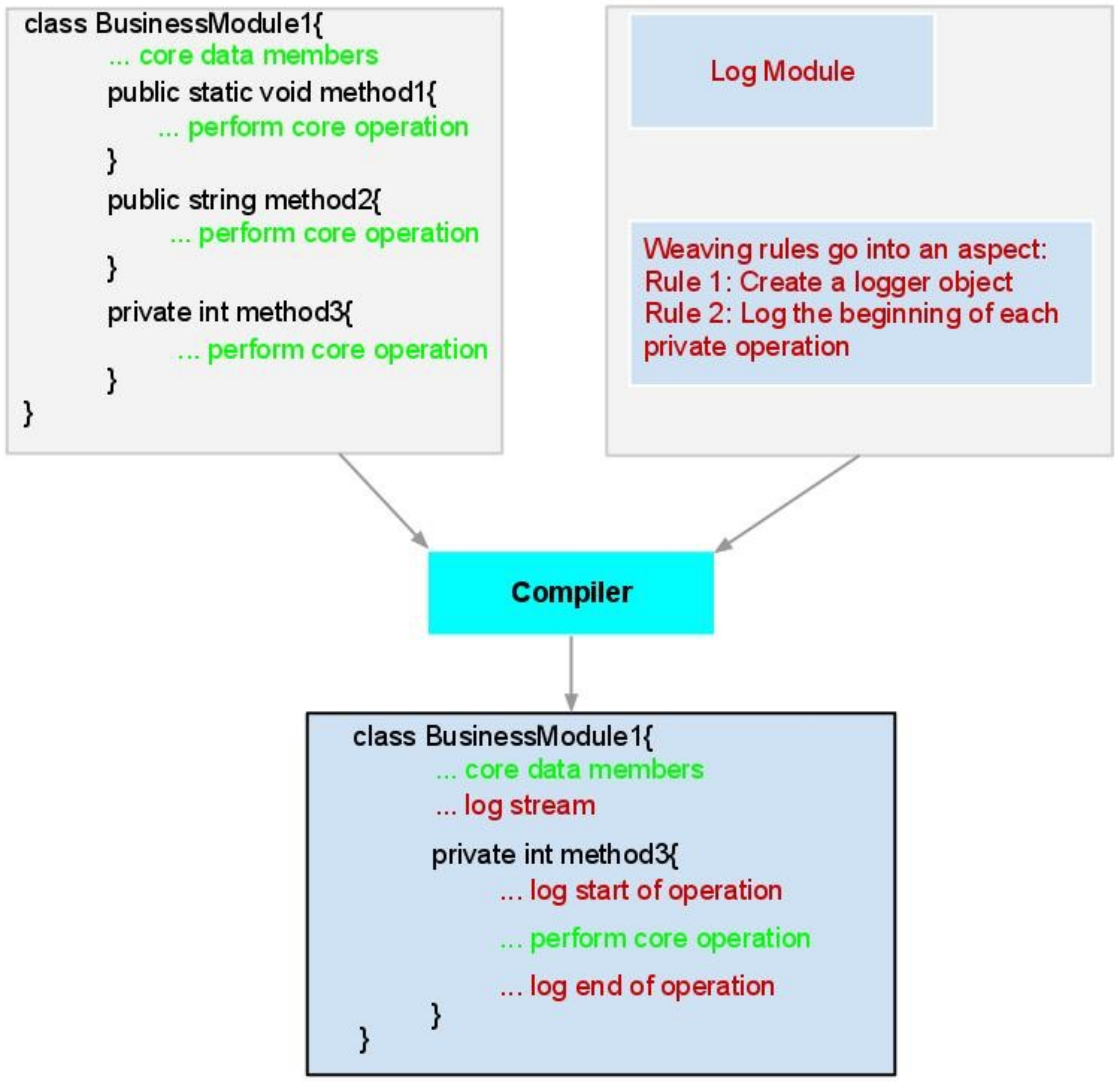
One of the most important design principle.

- A system/program should be partitioned into different sections; each section should address a separate concern.
- Why is this important?
  - Easier to understand, reuse, change… (this is a recurring theme, isn’t it?)
- Cross-cutting concerns
  - Concerns that affect the whole system (logging, security check, etc)
  - Harder to centralize

AOP

Aspect Oriented Programming

- A new programming paradigm where a single concern is expressed as an “aspect” of the program.
- Aspects can be “weaved into” the remaining system at “join-points”, i.e., the points of cross-cutting concerns.



```
class BusinessModule1{
        ... core data members
        public static void method1{
            ... perform core operation
        }
        public string method2{
            ... perform core operation
        }
        private int method3{
            ... perform core operation
        }
}
```

Weaving rules go into an aspect:
Rule 1: Create a logger object
Rule 2: Log the beginning of each private operation

```
class BusinessModule1{
        ... core data members
        ... log stream
        private int method3{
            ... log start of operation
            ... perform core operation
            ... log end of operation
        }
}
```

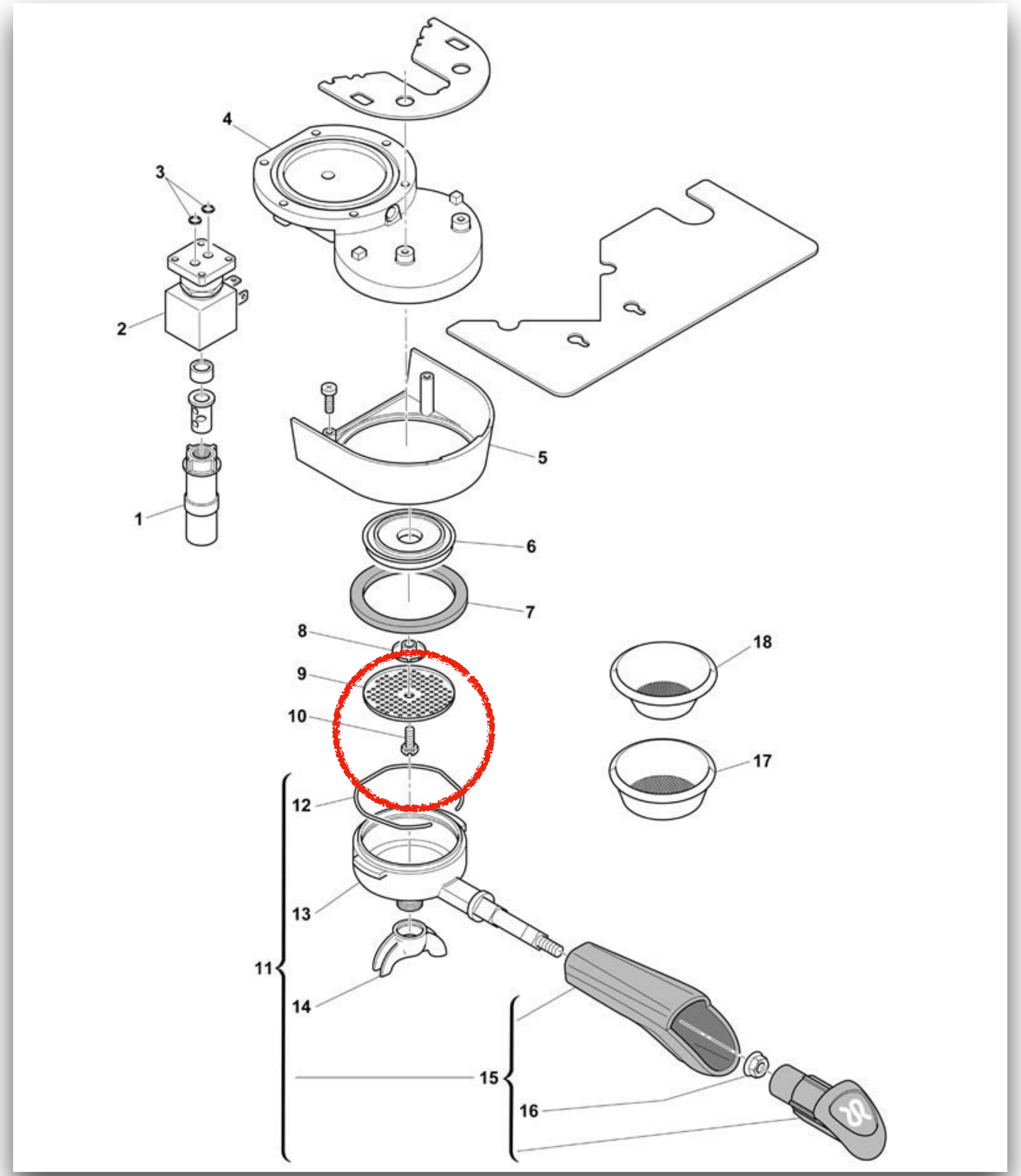
https://www.programcreek.com/2011/08/what-is-aspect-oriented-programming/

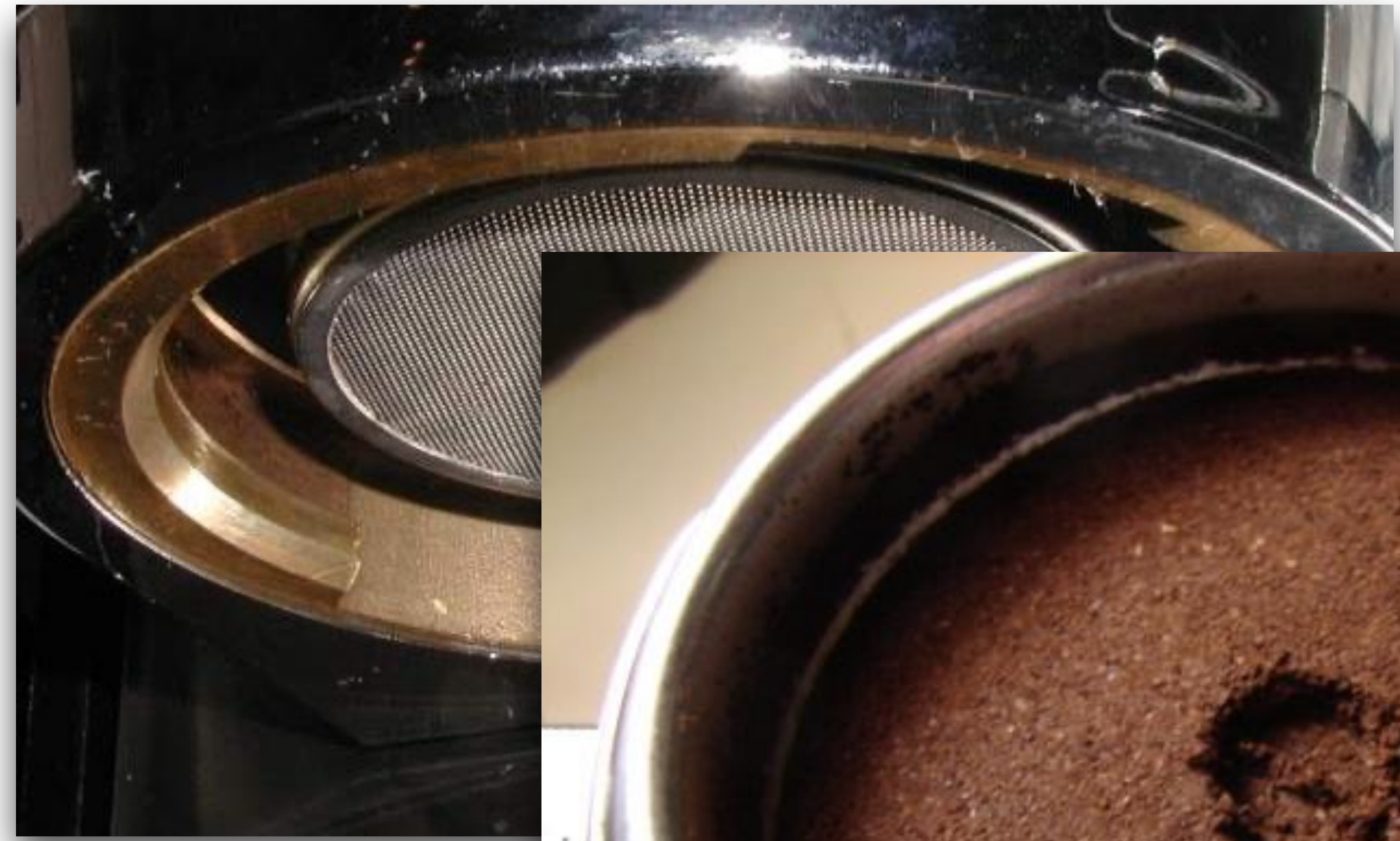
Modularity

- A system is highly modular if its components can be separated from each other to be re-combined in a different way.
- If a system adopts good abstraction (i.e., it is connected with good interfaces) and separation of concern is well implemented, then we call the system “modular”
- Systems should be built as loosely coupled modules (i.e., can be separated easily), each of which are highly cohesive (i.e., good separation of concerns)

An anecdote about modularity

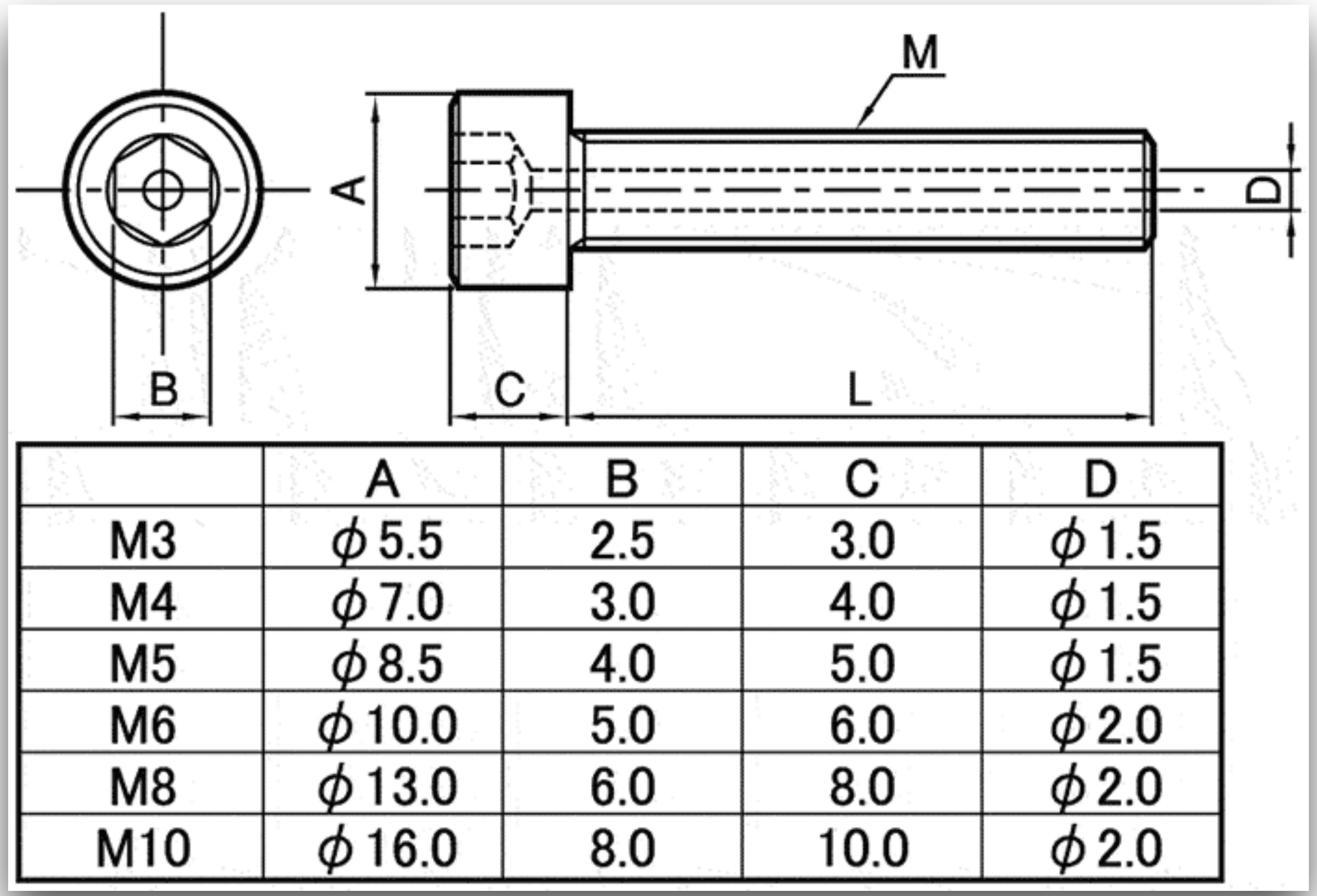
(Confessions of an espresso nerd)





An anecdote about modularity

(Confessions of an espresso nerd)



	A	B	C	D
M3	ϕ5.5	2.5	3.0	ϕ1.5
M4	ϕ7.0	3.0	4.0	ϕ1.5
M5	ϕ8.5	4.0	5.0	ϕ1.5
M6	ϕ10.0	5.0	6.0	ϕ2.0
M8	ϕ13.0	6.0	8.0	ϕ2.0
M10	ϕ16.0	8.0	10.0	ϕ2.0

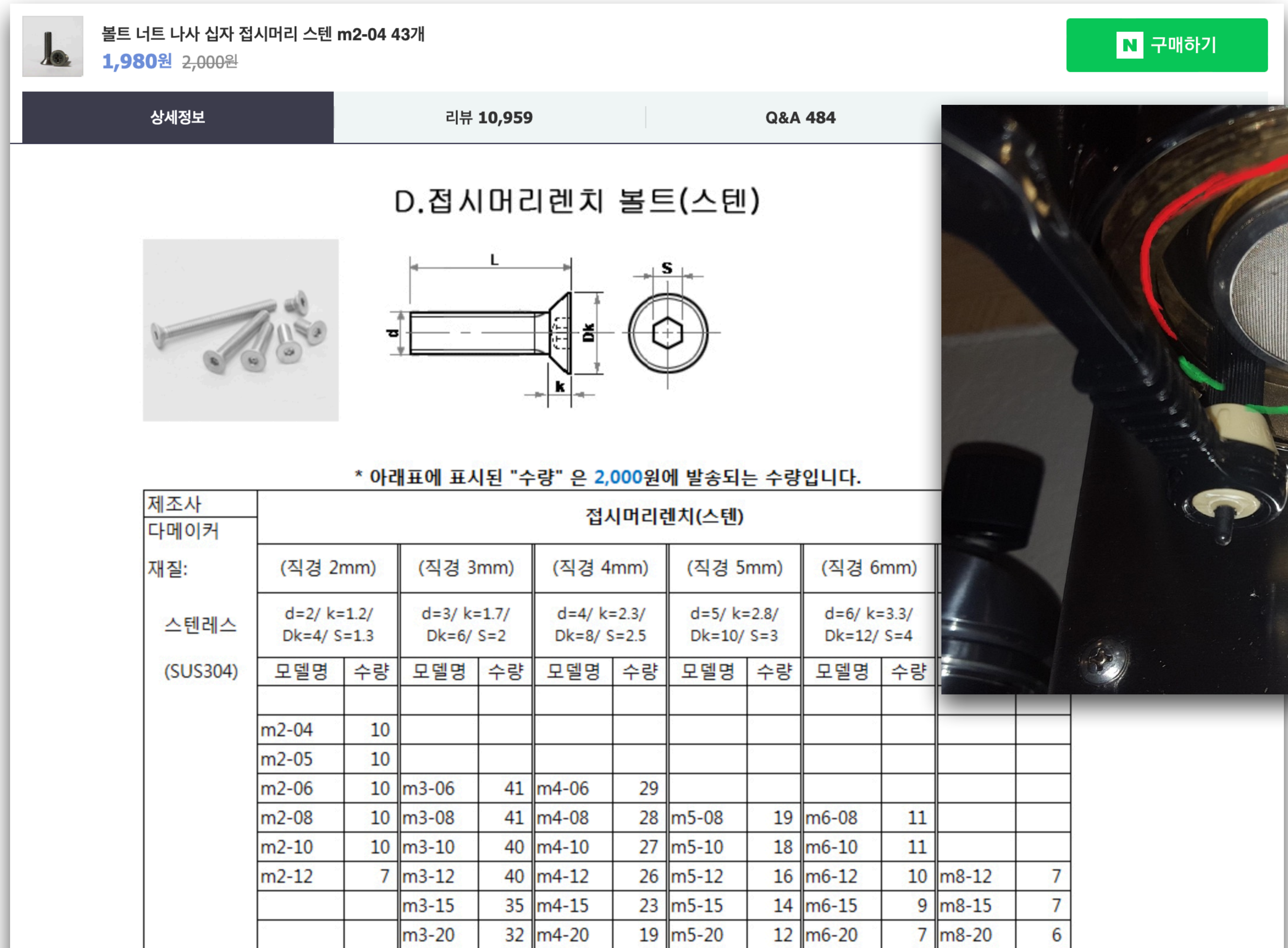
+

+

An anecdote about modularity

(Confessions of an espresso nerd)

=



볼트 너트 나사 십자 접시머리 스텐 m2-04 43개

1,980원 ~~2,000원~~

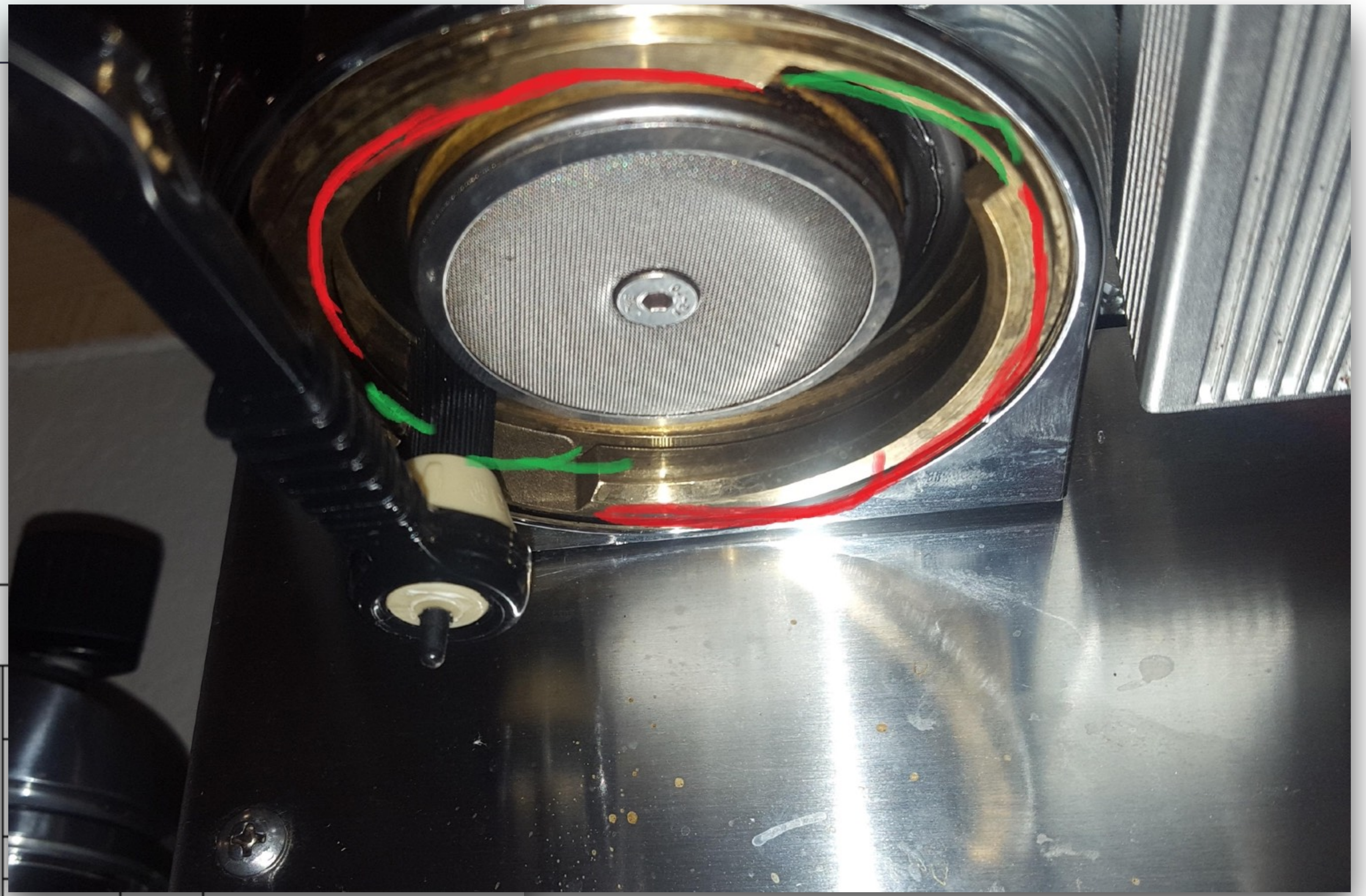
N 구매하기

상세정보 | 리뷰 10,959 | Q&A 484

D.접시머리렌치 볼트(스텐)

* 아래표에 표시된 "수량" 은 2,000원에 발송되는 수량입니다.

제조사	접시머리렌치(스텐)											
다메이커												
재질:	(직경 2mm)		(직경 3mm)		(직경 4mm)		(직경 5mm)		(직경 6mm)			
스텐레스	d=2/ k=1.2/ Dk=4/ S=1.3		d=3/ k=1.7/ Dk=6/ S=2		d=4/ k=2.3/ Dk=8/ S=2.5		d=5/ k=2.8/ Dk=10/ S=3		d=6/ k=3.3/ Dk=12/ S=4			
(SUS304)	모델명	수량	모델명	수량	모델명	수량	모델명	수량	모델명	수량		
	m2-04	10										
	m2-05	10										
	m2-06	10	m3-06	41	m4-06	29						
	m2-08	10	m3-08	41	m4-08	28	m5-08	19	m6-08	11		
	m2-10	10	m3-10	40	m4-10	27	m5-10	18	m6-10	11		
	m2-12	7	m3-12	40	m4-12	26	m5-12	16	m6-12	10	m8-12	7
			m3-15	35	m4-15	23	m5-15	14	m6-15	9	m8-15	7
			m3-20	32	m4-20	19	m5-20	12	m6-20	7	m8-20	6

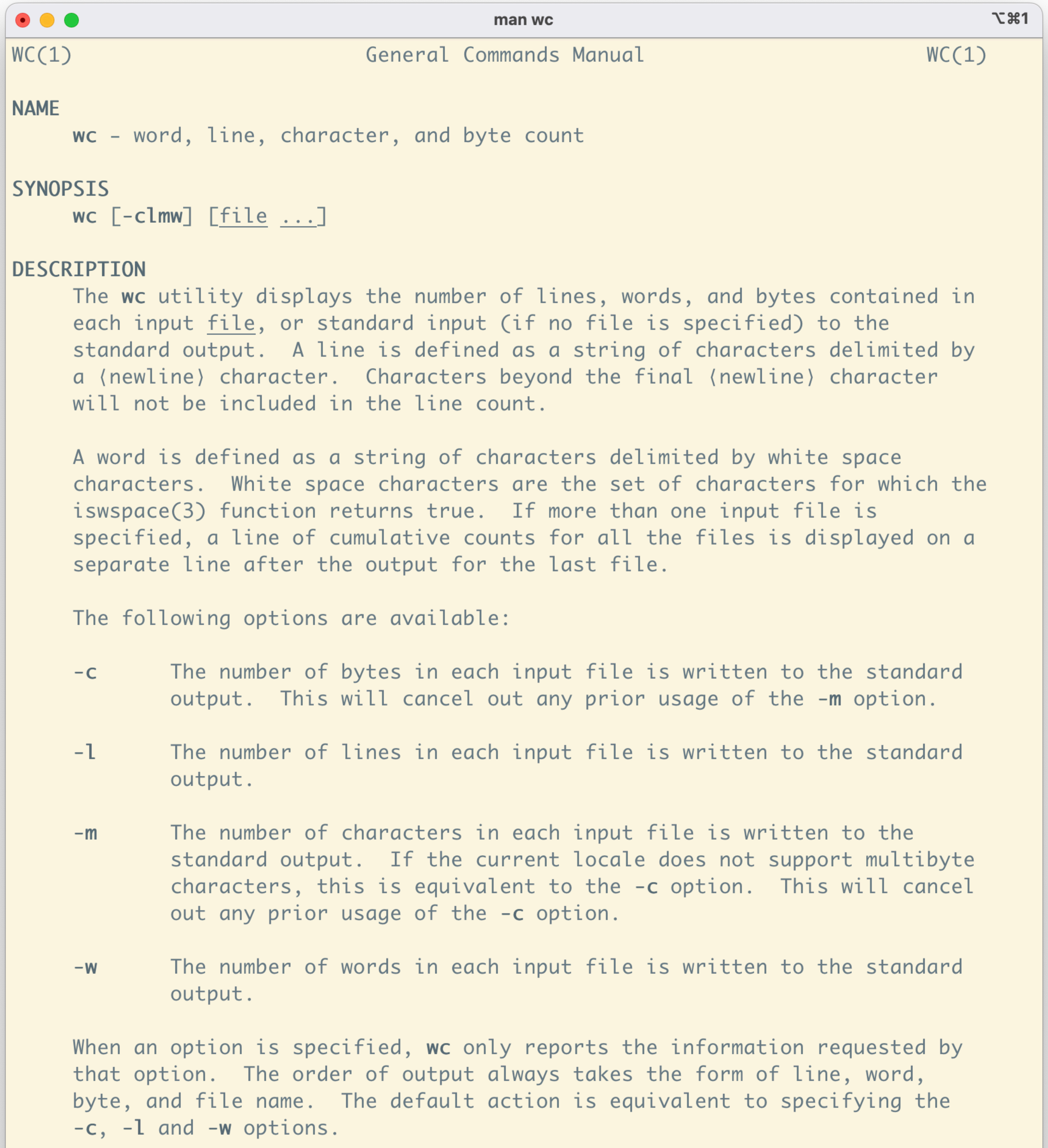


Can you think of a similar degree of modularity in SW?

- Hardware, maybe yes: we have reasonably successful interfaces (USB, Ethernet…), many electronic components are standardized, etc etc
- Software?
  - Unix pipelines?
  - Maybe only at a very coarse granularity level: for example, in my workflow of preparing my lectures, I can use PowerPoint or Keynote…?
  - APIs…?
    - Replacing PyTorch with TensorFlow?

Example: Unix Pipeline - wc

- "I want to count lines in a file"
  - `cat my file.txt | wc -l`
- "I want to count files in the current directory"
  - `ls -1 | wc -l`



```
man wc                                                    ⌥⌘1

WC(1)                    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⟩ character
     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
             characters, 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.
```

Example: Unix Pipeline

- “Unix Time-Sharing System: Forewords” by McIlroy, Pinson, and Tague, The Bell Systems Technical Journal, 1978
- Unix philosophy is:
    - **<u>Make each program do one thing well</u>. To do a new job, build afresh rather than complicate old programs by adding new "features".**
    - **<u>Expect the output of every program to become the input to another, as yet unknown, program.</u> Don't clutter output with extraneous information. Avoid stringently columnar or binary input formats. Don't insist on interactive input.**
    - Design and build software, even operating systems, to be tried early, ideally within weeks. Don't hesitate to throw away the clumsy parts and rebuild them.
    - Use tools in preference to unskilled help to lighten a programming task, even if you have to detour to build the tools and expect to throw some of them out after you've finished using them.

Redundancy

i.e., having more than strictly necessary

- Is this good or bad? :)

Reusable Well-designed Modules = Components?

Component Based Design / Engineering

- This used to be a BIG thing: your program can be broken down to components, each of which does some specific thing well, and is reusable.
- Visual programming IDEs like Delphi captures this ideal; GUIs are inherently composable, and many GUI libraries (such as java Swing, or the default Android widgets) can be viewed as components.
  - Rapid Application Development (RAD) using Delphi
    - https://www.youtube.com/watch?v=m_3K_0vjUhk (Windows 3.11)
    - https://www.youtube.com/watch?v=aFaLm41CDI4 (now)

Reusable Well-designed Modules = Components?

Component Based Design / Engineering

- But it seems like we are never “done” with inventing the wheel yet another time… especially with the most “component-esque” part of our toolbox, GUIs!

React JS

Vue.js

SVELTE

ember

jQuery JS

ANGULARJS

BACKBONE.JS

Bad Redundancy

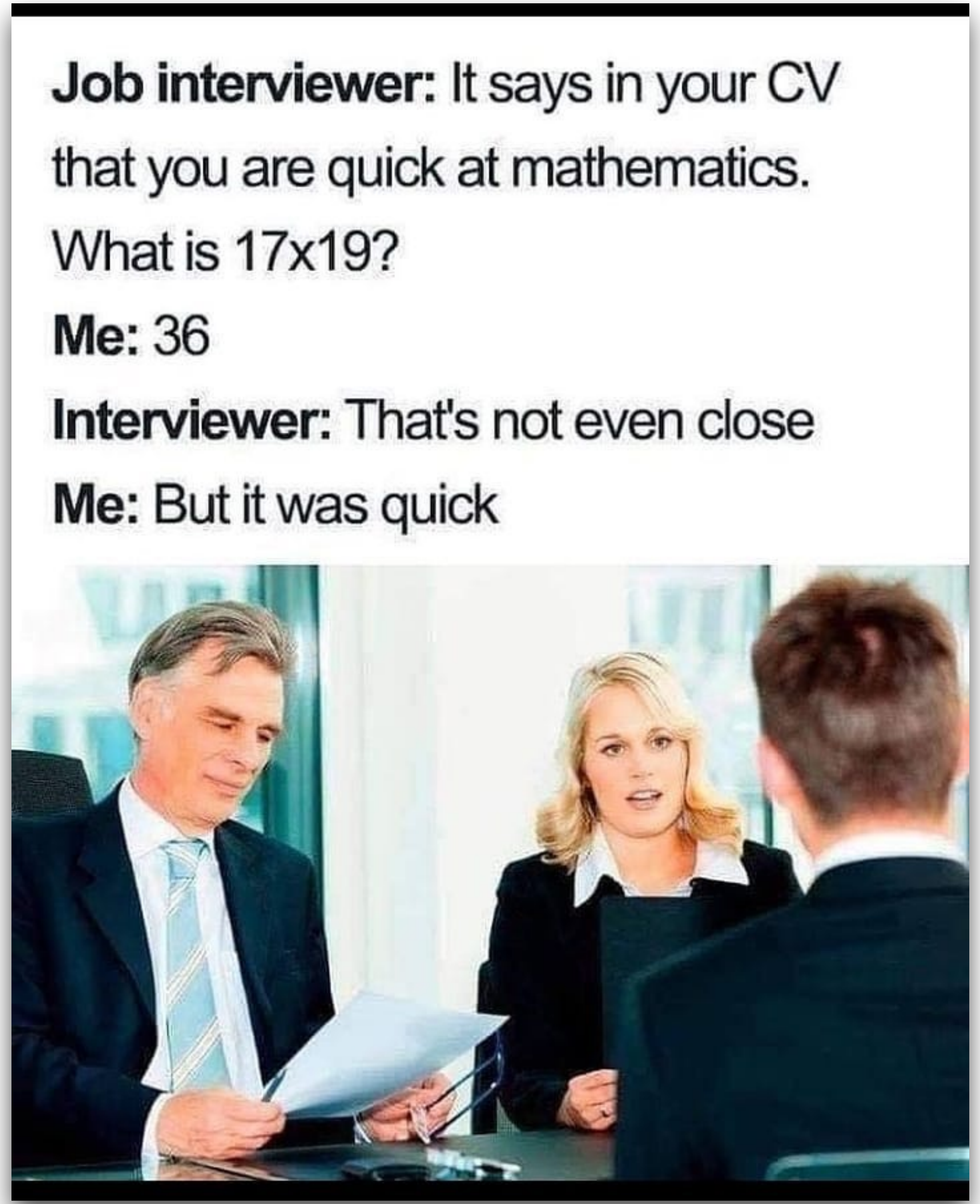
- Code Clone is essentially the same code snippet pasted across different locations (creating redundant code).
- Why is this considered a bad practice?
  - Plagiarism / Code Provenance (IP issues)
  - Bug fixing/feature addition becomes more difficult (what if you only change parts of it?)
- What should we do?
  - Refactor the code so that the functionality exists only once

Good Redundancy

- Duplicating servers for higher availability.
- Having multiple network service providers so that internet remains available.
- At the strategic level, even geographic redundancy may be needed (so that your data survives any natural disaster).
- N-version Programming
    - Develop *N* versions of critical system independently
    - Use majority voting to decide the final outcome

Correctness

- Your software needs to be correct in its functionality.
- How do we ensure this?
  - Experimental methods (i.e., testing), or analytic methods (i.e., formal verification)
  - Ensure correctness of the intermediate steps (e.g., static analysis)
  - Use proven libraries and components



Job interviewer: It says in your CV that you are quick at mathematics. What is 17x19?

Me: 36

Interviewer: That's not even close

Me: But it was quick

Reliability

- The probability of SW operating without any failure for specified duration of time in a specific environment.
- It is actually easier if we are retrospectively analyzing a failure; harder to evaluate reliability of a software system that has yet to fail.
    - $P = \frac{\text{\# of failing cases}}{\text{\# of total cases under consideration}}$
    - Can be hard to estimate, requires heavy probability theory
    - Note that the definition is parametric to time and environment

Reliability

A real world example from Gulf War, 1991

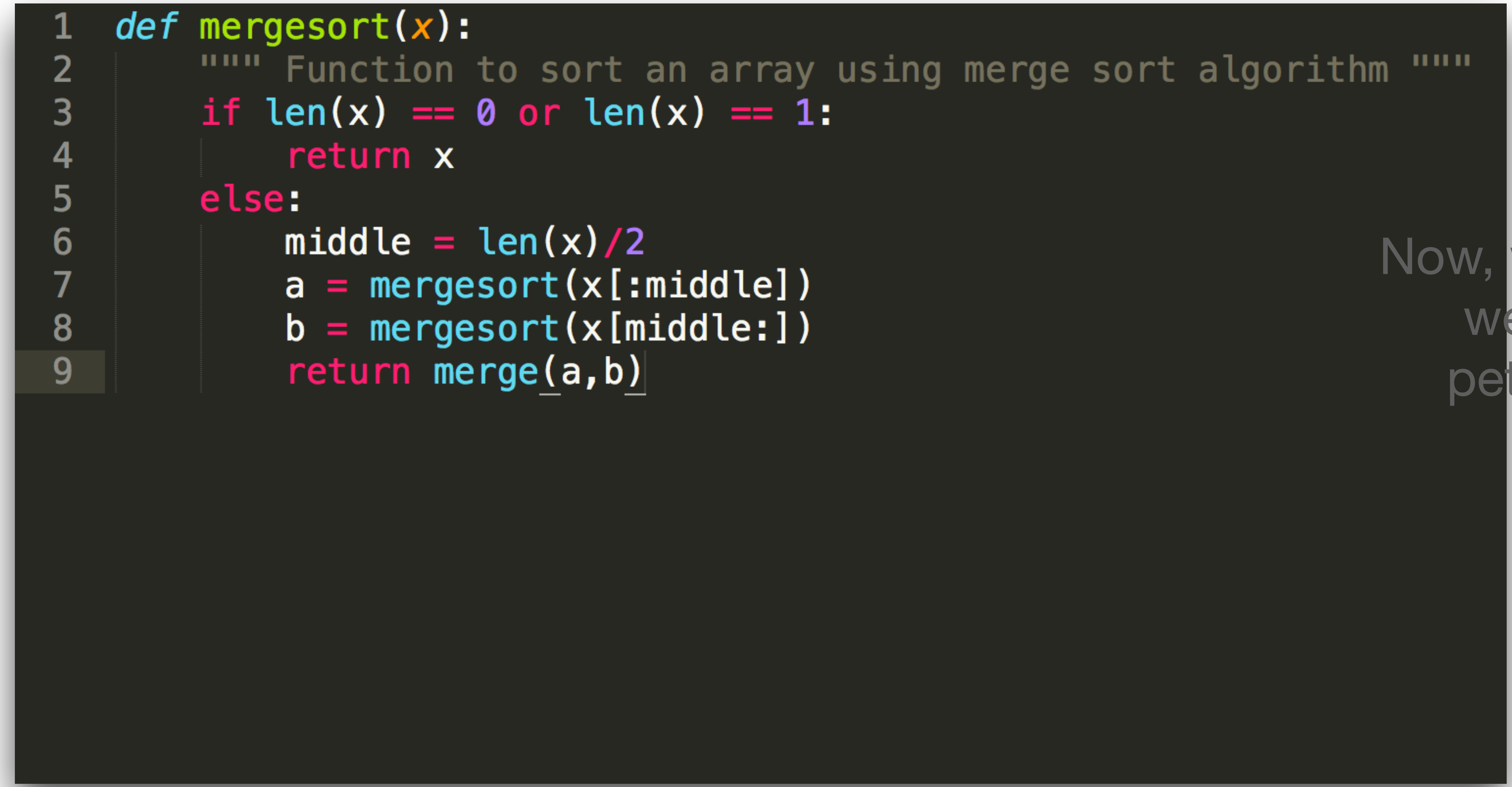
- The SW in Patriot missile defense system multiplied $\frac{1}{10}$ to system clock to get the “time from booting” in seconds (with two digits below 0) in 24bit floating point register.
- $\frac{1}{10}$ in 24bit floating point register can only be represented as non-terminating binary expansion cut at 24th place
  - $\frac{1}{10} = \frac{1}{2^4} + \frac{1}{2^5} + \frac{1}{2^8} + \frac{1}{2^9} + \frac{1}{2^{12}} + \frac{1}{2^{13}} + \dots$
- After 100 hours, the chopping error amounted to about 0.34 seconds: enough for an incoming Scud missile to travel more than 500 meters, resulting in 28 deaths.
- Why do you think this system was believed to be reliable? How could we have avoided this?

Scalability

i.e., the capacity to be changed in size

- In theory, software is not physically restricted in its size; in practice, scalability does not come free.
- What is the most efficient sorting algorithm? :)

Science of Sorting



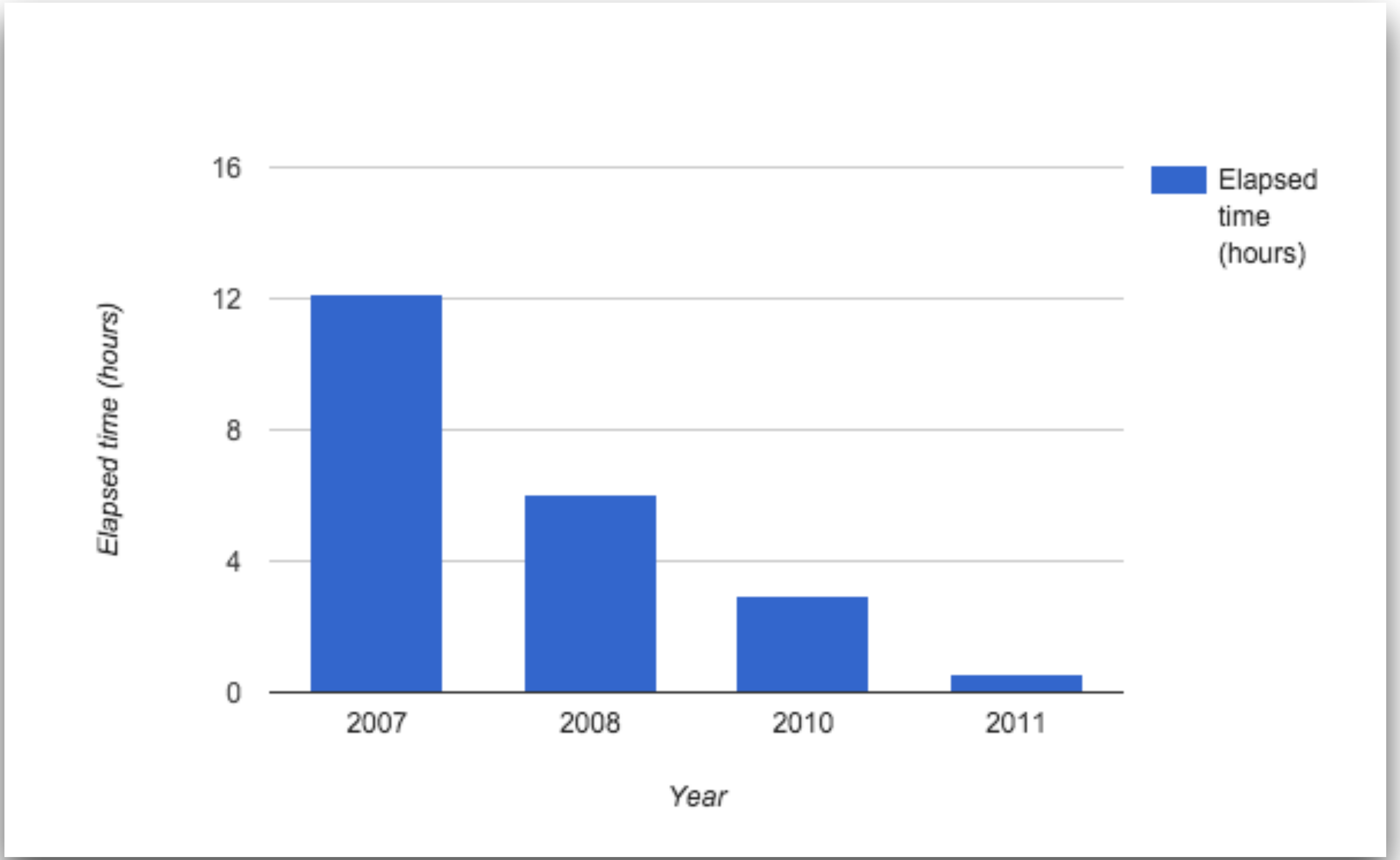
```python
def mergesort(x):
    """ Function to sort an array using merge sort algorithm """
    if len(x) == 0 or len(x) == 1:
        return x
    else:
        middle = len(x)/2
        a = mergesort(x[:middle])
        b = mergesort(x[middle:])
        return merge(a,b)
```

Now, what do we need if we want to sort 1 petabytes of data?

#precise #O(nlogn) #proven

Engineering the scalability of sorting

from “History of massive-scale sorting experiments at Google”

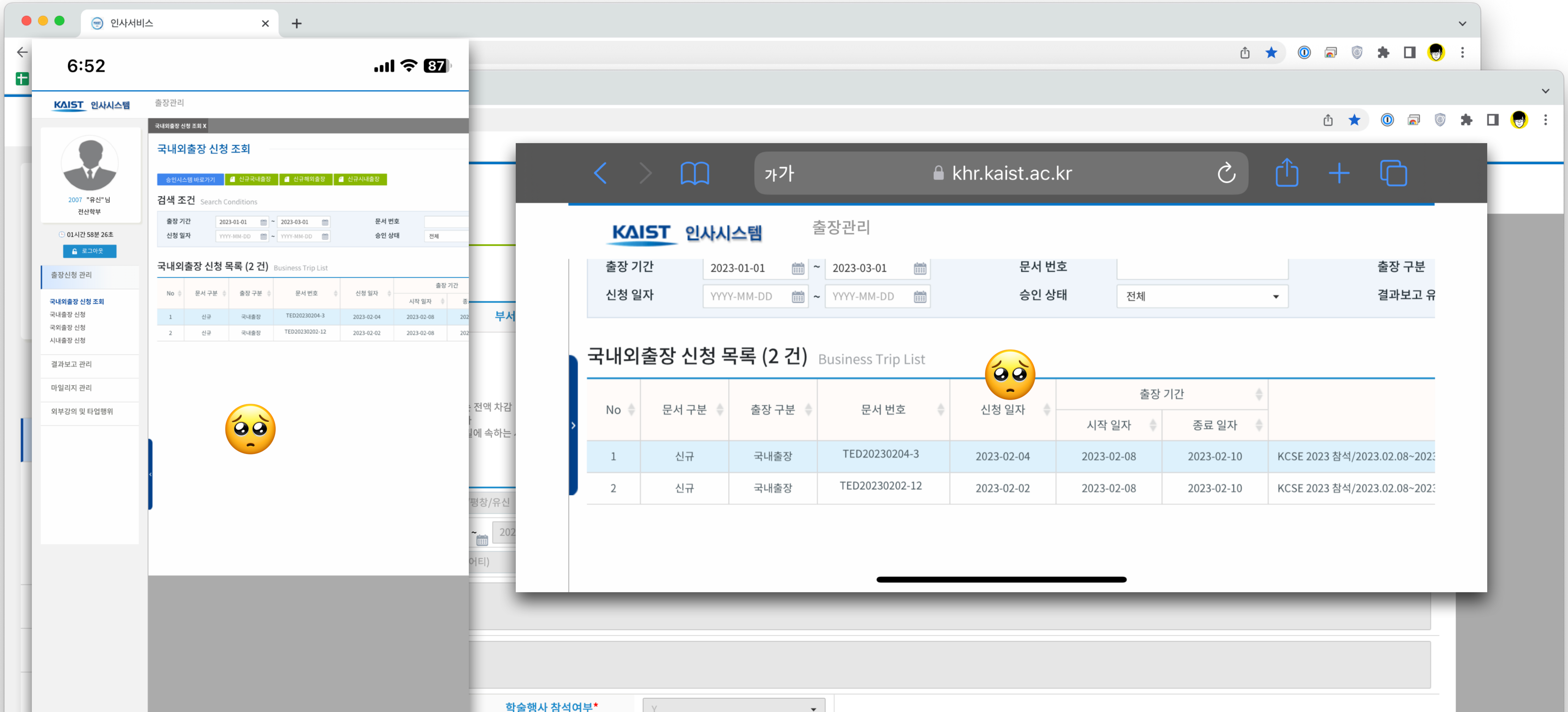


- Tuning cluster configuration
- Changing the file system entirely + new encoding to reduce write amount
- Hardware tuning (I/O block size + SSD)
- Correctness check only came in 2010 :)

https://cloud.google.com/blog/products/gcp/history-of-massive-scale-sorting-experiments-at-google

Usability / Accessibility

“It works” and “It is nice to use” are different.



국내외출장 신청 조회

검색 조건 Search Conditions

출장 기간	2023-01-01 ~ 2023-03-01	문서 번호	
신청 일자	YYYY-MM-DD ~ YYYY-MM-DD	승인 상태	전체

국내외출장 신청 목록 (2 건) Business Trip List

No	문서 구분	출장 구분	문서 번호	신청 일자	출장 기간		
					시작 일자	종료 일자	
1	신규	국내출장	TED20230204-3	2023-02-04	2023-02-08	2023-02-10	KCSE 2023 참석/2023.02.08~202
2	신규	국내출장	TED20230202-12	2023-02-02	2023-02-08	2023-02-10	KCSE 2023 참석/2023.02.08~202

Usability / Accessibility

“It works” and “It is nice to use” are different.

Maintainability

- In real SW project, you are never “done”: your code/service/product may outlast your presence in the organisation, and they keep evolving.
- Investigations of real world projects show that maintenance cost is over 60% of Total Cost of Ownership (TCO).
- Changes made over lifetime of SW are of multiple types (IEEE1219/P14764):
    - Perfective changes (i.e., adding features): more than 50%
    - Corrective changes (i.e., fixing bugs): about 20%
    - Adaptive changes (i.e., adapting to new environment): about 20%
    - Preventive changes (i.e., preventing latent faults) - new addition in ISO/IEC 14764

Maintainability

- Whether your software allows smooth evolution is a fundamental concern.
- Many relevant discussions stem from this:
  - Documentation
  - Code comments: do you support, or not support, comments?
  - Refactoring

Testability

(a related concept)

- It should be easy to test your software… but what do we mean by “easy to test”?
- Testing is done at multiple levels: unit (i.e., individual functions and classes), integration (i.e., testing the connections between units), and system (i.e., testing everything put together) - a good modular design naturally supports this well.
- Testers and developers may be different people (more on this later) - a highly readable, well documented code is naturally easier to test.

Security

- A good software should be secure: it should protect the information it processes.
- Thorough consideration of software security can be pervasive, affecting everything from architecture & design, through writing good code, eventually to usability.

Functional/Non-Functional Requirements

- Functional Requirements: what is expected in terms of input-output behaviour (i.e., correctness)
- Non-Functional Requirements: Properties related to the general operation of the software system, instead of specific behaviour and correctness
    - Many of the qualities we have examined: reliability, security, accessibility, performance…
    - Harder to analyse or test!
    - Let’s look at a few examples…

Worst-Case Execution Time Analysis

- For certain real time embedded systems, you need to know how long the program can take when executed
  - Airbag controllers: there is a specific time window for the airbags to be effective - triggering earlier or later than the time window is unsafe.
  - We are used to analysing algorithmic complexity, but
    - the analysis only asymptotic - it does not tell us the concrete wall clock time
    - it does not tell us anything about the specific hardware platform your code runs one

Side-Channel Attacks

(a simple example)

- Side channel attack exploits information that can be gathered because of the way SW is **implemented**, not designed.
- What would be the “timing attack” for the code on the right”
- How about “power analysis”?

```
bool check_password(const char input[]){
  const char correct_password[] = "hunter2";

   if (strlen(input) != strlen(correct_password)) return false;

   for (int i = 0; i < strlen(correct_password); i++){
       if (input[i] != correct_password[i]) {
           return false;
       }
   }

   return true;
}
```

Summary

- What we covered today is by no means an exhaustive list of things you need to consider when developing software.
- Given the multitude of quality criteria you need to consider, it is best that your team has
    - a diverse and suitable set of expertise
    - high diversity in the team itself