



Software Engineering from First Principles

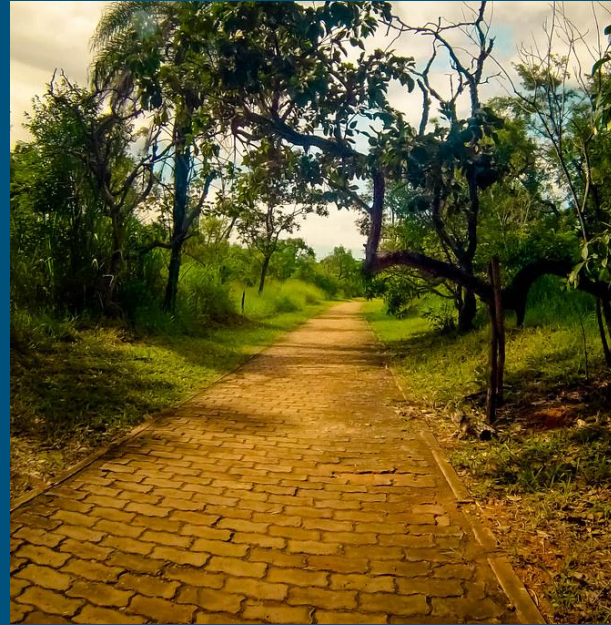


Zongyao Mao



First, a fun exercise

“All roads lead to Philosophy”



First Principle

A first principle is a basic proposition or assumption that cannot be deduced from any other proposition or assumption.


Examples of First Principles

First Principle of Newtonian Motion:


Every body continues in its state of rest, or in uniform motion in a straight line, unless it is compelled to change that state by forces impressed upon it.

Why bother?

Learning about the underlying foundations of the things you use can help you use them better.



First Principle: Abstraction



“Every name is an abstraction”



Data structures are abstractions

Queue, Stack, LinkedList: 1-dimensional data management

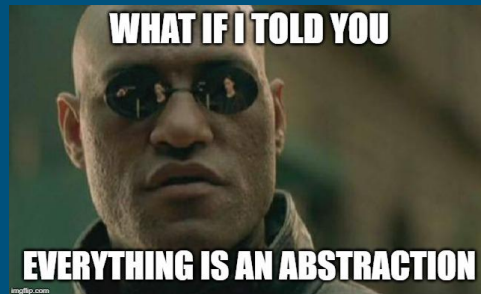
Tree: Hierarchy management

Graph: $N \times N$ relationship management

Underlying every data structure operation is a complex set of instructions to manipulate it in a way that makes sense to the user.

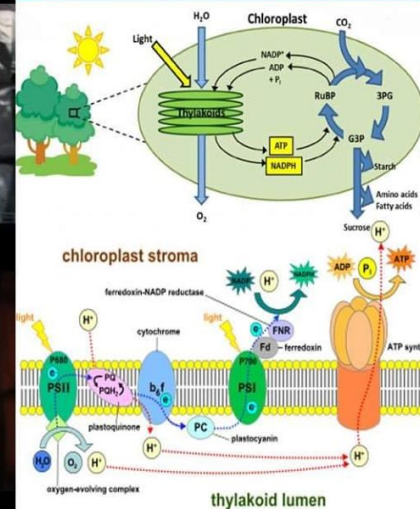
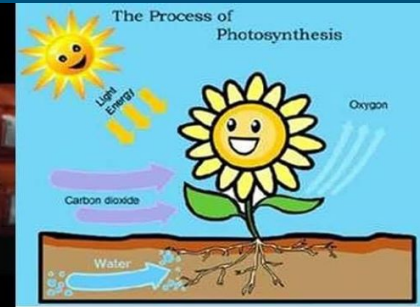
What do these names abstract?

- Framework
- Function
- Programming Language
- OOP
- TCP
- Kafka
- Agile
- Hackathon
- Pair programming



Implication #1

There are good and bad abstractions.



Implication #2

If you cannot find the problem, it's probably hidden by abstraction in a lower, more complex layer.





Second Principle: Tradeoff



“All abstractions have tradeoff”



Your first tradeoff lesson in CS

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	Worst
Array	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Stack	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
Queue	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
Singly-Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
Doubly-Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
Skip List	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n \log(n))$
Hash Table	N/A	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	N/A	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Binary Search Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Cartesian Tree	N/A	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	N/A	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
B-Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(n)$
Red-Black Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(n)$
Splay Tree	N/A	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	N/A	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(n)$
AVL Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(n)$

<https://www.bigocheatsheet.com/>

Example: Protobuf

JSON is readable. Why does Protobuf exist?

- Low-bandwidth scenarios
- Performance

To compensate, we created Protobuf content viewers. What's the tradeoff?

- Need for extra tools to perform daily functions

Example: UDP

TCP offers guaranteed communication, so why did we create UDP, a protocol that doesn't guarantee communication?

- Connection required: setup time, notion of sessions
- Slow: retries potentially slow down throughput
- Guaranteed: not all scenarios require guaranteed communication

Even TCP's strength (guaranteed communication) can become its weakness in some scenarios.

Example: Programming languages

Writing in assembly allows us to super-optimize our code. Why do we write in a higher-level programming language?

- Readable
- Maintainable

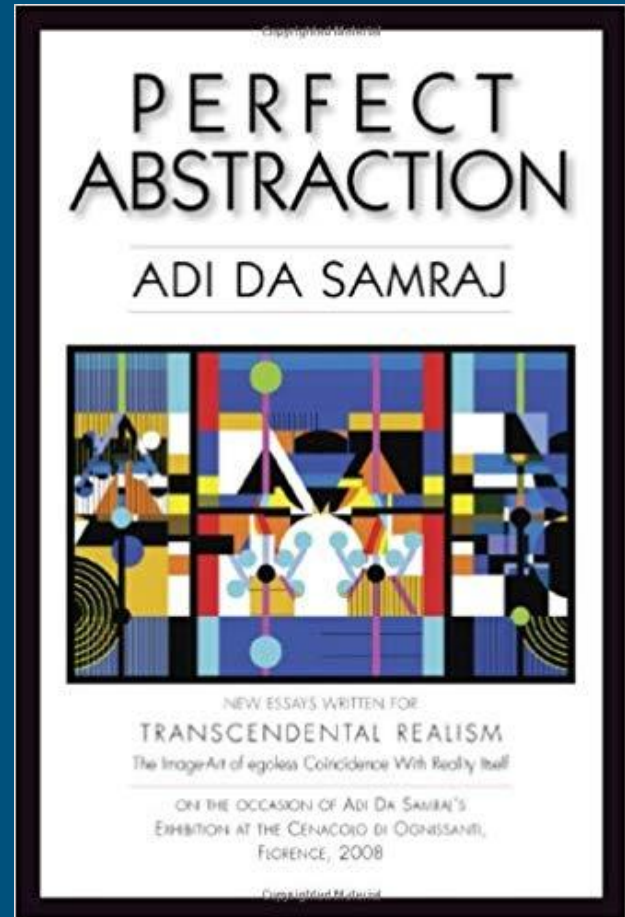
Does an optimizer solve the tradeoff? What did we lose?

- Additional compile time needed to optimize code
- Debuggability

Implication #3

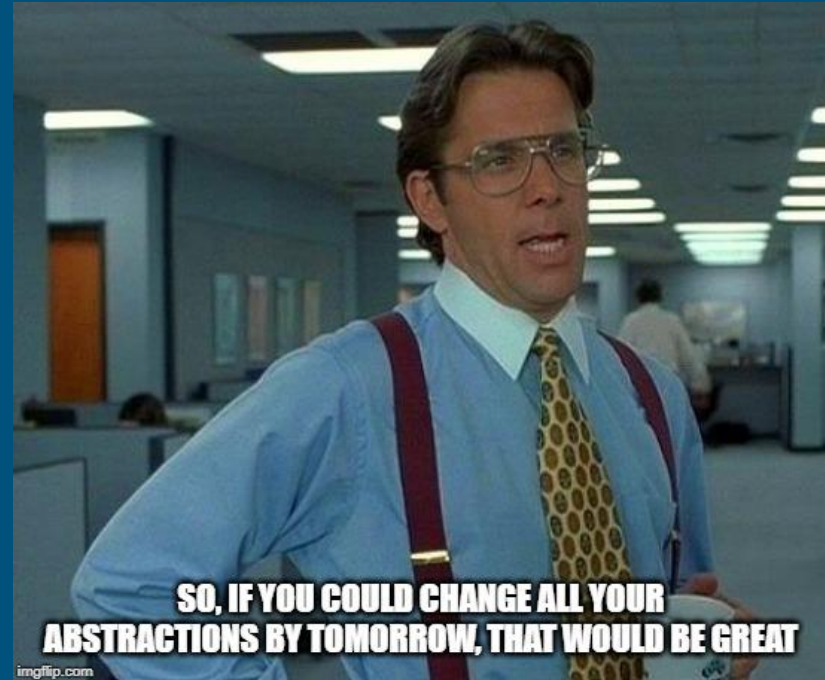
There is no perfect abstraction for every single use-case

(But apparently they exist in art)



Implication #4

Even well-designed abstractions can become worse over time as requirements change your tradeoffs.



Implication #5

When you bring something new into your product, you're implicitly also bringing in its abstraction and tradeoffs.





Applications



a.k.a. “I learned 2 new principles but it
wasn’t tested in exams”



Changing project requirements

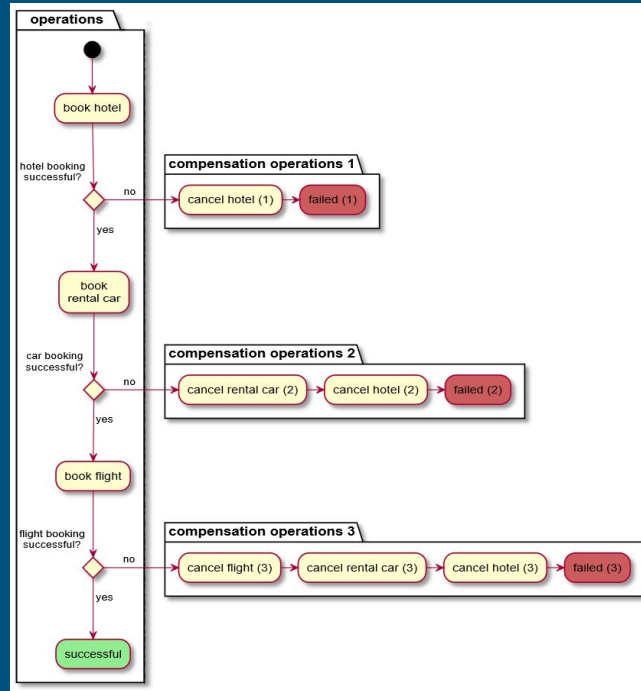


Microservices



<https://commons.wikimedia.org/wiki/File:Services4.png>

The Saga (Pattern) Continues



https://commons.wikimedia.org/wiki/File:Saga_pattern-UML.png

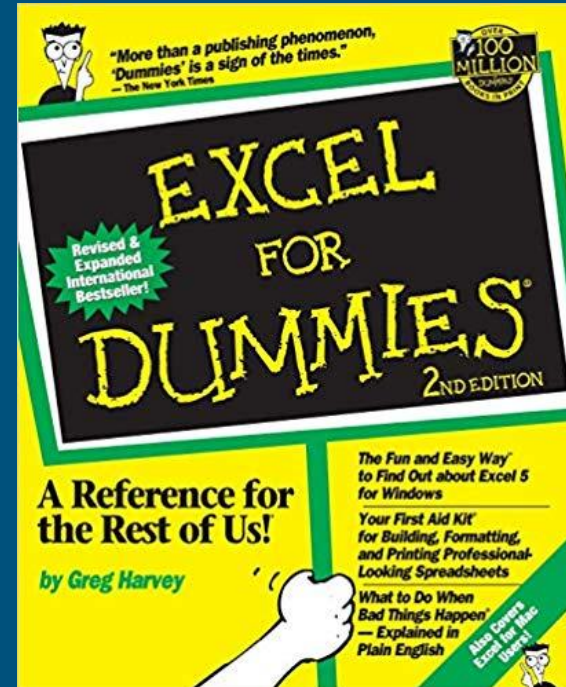
I need this library

```
npm ERR! npm v2.14.7
npm ERR! code E404
npm ERR! 404 Registry returned 404 for GET on https://registry.npmjs.org/left-pad
npm ERR! 404
npm ERR! 404 'left-pad' is not in the npm registry.
npm ERR! 404 You should bug the author to publish it (or use the name yourself!)
npm ERR! 404 It was specified as a dependency of 'line-numbers'
npm ERR! 404
npm ERR! 404 Note that you can also install from a
npm ERR! 404 tarball, folder, http url, or git url.
npm ERR! Please include the following file with any support request:
npm ERR!    /home/travis/build/coldrye-es/pingo/npm-debug.log
make: *** [deps] Error 1
```

<https://arstechnica.com/information-technology/2016/03/rage-quit-coder-unpublished-17-lines-of-javascript-and-broke-the-internet/>

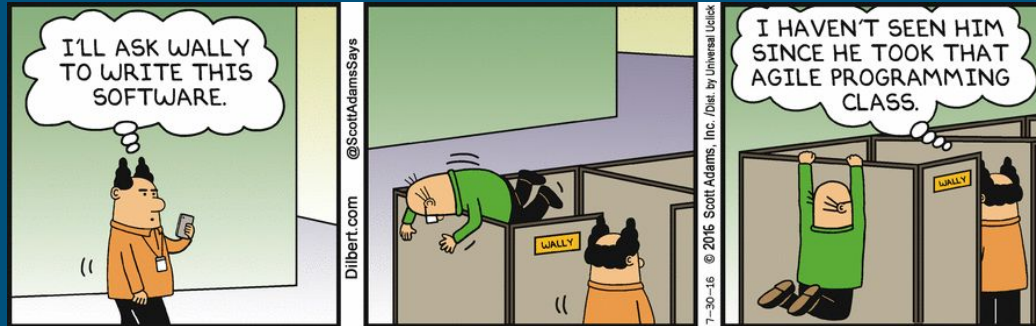
To SQL or not to SQL ...

... that is (not) the question.



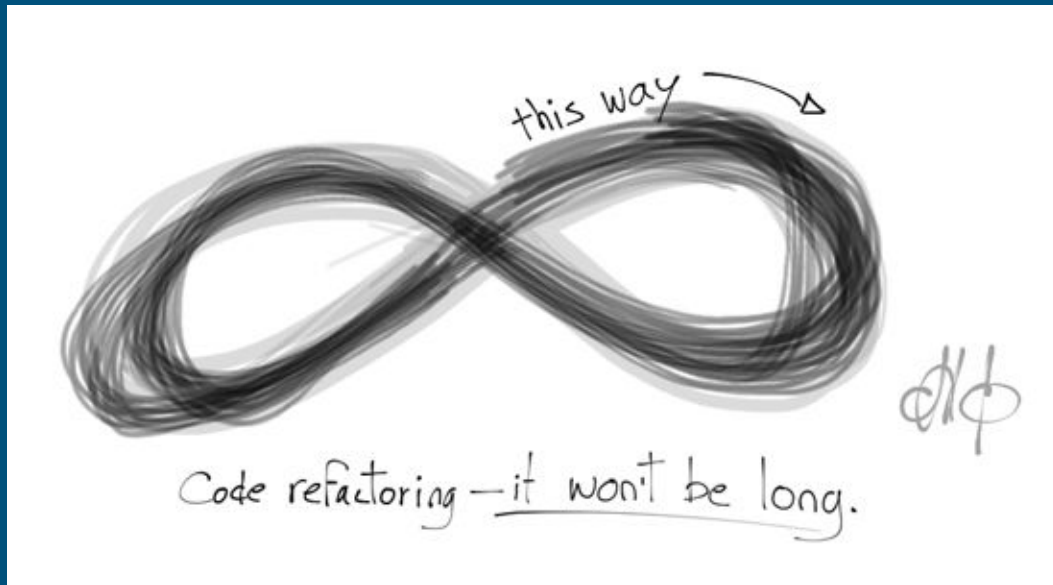
The only database you'll ever need
<https://www.wiley.com/en-gb/Excel+For+Dummies,+2nd+Edition-p-9781568840505>

“That’s not Agile”



<https://dilbert.com/strip/2016-07-30>

Need.. to.. refactor..



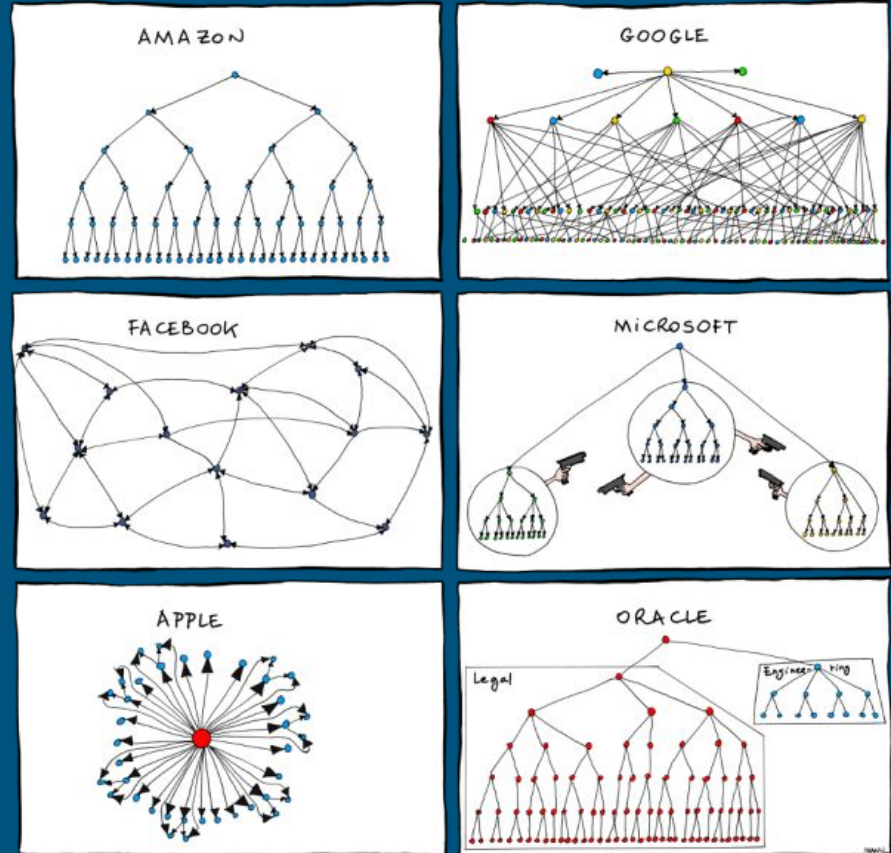
<https://swizec.com/blog/what-refactoring-is-and-what-it-isnt/swizec/4148>

Let's optimize this!



Creating the right team structure

Software Engineering isn't limited to just code



Exercise: Self-review

- Think of the last time you decided to use a new “thing” in your project
 - Patterns, libraries, framework, application, tools, ...
- Answer these questions:
 - What was the problem you wanted to solve?
 - What are your current system tradeoffs?
 - How does this new thing help you compensate?
 - What are your new tradeoffs?
 - What are your other options?

Takeaways

- Understand the abstractions and tradeoffs about *everything* you use.
 - Libraries, frameworks, patterns, processes, ...
- When your requirements change, examine whether your abstractions and tradeoffs have been broken.
- Rewriting is not always necessary, sometimes you can just create compensation for your tradeoffs.
- Don't be afraid to create new abstractions, but understand your tradeoffs.

Questions?

Slides:

<https://www.zongyaomao.com/talks/2020-01-21-Govtech-1st-principles.html>

Website: <https://www.zongyaomao.com>

LinkedIn: <https://www.linkedin.com/in/zongyaomao>