Maximising Profit From Programs

Samuel Desmond Savage, Anna Ferrari and Hannah Langridge

April 10, 2023

Contents

1 Abstract and Introduction

This paper first aims to provide absolute unambiguous clarity to ubiquitously used yet often ill-defined vocabulary within the fields of Software Engineering and Data Science. In sections Basic Definitions to Program Correctness definitions will include “Computer”, “Language”, “Computable Function”, “Parser”, “Compiler”, “Programming Language”, “Refactoring”, “Dead Code”, “Abstraction”, “Obfuscation”, Complexity as “Time Complexity”, “Space Complexity”, “Formula Complexity”, “Kolmogorov-Chaitin Complexity”, “Compile/Feature Complexity”, and also “Accuracy”, “Bug”, “Bug Density” and “Probability”.

The subsequent sections Maximising Profit From Programs and Practically Applying The Optimisation Heuristics we apply the definitions from the earlier sections to give formally defined easy to apply heuristics for ultimately solving the optimisation problem of making a program as profitable as possible to a given company. In these sections we heavily cover cognitive biases like “Imposter Syndrome” and “Not Invented Here”.

In the rest of the paper we will collectively refer to any professional that writes code for their job as a “developer”, so we mean this to include all possible variants, such as “Data Scientists”, “Software Developers”, “Software Engineers”, “Data Engineers”, etc.

1.1 Dedication

It’s estimated that Alan Turing’s creativity in mathematics and engineering, by essentially inventing the computer, to assist in codebreaking in the second world war reduced the length of the war by 2 to 3 years saving 14 to 21 millions lives. We dedicate this paper to his memory.

2 Basic Definitions

2.1 Definition - Language $\mathscr{L}$

A Language, which we denote using mathscr font; $\mathscr{L}$, consists of 'logical symbols' which occur in every language, $\forall, \exists, \vee, \wedge, \rightarrow, \neg, =, (,)$, where:

- $\forall$ – The universal quantifier symbol meaning for all.
- $\exists$ – There exists.
- $\vee$ – The logical connective for and.
- $\wedge$ – The logical connective for or.
- $\rightarrow$ – The logical connective for implies.
- $\neg$ – The logical connective for not.
- $=$ – The equality symbol.
- $(,)$ – Parenthesis. $\mathscr{L}$ has an infinite set of variables, $x_1, x_2, x_3...$. These can be thought of as names – there is an infinite number of names in any language.

$\mathscr{L}$ also consists of 'non-logical' symbols which are specific to each Language.

- A set of Predicate symbols, P, often written as $P_1, P_2, P_3...$
- A set of Function symbols, F, often written as $F_1, F_2, F_3...$
- A set of Constant symbols, C, often written as $c_1, c_2, c_3...$

Predicate and Function symbols each have an arity, which refers to the number of arguments each takes. An arity is often noted in superscript above the predicate and function symbols where n represents the arity.

- $P_1^n, P_2^n, P_3^n, ...$ shows predicate variables each with a arity of n.

As the predicate and function symbols alongside their corresponding arities differ between languages, a language is essentially defined by the tuple $< P, F, C >$. As the logical symbols occur in every language, they are not included in the tuple. Constant symbols do not have an arity.

2.2 Notation - Strings

For symbols $S_1, S_2, ...$ if we write "$S_1S_2...$" this is shorthand for the vector $< S_1, S_2, ... >$, which we call a **string**. A **substring** of a string "$S_1S_2...S_n$" is a string "$S_iS_{i+1}...S_{i+m}$" where $i + m \leq n$. Let X be a substring of Y we also say X **occurs in** Y. Sometimes we will omit the quotes " and " if the context is clear. Finally for any string x, $|x|$ is the length of x.

2.3 Definition - Term

We inductively define a term as follows:

- A variable or a constant e.g. $x_1, x_2, x_3...$
- A string "$F(t_1, ..., t_n)$", where $t_1, ..., t_n$ are terms – a function symbol applied to other terms.

2.4 Remark

Generally expressions of operators are written in in-fix notation such as (0 + 1) which some people find easier to understand compared to pre-fix notation +(0,1) – both mean the same thing. Computer parsing of pre-fix notation is generally more efficient (or the similar postfix notation (0, 1)+), but modern computers rarely struggle to parse any notation. +(0,1) is also called an expression in computer science. The '+' is the function symbol and the terms are 0 and 1.

Using pre-fix notation, $+(+(0, 1), 1)$ is also an example of an expression/term.

2.5 Definition - Atomic Formula

Given any predicate 'P' and terms $t_1, ..., t_n$, then,

- $t_1 = t_2$ is an atomic formula
- $P(t_1, ..., t_n)$ is an atomic formula

2.6 Definition - Formula

We define (well-formed) formula inductively as follows. For any sequence of symbols ϕ and ψ,

- If ϕ is an Atomic Formula then it is a Formula
- Negation: If ϕ is a formula, then so is $\neg\phi$.
- Binary connectives: if ϕ and ψ are atomic formulae, $\phi \rightarrow \psi$ is a formula. This applies with other logical connectives.
- Quanitifiers: If ϕ is an atomic formula and x is a variable, $\forall x\phi$ and $\exists x\phi$ are both formulas.

2.7 Example

- $E = mc^2$, the mass-energy equivalence formula is an example of an equality statement with two terms either side.
- $\in (x, \mathbb{R})$, usually written in infix notation, $x \in \mathbb{R}$, is an example atomic formula using a predicate

2.8 Definition - Free variable

Free variables within formulae are defined inductively.

- In the atomic formulae, ϕ, the variable x is free in ϕ iff x occurs in ϕ.
- x occurs free in $\neg\phi$ if and only if x occurs free in ϕ.
- x occurs free in $\phi \rightarrow \psi$ iff x occurs free in either ϕ or ψ.
- x occurs free in $\forall y\phi$, iff x occurs free in ϕ and x is not the same symbol as y.

2.9 Variable Substitution Notation

Let the variable y occur in ψ, for any string x, we write $\psi(y \rightarrow x)$ be the result of replacing y with x. Sometimes this is written simply $\psi(x)$ if y is implicit or irrelevant, similarly for more than 1 substitution we write $\psi(x_1, ..., x_n)$. In most literature usually this notation is reserved for when the variables occur free in ψ, but that convention does not hold in this paper.

2.10 Definition - $\mathscr{L}$ - Sentence

A sentence is defined as formulas that do not contain any free variables.

2.11 Example

- $x + 1 > 0$ is not a sentence as x is a free variable.
- $\forall x\, x + 1 > 0$ is a sentence, as the x has been quantified.

2.12 Notation

The field of Predicate Calculus contains definitions of the semantics and truth of a sentence, and for any sentence s we denote that it is true with respect to all possible semantics by $\vDash s$. This paper will not give a full definition of $\vDash$. Also $s_1 \leftrightarrow s_2 := (s_1 \rightarrow s_2) \wedge (s_2 \rightarrow s_1)$. So when you see $\vDash s_1 \leftrightarrow s_2$ it is to be read as "s_1 and s_2 have equivalent meaning" and this informal understanding should suffice for this paper (if it doesn't, then recommend reading say "A Mathematical Introduction To Logic" - Enderton, "Introduction to Mathematical Logic" - Mendelson) or "Model Theory: An Introduction" - David Marker.

3 Definitions for The Theory of Computation

3.1 Definition - Turing Machine (Computer)

A Turing machine (TM) is a 7 tuple
$< Q, \Gamma, b, \Sigma, \delta, q_0, F >$.

$$M =< Q, \Gamma, b, \Sigma, \delta, q_0, F >$$

- Γ is a finite, not empty ($\Gamma \neq \phi$), set of 'tape alphabet symbols' – symbols which can be written on the tape.
- b is a member of the Γ set of 'tape alphabet symbols', $b \in \Gamma$, and corresponds to blanks.
- $\Sigma \subseteq \Gamma \backslash b$, the is a set of 'input symbols'. These are symbols which are allowed to appear on the initial contents of the tape.
- Q is a 'set of states' and is finite.
- q_0 is the initial state.
- $F \subseteq Q$ and represents the 'final states'.
- $\delta : (Q\backslash F)$ x $\Gamma \nrightarrow Q$ x Γ x $\{L, R\}$
  - Here L corresponds to 'left shift' and R corresponds to 'right shift'.
  - The first argument to δ is a state (Q) not including a final state (F) and the second argument is a 'tape alphabet symbol' (Γ). Any tape and state symbol pair is then mapped to another tape and state symbol pair as seen above. The second set has three arguments as it includes the third argument, either left or right $\{L, R\}$.
  - It is described as a partial function as it is not defined on every input, meaning if it is given an input of a state and tape symbol for which it is not defined, the machine will stop.

Let $\mathbb{T}$ be the set of all Turing Machines.

3.2 Example

The above image (TODO FIX) is a TM which accepts or rejects inputs based on whether they are palindromes or not. The head moves to the first letter of the input and identifies it as a 'b', then removes it. The head then moves to the right until it hits the blank space, thus identifying the space before as the the last character of the input, it moves one to the left and identifies that character. If also a 'b' like in this example, then the head moves back to the first character (the first 'a', as the 'b' has been removed) and the process begins again. If the character wasn't a 'b' then the TM would reject the input as it wouldn't be a palindrome. If the process continues until all the letters have disappeared, then the TM would accept the input, as it would be a palindrome.

3.3 Definition - Binary Strings

We define the set of Binary Strings as follows:

$$\mathbb{B} := \bigcup_{N \in \mathbb{N}} \{0, 1\}^N$$

3.4 Example

- If $N = 2$, then the Binary Strings could take the form of $< 0, 0 >, < 1, 1 >, < 0, 1 >$, or $< 1, 0 >$.

3.5 Definition - Computable function

A binary function ($\mathbb{B} \to \mathbb{B}$) is computable iff there exists a TM which, given any input, will return the correct output for that function. Such that, for all x within the binary string ($\forall x \in B$), if x is the initial tape values, the tape values are $f(x)$ when the TM halts.

3.6 Definition - Induced/Represented Function

Conversely, any TM M "induces", or "represents" a computable function $R(M) : \mathbb{D} \to \mathbb{B}$, where $\mathbb{D} \subset \mathbb{B}$ which contains exactly the elements of $\mathbb{B}$ such that M halts.

3.7 Definition - Parser

p is a Parser iff $p : \mathbb{B} \to \mathscr{L}$-sentences. The "Canonical Inverse" of a Parser p, written p^{-1} is $p^{-1} : \mathscr{L}$-sentences $\to \mathbb{B}$ where for any $\mathscr{L}$-sentences s and any binary string b such that $p(b) = s$, then $|p^{-1}(s)| \leq |b|$. WLOG the Canonical Inverse is unique. Strictly speaking a parser is a partial function, i.e. it is not defined on every binary string.

3.8 Definition - Compiler

c is a Compiler iff $c : \mathscr{L}$-sentences $\to \mathbb{T}$.

3.9 Remark

The term "parser" is also used to refer to functions $\mathbb{B} \to \mathbb{B}$ that transform the binary representation of something. For example the date-time parser, p, returns the amount of seconds that have passed since 01/01/1970 00:00:00, thus $p(\text{"}25/10/202109 : 42 : 00\text{"}) \to 1635154920$.

3.10 Definition - Programming language

A programming language in a Language $\mathscr{L}$, or $\mathscr{L}$-Programming Language, is a 3-tuple, $< \mathscr{L}, c, p >$, where c corresponds to a 'compiler' and p to a parser, such that, for sentences s_1 and s_2, if they are equivalent, $\vDash s_1 \leftrightarrow s_2$, then $R(c(s_1))) = R(c(s_2)))$. We say $\mathscr{L}$ are the Programming Language Features (or just Features) of an $\mathscr{L}$-Programming Language.

3.11 Remark

Notation as above, observe that we don't require that $c(s_1) = c(s_2)$, this is because equivalent sentences when compiled can give different Turing Machines that although *functionally* equivalent may differ in computational steps.

3.12 Notation - Program, Program Binary, and Abstract Syntax Tree

We say a $\mathscr{L}$-sentence is a $\mathscr{L}$-program or "Abstract Sytnax Tree", and can call the input binary string the $\mathscr{L}$-program-binary.

3.13 Definition - Transpiler

Given programming languages $< \mathscr{L}_1, c_1, p_1 >, < \mathscr{L}_2, c_2, p_2 >$, we say $t : \mathbb{B} \to \mathbb{B}$ is a transpiler from $\mathscr{L}_1$ to $\mathscr{L}_2$ iff $\forall b \in \mathbb{B}$ $R(c_1(p_1(t(b)))) = R(c_2(p_2(b)))$.

3.14 Example

The assignment of 10 to the value *number* in R is:
```
number ← 10
```

Through the use of a transpiler, this code can be transformed into Scala code:
```
val  number = 10
```

3.15 Definition - Refactoring

s_1 is a refactoring of s_2 iff $\vDash s_1 \leftrightarrow s_2$.

3.16 Definition - Dead code

s_1 is said to contain dead code if symbols can be removed from it to make some s_2, such that $\vDash s_1 \leftrightarrow s_2$.

3.17 Remark

Dead code causes the program to be more complex than is necessary and can make it more confusing to interpret. A program containing no dead code we'll call "maximally alive".

3.18 Example

- The following example contains some dead code:

```
a = 1
b = 2
c = 3
a + c
```

  As shown, 'b' is assigned a value, but is not used in any of the code, therefore this is considered Dead code.

3.19 Definition - Dead code remover

For any s_1, the function $D : \mathscr{L}\text{-Sentence} \to \mathscr{L}\text{-Sentence}$ removes any dead code from a sentence to make $D(s_1)$ maximally alive. WLOG we can assume this function to be unique.

3.20 Notation

$\mathscr{L}\text{-Alive} := D(\mathscr{L}\text{-Sentence})$

3.21 Definition - Formula Complexity

For any $\mathscr{L}$-formula F, the formula complexity, $|F|$, is the length of the sequence.

3.22 Example

- $|P(x_1, x_{4000}, c, f(x))| = 13$
- $|\text{length}(x) + 10 = 50| = 10$

3.23 Definition - Abstraction

s_2 is an "abstraction" of s_1 iff $|s_2| < |s_1|$ and $\vDash s_2 \leftrightarrow s_1$

3.24 Remark

After the removal of dead code, abstraction (a type of refactoring) is the proceeding condensing of code to increase clarity and efficiency whilst retaining the same output as before.

3.25 Example

- If we take the following code to be s_1, program which prints out the numbers from 1-100 twice:

```
val  x = "1,  2,  3,  4,  5,  6,  ...   ,  100"
print (x)
print (x)
```

  Then the following code, s_2, would be an abstraction as it is shorter in length than s_1 but still equivalent to it:

```
val  x = (1  to  100).mkString(‘‘, ")
print (x)
print (x)
```

3.26 Definition - Obfuscation

"Obfuscation" is when $|s_2| > |s_1|$ and $\vDash s_2 \leftrightarrow s_1$

3.27 Remark - Naming Things is Hard, so Avoid it when possible!

There are two hard things in computer science: cache invalidation, naming things, and off-by-one errors.

In practice developers often subjectively debate a lot as to whether a refactoring is an Abstraction or an Obfuscation. This paper offers formal definitions that can aid communication between developers by giving them an unambiguous and objective common vocabluary. Note: under our definitions a program-binary may get longer for an abstraction or shorter for an obfuscation, since what matters here is not the total length of the sequence of *characters* in a program's file(s), but the total length of the sequence of *symbols* from the programming language (i.e. the size of the Abstract Syntax Tree). It would be false to consider a *character* as a *symbol*. So if cosmetic whitespace is added to a program this may make the file(s) longer, but it doesn't impact the Formula Complexity since cosmetic whitespace isn't an actual symbol of the programming language - that is to say it's eliminated after parsing.

Sometimes developers increase the Formula Complexity (size of the AST) but will argue that the resulting code is "easier for them to read" even though there are now more symbols to read. In later sections we give a formal treatment of the actual financial cost to a company of "readability" in terms of 4 complexity classes. For now, we observe that perhaps what the developer means to say is that "According to my cultural, linguistic, experiential and cognitive context from whence my current mind arises my brain can expend less energy to parse, internally compile and internally assign semantics for this particular given sequence of characters". Now our definitions are intentionally devoid of cultural, linguistic, experiential and cognitive context - i.e. devoid of subjectivity, they are purely platonic, and therefore objective.

Typically when there is this friction between what is objectively simpler platonically vs what someone subjectively finds easier to read due to conditioning this results in code where the author imposes their own meta-language about what they *think* the code is doing in non-useful ways. Comments are one trivial example, but since the parser will discard them they don't cause much problems; other developers can ignore them or even delete them. The most non-trivial common form of this is variables or functions that are only used once so that the author can *name* some expression or block of code (henceforth we will call this an "indirection") in order to impose the meta-language corresponding to their internal monologue (or *thought processes*). This kind of imposition in of itself has no value except to that particular developer, but when combined with Unit Tests this is a totally different matter and can have great value. We will cover the value of Unit Tests in detail in later sections when we deal with Program Correctness and Tradeoffs.

A perfect analogy here would be an English person annotating a book written in French to help their own reading of the book. To a person fluent in French the English annotations would *impede* reading the book not help.

Naming things is hard because good names need to have some meta-linguistic properties such that (ideally) all people can gain an immediate intuition about what is being named from the name. In practice this is rare to impossible, usually names are closely coupled to a subset of linguistic contexts. In fact human recognisable phenoms, sounds and writing systems are pretty finite, so no single objective linguistic context can exist when there are infinitely many concepts to classify. Therefore when we don't really need to give a name to something it's better to give no name at all.

Poincaré (a French mathematician) once gave a poetic definition of abstraction by saying "[abstraction] is the art of giving the same name to different things". Indirections are abstraction failures by this definition also. Giving a name to something is only useful provided that name applies to multiple things (i.e. multiple parts of a programs execution). The very point of a name for a thing is to shorten communications when referring to that thing so that the entire thing need not be re-described every time. An indirection only lengthens the communication.

Finally we observe that junior developers learning a new language are more likely to commit indirection because they have not yet fully internalised the syntax and semantics of the programming language to the degree where they can effortlessly read it. Consequently they name bits of code to remind *them* what a bit of code does (similarly with comments) but non-junior developers will not find these useful as they can read the code easily without the redundant names. In fact to a non-junior developer the indirections usually just slow down the reading as its extra symbols to parse, and the names are rarely perfectly chosen.

3.28 Example

- If we take the previous example of s_2 :

```
val  x = (1  to  100).mkString(‘‘,")
print (x)
print (x)
```

s_3, as shown below, would be an obfuscation (by indirection) of s_2.

```
val  list = (1  to  100)
val  comma = ‘‘,"
val  x = list.mkString(comma)
print (x)
print (x)
```

Perhaps the author isn't familiar with the syntax (1 to 100) and like to remind themselves that this is a list, hence give it the name "list". To an experienced developer in this language, this extra name does absolutely nothing to help them, and just makes the program longer.

4 Complexity Theory

4.1 Definition - Big O Notation

Given functions f and g on numbers, we say $f \equiv O(g)$ iff positive constants c and k exist such that $0 \leq f(n) \leq cg(n)$ for all $n \geq k$.

4.1.1 Further Notation

Big O Notation has a number of naunces:

- In a lot of literature $=$ is used in place of $\equiv$, which we deem poor nomenclature as it overloads $=$.
- Furthermore, sometimes $f \equiv O(g)$ is written "$f(x) = O(g(x))$ as $x \to \infty$".
- Furthermore, sometimes the $x \to \infty$ is ommitted entirely when using Big O Notation, which can be awkward because x becomes unbound, and so we are not really writing in complete *sentences*. You'll often see statements like "The complexity of this algorithm is $O(n^2)$".
- Given functions g and h, $O(g) \equiv O(h)$ means $\forall f, f \equiv O(g) \Leftrightarrow f \equiv O(h)$
- Similarly $O(g) > O(h)$ means: $\exists f, f \equiv O(h) \wedge \neg f \equiv O(g)$ *and* $\forall f, f \equiv O(g) \Rightarrow f \equiv O(h)$.

It's because of these naunces that Big O Notation is often used quite informally and it's up to the reader to make a canonical guess as to the mathematical meaning.

4.2 Remark

As can be seen in the figure above (TODO fix image), $cg(n)$ (after k) is always above $f(n)$ and then both functions will become asymptotic.

4.3 Theorem

If the following limit exists and is non-zero, $\lim\limits_{x \to \inf} \frac{f(x)}{g(x)}$, then $f(n) \equiv O(g(n))$.

Proof

Exercise

4.4 Definition - Time Complexity

Let M be a TM that halts on any input, the time complexity of M, w.r.t. $g_1, ..., g_n$ (where $g_1, ..., g_n : \mathbb{B} \to \mathbb{N}$), is a function $f : \mathbb{N}^n \to \mathbb{N}$ given $x_1, ..., x_n \in \mathbb{N}$, $f(x_1, ..., x_n)$ is the maximum number of steps M uses given any input $b \in \mathbb{B}$, such that, $g_1(b) = x_1, ..., g_n(b) = x_n$.

4.5 Example

- The following code executes a program which returns true or false depending on whether the input is a palindrome or not:

  ```
  def isPalindrome(s:  String):   Boolean = {
  s == s.reverse
  }
  ```

  If we take the length of the input to be N (i.e this could be g_1). The code has to run through the entire length of the input in order to reverse it and run through it again to compare the input to the reverse for the equality statement. Therefore, the time complexity of this is $O(2N)$. However, $O(2N) \equiv O(N)$ (TODO proof exercise). Therefore the time complexity of this code is O(N).

4.6 Remark

In practice, we do not need exact proofs of time complexity, however estimates of it are useful.

4.7 Definition - Space Complexity

Let M be a TM that halts on any input, the space complexity of M, w.r.t. $g_1, ..., g_n$ (where $g_1, ..., g_n : B \to \mathbb{N}$), is a function $f : \mathbb{N}^n \to \mathbb{N}$ given $x_1, ..., x_n \in \mathbb{N}$, $f(x_1..., x_n)$ is the maximum number of tape spaces used given any input $b \in B$, such that, $g_1(b) = x_1, ..., g_n(b) = x_n$.

4.8 Example

- Using the same code as above, which executes a program which returns true or false depending on whether the input is a palindrome or not:

  ```
  def isPalindrome(s: String): Boolean = {
  s == s.reverse
  }
  ```

  The space complexity for this code is O(N), as the program has to store the reversed input which is of length, N.

4.9 Remark

In practice, we do not need exact proofs of space complexity, however estimates of it are useful.

4.10 Definition - "Kolmogorov - Chaitin Complexity"

Given $x \in \mathbb{B}$, and a pre-chosen $\mathscr{L}$-Programming Language $< \mathscr{L}, c, p >$, we say the "minimal description w.r.t P" of x written $d(x)$ is the shortest $\mathscr{L}$-program P such that $c(P) = M$ where TM M on the empty binary string as input halts with x on its tape, then the "Kolmogrov Complexity w.r.t $\mathscr{L}$" of x written $k(x)$ is $|d(x)|$.

4.10.1 Remark

In practice we can fix $\mathscr{L}$ depending on some context. Note that k is not a computable function, so we cannot calculate its value, we can only informally intuit its value or use it only analytically. Furthermore this definition applies to data, since $x \in \mathbb{B}$. We introduce this definition for historical completeness but will only use it to introduce a variation that we can use to apply to code.

4.10.2 Remark - What is information?

We can say this is a function on "data" directly that outputs some positive quantity of "information". Suppose we are only interested in the distribution of the data, that is we wish to measure the "information" in a probability distribution (probability will be defined formally later). This latter notion of information is usually called "Information Theory" and uses two functions: Entropy and Kullback-Leisler Divergence, which won't be covered in this paper - typical applications include Machine Learning.

4.11 Definition - Can Compile (Programs Representing Compilers)

Given Programming Languages $< \mathscr{L}, c_L, p_L >$ and $< \mathscr{M}, c_M, p_M >$, we say an $\mathscr{M}$-Program m "Can Compile" an $\mathscr{L}$-Program P if and only if $R(c_M(m))(p_L{-}1(P)) = c_L(P)$. I.e. the program m parses and compiles the program-binary for P outputting a TM that is the same as the TM that would be output if we compiled P directly using c_L.

4.12 Remarks

Up until this point we have been treating "compilers" as purely platonic mathematical functions with no canonical way to physically represent them in the context of computational theory. With this definition we can now think of compilers as programs. It's useful to refer to $< \mathscr{M}, c_M, p_M >$ above as a "meta language w.r.t. $< \mathscr{L}, c_L, p_L >$".

4.12.1 Bootstrapping

Historically as programming languages have progressed it's necessary to use older languages to "bootstrap" compilers to invent new languages. The most fundamental and primary language is Machine Code, say $\mathscr{L}_0$, and this has the property that program binaries are isomorphic to their programs. Next is Assembly Language, say $\mathscr{L}_1$, and the first compiler for Assembly Language would have had to be written in Machine Code. Some time later the C programming language was invented, and it's first compiler would have been written in some prior language, and so on until we get to Scala.

Note that once the first compiler written in $\mathscr{L}_n$ for a language $\mathscr{L}_{n+1}$ is compiled, we can then write another compiler for $\mathscr{L}_{n+1}$ *in the same language* $\mathscr{L}_{n+1}$. This is generally the case, and it's true for Scala - i.e. the latest Scala compiler is written in Scala. Of course if language features are added to a language we have to bootstrap again before we can use those language features in the implementation of the compiler.

4.13 Definition - Set Of Sufficient Compilers

Given Programming Languages $< \mathscr{L}, c_L, p_L >$ and $< \mathscr{M}, c_M, p_M >$, and given a $\mathscr{L}$-Program P, the "Set Of Sufficient Compilers for P w.r.t the meta-language $< \mathscr{M}, c_M, p_M >$" is the set of $\mathscr{M}$-programs that can compile P and any Q where Q uses the same or a subset of Language Features of P.

4.14 Remark

Intuitively speaking the Set of Sufficient Compilers for a P can contain compilers that can ignore the language features not used by P, and so will typically contain "simpler" compilers than compilers which can compile arbitrary $\mathscr{L}$-Programs.

4.15 Definition - Compile Complexity (AKA Language Feature Complexity)

Given a Programming Language $< \mathscr{L}, c_L, p_L >$ we define the Compile Complexity of an $\mathscr{L}$-Program P w.r.t a Programming Language $< \mathscr{M}, c_M, p_M >$, which we call the "meta-language", by

$$\mathscr{C}(P) := \min_{m \in \mathbb{M}_P} |m|$$

where $\mathbb{M}_P$ is the Set Of Sufficient Compilers for P.

4.16 Remarks

Here intuitively the shortest m from the Set Of Sufficient Compilers for P is the "simplest possible compiler that can handle the features in program P", and that the Formula Complexity of P *is a numeric quantity* of the complexity of those features.

Here at SomeCompany, we always use Unix/Linux based Operating Systems, and almost all of the Unix/Linux kernel is written in C. This justifies picking M to be the C programming language. Furthermore as per the Bootstrapping remarks, C was the latest language used common to the bootstrapping chains for all programming languages used at SomeCompany (like R, Python, Scala, TypeScript, etc) - that is to say it's the "Greatest Common Divisor" in our context.

Put another way, in order to understand the meaning of all language features in all the languages we use one only need to understand C *and no earlier language*, since C was at some point used to construct compilers for these languages. We don't want to go back further than C, since C is also the most sophisticated ("greatest") language. We want to pick a meta-language that is maximally sophisticated while also common to all used language's bootstrap chains.

In principle the meta-language choice can change as we change what languages we use, or if language creators decided to fork their bootstrapping chains and use some language other than C (like Assembly). In practice this is unlikely to ever happen, if it did happen it would probably happen in conjunction with some major revolution in Computer Science - perhaps an entirely new Operating System. Even then, it's still likely that future technology will in some way rely on or use C.

It also happens to be that C is a very nice programming language because it's very close to Assembly, yet provides structured programming constructs. I.e. with experience, it's possible to read C programs and know almost exactly what computational steps will be taken by the computer. There are other languages which have this property also though, like Go and Rust.

4.17 Remark - Real Life Example

Note Google got so fed up with abuse of language features i.e. using more advanced features than necessary, they invented Go which has few features. This is because they have approximately 20,000 developers and they need consistencies in style so that any developer can understand any part of the code. (we will more formally clarify "feature abuse" in later sections using Big O Notation).

4.18 Remark - Practical Application

When comparing programs to find simpler programs w.r.t. Compile Complexity we don't actually calculate $\mathscr{C}$ exactly, rather we can intuit the complexity of language features used by a program by considering the length of fully self-contained natural language definitions of the language features. Put simply (pun intended), when a short straightforward definition of a language feature cannot be delivered, it's likely that language feature is a candidate for abuse.

Another simple practical method to identify feature abuse is to heavily scrutinise changes to programs that introduce a previously unused language feature relative to changes that merely re-use features already introduced. We will return to this with more formality later.

5 Program Correctness

See https://en.wikipedia.org/wiki/Measure_(mathematics) and https://en.wikipedia.org/wiki/%CE%A3-algebra and https://en.wik
theoretic_probability_theory

5.1 Probability

Given any set Ω, called the "Sample Space", and a σ-algebra F on Ω, a function $P : F \to [0, 1]$ is a "Probability Measure" (or "Probability Distribution" or just "Probability") if and only if P is a measure and $P(\Omega) = 1$ (i.e. it has "Unit Measure").

5.2 Notation - Power Set

Let $\mathbb{P}(S)$ be the power set of S.

5.3 Program Accuracy

Let $p : \mathbb{P}(\mathbb{B}) \to [0, 1]$ be a probability distribution, let $A : \mathbb{B} \to \mathbb{B}$, which we will call "Actual Desired Program Function", then the "Accuracy of a Program" Q in some programming language $< \mathscr{L}, c, p >$ is the probability that $R(c(Q))(X) = A(X)$ w.r.t p written $a_{pA}(Q)$ where X is a random variable from $\mathbb{B}$. I.e. it's the probability that the program outputs the desired result w.r.t. the distribution p and desired behaviour A.

5.3.1 Remark - Business Analysts and the Product Owner

In practice p and A are not always fully known. Usually only a small portion of possible inputs from $\mathbb{B}$ have non-zero probability. It's usually up to Business Analysts and the Product Owner to specify both A, and sometimes p. A scientific technique to determine p is to let a program run for some interval of time, and record it's inputs, this can then be used to estimate a probability distribution via usual statistical means.

5.4 Definition - Bug

Notation as above, a bug is an input $x \in \mathbb{B}$ such that $R(c(Q))(x) \neq A(x)$.

5.5 Definition - Bug Density

There are two useful definitions.

- (By input) The proportion of inputs that are bugs
- (By program length) The number of bugs over the Formula Complexity

5.5.1 Remark

Typically the former is more meaningful, especially when combined with a utility function and probability distribution, since it doesn't really matter if a program has a lot of bugs on inputs that either have little business value, or are improbable. We'll return to this later.

5.6 Important Remark - Determinism

So far programs have implicitly been deterministic, that is $R(c(P))$ always gives the same output on the same input. In practice, if the input of a program depends on time, or a program is actually multiple programs running in parallel (and the time it takes to perform a single step of execution is determined by something unknown (so practically speaking it's non-deterministic)) then real life program behaviour can be non-deterministic, i.e. it's possible that $R(c(P))(x_1) \neq R(c(P))(x_2)$ for inputs x_1 and x_2 where x_1 and x_2 only practically differ by some time component.

When a program is highly non-deterministic it makes all of the following sections of this paper somewhat irrelevant. In fact if a program is highly non-deterministic it's almost impossible to reason about the real life costs, accuracy and utility of the program. Non-deterministic programs are extremely risky liabilities to a company because at some point they could suddenly stop working completely as the company scales, and by this point, it may be difficult to impossible to debug it.

Therefore henceforth we will continue to assume programs are implemented to be largely non-deterministic, and in practice if a program is non-deterministic then the absolute first priority should be to make it deterministic. Making a non-deterministic program deterministic is usually easy via a number of techniques:

- Fixing or parameterising seeds for Random Number generation rather than using Time.
- Use Functional Programming for Concurrency and Parallel execution. Functional Programming aims to eliminate "side effects" from programs, which basically means the input of the program is not influenced by time.

5.7 Program Utility

A function of the form $u : \mathbb{P}(\mathbb{B}) \rightarrow [0, 1]$ with $u(\mathbb{B}) = 1$.

5.7.1 Remark

This function is supposed to denote the relative financial importance of potential inputs to the program. Some inputs to a program are more important to the business than others. It is the job of the Product Owner to specify u.

5.8 Expected Revenue Generated

The "expected revenue generated" by a program P, we'll denote as $\mathscr{R}(P)$ is

$$c \sum_{\{x \in \mathbb{B} | A(x) = R(c(P))(x)\}} u(\{x\})$$

where c is some constant. In other words, the revenue is the sum over all possible inputs of the utility value for inputs where the program is correct.

6 Maximising Profit From Programs

We will introduce some functions that each aim to correspond to real life costs of software engineering.

6.1 Cost Functions

Given a program P in some programming language.

6.1.1 Human Resource Cost Functions

- **Raw Read Cost** Let $\pounds_f : \mathbb{N} \to \mathbb{N}$ be the cost of developers time to read program P given $|P|$ as the input to $\pounds_f$ in order to add new features, fix bugs and general maintenance of the program.
- **Feature Complexity Cost** Let $\pounds_{\mathscr{C}} : \mathbb{N} \to \mathbb{N}$ be the cost of developers time to learn the features used by the program sufficient to understand the program given $\mathscr{C}(P)$ as the input to $\pounds_{\mathscr{C}}$ in order to add new features, fix bugs and general maintenance of the program.

6.1.2 Computational Resource Cost Functions

- **CPU Cost** Let $\pounds_T : \mathbb{N} \to \mathbb{N}$ be the cost of CPU to execute program P given the average number of compute steps of the program weighted by the probability distribution of the inputs.
- **Storage And Memory Cost** Similarly $\pounds_S$ for space used.

6.2 The Cost Minimisation Heuristic

In general in most companies it's true that

$$O(\pounds_f) > O(\pounds_{\mathscr{C}}) > O(\pounds_T) > O(\pounds_S)$$

Henceforth we will slightly abuse notation and assume it's implicitly obvious what is meant by $\pounds_x(P)$ for each x as above for some program P.

6.2.1 Justification

- **Humans cost more than computers** Except for companies dealing with Big Data or very high-frequency transactions, in 2022 generally the cost of paying developers is much greater than the cost of hardware. This justifies the middle inequality.
- **Learning a codebase costs more than learning languages** Except for very esoteric, academic or badly designed languages, learning a programming language and it's features is a relatively small one-time-cost. Nevertheless it isn't entirely a constant function because programming languages can change and grow (C++ and Scala are good examples), new developers from different backgrounds join the team, and in practice developers are unlikely to learn and remember all the language features for feature-rich languages like Scala. Therefore developers typically learn language features as-and-when necessary in order to understand a codebase - it's a cost that changes as a program changes. Now by comparison, learning a codebase always increases as the codebase increases. The bottom line is that it's fair to say (and we are confident no developer will disagree) that the time it takes for developers to learn sufficiently many language features to understand a codebase is significantly less than the time to read and understand the codebase. This justifies the first inequality.
- **CPU costs more than RAM and Storage** This is just a known fact in computing that commodity RAM and storage is relatively cheaper than CPU, and also that it continues to get cheaper because CPUs are becoming increasingly harder to optimise as we are reaching the physical limitations set by the minimum possible size of a transistor switch. It's easy to add RAM and storage almost indefinitely to the point where most applications can cheaply buy more RAM and storage they can ever dream to consume. This justifies the last inequality.

6.3 The Profit Maximisation Heuristic

Maximising profit amounts to maximising:

$$\mathscr{R}(P) - (\pounds_f(P) + \pounds_{\mathscr{C}}(P) + \pounds_T(P) + \pounds_S(P))$$

6.3.1 Justification

Note that we do not need to explicitly include a "Write Cost" component, i.e. the time it actually takes to manipulate characters in a program to add a feature or change a program. This is because

- **Implicit Coverage** The other components of the above expression cover Write Cost implicitly, higher complexity classes imply higher Write Cost. The utility function covers weighing the value of the feature to be written.
- **Typing is easy** typing is just a negligible cost, the time to write code is almost entirely covered by the time taken to understand the existing code and time taken to understand language features
- **Codebases grow** the time to write code decreases exponentially as a proportion of the time to read and understand code as a codebase increases in size. It's very common that a developer can spend days or even weeks to implement a feature or fix a bug that only requires a single line of code, or just a handful of lines of code. It should be obvious that the time to type one line of code hasn't been spread out over the days/weeks, rather the developer spends days/weeks understanding the problem, usually by reading existing code, then spends 1 minute typing the actual changes.

6.4 Definitions - Inherited Complexity VS Native Complexity

In practice it's often the case that a large amount of the code for a program is written by third parties in the form of libraries and frameworks. So it's useful to consider:

- **Inherited Complexity** The length of the subsequence of P where this subsequence is exactly the third party code
- **Native Complexity** The length of the subsequence of P where this subsequence is exactly not the third party code

We can then similarly define $\pounds_{fI}$ and $\pounds_{fN}$ respectively.

6.5 Remark - Inherited Complexity Exclusion Fallacy

When heuristically (or formally) aiming to measure the Formula Complexity of a program if Inherited Complexity is omitted this is the "Inherited Complexity Exclusion Fallacy". For many programs the vast majority of the complexity is in the Inherited Complexity, and so developers could falsely believe that their program is "simple" because they have omitted Inherited Complexity from their calculations.

For example a developer may say "we can replace these 100 lines of code with one 1 line of code to call framework X", but the implementation in framework X may be a billion lines of completely impenetrable buggy code. What they really mean to say is "we can replace these 100 lines of code *we authored* with some code *we did not author*."

There are many problems this creates, to highlight but a few:

- **Unknown bugs** The program has many bugs that appear unexpectedly because those bugs are actually bugs in the third party code
- **Unknown behaviour** Even when the third party code is bug free it can be used erroneously because either the documentation is poor, or the developers interpretation of the behaviour is incorrect. This is especially true when developers do not read the third party code to properly understand what it **really** does. Developers can assume certain behaviours of third party code because that's what the documentation says (or what they think it says).
- **Language indirection** Similarly, when developers rely on documentation, written in Natural Language, instead of reading the third party code this introduces huge indirection and inaccuracy of understanding since Natural Language is extremely vague by comparison to a programming language.

6.6 Remark - Cognitive Bias's

6.6.1 Imposter Syndrome

"Imposter Syndrome" is the implicit assumption that:

$$O(\pounds_{fI}) < O(\pounds_{fN})$$

This assumption is dangerous and often wrong (see also https://en.wikipedia.org/wiki/Impostor_syndrome). Here, the developer assumes they are less competent than the average developer, or at least less competent than the developers that wrote the third party code they come to rely on. It's only under this assumption that it could make any sense to say "we can replace these 100 lines of code *we authored* with some code *we did not author*."

Therefore developers with Imposter Syndrome often outweigh the benefit of using third party libraries and frameworks vs writing the code themselves.

6.6.2 Example - log4j

A common example of Imposter Syndrome in practice is how many companies use a library for logging called log4j. Firstly it should be noted that the interface for log4j is side-effecting, that is it's behaviour is determined by config files (rather than programatic parameters), that in principle can change during runtime execution and thus the program becomes non-deterministic. Recall from the determinism section that when programs are non-deterministic they become extremely difficult to reason about. Furthermore the language of the config files is very poor and error prone as it is a Markup language, not statically typed and not even Turing Complete.

The author has been told that log4j has in the order of 600,000 lines of code (easy to verify on Github), in practice few to zero developers ever actually read all the code that is implicitly inherited via usages of its interface. In fact it's extremely difficult to even know what lines of code from log4j are even used exactly because of the above point regarding its interface being via config files. When an interface is controlled via config files it's not possible to use modern IDEs (program editors) to "step into" the code from the Native Code.

Such vast amounts of inherited complexity massively undermine the developers chances to understand the program in its totality. This has been demonstrated factually by the recent log4j security vulnerability (see https://www.ncsc.gov.uk/information/lo
vulnerability-what-everyone-needs-to-know). Literally 1000s or 10,000s of developers had no idea that log4j had a security vulnerability because of the Imposter Syndrome cognitive bias and the Inherited Complexity Exclusion Fallacy.

By comparison at SomeCompany, our core logging library is basically about 150 lines of code, with a further few 100 lines of code to manage log backups, aggregation and rotation. Of course with such a small codebase it's very easy to be sure our logging doesn't have any bugs, is deterministic, etc, etc.

If we are aware of and correctly use the The Cost Minimisation Heuristic from this paper without committing the Inherited Complexity Exclusion Fallacy then huge costs to companies can be saved.

6.6.3 Example - Terraform

A common "industry practice" in 2022 is to use templating Markup languages (often based on the now fashionable YAML), like Terraform, for infrastructure code, rather than using the same language as the business logic via Cloud APIs. We would hazard a bet that few people who use Terraform have ever actually read the Terraform code and thus rely heavily on its documentation as to its actual behaviour. Although generally this works fine it creates a number of problems, many of which extend beyond the scope of this paper (most notably being lack of Turing Completeness, Type Safety, Connascence handling, Unit Testability and Dev and Ops silos of developers that cannot work across domains due to the difference in languages), but for now we just observe that Terraform incurs a large Inherited Complexity often not captured or considered by developers.

6.7 Remark - Exceptions To The Above Reasoning

Sometimes in order to implement a feature or fix a bug would require writing a large amount of (difficult) code if a third party library was not used. Logging and infrastructure code is obviously not difficult to write and obviously not very much code, so it certainly is not an example. In fact by all 4 measures of complexity, logging and infrastructure code is the simplest part of a codebase relative to say business logic and algorithms.

When the problem space is highly theoretical, abstract **and** domain agnostic these are candidates for use of third party code.

There are classes of examples that fit this criteria fairly unambiguously: Parsers, Compilers, Protocols, Parallel Computing and Concurrency, Domain Agnostic Algorithms from Computer Science (e.g. Sorting algorithms, Hashing algorithms, etc). In other words, if a problem is highly algorithmic, it's likely a good case for using a third party library or framework. Conversely if a problem is not algorithmic, but more about run of the mill integration, ops, etc, then a native implementation is usually better - e.g. logging, infrastructure, CI/CD, deployment, alerting and monitoring, orchestration, etc, and to some degree even also unit testing abstractions (i.e. avoid heavy use of unit testing frameworks, mocking libraries, etc).

Another fair heustric is to consider the minimum amount of code to implement the capability, if this is relatively large (like 1000s or 10,000s of lines of code) then third party code should be *considered*. Nevertheless sometimes what off-the-shelf already available third party code exists may be sufficiently poor that you need to write 1000s of lines of code for a home grown solution in order to meet quality assurances.

If it's relatively small, like 100s of lines of code, like that of logging or most infrastructure, then it's probably best to avoid third party libraries or frameworks.

6.7.1 Example 1 - Spark for Distributed Computing

At SomeCompany we use Spark for distributed computing. The most notable algorithmic concerns that are provided by the framework include: Shuffle Algorithm and distribution of processing across multiple cores and machines. We currently also leverage Spark for reading and writing Parquet, but it has since been recognised that using a lower level library would be more flexible.

We do **not** use the SQL or Datasets/Dataframes APIs for Spark as these provide little to no algorithmic code that cannot be written using the lower level RDD API with a better design - particularly a Functional design. The only real advantage to these APIs is serialisation, but serialisation is buried so deep into the APIs it's not possible to leverage this feature without being exposed to many design problems and abstraction leaks.

So although we use Spark, we only use the parts of Spark that are *actually* difficult to implement ourselves in full - i.e. Shuffle Algorithms and distribution of processing. All the bell's and whistles and supposed optimisers tend to get in the way rather than help, especially for a team of developers with extensive experience with data processing and functional programming.

6.7.2 Example 2 - Circe for JSON Parsing

At SomeCompany we use Circe for parsing JSON.

6.7.3 Domain Agnosticism

Infrastructure code can sometimes require > 100s of lines of code, but, contrary to popular belief, it isn't domain agnostic code. There are often many forms of connascence between business logic and infrastructure, for example: amount of compute resources, locations of storage mechanisms (topics, databases, tables, buckets, etc). Examples of this non-agnosticism often manifest when you find yourself modifying some codebase in a normal programming language for the business logic, then you have to modify the infrastructure code in some other language in some compatible way (like adding a server, changing a database name, or changing an algorithm to require greater compute resources). This connascence is Strong connascence and thus makes bugs and mistakes likely.

This kind of problem becomes especially bad when the infrastructure code is maintained by another team (because typically most developers of ordinary languages don't like learning tools like Terraform, so prefer "Platform Engineers" to do this work, and visa versa). In order to implement a feature two teams and two languages must be involved. **Even worse** than even this, is when the codebases are stored in different repositories so it's not possible to easily synchronise releases or version control history.

Note that Shuffle Algorithms and JSON parsing by comparison are genuinely domain agnostic. Shuffling a dataset to say sort or join it is going to be largely the same for any domain with a non-trivial dataset size. And JSON parsing never changes from domain to domain at all in any way.

6.8 Not Invented Here

It's sometimes the case (less often than imposter syndrome) that developers have an implicit assumption that:

$$O(£_{fI}) > O(£_{fN})$$

I.e. the developer believes that if the code is "Not Invented Here" then it's bad code.

6.8.1 Remarks

Generally "Not Invented Here" can be a problem when the developers estimation of their own ability is inaccurate, that is they are not actually much better than the average but believe they are. This can result in what are called "Home Grown Frameworks", that new developers to the codebase will not like because the new developers will have to learn this home grown framework while they may already know some publicly available third party framework. The fact that the developer has to learn a new framework shouldn't be the main concern, the main concern should be whether or not the home grown framework actually does a better job than the third party frameworks, such as: what is it's Formula Complexity and how does it impact Program Accuracy.

Humans generally have a cognitive bias towards anything they already know, and so it's more common that developers meta-falliciously claim "Not Invented Here" fallacy is present ahead of fully and properly appreciating the relative strengths and weaknesses of the in-house framework.

The crux here is that developers must avoid acting on cognitive biases by introspection and rational thought processes (recommend the book “Thinking Fast and Slow”). Jumping to meta-cognitive biases and dismissing other developers design decisions as cognitive biases can be just as damaging to productivity as the cognitive biases themselves.

6.8.2 A Reasonable Rule of Thumb

When evaluating whether a third party library or framework is a good idea, the following rough rule works quite well: If the amount of time it takes to read the documentation **and live code** of a library or framework is about the same or more amount of time it would take to implement a “home grown” approach then generally the “home grown” approach will be better suited to the problem you are trying to solve and induces less overall complexity.

As noted above for badly designed frameworks that rely on Markup or Config languages for their interface, it’s non-trivial to determine what exactly the live parts of the code will actually be. The time it takes to figure this out should be included in your measurement. (Though we would say if a framework doesn’t have programmatic Turing Complete Type Safe and preferably Functional client APIs, then this alone is grounds for dismissing the framework as poorly designed.)

Of course if you already know a library/framework then the time will be less, but this is subjectively so. To recover some objectivity you’ll have to try to remember the amount of time it took to learn the library/framework in the first instance.

Observe that this rule of thumb will be applied differently depending on the experience of the developer with programming languages. Developers that are very comfortable writing code (especially in well designed languages) will thus generally prefer to write their own utilities rather than relying on third party code, **especially if the third party code is written in an inferior language**.

Note that software engineering is not yet a multi-generational discipline when say compared to masonry, civil engineerings, mechanical engineering, etc. In fact Robert C Martin reports that the number of software developers has roughly doubled every 5 years. This means that 50 % of all software developers at any given time have less than 5 years of experience. This makes software engineering a highly unusual discipline. It’s then no surprise that the behaviours of inexperienced programmers becomes a defacto standard, and thus Imposter Syndrome being a more prevalent bias than Not Invented Here.

7 Practically Applying The Optimisation Heuristics

We have hopefully already sufficiently covered the notion of Inherited Complexity and the cost of neglecting its impact. Now we will consider each component of the formula in The Profit Maximisation Heuristic.

The inequalities in The Cost Minimisation Heuristic are often easily applied when “all else is considered equal”. That is attention to the larger sides of the inequalities will generally give better results than attention to the smaller sides. Nevertheless since the inequalities are actually Big O Notational, it’s not *always* true, particularly when “all else cannot be considered equal” and when large jumps in complexity classes occur.

Therefore the pernicious and taxing problem of engineering optimisation is managing **tradeoffs** and continuously returning to first principles & primary definitions. It’s much easier, but of no use, to justify a particular engineering, design or architectural choice with a generic slogan unrelated to the domain, such as “this is industry practice”, “everyone else does this”, “this is best/standard practice”, etc. Here we would be neglecting tradeoffs, which nearly always apply specifically and narrowly to the actual domain we are working in. What works for Google or Facebook, may not work for a regular enterprise, and what works in one industry, like Finance, Insurance, Retail, Banking, AdTech may not work in the other. Furthermore each domain has multiple subdomains.

7.1 Creative Discipline - No Free Lunch Theorem

It is a mathematical fact, which has been formalised in various forms (e.g. see “No Free Lunch Theorem”), that solving a specific problem with a specific solution will perform better than a generic solution.

Therefore creativity is not only a major satisfaction to be experienced when engineering, but we must be disciplined in always exercising creativity. Here by Creative Discipline we mean always returning to the first principles and primary definitions in this paper.

7.2 Creative Suppression

Unfortunately the problems of infrastructure management highlighted in earlier sections (multiple codebases, strong connascence, etc) arise exactly because arguments like “this is industry practice”, “everyone else does this” are recited verbatim rather than focusing on first principles and primary definitions, such as by addressing each specific requirement in turn.

This mentality is extremely dangerous not just in programming but in society in general. The obvious and direct relationship is that society is more and more being run by software. Software is flying our planes, running our banks, modelling pandemics, climate change and economics, etc, etc. A widely believed but principally wrong view prevalent in software engineering is likely to have catastrophic consequences in the near future, perhaps subtly so.

7.2.1 The Alan Turing Irony

For example it’s estimated that Alan Turing’s creativity in mathematics and engineering, by essentially inventing the computer, to assist in codebreaking in the second world war reduced the length of the war by 2 to 3 years saving 14 to 21 millions lives. There is a multi-faceted irony here:

- Alan Turing solved the problem of codebreaking in a completely unseen way by returning to first principles and mathematical precision, which was contrary to the standard approaches at the time.
- The problem arose because there existed an entire society under which beliefs and ideas contrary to the standard in that society were suppressed, i.e. Nazi Germany.
- Alan Turing’s work was completely secret and unknown to the general population at that time - his work was not immediately obvious, like that of a pilot or soldier.
- Alan Turing was later persecuted to the point of suicide by the society and people he worked to protect.
- The persecution could have been avoided if every individual in the society actively engaged in creative discipline by always questioning existing *ethical and legal* systems and conventions

Therefore we observe an *indirect* relationship between creative discipline in programming and in society, that the mental process of questioning an engineering approach according to first principles is isomorphic, i.e. essentially the same as, the process w.r.t. laws and ethics. A society that lacks innovation in one area will likely lack innovation in the other.

For some light relief and entertainment when you are not fixated on reading this paper, we recommend the films “The Imitation Game”, a fact based biography of Alan Turing, and “Jojo Rabbit” a fictional black comedy about a Nazi boy in Nazi Germany.

Anyway, in later sections we will explore in length the common tradeoffs encountered when trying to apply The Cost Minimisation Heuristic.

7.3 Competitive Advantage

The biggest flaw in the "everyone else does this" type reasoning is that **By Definition** all private companies in a healthy capitalist society make money by having some competitive advantage, which **trivially** implies that for a private company to be successful it must be **different** to it's competitors in at least one way[1]. The same analogy can be seen in evolutionary theory; species are always changing and adapting to their environment, never copying exactly even their immediate ancestors. Any species that copies exactly its ancestors necessarily goes extinct at some point in the future.

Similarly we can analyse this reasoning with some trivial statistics. If you are trying to be "like everyone else" then you will necessarily coverage upon the global average competency. Therefore the best you can possibly achieve is just average. Any mistake along the way, which let's assume is inevitable, means you have probability 1 that you will be below average. Is that really a logical optimisation strategy?

7.4 Technology Age and Open Source

Another fallacy here is automatically assuming the newest and latest technologies are the best. Software Engineering and Data Science are littered with Cargo cults, bandwagons and fanboy-ism (or fangirl-ism). The age of a technology is independent of it's quality of design and implementation, the only advantage that newer technologies have is that they should in principle learn from the mistakes of older technologies. Unfortunately this is not true as many technologies are invented either in pure ignorance of previous technologies, or invented for financial gains (that have a Conflict of Interest with the financial gains of *your* company). This is most notably the case when a technology is not a truly Open Source project, by "truly" here, we mean that the source code is freely available, and also there does not exist any company where it's sole business model is to support the code.

When companies exist solely to support software it isn't necessarily in their best interests to make the software the best it can be for a number of reasons:

- **Vendor Lock In** Software makes it difficult to integrate with other software, because other software may have functional overlap and thus be competition for the creators.
- **User Bloat** Software is designed to require many users as their commercial model depends on the number of users.
- **Hardware Bloat** Similarly with hardware (e.g. vendors charge based on the number of nodes used)
- **Intentional Obsolescence and Limited Lifespan** Software is designed to stop working, or to just generally be crap, so that customers must pay for upgrades and updates in future thus generating a recurring revenue stream.
- **Competition in a Parallel Industry** A company may operate in multiple industries, some if which may overlap with your industry. It might be more profitable for a company to impede you with bad software so that it can outcompete in some other industry while internally using completely different software. It's been alleged that tech giant companies employ this strategy offering inferior Software As A Service to consumers as to what they use internally, but we are not aware of any direct evidence of this.

7.4.1 Examples

Version control systems, like git and mercurial, operating systems like Linux, and Shells, like Bash / Unix Terminal are all examples of quite old technologies that have not been improved upon by newer technologies. Usually attempts to "improve" upon these come in the form of modern Guided User Interfaces (GUIs), that is non-terminal based interaction. GUIs are vastly inferior to Command Line Interfaces in almost all ways except a small class of visualisation problems but they are very easy to sell because they have pretty colours and shapes. Therefore financially motivated organisations create technologies with shiny GUIs.

[1]Except when either corrupt or incompetent governments privatise organisations that automatically have a monopoly, like public transport companies in the UK

7.5 Flexibility

We can say a tool is flexible if and only if it is Turing Complete https://en.wikipedia.org/wiki/Turing_completeness.

Almost all programming languages are Turing Complete, while Markup languages are generally not Turing Complete (LaTeX being an exception!). GUIs are almost never Turing complete, some nice exceptions are the video games Factorio, Minecraft and Dwarf Fortress.

7.6 Composability

We can say a tool has high composability if and only if it has a well defined (and preferably narrow) Application Programmers Interface (API).

7.6.1 Examples

Good examples

- Unix commands have a very simple interface: **Inputs:** one Array of Strings, one input streams of bytes, **Output:** two streams of bytes as output (usually printable ASCII characters), and one special byte as an "exit code".
- RESTful HTTP APIs used in Web Development. Although HTTP is an ugly protocol for back end communication, and usually is only used insofar as to wrap some JSON, it has become quite fashionable to wrap applications/tools in HTTP interfaces. When a HTTP interface is also RESTful, this makes it even more composable (see https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm).
- Programming client libraries are fairly composable by definition, especially provided they do **not** employ Inversion of Control.
- Functional Programming libraries are by far the most composable software tool that currently exists.
- Modern Document Databases allow the storage of data in arbitrary formats and provide many client libraries and RESTful HTTP APIs

Bad examples

- SQL Database technologies heavily constrain the structure of data to almost always flat/un-nested data structures. Therefore composing these technologies with other applications becomes challenging (at least when compared to Document Databases).

7.6.2 Longevity in Software Choices

Software choices that are likely to survive the test of time will be highly Flexible and Composable, and usually they "do one thing and do it well" (unix philosophy). The more composable some software is, the less likely it will be that it needs to be completely removed from a technology stack because it can compose with newer parts of the stack. Similarly highly flexible software is unlikely to encounter future requirements that it cannot meet. We explore some examples:

1. **No-Code / Low-Code VS As-Code**: A lot of vendors peddle software that doesn't require the user to input any code (it's just a GUI), inevitably these solutions at some point encounter a problem they cannot solve. The customer is then stuck, they cannot seamlessly compose their existing software with others, nor can they meet the new requirement with the existing software. By contrast, companies that insist on all software being "as-code" (ideally real turing complete programming languages[2]) can satisfy more new requirements and usually can integrate new technology into their stack.
2. **Bash / Shell Scripting VS Ops Frameworks** Bash is a highly flexible and composable operational tool, and although invented 33 years ago (1989), was designed with such powerful flexibility and composability that it can still do everything any Ops Framework can do today. Furthermore modern Ops Frameworks (notably CI/CD tools employing YAML interfaces) lack flexibility. It's no surprise then that CI/CD tools are constantly being replaced and re-invented, the fashion used to be Jenkins, then it became Circle CI, then Cloud providers started giving tools like AWS CodeBuild, now even Github has given Github Actions. Since every tool is inflexible and poorly composable, it's inevitable that they get superceded by something newer and slightly more shiny. To avoid these CI/CD tools getting in the way, one should only use them as a means to run a bash script, and all the important logic should exist in the bash script, not within some YAML configuration monstrosity of the tool.

[2]A compromise can be configuration as-code (config files or better via APIs) or infrastructure-as-code

3. **Bash / Shell Scripting VS Microsoft Windows** Similarly Windows has always had terrible operational flexibility due to reliance of proprietary huge monolithic applications. They tried inventing their own shell closer to Bash relatively recently in the form of PowerShell, but it was quickly realised that the design was terrible. Eventually Microsoft gave up trying to out-do Bash and in Windows 10 Bash is now kind of an option (but still not as good as a native Unix OS).

4. **S3 or Kafka as a Storage Mechanism VS Databases/DataWarehouses** Too big to get into in this paper, but worth noting that more and more companies are recognising that it's better to store data in S3 or Kafka and only use classical Database/Datawarehouse technologies to provide views and ODBC interfaces over the top.

7.6.3 Easily Reversible Decisions vs Architectural Dead Ends

Points 2 and 4 above are examples of easily reversible decisions that take for form of using a stable and highly flexible underlying technology (e.g. Bash or S3) and then use the plethora of unstable inflexible shiny stuff as a thin layer on top. This means we can easily swap out technologies as shinier technologies are released, so we avoid vendor lock in.

Here at SomeCompany, a good example of leveraging S3 is that this enabled us to switch our SQL Query Engine from AWS Athena to SnowFlake because at the time SnowFlake was considered a better suited technology for our uses cases. It's exactly because we used S3 for the storage mechanism, say instead of a database or datawarehouse, that this migration was easy. If we didn't, we would likely have had a long drawn out data-migration exercise, or worse, an incomplete migration leading to duplication and legacy in the architecture.

Similarly we have changed CI/CD tooling three times in less than three years, each time this has been a fairly trivial thing to do because we just point the tool at a bash script that handles everything (note that the bash script actually calls a custom built ops tool written in Scala, and so we don't have complex logic written in bash).

To avoid Architectural Dead Ends the things to watch out for are:

- Imposter Syndrome
- Slogan and verbatim arbitrary fashions like "bash scripts are bad"
- Short sighted architectural planning, i.e. optimising a tech stack for a 2 year software lifecycle for software expected to exist indefinitely
- Cake cutter lead engineering, i.e. where a team of engineers copies approaches that they have seen work at other companies. This isn't bad in of itself, it's usually sensible to draw on past experience, but it needs to be watched carefully to ensure a new better approach isn't overlooked. Cognitive laziness is the root of cognitive bias; before employing past approaches always spend a few joules of energy thinking about how well the approach meets the requirements *now*.

7.7 Expected Revenue Maximisation

Recall the definitions, revenue is determined by Program Accuracy and Program Utility, so it makes sense to optimise each in turn. Specifying the latter is the job of the Product Owner, so we won't cover that. Accuracy is the job of developers, and here are some common techniques to ensure high program accuracy in rough order of precedence:

- **Unit Testing and TDD** We save the detail for the next sections regarding tradeoffs. We recommend reading up on different approaches to TDD, particularly Chicago style vs London style, Triangulation TDD, strict 3-laws TDD, Red Green Refactor and Extreme Programming. We also recommend reading up on Inversion of Control (IoC), which is a technique that often has to be employed in order to write unit tests (especially in non-functional languages).
- **Minimising Connascence** Bugs introduced as a result of Strong Connascence basically occur when one part of a codebase needs changing in conjunction with other parts simultaneously and it is not easy to detect/determine what those parts of the code are. See also https://en.wikipedia.org/wiki/Connascence. Static Typing, TDD, Simple Design via XP and Functional Programming all reduce Connascence.
- **Static Typing** Too big a subject to get into here, but again we will go into some more detail in the next sections regarding tradeoffs. But in a nutshell, static typing means that common and simple errors get caught by the compiler and get displayed to the user (often seamlesly as part of the IDE). So you need not run the program to find such errors in statically typed languages.

- **Functional Programming (FP)** Again beyond the scope of this paper. As noted in earlier sections FP can minimise non-determinism. Other benefits of FP include: referential transparency, side effect elimination, lower Formula Complexity and lower Compile Complexity (because in FP a lot of language features and design patterns are not needed as they are covered by FP itself). FP tends to increase computational complexity, especially in RAM, but as noted this is usually the least of our worries. Historically FP was not practical because RAM was expensive, this again highlights the importance of re-reviewing practices for the specific problem being solved and not applying a practice generically. Unfortunately OOP is still the most common programming paradigm yet this is best suited for low level mutable state encapsulation for efficient computation on older computers.
- **Alerts and Status Checks** Mini side programs that periodically probe the main program to check it's functioning as expected in either a full production environment or a test/pre-prod environment. These are the "last resort" catch-all fail-safes of ensuring Program Accuracy. They are very important and should exist, but when they are triggered in production it generally implies a failure in one of the above.

Perhaps in the future FP will be superceded by something else, our hope is that some day developers will specify a programs interface using mathematical notion (ideally typeset in LaTeX, not some ASCII based corruption of mathematical notation like that of Hol, Coq, Agda, Idris etc), and then a magical Gödel Machine like compiler running on the cloud will automatically generate an optimal program. Alas, right now (2022), statically typed functional languages are the best we have.

7.8 The Root of All Evil - Premature Abstraction and Optimisation

7.8.1 Definition - Premature Abstraction

"The best architecture decisions are made as late as possible, because the largest amount of information will be available. Maximum information facilitates optimal decisions" - Paraphrased from Robert C Martin.

A Premature Abstraction is when a developer attempts to introduce an abstraction but causes an increase in Formula Complexity (recall, an Obfuscation), or a large increase in Compile Complexity.

As discussed in the remark regarding naming things the most common form of this is introducing a variable, function, class, trait, interface, etc for something that is only used once.

In fact sometimes even if some code repeats itself twice it can still be premature to attempt an abstraction if that code is very short. The cost in Formula Complexity (or Compile Complexity) of introducing the abstraction can still exceed the savings in Formula Complexity from deduplicating the code. Generally large blocks of code can be abstracted when they occur only twice, but for small expressions it can be better to wait for three or even more examples.

That said introducing a name for some expression or block of code can be useful for unit testing, which can increase the accuracy of the program. We'll go into this in more detail in later sections.

The worst form of Premature Abstraction is when abstractions are introduced that have *zero* usages, i.e. dead abstractions. These tend to occur as a result of Big Up Front Design (BUFD).

The more premature an abstraction the less likely the abstraction will actually work in the future. It becomes more likely that the abstraction will need to be completely redesigned. A codebase that has a wrong abstraction is much harder to refactor than a codebase with no abstraction in its place, because often the first step in fixing a wrong abstraction is the mechanical process of inlining. Inlining variables and even functions can be done automatically using modern IDEs, but classes and Object Oriented abstractions require a lot of manual fiddling. The most notorious problem with replacing Object Oriented abstractions is the scope-decoupling of data with function.

7.8.2 Evolutionary Design and Extreme Programming

One technique to prevent premature abstraction is Extreme Programming (XP), here the general rule is that you can't introduce code until you have a failing unit test, and you can only refactor (and introduce abstractions) when the unit tests pass. This means you can't introduce an abstraction before you have (usually many) concrete cases for it. Furthermore XP has a lot of literature regarding "Evolutionary Design" and "Simple Design", which is basically an informal description of the formally defined heuristics in this paper by emphasising the importance of minimising complexity.

Recall Naming Things is Hard. The main result of XP techniques is that naming corresponds to only the naming of patterns *in the code* rather than the less objectively helpful imposing of names arising *from the mind and to thoughts.* Put another way, the named patterns in XP, *really exist* as discretisable computational patterns (electrons in a circuit) or at least discretisable platonic patterns. Failures to to minimise complexity generally name patterns that *ethereally* exist as momentary thoughts in a continuous soup of internal sounds and images.

OOP is especially bad for abstraction failures because at it's core it anthropocentricises programming by requiring that every subcomputation of a program is somehow represented as noun - hence *Object* Oriented Programming. This is a shocking approach since programs are all about *Doing* things, i.e. verbs, the only objects/nouns that really appear in programming is data structures. As noted earlier, OOP can help with a class of problems regarding low level optimisations that require handling mutable state, but even this view is being challenged in recent years in the form of the fairly new languages Go and Rust.

The author recommends Rust as the currently best in class programming language for writing low level optimised code.

7.9 Tradeoff Between Revenue with Costs

Engineers must work closely with Product Owner to weigh costs (complexity) of features against their gains (utility). This is highly non-trivial and there is no easy mechanical process to achieve this. The best the industry has is "lightweight processes" again based on XP (like Agile Development Methodology).

Although Agile is not perfect, the commonly used alternative "Waterfall" is considerably worse. Agile achieves better results by breaking down features into the smallest possible "vertical increments" which ensure the shortest possible feedback cycles with respect to production deployment and feedback from application users/consumers.

A good analogy is navigating by frequently checking your current position and location against charts and maps, vs say infrequently checking position and using long plans for navigation. Agile is like finding a pub using GPS and a map built into your phone, while Waterfall is like following a sequence of instructions e.g. "left, second right, walk until you get to the church, take a right at the tree, ..." When following a long sequence of instructions any small mistake early on may only be realised after hours of frustration, and backtracking can be super hard.

7.10 Tradeoff Between Formula Complexity and Compile Complexity

Sometimes a lot of code can be deduplicated and abstracted with the use of a language feature, thus reducing Formula Complexity and increasing Compile Complexity. Usually this is the best approach, but sometimes the reduction in Formula Complexity is relatively small but the language feature is especially complicated, or perhaps a simpler language feature can achieve the same job.

A fairly well known example is Macros, a very powerful language feature. Even though Macros can solve many (if not all) abstraction problems, there nearly always exists a simpler language feature that also solves the problem. It's only when all simpler language features have been exhausted, demonstrably so (i.e. with some cogent argument), **and** there is a lot of code that can be deduplicated, then it can be justifiable to employ macros.

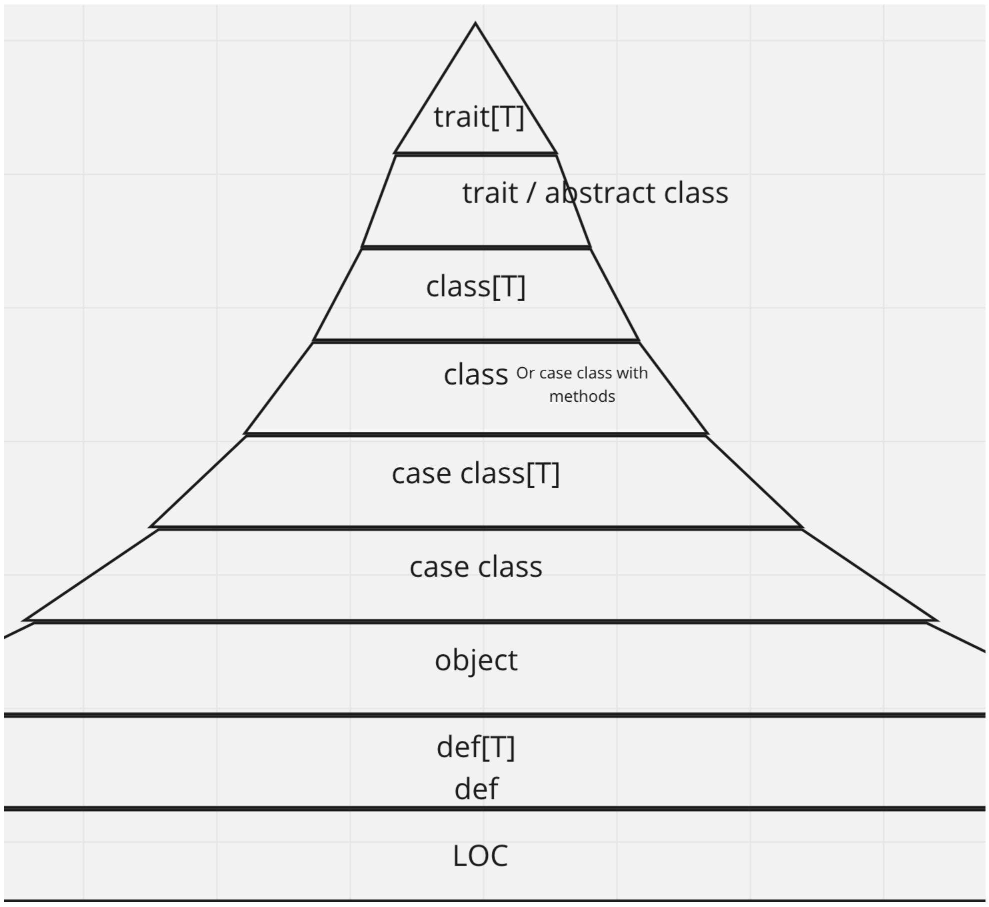
It's perhaps hard for most to determine, even just roughly, the Compile Complexity of a program or specific feature. It's unlikely developers will actually start searching the space of Sufficient Compilers to find the smallest one. Here are some rough tips:

- Language features common to many languages are likely to be simpler, because compiler developers can be assumed to be fairly "lazy" and wish to avoid implementing complex features.

- ... similarly especially older languages, since older languages themselves lacked powerful languages to write their initial compilers. Surprisingly to some, Functional Programming is actually a fairly simple language feature. For example Lisp, is a super simple language with barely any language features (it's even homoiconic) yet Lisp is a functional language and first appeared as far back as 1958!

- Try writing your own compiler as an exercise to get a feel for what it's like to implement language features. *(Having tried this myself I can say with certainty that unconstrained Macros are a very difficult feature to implement, at least for me, as I kept encountering performance issues due to Catastrophic Backtracking - a problem of parser theory relating to Context Sensitive Grammars. In the end I gave up and imposed some minimum terminals to reduce Context Sensitivity. Another example was multiple inheritance (and the related Early Initialisation) for scopes due to the problem of the Deadly Diamond of Death. Scala resolves the Diamond Problem with traits that use right-first depth-first search, which I assure you isn't trivial to implement. Therefore traits are an often abused feature in Scala).*

- For Scala you can use the image below as a rough guide as well as this page https://www.scala-lang.org/old/node/8610, though better to understand **why** this is true by understanding compiler theory rather than to memorise this (lest it become yet another cargo cult). Features at the top of the image should be avoided especially when features lower down can be used instead. We have found that it is fairly easy to write entire applications, without significant boiler-plate or duplication, only using the language features up to and including "case classes".



7.11 Tradeoff Between TDD, IoC, and Formula Complexity

As noted in the Naming Things is Hard section Unit Tests can add great value to a codebase. Recall the section was discussing the impact of extracting expressions or blocks of code to variables or functions that were only used once and so could be inlined, and increase Formula Complexity. Now if we extract an expression or block into a function, *but generally not a variable (at least not local variables)*, it then becomes possible to Unit Test this function. Note we refer to the main program as "main code" and the code for the Unit Tests as "test code", and we generally don't consider test code when calculating complexity classes as it is ignored by the compiler.

This *can (but not necessarily)* have the following benefits:

- **Improve Readability** by providing a representation of the code for developers to **optionally** read in the form of a finite set of concrete examples employing a usually vastly reduced Feature set, and thus vastly lower Compile Complexity. The representation is usually partial, as the finite set of examples does not usually "triangulate" the entire functionality of what is usually an effectively infinite input space.

- **Triangulation** We can roughly define triangulation as a finite set of examples that provide data points on every dimension of the input parameter space such that the simplest function (according to FC and CC WLOG) that satisfies these examples must be the correct implementation. *(For the passionate reader, this is analogous to the Downward Löwenheim–Skolem Theorem (https://en.wikipedia.org/wiki/L%C3%B6wenheim%E2%80%93Skolem_theorem).).* Not only can this greatly improve readability but also ...

- **Accuracy** Unit tests, especially with full triangulation, likely increase Program Accuracy, which in turn will likely increase revenue, and thus profit. Of course it's still possible that the tests themselves are wrong, but we generally like to hope that since tests are based on concrete examples it's less likely than making a mistake in the main code.

- **Property Checking** Some languages, e.g. Scala via ScalaCheck, have Unit Test frameworks for the probabilistic assertion of First Order properties on a function. These have slightly higher Compile Complexity than concrete examples but usually still lower than the main code. These tests can check certain properties hold for a large sample of inputs, and when the check fails the frameworks provide "Shrinkage Algorithms" that auto-magically find the simplest (in cardinality and dimension) counter example for the given property. These tests are especially useful for improving Program Accuracy and readability for particularly algorithmic code where mistakes are likely and full triangulation is intractable/infeasible. Furthermore the Shrinkage Algorithms generation of counter examples often drastically reduce debugging time as the developer is given a concrete simple example showing how a property fails.

- **Reduce Complexity of Main Code** When "Red, Green, Refactor" or Three-Laws TDD is followed with some rigour, particularly for Pure Functions and algorithm writing, an interesting result arises - all four complexity classes are generally fairly optimal and generally better than code written without TDD. That is, programs are shorter, use less features, run faster and run in less time. Robert C Martin demonstrated this by example in a video on YouTube called "The Transformation Priority Premise". A formal proof might be possible using Information Theory or a variation of the aforementioned Downward Löwenheim–Skolem Theorem. Note that this is only generally true when the function being tested is "Pure", otherwise sometimes Unit Testing can have the opposite effect and make main code more complex.
- **Facilitate Refactoring** Sometimes, and only sometimes, when many unit tests exist they make it easier to refactor code because the developer doing the refactoring has some assurances that the code retains it's Program Accuracy - that is to say the refactoring is a true refactoring and doesn't change program function. Usually they make it easier to refactor the function in question, but can have the opposite impact if the signature or entire design of a codebase needs refactoring. When we wish to refactor across function boundaries unit tests can get in the way as we can need to update all the unit tests to the new design.

Note that if a function is only used once in the main code but has Unit Tests in the test code, the difficult job of naming the function only arises to facilitate the Unit Tests. The thought process is "I want to unit test this single block of code, I'll have to think of a name for it so I can call it from test code", rather than "I want to give this single block of code a particular name according to my own mental model".

7.11.1 Test Code vs Main Code Tradeoffs

As noted above under **Reduce Complexity of Main Code** and **Facilitate Refactoring** the opposite of these can happen when writing unit tests. This is something to be especially cautious of when Inversion of Control and "Interfaces" (or classes, abstract classes, traits, etc) are used *in order to make the code unit testable.* For example suppose a function calls an API, it's common to wrap the API in an "interface", and inject the interface at runtime using Dependency Injection to achieve Dynamic Dispatch so that unit tests can inject a "fake", "mock" or "stub" API in place of the real API as the real API may be unavailable or awkward to call at the point in which unit tests run. In fact Unit Tests are often defined in such a way that they must be able to run on a machine without a network connection or forking processes.

An alternative and much safer approach to Inversion of Control (Ioc) and Dependency Injection (DI) is Pure Function Extraction with Integration Testing or End to End testing. Here the developer pulls out as much of the pure functional code from a function into lower level functions and writes unit tests for this logic, then to give some assurance the high level function that wires together these function still works, they write an Integration Test, which calls a real life (but test instance) of the API on a dedicated server (or sometimes even locally).

Integration Tests are generally slower and harder to write, but give greater assurances as to Program Accuracy as they cover the actual dependencies, rather than, a mere interpretation of what the dependencies are. Beyond Integration Tests are Status Checks or End-to-end tests that rely on full environments that aim to simulate a production environment.

When managing this tradeoff generally start with Pure Function Extraction with Integration Testing, then if something makes Integration Testing particularly difficult then consider (a) if the Integration Test really adds much value over and above the unit tests for the extracted pure functions combined with some Status Checks, and so could just be omitted entirely, (b) if using IoC and DI could add significant Program Accuracy without some huge jump in some complexity class.

Unfortunately this tradeoff is overlooked because heuristics are arbitrarily chosen with no mathematical backing tied to company profit. For example, that "a codebase must have 100 % test coverage". 100 % test coverage does not imply 100 % Program Accuracy, and following an arbitrary rule like this divorces the thinking process from the more useful Profit Maximisation Heuristic.

7.12 Tradeoff Between Static Typing and Error Complexity

We can loosely define Error Complexity as the length of the error message (when perfectly self contained). Now as mentioned in previous sections Static Typing can improve Program Accuracy and also developer productivity because the compiler can check for certain classes of errors and mistakes. This is usually the preferred approach, but sometimes too much static typing, especially very advanced static typing (coming from, say, Martin Löf Dependent Type Theory, or Heterogeneous

Collections) can result in very difficult to understand error messages. Sometimes the error messages output by the compiler can be huge. Here is an example from C++, that at a glance appears to use some unbounded polymorphic recursive duck-typing https://codegolf.stackexchange.com/a/1957, the error message is 13507 characters while the actual code is only 100 characters. This example doesn't even use particularly advanced static typing features.

When a type system is used to detect fairly straightforward errors, like providing an empty list to a function that expects non-empty lists, we get fairly straightforward error messages to read. Getting too clever here, say checking that the numbers in a list all occur an even number of times, is possible using advanced static typing (dependent typing) but the error messages may turn out to be total gibberish.

The alternative here is to add what are called "assert conditions", "Asserts" or "require conditions" into your code, usually at the start of functions to check that input parameters are of the required form *at runtime* and they throw an easy to read exception when the unit tests are run. In the world of Python, Ruby, R and other Dynamically Typed languages they have to write these Asserts for very trivial things, like checking a number is an integer or something. Therefore acheiving Program Accuracy in Dynamically Typed languages is a very tedious and repetitive job. Nevertheless employing this technique for non-trivial things can sometimes be a necessary tradeoff because error messages generated by static type-systems are just too difficult to comprehend.

This whole subject of compiler theory and how error messages are generated is a fascinating topic in of itself, and we believe the crux lies with poor internal proof-generation because their proof system isn't a full hybrid Hilbert-Ackermann deductive calculus, i.e. they don't implement full axiom schemas for first-order predicate calculus complete with Zermelo–Fraenkel Set Theory. Instead compilers based on Martin-Löf Dependent Type theory (like Agda, Idris, etc) are based on constructivism, which in the context of computers seems to make sense, since computers are functionally bound to countable, or just finite, sets, all of which ought to be constructible. Nevertheless even if we cannot construct a set in a computer, a system of reasoning that includes it results in significantly more elegant proofs, which could result in significantly more elegant error messages. Alas, we digress.)

7.13 Tradeoff Between Human Cost and Computational Cost

This is one of the easiest tradeoffs to deal with as both are directly measurable. Human cost is simply sum of salaries and computational cost is simply the cost of infrastructure provided by a Cloud or bought directly. When computational cost exceeds that of the salary of a single developer, or let's say 10 % of the sum of salaries, then it's perhaps time to start considering optimising computational cost.

When it's known that the domain will involve some Big Data or High Frequency Transaction processing, it's known a priori that computational cost will become a concern (by the definition of Big Data). It can be known that certain algorithms, that have high Computational Complexity classes, like say $O(x^3)$, $O(x!)$ or $O(2^x)$ will be prohibitively expensive when run on datasets with millions or billions of records. You shouldn't need to benchmark an exponential algorithm to know that $2^{1,000,000}$ computational steps is going to be expensive no matter how cheap the CPUs.

Therefore when implementing algorithms in Big Data we immediately sacrifice Formula Complexity or Compile Complexity to ensure that algorithms generally have Computational Complexity classes less than $O(x\log(x))$ and often aim for $O(x)$. Observe that complexity classes involving components with monotonically decreasing derivatives, like $\log(x)$ or $\sqrt{x}$, these components get progressively more irrelevant the larger the data. Put another way, the rate at which additional hardware must be obtained to run the algorithm decreases as the company grows.

7.13.1 Horizontal Scalability on The Cloud

When an algorithm scales linearly with the number of machines executing the algorithm we reduce the execution time (on the Cloud) at no extra cost by simply increasing the number of machines. There is no extra cost, because we will need the machines for proportionally less time, and Clouds charge based on time. If we double the machines, we half the amount of time we need them for. This doesn't work arbitrarily due to constant overheads, there is a cut off point at which adding more machines doesn't improve performance.

8 Appendix

8.1 Very Basic Mathematical Notation

TODO: Set, iff, function, at least a reference to ZFC, etc

8.2 Big O - Examples

Big O describes the worst case scenario regarding execution time and memory used (the asymptotic upper bound). Big O notation of an algorithm describes how quickly the run time grows relative to the input, N. The following are key examples of Big O notation:

- O(1) – Constant – Describes any algorithm which has a constant execution time and memory usage regardless of the magnitude of the input data.
- O(N) – Linear – Describes an algorithm with a performance which has a linear, directly proportional relationship with the size of the input dataset.
- O(N^2) – Quadratic – Describes an algorithm with a performance which is directly proportional to the square of the size of the input dataset.
- O(2^N) – Exponential – Describes an algorithm which has an exponential relationship between performance and dataset size.
- Olog(N) – Logarithmic – This type of Big O notation has a less simple explanation than the others. An example of a logarithmic algorithm is a binary search technique which searches datasets by finding the median of the sorted data and compares this value to a target value. If they do not match, it either performs the same method on the lower or upper half of the dataset depending on the value it returned prior. This process continues until the target value is matched or there is no more data to split. It creates a growth curve which has an initial steep peak but then begins to flatline. This makes it one of the more efficient algorithms as doubling the dataset has less of an affect on performance.