Re-Thinking Software As Engineering

Why This Document

In the course of past 3 decades or so, there has been a great tendency to complexify or rather "enterprisify" incredibly simple common senses into a form of cargo cult engineering. As computing power increased, the discipline of Electrical Engineering diverged itself into Electronics with the discovery of semiconductors and finally with arrival of cheap computing machines - make "Computer" science as a household and synonym for "developing software".

This document put these changes into perspective, and ask for an introspection of the following questions:

1. Why we do
2. When we do
3. What should we do
4. Where we should do what we must do
5. Who should do what?

Off we go then.

Is Developing Software Engineering ?

First and foremost there is a huge debate raging over this. This is an age old debate - raging from "Are mechanics Engineers?" . The answer is of course, YES. Some mechanics are of course Engineers, but not all mechanics are Engineers. By definition all Engineers are taught to be fully capable as a mechanic in their respective fields.

Computer is no exception. The curriculum share a hefty bit borrowed from Electronics and Electrical - and when one is old timer like us, bits and pieces from mechanical civil and metallurgy too.

What is Engineering ?

Here we ask the question - what makes a discipline Engineering? Engineering, has a very simple definition.

Using scientific principles to economically change society one innovation at a time.

No more and certainly no less.

What are scientific principles? It is adherence to Logic and Data - gathered by impartial unbiased experiments.

If one is too much into logic and not looking at any data, we are looking at abstract mathematics and foundations of mathematics, no small feat of human mind, by any means. After all, pursuit of those generated Kolmogorov axioms and pricing theory as results from those abstract thinking.

If one is only data driven - then we are literally looking at mechanics at an engine shop who does not know why they have to change the knob when the pressure at a critical level. Such thinking will keep the engines running, but will not build newer better engines.

Hence, **pursuit of Logic IN Data is what pursuit of science is**, Engineering just adhere to it to make the fruits of sciences available to the society.

What makes Software Not Engineering Today?

In the last 30 years or so, a lot of literature has been formed - which, for the lack of better terms will qualify as Cargo Cult. No-one, having one ounce of scientific experience will give any due to them. These viewpoints generated way more debates than it should because these are biased personal opinions - not result of any impartial impersonal evaluation of data.

Let's go about some of those debate which were raised.

1. **OOP** - Arguably the most popular debate of all time. Quite literally more than half of all literature on "Software" are devoted to only this, and methodologies about this. Detractors argue this is a fault of Alan Kay, and Bjarne Stoustrup, however I would argue than people who did not study what Kay and Bjarne has to say about real world modelling. None, and let me repeat none - actually read the 1997 edition of the C++ Programming language - where chapter IV is the source of almost all trouble we see today and naturally the counter point of Alan Kay.

2. **Type War** - the most dangerous war in the Software. Type vs Untyped debate rage supreme, even today. Eventually like Cold War, an uneasy peace has been established.
    1. Web is untyped or dynamically typed (ECMA Script, HTTP etc etc)
    2. UI is rarely Typed (Web, WPF, Android ...)
    3. Enterprise Computation is fully typed (C#, Java , etc etc)
    4. Administrators are partially typed (Python, Powershell, Shell and did we forget DB and SQL?)
    5. And then the hardware engineers look at chips and assembly and wonder what is the fight that is going on. Software, quite humorously is full of humorous people, they think.

No matter how funny it sounds, let me assure you, my readers, that the reality is far from that. It is literally like a holy war out there, where no one yields completely. With the advent of ML - Python simply won there, but Julia came along with types. Guido, simply conceded Google (Go, anyone?) and moved to Dropbox.

You might say I am digressing, but no. Wait. None of these debates and holy wars or crusades were done and fought objectively. All of these are being fought about "**S**ome **R**andom **E**lder has **S**ome **R**andom **O**pinion" I call it **SRESRO** principle.

Naturally, people who are actually working on Windows, Darwin or Linux **kernel** - make joke about **SRESRO**. After all Dave Cutler literally created the AMD64 architecture to generate address virtualisation and expansion. Torvalds, who does not require introduction - and everyone knows he does not like C++ at all. See what I mean? **SRESRO** again! It is hard to come out of the mentality!

Software, in the last 30 years became a classic case of Cargo Cult Engineering. To summarise :

1. There has been no experiments to measure any DOGMA's practical efficacy or efficiency and ROI
2. There has been no foundational discovery - from a pure scientific standpoint (locked into OOP model)

It is very clear, by Logic and Data that we are looking at clear Cargo Cult which just believes what the Elders said.

How Software Can become Engineering?

By simply bringing about Scientific Principles. Ask any random **software Engineer** what are **Scientific Principles**. See how the individual falters - and says that is not software. It is, quite naturally a piece of Art.

Newsflash, I do have a first year degree in Painting, and I can tell you Software is not Arts. Not the liberal kind anyways. Now, I know some Universities used to give MA in pure mathematics - Software belongs to that M.A. category - because there are theorems which proves it. It is not **SRESRO**.

This is what we call **Janus syndrome**. In front of an **Engineer** people say Software is **Art**, while in front of a lay-person they would say it is **highly technical** thing. (Same person who's name is imbibed in the month of January past and future !)

People even created a name for it, "Technology". And, much to the everlasting shame to Claude Shannon, they added "**Information**" in front of it, by calling it "**IT**" Information Technology. Any communication engineer literally laughs at it. I would say it also alludes to the horror novel "IT" from Stephen King.

Let's demystify this a bit. All program, a logical valid program is equivalent to a mathematical theorem, proven. There is a theorem around it - and that is called **Curry Howard Correspondence**. Let's repeat. Any logical, testable, proper program - which is correct IS in fact an established Mathematical Theorem.

Software by definition is scientific then, more into Logic than Quantum Mechanics really is - where people still do not understand the cause of Uncertainty and treat it as an axiom - quite literally we stopped digging there (not really, read The Trouble With Physics)

Software can be made Engineering by inverting the trait of **SRESRO** - by **demanding proof**. The problem with **SRESRO** is they appeal to common sense. Common sense is the most uncommon thing in the universe. Many common sensical things are not true at all - Banach & Tarski showed it prior. Science IS NOT Common Sense, not all the time. Neither so is Engineering.

As B. Russel said - if someone claims that between Mars and Jupiter - there is a tea cup too small to be detected by RADAR or a Telescope - someone then has to bring the proof that it is so.

If some manual says : "this is recommended practice" - source code needs to be seen to find out if the practice makes any sense or it is a clever ploy to discourage people into finding issues in the software construction itself. Mark Russinovich is a stalwart in this regard. His legendary handling of Terminate and Stay Resident Rootkits are taught by every kernel instructor.

Never Believe the Manual. Read the source. If source not available, reverse Engineer the source code.

Mark disassembled the windows source code and wrote a whole manual - which was published later by Microsoft Press as the only authoritative book on Windows Kernel, it still is. Every Windows Kernel developer has to read it, and check the source.

After all Science is about Reverse Engineering Nature.

Software, if anything different - is taking a deviant path from other scientific discipline then even with Open Source is the primary mover! People do not read open source. They should read it more often.

Path Forward

By definition we have to invert our priorities. Note the definition of the Engineering. Again.

Using scientific principles to economically change society one innovation at a time.

What comprise of society? Of course, as we all are developing software for some users, they are a subset of the society we want to change via innovation. How can we focus on changing their lives?

Naturally, the path is simple.

1. Deliver customer deliverables in:
    1. Lowest possible time
    2. Highest possible quality within that timeframe

    This is what is called being Agile. **Agile is not scrum**. In fact, it can be argued that **Scrum is anti agile**. Anyone having a difference in "opinion" may look into the agile manifesto itself. Data and Logic, remember?
2. Using minimal resources
    1. Such as to minimise cost
    2. Minimise maintenance

This is what is called being economical. There is a reason why Concorde was a failure. It was not economically viable.

In the following section we shall discuss how to take this path forward.

Deny Dogma

There are only two types of "Software" folks in the world.

1. Who wrote a lot of code, which got shipped to millions of customers with no error correction route
2. Who wrote a lot of blog and shared knowledge to millions of people - w/o writing any actual code

DOGMA are created by people of [2] category. It is incredibly important to understand just because someone came here first does not mean they have some elderly supremacy over other ideas.

Leslie Lamport is the author of Latex and is a literal Computer Scientist. He is a legend in concurrency. Everyone talks about CAP Theorem - which is not even a theorem - but no one really talks to Lamport Theorems which actually are!

No one has ever heard him talking about dogmas. And even he can be wrong. Science has no place for idol worship.

Many DOGMAs are Remnants of OOP

OOP was a dangerously deranged idea which was sold as a magic bullet to all the problems of building large scale software. While it is wonderful on a graphical system almost anywhere else it is an incredibly bad influence. Anyone thought about Object Orient SQL. Oh Wait. That is Cassandra. And MongoDB programmatic access. Anyone noticed how pathetic they become? It is easy to gather usability metric against code connecting to db and sql query running on db.

Let's discuss some those principles.

One of the most interesting group of principles is **SOLID**.

It has became popular and FAD and DOGMA and now there is a **FLUID** principle from 2011.

SOLID stemmed from the brilliant book - The Design of the Unix Operating System - for those who did not study it, it is a must read. Unfortunately it was discussing SOLID at a functional level - not at a system or as OOP calls it "Class Level".

S

Single Responsibility Principle.

Fascinating thought process, but naturally does not work. Because it is excruciatingly vague. And that is where the **ART**, seem to come in. At a pure functional level it is vacuously true, well almost. By definition a function is a subset of a cross product domain set into range set with the additional property that same input can not produce different output.

$$f : D \to R \, ; f \subseteq D \times R \, ; \; f(x) = r \wedge f(y) = r \implies x = y \tag{1}$$

Given it is defined as such, Single Responsibility principle is a THEOREM about functions. But an object is a state machine by itself. It maintains states. At this point it cease to become science and becomes a novel idea with no logical value.

O

Open Close Principle.

Arguably the vaguest of all - it states Modules should remain Open for Extensions but Closed for Modifications. Meyer, when he conceived this, he was considering **Modules**.

> **Design by contract** (**DbC**), also known as **contract programming**, **programming by contract** and **design-by-contract programming**, is an approach for designing software.
>
> It prescribes that software designers should define formal, precise and verifiable interface specifications for software components, which extend the ordinary definition of abstract data types with preconditions, postconditions and invariants. These specifications are referred to as "contracts", in accordance with a conceptual metaphor with the conditions and obligations of business contracts.

Notice the wording. Do you think we develop and design software this formal way? If yes, please stick to it. Given someone is quite literally following design by contracts - Open Close totally make sense.

But it was stole in the name of OOP and it totally has nullable logic value since then.

Also, it quite literally goes against - Composition over Inheritance. Which party you will join? The overwhelming modern developers in the last 10 year or so - has argued that it is not only confusing, BUT simply wrong for the same reasons.

L

Liskov Substitution Principle

Out of all these principles the only one which has a pure theoretical scientific backing is **L**.

Named after Barbara Liskov, a stalwart by no less means the principle reads:

> Let $\phi(x)$ be a property provable of the object of type T.
>
> Then $\phi(y)$ should be true for any object which is S where S is a subtype of T.

Ah. These are music to my ears, because I can see quite literally science here, Logic here. At least this principle has actually something that is non vague!

It makes total sense! Till it does not.

But, does this happen? Objects do not behave like that, not the sort of objects we know, for sure!

This is the only one principle which is stemming from Poof Theory and Type theory (there are multiple semester courses for those) :

> In an interview in 2016, Liskov herself explains that what she presented in her keynote address was an "informal rule", that Jeannette Wing later proposed that they "try to figure out precisely what this means", which led to their joint publication[1] on behavioral subtyping, and indeed that "technically, it's called behavioral subtyping".[3] During the interview, she does not use substitution terminology to discuss the concepts.

We can respect that. That is a woman of science, admitting that its quality snake oil.

Unfortunately, given medium blog posts do not cover these - and books are seldom read - these remained a mystery till Wikipedia came into foray. Data and Logic. Not **SRESRO**.

I

Interface Segregation Principle.

No client should be forced to depend on methods it does not use.

What does that mean? Can I violate it, ever? Like ever? Turns out like you can. Just have a base class having lot of fancy stuff it will be doing.

> The design problem was that a single Job class was used by almost all of the tasks. Whenever a print job or a stapling job needed to be performed, a call was made to the Job class. This resulted in a 'fat' class with multitudes of methods specific to a variety of different clients. Because of this design, a staple job would know about all the methods of the print job, even though there was no use for them.

This is exactly the problem of subclassing from a base class, which has lot of methods which never gets used, ever.

Question yourself, is it possible to do it, ever? The answer is avoiding inheritance completely and using Trait model, we shall talk about it a bit later to the end. This principle says - do not subclass almost ever, for specialisation. Use Composition, sort of.

D

Dependency Inversion Principle.

In short, in a module dependency graph, we should be able to do a topological sorting - and there should be no cycle.

Should there be? Biologists are not too sure. This comes under a pretty standard system called dynamical system where causes and effects are non linear, most of the time. For example see Logistic Equation.

Real world is non-linear. Linearity of this form is quite literally like inducing temporal traversal order on a connected system. Only in an extremely simplified system such things can be done.

Did Computer scientists thought about it? Bet they did. That is why there are "Event Driven Programming" . In a connected system the only way forward is events - which are always non linear.

Should the processing be Markovian? For those who are unaware, a Markovian process is where next step is only dependent on the last state, while a non Markovian process might have more states.

This would require someone to store all the states changes of the eventing system - which is or course what a Snail Variable is.. The observable, albeit not entirely correctly implemented.

Again this exists, because, there are Objects. If a behaviour was modelled in terms of pure function, such thing will be written via a compositional logic.

Get Shit Done

Given that OOP itself is a pretty remnant of the past - and almost all languages are evolving into part functional nature it is highly recommended to discuss the path forward.

The following principles are derived directly from Engineering.

1. Develop for today (You are not gonna need it - YANGNI)
    1. Do not forget tomorrow (Do not Repeat Yourself)
    2. Totally ignore days after tomorrow (things change every month)
2. Fastest finger first wins
    1. When the quality is high enough (happy customer)
    2. Tomorrow is taken care of (build for immediate future)
3. Smallest Code wins
    1. Given Tomorrow people would be able to understand it - (Principle of Least Astonishment)
    2. Code is provably correct (CHC)
4. Developers gets judged on:
    1. Less experience - how fast code and how much with quality
    2. Higher experience - how much code they removed from existing source code base
5. Only measure of code compaction is - Kolmogorov complexity

6. Code complexity measures like Cyclomatic are dangerously modular (Assembly Line Complexity)

Hence, the motto should be get shit done (A LinkedIn teaching) - faster, better, cheaper than anyone. That is engineering.Of course assembly can be itself an Engineering feat! See "The Toyota Way" to understand what assembly line really means.

Further Discussion on "Design" & Metrics

It is absolutely important to understand that **NO Code is the best code**. Unlike some consultancy firms, in a product company, any product company, really, no one cares about the lines of code that have been written. Again, time to see Kolmogorov Complexity.

Productivity is measured, quite literally by

$$Productivity = \frac{Outcome}{effort} \tag{2}$$

Hence, infra should be written in a such a way that coding should be breeze. Building such an infra is Engineering. Compare creating a locomotive engine - vs piloting it. Maximise outcome, while minimising effort is the key.

1. Code LESS
2. Code Meaningful (Maximise Information Entropy)
3. Deny Dogma

It should be called **LMDD**.

Perils of Inheritance

Humans love it. Biologists love it, but then, inheritance has cost in terms of performance and complexity. Most people imagine inheritance is a hierarchy, and they are mistaken - because it is a graph. There is no family tree, it is always a Family Graph. And then there are foundational problems in inheritance : most easy one which gets taught to the sophomore folks is Circle Ellipse problem. Who should inherit who ?

What "developer" tend to forget is there is still a Curry Howard Correspondence. A compiled, correct code must adhere to a theorem - and that stems from Axioms. While almost everyone understood the nuance of "Diamond" problem - almost no one realise it is an unavoidable conclusion from Incompleteness Theorems from Godel. In short, it will remain, forever.

In summary, inheritance and polymorphism due to that is result of applying DRY. But obviously, there are costs. These costs are known as Abstraction Costs. Rust makes it painfully obvious. For example, here is the code that literally finds out in the inheritance chain - which method to call (dynamically typed language interpreter) :

```
data class XMethod(
   val name : String,
   val body : Array<Byte>
)
data class XObjectType(
 val myId : String,
 val myParent :  String,
 val methods : Map<String,XMethod>
)
fun findMethod(someInstance : XObjectType, method: String,
Map<String,XObjectType> repo) : XMethod? {
  var cur = someInstance
  while ( cur.myId.isNotEmpty()){
    if ( cur.methods.containsKey(method) ) return cur.methods[method]!!
    cur = repo[cur.myParent]!!
  }
  return null
}
```

This code above shows the problem with nested hierarchy of OOP - the abstraction cost. Caution : Statically Typed languages do not do these - they have 2 magic tricks: Virtual Function Pointer Table and Pointer to Virtual Function Pointer Table, which gets generated at compile time - and maintains tables which can be actually of the order $O(n^2)$ where n is the number of classes in the hierarchy.

One would be blissfully unaware of such things - unless of course they developed languages multiple times, to solve various problems.

There is a tendency for people to start inheriting from a base class. No one noticed that it goes totally against the single responsibility principle. Again they are DOGMA.

Many older, poorly written frameworks (and many modern, poorly written ones) rely upon Base class. Java Servlet was the first of them - the most notable ones, and just coming after COM/DCOM WPF did not do justice there either.

The entry point is you have to inherit from a base class. That is catastrophic because neither Java or C# support multiple inheritance. Once you done with it - there is no escape, you can not ever ever inherit from anyone else.

Most people do not understand what an abstract class is. It is one who is actually implementing an algorithm. Notice the abstract class `AbstractMap` in Java sdk. One should NEVER implement and abstract class like that, ever. That is a demonstration of what is to be totally avoided. Specifically look at the functions which every child of this class overrides. It seems people were learning OOP while designing Java JDK.

Almost Abstract Class - Interfaces with Default

It was not lost entirely on common sense of the people. That is why, when the functional rewrite happened, new features came along which supported Interfaces with proper implemented methods - called default methods. Otherwise, all classes inheriting from them have to rewrite all the implementation.

So, modern Java has a tiny advantage. They have interfaces with proper methods defined, which makes them dangerously close to the Abstract class.

What does that give you? That is the one which gives you flexibility. You do not assume (adhering to SRP) - one class does it all. You ask - is this control closable? Is that component network-aware?

An interface is what really what a responsibility looks like.

Here are some metrics:

1. Base class ? That is bad
2. Lotta interfaces? May be good given
    1. A codebase where one interface gets used only once is a debacle.
    2. Interfaces are real traits of a responsibiity

Problem is, they can not store states. But of course, they can, if you force an interface to be a View.

Onto State Management and Reversible Computation

What are objects anyways? The foundational question gets answered in the most ridiculous way possible. However, formally there are two answers.

Objects are:

1. States Machines.
2. Property Buckets

In a language like JavaScript or Python objects do show the Property Bucket characteristics, while Strostrup's objects are state machines.

JVM and CLR are in between, thanks to reflection they can actually be made into either!

Alan Kay - the literal founder of the OOP believes objects should talk to each other via events. While this is the domain of languages such as Erlang, and is possibly the ONLY Language which is true to what Alan suggested. Unfortunately, for demented souls like us relying on JVM and CLR that is not going to be true. But even Alan's sense too - they are state machines never the less.

Object Relational Mapping - ORM

What people do not realise that there is a fundamental relation between event and objects methods.

Objects methods are only of 2 types as far as state change is concerned.

1. Write - change state of the object
2. Read - does not change state of the object

The [2] category methods are actually called "View" to the object system. Thus, a view shows only a projection of the objects current states ensemble.

An event is something that changes states of an object - what it could be, we wonder? Of course, nothing but a write function call over an object. That is the lowest granularity event that can be supported on that object.

Consider the same as the database view. This relationship between database entity and object entity generated the while discipline of ORM - Object Relational Mapping.

1. Hibernate focuses on Property bucket representation - storing fields in columns
2. iBatis focuses on event representation

Recall the triggers in a database ? They are quite literally event hooks.

This brings the fundamental question - why we can not then write SQL for object query? Some smart people thought about it, and most notable of such implementation is Type Aware LINQ a default part of .NET. Sadly, it seems from an Engineering standpoint JVM ecosystem is lightyears behind the CLR.

View Models & Data Binding

Views, are then, pretty obviously just a method call by gathering current frozen state of an underlying object. Which object ? That object is called a ViewModel. That is it. There is really nothing fanciful about it. But implementing ViewModel is an incredibly complicated job - one has to freeze something to give only read only access to the underlying states.

Unfortunately, NONE of the examples on internet understand what it supposed to mean or do. World has fallen into the trap of a Cargo Cult Engineering here.

When MVVM was invented in Microsoft - it stemmed from removing logic from the UI - so that UX designers can design better and ViewModel focusses on data binding.

What is data binding? To understand data binding one has to understand the concept of data source, again, invented in Microsoft and then quietly stolen and used incredibly misappropriated across all over in everywhere.

The meaning of Boundable View is :

1. One gives a collection or any other object to the View
2. View just knows how to render it.

3. All of these has to be done automatically - w/o any source code

Consider the view that gets created via a metadata called a view definition? That is the true meaning of the Data Binding and View Model.

Only time one should focus on ViewModels if one is quite literally creating a custom control. Nowhere else.

It is frustrating to see there is no such control already available in frameworks which is more than 10 years old where we are not able to do this :

```
data class MyData(
   val oneFieldOfObject : String,
   val anotherFieldOfObject : String
)
val someList = listOf( MyData("A","a") , MyData("B","b") )
// finally
radioControl = radioGroup{
  this.dataSource = someList /* dynamically bound, if someList changes it would autorender */
  this.textProperty = "oneFieldOfObject"
  this.valueProperty = "anotherFieldOfObject"
  onSelectChange { selectedItem ->
    /* do some magic here if at all
    ideally it can auto bind to something else
    */
  }
}
```

This is rudimentary, and creating such a code is quite literally in a fresh graduates capability, but it does not exist. Apparently someone autogenerates the class for the same, it is bad Engineering. Let's keep it there. There is nothing new, these concepts are quite literally 10 years old. Old wine in a terribly new bottle you say? WPF had this in the first version. I have a patent for the same.

Summary is, ViewModel should be an automatic infra, not required to create at all - anywhere. It's only jobs are:

1. Data Binding
2. Event Handling

Both are to be automatically done - in Pareto optimal cases. Focus on creating such components. I fail to see any focus is being done on creating some of these. That would be, as it depends on data, number, an Engineering.

Information Conservation - Reversible Computation - Immutability

Most people who approach the problem of functional paradigm from the classic Curry way (Haskel) - do miss the point of Entropy and the reversible computation. Ask yourself the question - given only output of the AND function can you tell what inputs were used to generate the output? You can not, right? This is from where information conservation starts.

It is easy to just invert the AND operation if you keep one input with the output itself. Feynman is attributed to reversible nature of computation - but - it has huge implication in the asynchronous programming.

If you make data immutable it generates referential transparency, and thus, if you store all steps you would be able to playback anything ever happened to your system. In the colourful language "Software Development" it is known as playback - ability.

While this is done by almost all hedge funds for legal and financial reasons - it is not a common practice because of the huge amount of data it would generate.

Consider the risk of memory when we sort a 1 million record and it produce another 1 million record, sorted as a separate list. Immutability has a cost and a terrible price. These are Engineering choices.

Quack Like a Duck, Typing and Traits , Mixins

For those who come from any dynamic language - they are familiar with what is called as Duck Typing. If something quack likes a duck then it must be a duck.

Modern languages, like Go for example took this definition by heart. In go, one do not even say if a structure is implementing any such Duck-i-ness.

```
type Animal interface {
    Speak() string // that is all folks!
}
type Duck struct {
}

func (d Duck) Speak() string { // it is not even inside it's definition
    return "Quack!"
}
```

There is no `implements` keyword in Go; whether or not a type satisfies an interface is determined automatically. This in Python would be called "Duck" typing, really (well, not really).

Go is dangerously close to what is called Trait, and for the unaware interfaces are a type of shell-traits.

> *Scala traits* were designed from scratch as building blocks for modular components composition. They are multiple inheritance friendly and don't have diamond problem by having strict rules on evaluation order of mix-ins due to linearization. They also support state, can reference the implementing class and place restrictions on which type can mix-in them. Look at Scala collections library where traits are used thoroughly.

So, the fact is, Java (and subsequently Kotlin) is not that much evolved in terms of "Computer Science" - Scala is still lightyears ahead. Ordersky, is a genius. I have said it a million times, and I will say it again. Why, why why Ordersky did not design Java?

On the other hand some dynamic languages do support a limited form of multiple inheritance - most notably Ruby and Python. Even in C++ there is **Mixin**. This is a mixin from Ruby :

```
module Debug
  def whoAmI?
    "#{self.type.name} (\##{self.id}): #{self.to_s}"
  end
end
class Phonograph
  include Debug
  # ...
end
class EightTrack
  include Debug
  # ...
end
ph = Phonograph.new("West End Blues")
et = EightTrack.new("Surrealistic Pillow")
ph.whoAmI?  # "Phonograph (#537766170): West End Blues"
et.whoAmI?  # "EightTrack (#537765860): Surrealistic Pillow"
```

> you can *include* a module within a class definition. When this happens, all the module's instance methods are suddenly available as methods in the class as well. They get *mixed in*. In fact, mixed-in modules effectively behave as superclasses.

And that is why Ruby was said to be "beautiful language" if only it has speed. Wait. `JRuby` has similar speed as Java!

Final Words

In 1930s - Chemists celebrated what is known as Hybridisation. By 1950 Scientists already found it is ineffective in anything beyond rudimentary organic compounds - and Molecular Orbital theory came in foray which is a subtopic of quantum chemistry. Unfortunately all of us, studied a 70 years old theory in pre college, because that was easily understandable, not because it was correct assessment of reality.

Science does not depend on easy. Science is the pursuit of truth, and Engineering must be motivated by scientific progress.

Thou shall speak truth always - while is nice starting point - it's usage , as everyone knows is limited. John Nash & our own Von Neumann is attributed to create the discipline of evolutionary psychology to state otherwise. Ethics, as applied mathematics suggests - is plain and simple Evolutionary Stable Strategy.

Likewise, what was true - in 1960 and 1990s even and in 2000 even - is not true anymore. Java was a revolutionary way of looking at development then - today it is the biggest laggard in terms of adapting from research. Given OOP quite literally means Java /C# and does not really mean Simula, or Prototypic Inheritance in ECMA Script or even C++ (For Bjarne Sake - Java was a clone!) - many of the concepts are archaic and useless for modern software development using modern tools. There are way more paradigms in the world to solve a problem that OOP, it is one of 100s. It is useful to study them all, in detail.

Rust frowns upon creating functions even - because function calls are slow. Rust believes in macros, and believe it or not, `println!()` in rust is implemented as a macro because they want to optimise on speed. All heavily used functions, as rust deals with it - are implemented as macros!

It is one thing to state Newtons Law, and another thing to comprehend it against Noether theorems. The truth is not Newtons Laws - Newton just discovered part of it, truth lies in Noether Theorems. No one can say Newton was WRONG. Newton started the journey which culminated by Noether and later picked up by Einstein to put it an a better place.

Same is true for Software in Modern era.

Hence the most important aspects of Engineering - Software.

1. **Deny Dogma**
2. **Ask question of 5 Ws**.

Let's summarise with the **5Ws**.

1. **Who**
    1. Wants software ? Customer and Business
    2. Builds Software ? People like us
2. **What**
    1. We do? Build software for business
    2. We want ? Customer Satisfaction
3. **When**
    1. Customer want it? Fastest possible way
    2. We think it is ready ? Open problem
4. **Where**
    1. To apply theory ? When there is a significant ROI measured in numbers (removal of lines of code)
    2. Not to apply theory ? Failing that previous point
5. **Why**
    1. Customer needs it ? That is how they earn money - so deliver faster

2. We build it? Because that is how we earn money - so deliver faster

The only true metric of success in Engineering is customer satisfaction using scientific principles so it becomes repeatable.

This document will be incomplete if I do not put the quote that started this revolution in terms of how software is bloated garbage today - Ryan Dahl's (yes, the same guy who wrote Node.js) rant about software:

> I hate almost all software. **It's unnecessary and complicated at almost every layer**. **At best I can congratulate someone for quickly and simply solving a problem on top of the shit that they are given.** The only software that I like is one that I can **easily understand and solves my problems**. **The amount of complexity I'm willing to tolerate is proportional to the size of the problem being solved.**
>
> In the past year I think I have finally come to understand the ideals of Unix: file descriptors and processes orchestrated with C. **It's a beautiful idea. This is not however what we interact with. The complexity was not contained**. Instead I deal with DBus and `/usr/lib` and Boost and ioctls and SMF and signals and volatile variables and prototypal inheritance and `C99_FEATURES` and dpkg and autoconf.
>
> Those of us who build on top of these systems are adding to the complexity. Not only do you have to understand `$ LD_LIBRARY_PATH` to make your system work but now you have to understand `$NODE_PATH` too - there's my little addition to the complexity you must now know! **The users - the one who just want to see a webpage - don't care. They don't care how we organize /usr, they don't care about zombie processes, they don't care about bash tab completion, they don't care if zlib is dynamically linked or statically linked to Node. There will come a point where the accumulated complexity of our existing systems is greater than the complexity of creating a new one. When that happens all of this shit will be trashed. We can flush boost and glib and autoconf down the toilet and never think of them again.**
>
> ...
>
> **The only thing that matters in software is the experience of the user.**

Further Reading & References

NOTE : This document takes a very wide sweep and there are some links to encompass the "opinion" which can be verified via data and logic. The references are presented here. By no means they are exhaustive, and new ideas gets added each day - and old ideas gets discarded and re-invented in new bottle (MVVM) each day.

1. Enterprisification - https://projects.haykranen.nl/java/
2. Cargo Cult - https://en.wikipedia.org/wiki/Cargo_cult , https://en.wikipedia.org/wiki/Cargo_cult_programming
3. 5Ws - https://en.wikipedia.org/wiki/Five_Ws
4. Agile Manifesto - https://agilemanifesto.org

5. Computer Science - https://en.wikipedia.org/wiki/Computer_science
6. Software Engineering Debate
    1. https://www.cs.usfca.edu/~parrt/doc/software-not-engineering.html
    2. https://www.theatlantic.com/technology/archive/2015/11/programmers-should-not-call-themselves-engineers/414271/
    3. https://softwareengineering.stackexchange.com/questions/111265/is-software-development-an-engineering-discipline
7. OOPs(?)
    1. Alan Kay - https://wiki.c2.com/?AlanKaysDefinitionOfObjectOriented
    2. Stroustrup - https://www.amieindia.in/downloads/ebooks/cplusplus.pdf#page=700
    3. Meyer - https://en.wikipedia.org/wiki/Object-Oriented_Software_Construction
    4. Exceptionally Bad - https://www.leaseweb.com/labs/2015/08/object-oriented-programming-is-exceptionally-bad/
    5. Disaster - https://medium.com/@konradmusial/why-oop-is-bad-and-possibly-disastrous-e0844fa96c1f
8. Type Systems
    1. Strong and Weak Typing - https://en.wikipedia.org/wiki/Strong_and_weak_typing
    2. Type Theory - https://en.wikipedia.org/wiki/Type_theory
    3. Category Theory and Types - https://ncatlab.org/nlab/show/relation+between+type+theory+and+category+theory
    4. Haskel Type System - https://softwareengineering.stackexchange.com/questions/279316/what-exactly-makes-the-haskell-type-system-so-revered-vs-say-java
    5. Scala Type system - https://docs.huihoo.com/scala/programming-scala/ch12.html
    6. Dynamic Typing Benefits
        1. https://wiki.c2.com/?BenefitsOfDynamicTyping
        2. https://softwareengineering.stackexchange.com/questions/122205/what-is-the-supposed-productivity-gain-of-dynamic-typing
        3. Comparing Type systems - https://pleiad.cl/papers/2014/hanenbergAl-emse2014.pdf
        4. End of Cold War - https://www.ics.uci.edu/~lopes/teaching/inf212W12/readings/rdl04meijer.pdf
9. Algorithmic Information Theory
    1. Shannon Entropy - https://en.wikipedia.org/wiki/Entropy_(information_theory)
    2. Kolmogorov complexity - https://en.wikipedia.org/wiki/Kolmogorov_complexity
    3. Chaitin Costant - https://en.wikipedia.org/wiki/Chaitin%27s_constant
    4. Mu Puzzle - https://en.wikipedia.org/wiki/MU_puzzle
10. Curry Howard Isomorphism - https://en.wikipedia.org/wiki/Curry–Howard_correspondence
11. Design of Unix Operating System - http://160592857366.free.fr/joe/ebooks/ShareData/Design%20of%20the%20Unix%20Operating%20System%20By%20Maurice%20Bach.pdf
12. SOLID
    1. Basis - https://en.wikipedia.org/wiki/SOLID
    2. Discussion - https://www.tonymarston.net/php-mysql/not-so-solid-oo-principles.html

3. How SOLID is SOLID - https://stackoverflow.com/questions/2997965/are-solid-principles-really-solid
4. Quality **IS SPEED**
    1. Why I Don't Teach Solid - http://qualityisspeed.blogspot.com/2014/08/why-i-dont-teach-solid.html
    2. Beyond SOLID - http://qualityisspeed.blogspot.com/2014/09/beyond-solid-dependency-elimination.html
    3. **SOLID IS Just Functional** - https://blog.ploeh.dk/2014/03/10/solid-the-next-step-is-functional/
5. We Should Drop Open Close Principle
    1. Drop it Now - https://www.codeproject.com/articles/607395/a-call-to-drop-the-open-closed-principle-from-the
    2. It is Wrong - https://naildrivin5.com/blog/2019/11/14/open-closed-principle-is-confusing-and-well-wrong.html
    3. Better Rules for Software Design - https://naildrivin5.com/blog/2019/07/25/four-better-rules-for-software-design.html
6. FLUID Principles - https://www.slideshare.net/Kevlin/introducing-the-fluid-principles
7. Least Astonishment
    1. Standard - https://en.wikipedia.org/wiki/Principle_of_least_astonishment
    2. Review - https://wiki.c2.com/?PrincipleOfLeastAstonishment

13. Inheritance
    1. Objects Are Hard - https://www.keithschwarz.com/cs143/WWW/sum2011/lectures/120_Runtime_Environments_2.pdf
    2. Prototypal - https://javascript.info/prototype-inheritance
    3. General - https://en.wikipedia.org/wiki/Inheritance_(object-oriented_programming)
    4. COI - https://en.wikipedia.org/wiki/Composition_over_inheritance
    5. Source of Abstract Map - http://hg.openjdk.java.net/jdk9/jdk9/jdk/file/37a05a11f281/src/share/classes/java/util/AbstractMap.java
    6. Mixin in Ruby - https://www.cs.auckland.ac.nz/references/ruby/doc_bundle/ProgrammingRuby/book/tut_modules.html
14. Cost of Dynamic Polymorphism - Virtual Function
    1. https://arxiv.org/pdf/2003.05039.pdf
    2. https://stackoverflow.com/questions/667634/what-is-the-performance-cost-of-having-a-virtual-method-in-a-c-class
    3. https://softwareengineering.stackexchange.com/questions/191637/in-c-why-and-how-are-virtual-functions-slower
    4. https://stackoverflow.com/questions/10125140/alternative-schemes-for-implementing-vptr
    5. https://stackoverflow.com/questions/7013737/alternatives-to-vtable
    6. https://stackoverflow.com/questions/23628081/a-standard-way-to-avoid-virtual-functions

7. https://stackoverflow.com/questions/4352032/alternative-virtual-function-calls-implementations

15. Erlang
    1. Object Oriented - https://stackoverflow.com/questions/3431509/is-erlang-object-oriented/3433808
    2. Message Passing - https://erlang.org/doc/getting_started/conc_prog.html
16. Programming Paradigms - https://en.wikipedia.org/wiki/Programming_paradigm
17. Database
    1. ER Model - https://en.wikipedia.org/wiki/Entity–relationship_model
    2. Relational Algebra - https://en.wikipedia.org/wiki/Relational_algebra
    3. ORM - https://en.wikipedia.org/wiki/Object-relational_mapping
    4. LINQ - https://en.wikipedia.org/wiki/Language_Integrated_Query
18. MVVM - https://en.wikipedia.org/wiki/Model–view–viewmodel
19. Data Binding - https://en.wikipedia.org/wiki/Data_binding
20. Reversible Computation & Immutability
    1. Referential Transparency - https://en.wikipedia.org/wiki/Referential_transparency
    2. Reversible - https://en.wikipedia.org/wiki/Reversible_computing
    3. Arrow Language - https://etd.ohiolink.edu/!etd.send_file?accession=oberlin1443226400&disposition=inline
21. Quack & Duck
    1. Classic Duck Typing - https://en.wikipedia.org/wiki/Duck_typing
    2. Traits in Scala - https://stackoverflow.com/questions/16410298/what-are-the-differences-and-similarties-between-scala-traits-vs-java-8-interfa
    3. Go Interfaces - https://jordanorelli.com/post/32665860244/how-to-use-interfaces-in-go
    4. Rust Traits - https://blog.rust-lang.org/2015/05/11/traits.html
    5. Rust Macros - https://doc.rust-lang.org/1.7.0/book/macros.html
    6. Compiler Design - https://en.wikipedia.org/wiki/Principles_of_Compiler_Design
22. Review of Modern Development
    1. https://jganguly.github.io/assets/programming.html
    2. Design via Composition and Monads - https://www.linkedin.com/feed/update/urn:li:activity:6522136329810210816/
    3. Moving to Rust - https://jganguly.github.io/assets/rust.pdf
    4. Tenets of ZoomBA Language Development - https://gitlab.com/non.est.sacra/zoomba/-/blob/master/_wiki/00-Begin.md
23. Ryan Dhal Rant - https://gist.github.com/cookrn/4015437
24. People References
    1. MR - https://en.wikipedia.org/wiki/Mark_Russinovich
    2. Leslie Lamport - https://en.wikipedia.org/wiki/Leslie_Lamport
    3. Claude Shannon - https://en.wikipedia.org/wiki/Claude_Shannon
    4. Kolmogorov - https://en.wikipedia.org/wiki/Andrey_Kolmogorov
    5. Bjarne - https://en.wikipedia.org/wiki/Bjarne_Stroustrup

6. Alan Kay - https://en.wikipedia.org/wiki/Alan_Kay
7. Liskov - https://en.wikipedia.org/wiki/Barbara_Liskov
8. Bertrand Meyer - https://en.wikipedia.org/wiki/Bertrand_Meyer
9. Martin Ordersky - https://en.wikipedia.org/wiki/Martin_Odersky
10. Guido - https://en.wikipedia.org/wiki/Guido_van_Rossum
11. Linus - https://en.wikipedia.org/wiki/Linus_Torvalds
12. Dave Cutler - https://en.wikipedia.org/wiki/Dave_Cutler
13. Noether - https://en.wikipedia.org/wiki/Emmy_Noether

25. Other Useless References (Give or Take)
    1. Foundations of Mathematics (Prereq for Comp Science) - https://www.math.wisc.edu/~miller/old/m771-10/kunen770.pdf
    2. Scrum is NOT Agile - http://www.dennisweyland.net/blog/?p=43
    3. Imperative Code Complexity - https://en.wikipedia.org/wiki/Cyclomatic_complexity
    4. Probability Axioms - https://en.wikipedia.org/wiki/Probability_axioms
    5. Nash Equilibrium - https://en.wikipedia.org/wiki/Nash_equilibrium
    6. ESS - https://en.wikipedia.org/wiki/Evolutionarily_stable_strateg
    7. Evolutionary Psychology - https://en.wikipedia.org/wiki/Evolutionary_psychology
    8. Pricing Theory - https://en.wikipedia.org/wiki/Arbitrage_pricing_theory
    9. Trouble With Physics - https://en.wikipedia.org/wiki/The_Trouble_with_Physics
    10. Russel's Teapot - https://en.wikipedia.org/wiki/Russell%27s_teapot
    11. Banach Tarski - https://en.wikipedia.org/wiki/Banach%E2%80%93Tarski_paradox
    12. The Toyota Way - https://en.wikipedia.org/wiki/Toyota_Production_System