A Formal Analysis of Iterative TDD

Authors

Hemil Ruparel (hemilruparel2002@gmail.com)

Nabarun Mondal (nabarun.mondal@gmail.com)

To be submitted to *IEEE Transactions on Software Engineering* (https://ieeexplore.ieee.org/xpl/RecentIssue.jsp?punumber=32)

About

This document was created over authors discussing measurable efficacy of modern development practices. These discussions triggered a now famous LinkedIn post which got eventually shared by Kent Beck. Arguably, the formalism in that post needed a lot more work. In this paper we try to semi formally present the argument for Test Driven Development (TDD) and the reason behind it's efficacy in producing provable Software Systems. We also caution against the cargo cult development which is happening in the name of TDD which can be demonstrated to be not effective in the same regard.

Abstract

We first introduce some concepts. First we talk about what is (incremental) TDD as defined in Canon. Then to formalize it, we define the following concepts:

1. Functions as Point Mapping
2. Software as functions
3. Tests as higher order functions
4. Equivalence Partitions
5. Coupling in Software

Based on these we argue about the nature of the software development in terms of TDD and TDD gets formally defined.

In the next section we formalize Incremental TDD and the context in which Incremental TDD provably works.

We then argue from current development trends that Incremental TDD is definitely not being used in such narrow provable context.

We conclude by stating while evidently it can not work except in narrow contexts, that is the best tool we have when applied in those right contexts - e.g Unit Like Tests where implementation has almost non existent coupling.

What is (Incremental) TDD ?

Canon Definition

From the canon article taken as the "Definition of TDD" :

1. Write a list of the test scenarios you want to cover
2. Turn exactly one item on the list into an actual, concrete, runnable test
3. Change the code to make the test (& all previous tests) pass (adding items to the list as you discover them)
4. Optionally refactor to improve the implementation design
5. Until the list is empty, go back to [2]

Narrative

The objective for Incremental TDD is to ensure that we end up having a formally verifiable software in each step and in the end when all the "scenarios" are exhausted. Another "optional" objective is given as to "Improve the implementation" and it is not defined anywhere between one implementation to other what can be "improvement".

This is not a good starting point to formally analyze the methodology, as success metrics are not possible to be created on top of it. It is very imprecise, and open to interpretations.

In this paper we formally define such a methodology and provably demonstrate how provably correct software can emerge with clear metric on amount of code churn was done to attain it over the iterations - albeit in a very narrow context.

We call the "canon" practice followed in the industry as "Incremental TDD" for reasons which would be apparent after the definitions.

Definitions

We would need some definitions to formalize the (incremental) TDD pseudo algorithm.

Specification of Functions via Point Pairs

Any function, computable or not, can be imagined to be pairs (potentially $\aleph_1$) of input and output points in some abstract space. It makes sense to describe functions by defining their specific outputs at specific points or a large set of equivalent points. This list of pairs we shall call point specification or "specification" for brevity.

For functions which are well behaved this makes some sense. But even for well behaved functions this is not a good enough approximation.

Take a nice function like $f(x) = x$, a tautology but one can not define this function by adding pairs of specification values. A much more interesting function like $f(x) = sin(x)$ is much harder to describe, although we can always define them pointwise, and that would ensure the resulting "sampling" looks much and much like the target function, one must understand infinite pairs would be required to specify $sin(x)$. Even with $\aleph_0$ points specified, there would be set of infinite functions who are not $sin(x)$ but just gives off the exact same value at all those specific points. This has a name, called pointwise convergence.

Outside those fixed set of points the family of functions can take arbitrary values - and thus specification via point pairs arguably pose a problem.

Luckily, for software we can do much better, which is the topic for the next section.

Software

A software is defined to be a Computable Function - mapping abstract vector space of input to the output vector space. The notion of using vectors is due to all real software works with many inputs and hence the state space is multidimensional which is the exact same space as output.

$$S : \hat{I} \to \hat{O}$$

where $\hat{I} :=< x_i >$ is the input vector while $\hat{O} :=< y_j >$ is the output vector. These vectors are defined not in physics sense, but pure mathematical sense. The only change between the pointwise defined function vs specified software is about being "Computable".

Software Test

A "Software" test is defined as a higher order function :

$$T : t < \hat{I}_t, S_t, \hat{O}_e > \to S_t(\hat{I}_t) = \hat{O}_t$$

In plain English, a test is comprise of Input vector $\hat{I}_t$, the software under test S_t, and the expected output vector $\hat{O}_t$, it runs the S_t with the input, and checks whether or not the expected output $\hat{O}_t$ matches against the actual output of the system :

$$S_t(\hat{I}_t) = \hat{O}_t$$

and it simply checks whether or not $\hat{O}_t = \hat{O}_e$, hence the range of the test is Boolean.

A software test, then contains a single point specification for the desired Software.

A software test does not need to be computable. Unfortunately, any automated test, by definition needs to be computable. This also pose a problem for testing in general. A test that might not be computable is a human reporting software has hung or went into infinite loop. This is impossible to do algorithmically, unless we bound the time. This comes under Oracles in computation.

Branches and Partitions

Given software is written essentially using arithmetic logic and then conditional jump - this being the very definition of Turing Complete languages - that conditional jump ensures that the different inputs takes different code paths. A code path is a path (even having cycle) in the control flow graph of the software which starts at the top layer of the directed graph that is the code and ends in the output or bottom later.

Formally we can always create a single input node and output node in any control flow graph.

Treating multiple iterations of the same cycle as a single cycle, we can evidently say given the nodes of the graph is finite, there would be finite (but incredibly high) number of flow paths in the graph.

At this point we introduce the notion of equivalence class of input vectors to software. If two inputs $\hat{I}_x$ and $\hat{I}_y$ takes the same path P in the control flow graph, then they are equivalent.

This has immense implication in testing and finding tests. Because this induces an equivalence partitioning on the input space itself, because all $\hat{I}_x$ in the same equivalence class can be treated as exactly equivalent, because all of them would follow the exact same code path in the control flow graph. There is another related concept called boundary value analysis (BVA), but we would not go there, because that is not going to alter the subsequent analysis in any significant way.

This effectively means by isolating all equivalence partitions and choosing one input member from each of them we can test the system the most optimal way. For example, if there are A, B, C, D equivalent classes, then choosing $\hat{I}_A \in A$, only one would test the code path for A, similarly for the rest. So instead of infinite inputs, only 4 inputs would suffice.

This brings the problem, formally to finding the equivalent classes.

That is impossible without the implementation. It is absolutely wrong to perceive that this technique is a specification driven one alone. This is a gray box testing it requires assuming some implementation details. The Wikipedia article on this demonstrates this nicely.

But just how many equivalent classes would be possible? This definitely depends on the number of the conditional jumps. It is easy to prove that if there are B branches, then the bound of the equivalence class is $O(2^B)$ - a huge number. This also would be very important for a pragmatic discussion later.

The Equivalent classes would be called EQCP from now on because they partition the input set into Equivalent Classes. There would be many EQCP for individual "features" in "Software".

Coupling in Software

At this point we introduce the phenomenon of coupling between Equivalent Classes, when seen with respect to code implementation.

Given individual EQCP are nothing but depicting paths in the control flow graph (CFG), the coupling said to exists between EQCPs E_x with path P_x and E_y with path P_y if and only if $P_x \cap P_y \neq \emptyset$.

That is, if paths P_x, P_y has some common nodes, then E_x, E_y are coupled. In fact we can define the amount of coupling using similarity measures now, most easy one would be a Jaccard distance like measure:

$$C(E_x, E_y) = \frac{|P_x \cap P_y|}{|P_x \cup P_y|}$$

This essentially says - "Measure of the coupling between two equivalent classes is the amount of code shared between them relative to all code they have". We need to understand that even code shared for good reason, like applying DRY and not doing it even methodically also would create coupling via this definition. Any shared function between two EQCP would mean coupling exists. As we shall see Coupling becomes an incredibly key phenomenon while formalizing the stability of software under Incremental TDD.

Test Driven Development as Equivalent Class Specification

We can now formally define a software system specification in a finite, and correct way. If we can just specify the equivalence classes, then we can just fixed the software at those specification points and the resulting tests precisely, and correctly defines the software behavior. This must be taken as the formal definition of (non iterative, formal) TDD.

> Given an abstract (not written) Software S_a, let's imagine the equivalence classes E_x such that E_x, E_y are independent and specify the input and output expected from each equivalence classes.

This system is provably complete and correct, by construction. Every test just ensures all individual EQCP behavior is passed via construction. Given that was the entire specification, this means the system passes all criterion for the specification, and thus becomes provably correct.

This is the real superpower of TDD, formal verification baked in into development. This has been known for multiple decades.

Correctness of TDD

The correctness of TDD for a practical application hinges on the following :

1. Is the specification complete enough (to take care of all the equivalent classes)?
2. Is the specification non contradictory ?

That it is impossible to get (1,2) done together follows from Godel's Incompleteness theorems, but that is applicable to any specification, not only Software. Thus this argument should not be admissible as failure of TDD in itself.

Now we ignore the notion of contradiction and focus on completeness and stability when one tests gets added at one time (iterative TDD)

Completeness of TDD Spec

The business specification should be such that the formal specification of all possible Equivalence classes must be drawn from it. As it is bounded by $O(2^B)$ - this itself is not remotely possible. To understand how this bound works, a simple program unix `cat` has more than 60 branches. The equivalent class specification of this program is bounded by 2^{60} and the total stars in the universe is 10^{24} for comparison.

But this again does not disprove the crux of TDD, it only points to the fact that formal EQCP is a practical challenge and to be handled pragmatically.

Now an incremental TDD is when we add more specification to the mix of already existing ones one step at a time. This incremental TDD is what we discuss in the next sections as this is the one which proponents of TDD talks about.

Analysis of Incremental TDD

Development in TDD

Note that the methodology does not specify how to implement the paths of each equivalent classes in the code. Hence evidently there is no way it can ever improve on the "non correct aspect of quality" of software, one of them would be to lower coupling. In fact if not controlled this would bring in way more coupling than it was required due to application of other principles like DRY. Because there are infinite way to conform to the "point wise convergence" but then the methodology does not specify any family of approach to do so. These are some of the key open problems of the methodology as it formally stands as of now.

The very best non coupled way to deliver would be no equivalence class share any code path. This would solve the coupling problem, but code would be massively bloated. Any other way would reduce the code but ensure the classes would be coupled tighter.

Coupling would have implication in iterative TDD, and we also show a provable methodology that can reduce code churn in the later sections.

Stability of Software Point Specification under Iterated TDD

Suppose, there is already an existing system in place with tests done the right way, at least catching up to 1% or 2% of the equivalence classes. Ideally we would want to have way more coverage, and that will be discussed later.

Is it possible to add more specification w/o rewriting existing equivalent classes?

The answer to this is key to the prospect of TDD.

Formally, Software S_r, as the equivalent classes $E_x \in \mathbb{E}_r$, and now more specification change is happening. The following questions need to be asked:

1. How many of the existing EQCP will not be effected by this?
2. How many new EQCP needs to be added?
3. How many EQCP needs to be removed?

As one can surmise, this is the transformation step of a fixed point iteration on the abstract space of the EQCP. We shall get back to it slightly later.

Loose Coupling - A Life Saver

The answer to all these questions is coupling and coupling alone. If the implementation of those equivalent class was such a way that there was minimal coupling, then possibly less classes would be impacted via this. But this is not a principle of TDD in the first place in any form in any practical application of software development. In fact software principle like DRY would mandate code sharing, and hence there would be some coupling.

Pent Up Branching

The answer to the question [2] is in isolation if there would be K branches to implement the delta specification - new feature then, the isolated equivalent classes would be in $O(2^K)$, thus, the minimum new classes needed would be bounded by this value.

At most it can impact every equivalence class and at least it adds $O(2^K)$ classes and hence tests.

Incremental TDD as a Dynamical System

Now, we get back to the problem of defining incremental TDD.

As discussed, this EQCP merging culminates into a lot of those equivalence classes being thrown out, new classes being created - a fixed point iteration on the abstract space of the EQCP itself, which we can now formally define as follows:

$$\mathbb{E}_{n+1} = \tau(\mathbb{E}_n, \delta_n)$$

Where at step n, $\mathbb{E}_n$ is the current set of EQCPs, while based on new specification (δ_n) and the $\mathbb{E}_n$ TDD τ produces new set of EQCPs ($\mathbb{E}_{n+1}$) for the next step $n+1$. This is the fixed point iteration of incremental software development from point pair specification or incremental TDD.

It is obvious that the first ever specification was done with empty equivalent classes and initial specification of δ_0:

$$\mathbb{E}_1 = \tau(\emptyset, \delta_0)$$

This now depicts a dynamical, complex system.

Stability Space

While EQCP space is nice to visualize what is happening for real in terms of Software Specification and Test cases, it is not descriptive enough to translate into numbers so that we can track the trajectory of the Dynamical System. For that we would need a metric, that would define how stable the system is over the iterations in terms of retaining past EQCPs.

Let's $N = |X|$ be the cardinality of the set X.

Let us create the stability metric as the ratio of new EQCP unmodified post application of change :

$$\Sigma_{n+1} = 1 - \frac{|\mathbb{E}_n \cap \mathbb{E}_{n+1}|}{|\mathbb{E}_n \cup \mathbb{E}_{n+1}|}$$

As one can see $\Sigma_n \in \mathbb{Q} \cap [0,1]$, and as close it would remain to 0, the better stable the system is. We call Σ the stability metric for the system.

It is also a metric space with distance between two stability points $a, b \in \Sigma$ the distance between them is defined to be $d(a,b) = |a-b|$.

Observe the following, if we ensure that no EQCP has any shared code, then the only way to make change is to simply add new code, and thus $\mathbb{E}_n \subset \mathbb{E}_{n+1}$, and that gives minimum value of Σ.

This value is :

$$\Sigma_{n+1} = 1 - \frac{|\mathbb{E}_n|}{|\mathbb{E}_{n+1}|} \approx 0$$

If this is put to practice, any value close to 0 shows there is a guarantee that Σ_n follows a very stable path. We note that it is impossible to reach value 0 under any circumstances other than when $\mathbb{E}_n = \mathbb{E}_{n+1}$ which means, the specification δ_n did not change anything in EQCP space, e.g. a complete dud or spurious specification.

Now the other side is when $\mathbb{E}_n \cap \mathbb{E}_{n+1} = \emptyset$, in this case the value goes to 1. Any value close to 1 shows the system has been very unstable between last to the current iteration. This is possible when "strong" coupling ensured that we need to rewrite the EQCP implementations entirely in code.

Guiding Stability Algorithm

Assuming coupling has to be present, one way to avoid unpredictable jumps in the stability , we can device our development strategy such that the Σ does not change drastically. At this point, if there were many alternative way to program (P_i) the δ_n change, we may want to chose the alternative P_x way to program which minimizes Σ_{n+1} . If we do, then the system remains stable in the short term. But this is a direct anti theses of "less code change", as it minimizing Σ_{n+1} culminate into more code change, because it would inherently try to lose some coupling! But even with this guided approach, the problems would no go away, which is the topic of the next section.

Chaos in Stability space

We now proceed to demonstrate that the iteration driven by δ_n in Stability Space has characteristics of chaotic system.

Given there is no universally agreed definition of chaos - we - like most people would accept the following working definition:

> Chaos is aperiodic time-asymptotic behavior in a deterministic system which exhibits sensitive dependence on initial conditions.

These characteristics would now be demonstrated for iterative TDD.

1. ***Aperiodic time-asymptotic behavior***--this implies the existence of phase-space trajectories which do not settle down to fixed points or periodic orbits. For practical reasons, we insist that these trajectories are not too rare. We also require the trajectories to be *bounded*: *i.e.*, they should not go off to infinity.

   The sequence $\Sigma_n \in \mathbb{Q} \cap [0,1]$ is bounded by definition. The trajectories are not rare, and it is practically impossible for the sequence to settle down to periodic orbits or converging sequence. Note that w/o the presence of coupling this sequence converges approximating 0.

2. ***Deterministic***--this implies that the equations of motion of the system possess no random inputs. In other words, the irregular behavior of the system arises from non-linear dynamics and not from noisy driving forces.

One can argue that the sequence is driven by δ_n - an external input, but it is not. Iterative TDD has this baked in, as part of the system iteration description , and the processing of it is algorithmic in the formal methodology which we present for formal correctness for the software. In fact we can argue that the sequence δ_n can be specified beforehand, and it would make it fully deterministic and it would not impact our analysis.

3. ***Sensitive dependence on initial conditions***--this implies that nearby points can be spread further over time while distant points can come close over time - e.g. stretching and folding of the space. In fact it is said to be:

   > Chaos can be understood as a dynamical process in which microscopic information hidden in the details of a system's state is dug out and expanded to a macroscopically visible scale (*stretching*), while the macroscopic information visible in the current system's state is continuously discarded (*folding*).

   This is evident in case of coupling.

   CFG comprise of the micro details which culminates into the the space of EQCP, and merging further specification over that produce the sequence Σ_n . Inherently a lot of micro details are being pushed into visibility and then again being discarded.

   Given two nearby points in Σ_n , say $a, b : |a - b| < \epsilon$, there is no guarantee that in next iteration how further apart the sequence would go, given even exactly same specification of δ_n . Let $\Sigma(p, \delta)$ be the next iteration sequence after starting from p in Σ_n post applying the same specification change δ.

   Then the $|\Sigma(a,\delta) - \Sigma(b,\delta)| \neq 0$ almost always for all practical purposes.

   Let us define the function $\lambda(a, b, \delta)$ as follows:

$$\lambda(a,b,\delta) = \frac{|\Sigma(a,\delta) - \Sigma(b,\delta)|}{|a-b|}$$

A **stretch** happens when $\lambda(x,y,\delta) > 1$ and a **fold** happens when $\lambda(x,y,\delta) < 1$. This is to say, stretch increases the distance between the trajectories starting with (a, b) while fold reduces it.

Evidently, if only folding happens, then every sequence would converge. This is an extreme view. In the same way if only stretching happens, then because the sequence is bound, it must converge again to 0 or 1. This is another extreme view.

We can safely say the probability that for every tuple (a, b, δ) that the $\lambda > 1$ would be 0. So goes the same for $\lambda < 1$.

It is much more plausible that a function like this would have some intervals where it would stretch and some intervals where it would fold depends on the δ. This is the most likely phenomenon which invariably would generate a sequences diverging and converging in Σ thereby producing the dynamic process that stretches and folds - and thus creating sensitive dependence on initial condition, the hallmark of chaos.

The above points make it very clear that the sequence Σ may show all properties of chaotic dynamics.

Practical Considerations for Software Development Under Iterative TDD

Identifying Chaotic Trajectory

Is there a guarantee that chaotic patterns would emerge on each case? No one knows. Chaos in software development has been discussed about although not in much formal details like this. If we are very lucky it would not, but it is hard to tell. Only by carefully monitoring the sequences we would be able to claim whether we entered any chaotic sequence or not and this formalism gives a metric such that the sequence can be tested for emergence of chaos - by using Holgers method. That would be the empirical way of measuring on each iteration how the progress is happening. Given agility is the name of the game now, we can add 52 data points a year for each project if weekly shipping of software is followed.

Domain of Stability for Incremental TDD

Let's imagine the worst case, almost all of the sequences would be chaotic.

What is so problematic about chaotic dynamics appearing in the phase of "stability" of EQCP ? This means there is a lot of churn in terms of the changes in the EQCP. And that means a lot of churns in the "pair points specifications" e.g tests which were to "hold the correctness of the software", implying a lot of implementation change MUST happen. If in one iteration which was created by a tiny change in specification impacted 50% of the test cases to refactor source code and tests thoroughly, evidently this would become a huge problem.

The chaotic thesis suggests that not this is only possible, but also highly likely due to the mixing of EQCPs in terms of coupling, and a direct result of code refactoring trying to apply DRY principle.

Hence the formal idea of just fixing input output points can not work generally unless we keep on reducing the scope of the specification. It is guaranteed to work at the Unit Test level by definition. Unfortunately the proponents of TDD want to make it work even at user specification level - where it entirely lose its rigor and has no provable applicability to either improve the quality of the product or the code itself.

The iteration over all existing equivalent classes and new ones makes it a fixed point iteration, and specifically if lucky it may be in the domain of stability, but such zones would be extremely narrow due to the large dimensions in which the system operates. This is well known in the domain of the complex system. It is what the original document wanted to talk about via Hill Climbing, which is not a terrible way of looking at TDD trying to slowly converging into the "Ideal final state of the software" at a step n. In fact the condition that $\lambda(a, b, \delta) > 1$ or $\lambda(a, b, \delta) < 1$ guarantees the space to become convex to allow Hill Climbing to find a global optima. It is not going to work out if there are too many extremes, and that is the chaotic mapping in the λ space.

It is immaterial for now, for theoretical purpose, whether optimum was reached or not, we are only focused on producing "formally provably correct" Software.

Path Forward - Approaches

From the last section to avoid these chaotic sequences we can try avoiding all of these by either:

1. Making the specification more relaxed - at that point it would specify almost nothing and there would be almost no chaotic behavior because of the state space of EQCP being reduced drastically. This is the placebo, the application of TDD w/o any formalism.

2. Or, we can try to decrease coupling, in which case it would bloat the software by not having shared code path - this would result is unimaginable bloat in the software - given we are looking at very large dimension of EQCP state space.

Evidently, then via [2] incremental TDD, therefore, can only be effectively done in practice when the $\mathbb{E}_n$ space is extremely small and the context of "Software" is very narrow.

Context Of Applicability

Not all is lost however. As it is proven, if we can go narrower and narrower, to the point when EQCPs stop sharing code with one another, TDD not only becomes formally correct, but also the ONLY methodology known to mankind to develop software. Do these exist? They do, these are the unit tests with guarantee of becoming chaos free! We can now formally define scope for formal incremental TDD, which is guaranteed to work - e.g. create formal verifiable correct software as follows without ever destabilizing source code:

> Unit like tests where implementation of such features do not share any source code, e.g. Independent   (completely decoupled) - such that in every iteration the decoupling holds true guarantee to hold to verifiable correct behavior.

And it is in this context TDD reigns supreme. Anything other than that - correctness or stability can not be guaranteed. Just like one can try to use a scalpel to dig a canal, it just won't work. Any effort of using the scalpel to create a canal is not only misguided, but futile, and not even wrong.

Do iterative TDD, just ensure all EQCPs are completely decoupled - that now becomes a provable stable methodology. Now, in practice it is hard to do, even for Unit tests, so a small amount coupling should not really harm the effectiveness via that much - but at that point Chaotic behavior stems in.

Principles like AHA, WET comes in extremely handy in this regard. Even with very less coupling there is no absolute guarantee of code stability, due to emergence of chaos but at least we are in the right track by being formally correct, and the resulting chaos can be tamed.

Placebo view of Popular "Business Specification based" TDD

The previous issues culminates into less and less specific specifications used in the industry. At that point they cover so less equivalence classes that TDD would lose all it's effectiveness which is to be found rigorously at the unit test level. Thus we do have a problem, if we specify more and more, the resulting software has high coupling thereby ensuring the iterations are destabilized. If we specify less and less the resulting diluted TDD is just homeopathy, water in the name of medicine but peoples believe making it "work" - a placebo.

Interestingly this is the most popular TDD in the industry because the only way non Unit Testing TDD can be done is by just showing they people are doing TDD, which is not even wrong. There is no effectiveness of TDD for anything that goes of unit test, as was proven. It gives a lot of people something to talk about and mental peace just like Homeopathy sans effectiveness other than placebo.

"Some input,output are verified" is not really an effective methodology, given the nature of the number of tests required runs in exponential numbers in terms of the EQCP for the features.

Closing Remarks

Formal incremental TDD, as presented here, is shown to produce correct software code. The issue with such production requires a lot more formal and practical considerations.

When done correctly (by reducing coupling) it ensures we can further add more features to the existing software while maintaining stability as well as correctness as we go.

If that reduction of coupling is not followed, then the addition of more equivalence classes could and most definitely would modify a significant amount EQCP mapping by ensuring one must rewrite a very significant amount of tests, as well as implementations. This is also seen in reality. Anything at any further higher level of abstraction that Unit like tests would have impact like placebo.

Hence we propose incremental TDD is to be done at the Unit Testing level only, where it works correctly and satisfactorily because of Units should be essentially maximally decoupled keeping an constant eye on the coupling generated by those tests being constantly added, which is hard, but not impossible to do and shows provable theoretical efficacy.

References

1. Aleph Numbers : https://en.wikipedia.org/wiki/Aleph_number
2. Point Convergence : https://en.wikipedia.org/wiki/Pointwise_convergence
3. Test Vector : https://en.wikipedia.org/wiki/Test_vector
4. Computability : https://en.wikipedia.org/wiki/Computability
5. Oracle Machines : https://en.wikipedia.org/wiki/Oracle_machine
6. Higher Order Function : https://en.wikipedia.org/wiki/Higher-order_function
7. Decidability : https://en.wikipedia.org/wiki/Decidability_(logic)
8. Turing Completeness : https://en.wikipedia.org/wiki/Turing_completeness
9. Control Flow Graph : https://en.wikipedia.org/wiki/Control-flow_graph
10. Jaccard Distance : https://en.wikipedia.org/wiki/Jaccard_index
11. Equivalent Class : https://en.wikipedia.org/wiki/Equivalence_partitioning
12. Gray Box Testing : https://en.wikipedia.org/wiki/Gray-box_testing
13. Test Vectors Considered Harmful : http://fate.tttc-events.org/proceedings/5.1.pdf
14. `cat` source code : https://github.com/coreutils/coreutils/blob/master/src/cat.c
15. Coupling : https://en.wikipedia.org/wiki/Coupling_(computer_programming)
16. DRY : https://en.wikipedia.org/wiki/Don%27t_repeat_yourself
17. Godel and System Specification : https://en.wikipedia.org/wiki/Gödel%27s_incompleteness_theorems
18. TDD As It was Intend to be : https://en.wikipedia.org/wiki/Test-driven_development
19. Complex Systems :
    1. https://en.wikipedia.org/wiki/Complex_system
    2. Dynamical Systems : https://en.wikipedia.org/wiki/Dynamical_system
    3. Fixed Point Iterations : https://en.wikipedia.org/wiki/Fixed-point_iteration
20. Definition of Chaos :
    1. Metric Space : https://en.wikipedia.org/wiki/Metric_space
    2. Chaotic System : https://en.wikipedia.org/wiki/Chaos_theory

    3. https://farside.ph.utexas.edu/teaching/329/lectures/node57.html
    4. https://mathworld.wolfram.com/Chaos.html
    5. https://math.libretexts.org/Bookshelves/Scientific_Computing_Simulations_and_Modeling/Introduction_to_the_Modeling_and_Analysis_of_Complex_Systems_(Sayama)/09%3A_Chaos/9.02%3A_Characteristics_of_Chaos
    6. Mixing : https://en.wikipedia.org/wiki/Mixing_(mathematics)

21. Software Development shows Chaotic Behavior : https://timross.wordpress.com/2010/01/17/software-development-and-chaos-theory/
22. Testing for Chaos in Time Series Data : https://www.sciencedirect.com/science/article/abs/pii/0375960194909911
23. Hill Climbing : https://en.wikipedia.org/wiki/Hill_climbing
24. Homeopathy - Delusion : https://en.wikipedia.org/wiki/Homeopathy_and_Its_Kindred_Delusions
25. Not Even Wrong : https://en.wikipedia.org/wiki/Not_even_wrong
26. Placebo : https://en.wikipedia.org/wiki/Placebo