ANALOG DRIVEN DEVELOPMENT

Harnessing the Conceptual Human Mind to Ensure Software Artifact Stability

Matthew James Swann, *Bachelor of Arts*

Permission is granted to Auburn University to make copies of this document at its discretion, upon request of individuals or institutions and at their expense. The above referenced author reserves all publication rights.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
Signature of Author  
  
  
  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Date of Graduation

PROJECT ABSTRACT

ANALOG DRIVEN DEVELOPMENT

Harnessing the Conceptual Human Mind to Ensure Software Artifact Stability

Matthew James Swann, *Bachelor of Arts*

Directed by David Umphress, *PhD*

<insert meaningless summary drivel here, good stuff comes later>

**Table of Contents**

Table of Contents

Table of Figures

Chapter One – Introduction

1.1 *Of Predators and Prey*

1.2 *Definitions*

Chapter Two – Background

2.1 *Test Driven Development : The First Baseline*

2.2 *The Nature of Inquiry : The Second Baseline*

2.3 *The Temporal Relationship of Thought and Expression : The Third Baseline*

Chapter Three – Implementation

Chapter Four – Validation

Chapter Five – Conclusions

References

Appendix Alpha

Appendix Beta

Appendix Gamma

**Table of Figures**

Table of Contents

Table of Figures

Chapter One – Introduction

Chapter Two – Background

Chapter Three – Implementation

Chapter Four – Validation

Chapter Five – Conclusions

References

Appendix Alpha

Appendix Beta

Appendix Gamma

**Chapter One – Introduction**

<insert significantly more meaningless summary drivel here, good stuff comes later>

*1.1 – Of Predators and Prey*

The process of software development could be described as a game of cat and mouse; however, it is likely more similar to the struggle between the road runner and that coyote who ceaselessly blows himself up. Non-success is a chronic illness more than a temporary writer’s block. Of course, it is easy to see the potential for struggle when we think the human mind and capacity for problem solving to both be analog in nature. A machine requires the rules of a digital world. The mind does not store information in bytes. A computer has no random thoughts. Stream of conscious thought is flighty. Modeling continuous space into a discrete digital realm is simply hard to do. Considering their respective definitions, perhaps analog and digital play like oil and water. This realization does not make the translation between a human’s idea and a computer’s behavior any easier. Not directly. It may, however, be the most empowering consideration, and therefore weapon, we as humans have in the fight to ‘make computers behave!’

Software development is a hunt. A hunt for the set of desired behaviors that will be evidenced by the machine. And the machine *must* behave is certain ways. The human, the animal, is tasked with modeling the elusive behavior and ensuring its stability. The hunt itself is consuming. As he closes in on his prey, the animal can taste success. He can smell the prize. Intellectual inertia torrents through the mind as fiery blood courses though veins. But in a single moment, a minor defect injection can scare off the trophy leaving the human bereft of energy, dissatisfied and hungry. Tired and irritated, the predator sulks back to his den planning the next day’s effort.

In order to better secure the prize, humans attempt to refine their hunting strategy. Be the refinement out of a desire to be perfect or to reduce effort, it still exists. One such refinement takes the form of Test Driven Development (TDD). Further explored in section 2.1, TDD is a process by which tests for the software are written before the software itself. TDD was originally the brain child of Kent Beck, who literally wrote the book on it. Ten years ago, Beck helped the software community refine development techniques by rearranging the software process itself. His argument: if we know what the software has to do, let us build those tests first, then when we write the code we’ll know if we are correct.

Ten years ago, Beck asked and answered the question “How can we better set ourselves up for success?” TDD works. It would be disrespectful to say the process is old hat, but to a degree it must also be refined. It is time to stand on the shoulders of our patron to reach higher. In time, someone will stand on ours. For this is the nature of advancement. One day every action I now execute will be an obscene over usage of effort and every word in this document will be obsolete. I only pray that between now and then my efforts will help to set the table for those who come next.

The refinement I propose herein centers around a re-harnessing of the human mind. Beck attempted to make the process better by forcing the diagnosing code to come first. The tests are written before the source. But Beck wanted to write one test at a time. And he did so using a computer. Analog Driven Development (ADD) will not write one test at a time, nor will it use a computer. ADD will resurrect old world engineering via pencil and paper, sketch books, and borrowed pens feverishly scribbling on the back of restaurant napkins. The eureka moment in between beers surrounded by eager onlookers and friendly waitresses sharing in a moment of pride as the animal crashes down upon his prey. ADD will remind us of our human strengths. And ultimately, ADD will dispose of the perverted thought that unfortunately curses the minds of our students.

“Listen, Timmy, you are going to have to learn to think like a computer.”

The only redeeming quality in this sentence is that it is such a beautiful example of what not to do. As a tangential note, I propose that those promoting this idea be forced to take a pay cut. It is nothing short of disrespectful to the power of the human mind. This suggestion in summary: take the most creative and forceful computing device we have at our disposal and make it work like a hunk of silicone manufactured by the lowest bidder. Do we need to talk like a computer? Yes, a necessary skill set. Understand how a computer ‘thinks’ and how to efficiently utilize its hardware? Necessary skill sets. But to sacrifice our power, creativity and ingenuity in order to dumb ourselves down to the intelligence level of a non-conscious tool? Sacrilege. God is pissed, and rightfully so.

I have suggested the man who wrote the book on test driven practice is in need of refinement. I have compared the highly intellectual endeavor of software engineering to an animalist instinct to feed. And, I have taken a passionate shot across the bow of our educators.

Discussion now as to the merit of these claims is pointless. I will have to show you. I will begin my exploration with a dive into an initial discourse on TDD, the nature of inquiry itself, and a brief touch upon the connection of language and concepts. Next, I will begin to explore how these three venues assisted in the design what has become Analog Driven Development. I will show the evolution of my test first practice from Beck’s work. I will validate the practice through explorations of code repositories built using ADD. And lastly, I will close with some final considerations. But first, definitions. Linguistically, we must be on the same page. Without that, there can be no communication.

*1.2 – Definitions*

Before diving deeper into the meat of this document, it would be beneficial to establish meanings for the words that will be used herein. The following is not a set of meanings from industry or literature, though in some cases they may match. The following definitions are precisely what I mean when I use the words.

*Analog* : involving continuous space; non-digital and therefore computer-less

*Inquiry* : the process of discovery; complete in novelty or an ampliative gain

*Production Code* : ‘source code’; executed code designed to fulfill requirements for a given project

*Programming* : the a process by which a conceptual abstraction is translated line by line into source

code

*Software Process* : methodology by which software is created in an intelligent, structured and

disciplined manner

*Test* : a mechanism for the atomic verification of a single unit of source code

*Test Code* : scaffolding by which production code will be measured and thereby verified

*Test Harness* : collection of tests, generally organized to match the package structure of a source

code itself

*T.D.D.* : Test Driven Development; a software development process employing a test first

programming strategy

*A.D.D.* : Analog Driven Development; a software development process originally fabricated

from T.D.D. in which complex testing scenarios are formally solved with the use of pen and paper

A significantly more detailed breakdown of TDD can be found in section 2.1. The majority of this document enhances the above definition of ADD.

**Chapter Two – Background**

<start with either Beck or Blachowicz, then the other current state of the Art likely then what it’s missing>

*2.1 – Test Driven Development : The First Baseline*

<BECK HERE>

*2.2 – The Nature of Inquiry : The Second Baseline*

James Blachowicz, PhD, author and former professor at Loyola University Chicago provides the necessary foundation for a definition of inquiry. In *The Nature of Inquiry*, Blachowicz suggests that inquiry itself is a dualistic process that mandates “the partial generation from experience of ideas which come to explain experience, and the partial generation from ideas of consequences which come to match experience.” Blachowicz goes on to simplify this definition into a two-sided process involving both experience and thought.[1] One must be able to interact with the known portions of the problem while wrestling with the unknown portions.

This dualistic consideration is important to the definition of inquiry as it contains the necessary pieces to solving any problem, be it in the venue of engineering, mathematics, logic, etc. The need to solve a problem requires one to know various pieces of information about the problem while simultaneously not knowing the problem in some way.[1] (For Blachowicz, this is the first law of inquiry.) Strictly speaking the solution to the problem is unknown, but other pieces of the puzzle may also be obfuscated. This could be a variable’s behavior over time or the effect of multi-variable interaction. However, the problem itself must have a definition. Without a bounding definition, no problem is resolved in an intelligent manner.

Accidental solutions may be discovered for various problems, but for the purposes herein the premise is that we have a specific software problem that must be solved. As such, there is a desired result and intelligent observation of the distance between the known position and the goal can be made.[1] (For Blachowicz, this is the second law of inquiry.) This provides a means for intelligent inquiry. Spontaneous inquiry and randomized creativity is outside the scope of this discourse.

Each piece of software that must be written is a unique problem requiring a unique solution. If a solution to a software problem already exists, generally the problem is not resolved again. Academic settings may require resolutions that have already surfaced for learning purposes. However, in practice, reusability is a primary tenant of development. This focus has a twofold purpose. One, reusing existing code promotes confidence if the code is known to “work”. Two, reuse detracts from overall development time. Therefore, almost every software solution is a solution unique unto itself even if the uniqueness takes the form of refinement. Facebook must only be made once. The database aspect of Facebook remains the same from access medium to access medium. The rendering of that information may change from device to device, but therein lays a novel problem requiring a novel solution. The code executed by my Playstation when I load Assassin’s Creed is the exact code run by every Playstation when Assassin’s Creed is loaded on each gaming console. It would not be Assassin’s Creed unless this held true. It might be a second installment of the game. It might be a similar game. But it would not be the same.

As each piece of software inherently contains the resolution to a novel problem, each piece of unfinished software necessitates inquiry. We must discover the solution to what it is we wish to build. Later I will discuss the location of problem resolution, but for now knowing we have a unique problem. Above, I discussed the ability to both knowing and not knowing the solution to a given problem. Meno’s paradox suggests that this type of knowledge is impossible :

“And how will you inquire into a thing when you are wholly ignorant of what it is? Even if you happen to bump right into it, how will you know it is the thing you didn’t know?”[10]

Firstly, and necessarily, it may be impossible to inquire into a thing that one is wholly ignorant of, for how would one know to inquire of it in the first place? The act of inquiry inherently requires an object. For there to exist a predicate to the question, there must be an acknowledgement of that very predicate. Secondly, this is not the situation being explored. When solving a software problem, one knows what the desired result is. This follows suit with the second law of inquiry. One also knows both the functional and aesthetic portions of the desired result. This knowledge can be converted into a first order map, a mechanism that intelligently determines a specific direction to head when traversing a problem.[1] The software developer also is well aware of several use cases or testing scenarios that ought be passed before the software is completed. Behaviors have been explicitly defined in their end, but not means. This amalgamation of knowledge paints a picture as to what the desired result of the effort is. We know exactly what we want the end behaviors to be and we know how we want the software to look and feel. We do not know how we are going to model those behaviors, their actors or their use interface. We have knowing while not knowing.

Less amorphously, we have two points in a journey to solving a problem. The origin is the current location. The desired result is our expected location upon completion. Simply having a task necessitates that the current location and the goal are not the same. Consequentially, if we were to represent our current location in reference to our desired location in some measurable manner, we would be able to diagnose the differences between the spots. This is a first order map. This is the mechanism for defining the avenue from A and B.[1]

When Kepler began to search for the true orbit of Mars, he began by examining a large number of observations as to the orbital pattern of Mars. Kepler “knew” these observations were incomplete as there was no correct mathematical explanation for the orbit of Mars. The incomplete observations gave Kepler a springboard. Kepler was able to compare his findings with those of others. Ultimately, Kepler was able to resolve the mathematical explanation for Mars’ orbit by figuring out how wrong the current solution was.[1] The resulting solution bloomed from an understanding of what already existed. Known elements helped to prescribe the behaviors of unknown elements. The solution began to betray itself through the observations.

Our definition of inquiry is: a process for intelligent generation of novelty. Firstly, known elements and unknown elements are segregated. Known elements are then reviewed in light of each other. As a conceptual understanding as to their whole is formed, the current assessment of the solution is compared to the desired result. The differential is quantified and systematically dissolved.

Discovery, the removal of unknowns.

*2.3 – The Temporal Relationship of Thought and Expression : The Third Baseline*

Any expression that has not spontaneously occurred from the human must first have been thought. A reflex is an example of spontaneous reaction from the physical body without premeditation. Story writing is not. Even if the story was written as a stream of consciousness, it must occur in the mind before the hand can begin to craft the letters representing the symbols representing the concept or thought that has occurred. This is a necessary tenant of language. Without expression existing within the confines of a mandated form, the communication does not occur. The story of Don Quixote is not a Spanish story. It is a story written in Spanish. Had the author decided to write in Italian, the book would be in Italian. But the essence of the story could remain the same. Perhaps linguistic differences change small portions an event or two. However, the tale of the ingenious gentleman occurred in the mind of Miguel de Cervantes Saavendra before it occurred on paper. I suppose I could be wrong, but I doubt the story was an involuntary set of muscle spasms that happened to manifest into one of the world’s literary classics. Spanish was an encoding of Miguel’s imagination.

Consider the following sentences. Which two are most similar in meaning?

1. “sabe mas el Diablo por viejo, que por Diablo”
2. “the devil knows more because he is old, than because he is the devil”
3. “age breeds knowledge”

Although sentences 2 and 3 are in English and contain similar meaning, numbers 1 and 2 are translations of each other. By definition a translation is a representation of the meaning contained in one language, yet represented in another. Both 1 and 2 contain a force and vivacity that directly compares the Devil’s wickedness to his age in terms of each quality’s ability to correspond to garnered knowledge. Sentence 3 has no such comparison and is therefore the most dissimilar. This example promotes the conclusion that language itself is simply an encoding of a concept. Though this example contains human to human communication in the form of spoken language, this analogy is also observable in congruent software architectures implemented in distinct programming languages. The backend for a website can be scripted in PHP, MySQL or Django while still containing the same database structure. A student versus teacher relationship can be modeled or keyed in any of these.

As the software’s language can be reduced to a simple encoding scheme for a known solution; we can begin to equate the design of a software solution with a problem to be resolved by the conceptual human mind. It is in this realization that both power and flexibility are restored to the human intellect. The search for a solution is removed from the confines of computational logic and Boolean algebra. The animal has been awoken by the realization that the fight has been revenued to the home field of continuous space analysis and playful tinkering. A decisive advantage. Unbridled and rejuvenated, the animal can attack the problem at will and without reservation. Once sufficiently hunted, the solution is transmogrified into the digital aspect of the chosen language, Java, Python, perhaps procedural C. The lines of code themselves will differ. Library imports and custom modules varying from implementation to implementation, but the solution will be translatable nonetheless.

In section 1.2 of this document, I take a moment to lay out some foundational definitions. Words that will be used repeatedly, many of which have evidenced already. In that section I explicitly suggest that for communication to take place, the two humans must be on the same linguistic page. The suggestion here is that the verbal or visual symbols of the language themselves must be the same for both people to communicate. If I say “dog” and you think “cat”, there is a problem. Someone might define programming as: “messing around with code until it works” or “writing software”. These two ideas could be seen as similar, or vastly different. “Messing with code” *could* be analogous to refactoring, or reworking a database schema, or removing a bug. “Writing software”, well that could be an entry level person working at Microsoft, or a student completing ‘Hello, World!’ for the first time. “Writing code” and “engineering software” should never be equated. The act of writing is simply the execution of a detailed intellectual endeavor. However, I do not believe that many ever would have proposed in a formal document that the most challenging software solutions *should* be made without a computer in the room. Or, that programming is nothing more than a line by line translation of that very solution and unfortunately requires a computer. Had I not explicitly set for my definition, you would be using yours… would you not? And that would be a problem. For it is my duty as the communicator to ensure I am as precise in meaning as possible. By giving you my definitions, I have avoided large missteps in the conceptual mapping of the visual and verbal symbols we call words. The odds for successful communication have sky rocketed. You will hear what I want you to hear.

Back to the machines. Consider the following line of code :

“int x = 4;”

This assigns the variable ‘x’ with the value four. Where ‘x’ exists in memory contains a value of ‘4’. What about the next line:

“long x = System.currentTimeMillis( )”

This assigns the variable ‘x’ with the current system time as a ‘long’ which is simply a digital reservation for a number. And this next set of code:

“ for n in Students.objects.all( ):

print n”

This grabs every student in a database and prints them out. These lines of code have no relation to each other. But they have something specific in common. They will each do exactly one thing. If a programmer in Spain wants to assign the value four to a variable named ‘x’ in Java, there is one way to do it. You can ask a human for the time in many ways. “What time is it?” “Do you have the time?” “Got a watch?” Java asks the question in one way. It is possible to design duplicative functions. It is indeed possible to have two different mechanisms for the same input and output pairing. However, there still exists a one to one relationship between the input and the output. One knows explicitly what they should receive as output. This is the very nature of an API. Having erratic behavior is considered a bug and bugs must be expunged. Proper behavior, the way “computers must behave!”, has one input for one designed output.

Now, we have explored the mundane explicitness of programming languages, does that not make them a better tool to solve computational problems? If the computer takes one input and produces a single trustable output every time, is that not better than starting out without even knowing what someone might mean when they use a certain word? A logical objection. But no. A human can look at three apples and divide them among three people. A human can experience the number one-third conceptually. One-third of the pile is given out to each person. A computer has no ability to experience and no ability to accurately represent one-third. Floating point arithmetic is powerful. But let us breakdown those very words.

*Arithmetic* : the branch of mathematics dealing with the properties and manipulation of numbers.

*Floating point* : denoting a mode of representing numbers as two sequences of bits, one representing the

digits in the number and the other an exponent that determines the position of the radix

point

*A sequence of bits*. This mathematical operation transforms a number, inherently an abstract conceptual mechanism for measurement and counting, into a sequence of bits. Computers have finite memory and finite bit reservation for numeric storage. Therefore, a sequence of finite bits will not be able properly represent one-third as point three repeating, 0.333333 ad infinitum. Does this cause a lack in precision that is noticeable? Yes. Banking software consistently wrestles with rounding issues. Flight simulators may be able force a training pilot to experience a simulated emergency. However, the pilot’s life is not actually in danger in the simulator. The *real* experience is not properly mimicked. This is not a discourse on emotional or psychological training. It is a note that computers do not always represent what it is humans experience. Does a machine sometimes represent exactly what a human experiences? I do not know, and nor does it matter. Safety protocols and numeric representation being different are both foundational pieces of software. If it is built upon a lie, then corruption permeates.

Looping back to the most recent objection to my course of thought, is it not better to just work in a computer’s digital mind set? No. Most recently, it has been discussed that computers are not always able to represent the reality of a concept. They are also not always designed to represent reality. We could build a flight simulator that kills a pilot if he or she does not act appropriately. But we do not. There has been a discussion as to nature of implication and how it affects the similarity between statements. And there has also been an exploration as to the difference in intellectual capacity of humans and computers. A computer cannot create. A machine cannot ponder, nor argue. It can only calculate by the rules we give it. We are imperfect creatures that produce less perfect creations of our own. Engaging a problem by the rules which we teach our less perfect creations seems a bit unwise from a gut reaction. The option is appealing because it appears to simplify the problem. “*I just need one line of code to print something I think. Let me start printing things… now where’s that line of code? Wait what am I printing again? Should I string literal or just string variable?*” When we reach the end of this example, we see the mess we’ve created. Digitally effected thought. No civil engineer allows the construction workers to begin building until she knows where every piece is going. The conceptual design, the theoretical mathematics of the beam structure, and the aesthetic placement of non-function accoutrements are all in place formally. And this is done before a single construction worker shows up on the job. The software solution ought to be discovered before the programmers show up to put it together.

**Chapter Three – Implementation**

<similarities to TDD, differences from TDD, VERSUS examples, SCOPE examples>

<Inherited logic from NoI; First order map references>

<How to>

Human is weak, tempted by machine. Therefore no computers in the room

I believe that software engineers would enjoy the ability to sketch out all their work in the shade of an old oak tree overlooking a river or pasture. Perhaps from the top of a building visualizing the moving traffic as data transactions. Then, we could tidy up our sketches when we felt sufficiently done. Finally, toss the sketches to a machine that translates the continuous space solution into digital form, source code.

**Chapter Four – Validation**

<All the wicked cool shit ADD does. List it, prove it. Get on with it>

**Chapter Five – Conclusions**

<Future work would be arrogant here. The process itself needs to simply be refined but to be refined it has to be used. I need to screw it up. I need to make it messy. I need to break it, so I can make it stronger>

<basic discipline yields marvelous results>

**References**

[1] J. Blachowicz, *Of Two Minds: The Nature of Inquiry*, State University of New York Press, Albany, NY 1998

[2] K. Beck, *Test-Driven Development by Example*, Pearson Education, Inc., Boston, MA, 2003

[3] W. Strunk, E.B. White, *The Elements of Style*, 15th Edition, Pearson Education, Boston, MA, 2009

[4] M.J. Swann, *Tailoring a Pre-Existing Software Process: A Logical Interface Between the Intellectual and the Empirical*, Auburn University, 2011

[5] F. Keenan, *Agile Process Tailoring and probLem analYsis (APTLY)*, Proceedings of the 26th International Conference on Software Engineering, 2004

[6] I. Yoon, S. Min, D. Bae, *Tailoring and Verifying Software Process*, Software Engineering Conference, 2001, ASPEC 2001 Eighth Asia-Pacific

[7] V. Basili, H. Rombach, *Tailoring the Software Process to Project Goals and Environments*, Proceedings of the 9th International Conference on Software Engineering, ICSE 1987, Review by B. Boehm

[8] T. Ashley-Farrand, *Shakti Mantras*, Random House, New York, NY, 2003

[9] J. Swift, *A Modest Proposal*. 1729. Quotidiana. Ed. Patrick Madden. 19 Dec 2007. 10 Jan 2014 <http://essays.quotidiana.org/swift/modest\_proposal/>.

[10] R.S. Bluck, *Plato’s Meno*, Cambridge Press, Cambridge, MA, 1964

**Appendix Alpha**

**Appendix Beta**

**Appendix Gamma**