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 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*

1.3 *Future Lab : The Venue of Advancement*

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 First Premise*

2.4 *Conceptual versus Mechanical Representation : The Second Premise*

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**

This document will explore the following perspective on software engineering: the discovery of a software solution does not necessitate the presence of a computer. Software itself is a logic game. Control structures, loops, and Boolean variables are all evolutions from logic. Mathematical computation is based in logic and reasoning. And logic is inherently a mental game. As such, the totality of a software solution can be forged without a computer. Of course, the solution must be run on the computer to indeed be software. But the solution itself may be worked out in the natural continuous space venue of the conceptual human mind.

To validate what I have discovered, I will gather significant portions of my education and life experience. To prove computers are unnecessary to solve software problems, I must step outside of mainstream computer science thought. I will combine cognitive psychology, formal logic and current engineering practices to explain this mechanism. This document exists for the sole purpose of exploring the resultant. A combination of the arts and the sciences. A combination of creativity and structure governed by the laws of logic. A combination of my years earning a Bachelor of Arts and now a Masters of Engineering. The fusion allows me to be better than had I only studied one, arts or engineering. This is my work and this is my proof.

*1.1 – Of Predators and Prey*

The process of software development can 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 in the realm of software engineering is a chronic illness more so than a temporary cold. It is easy to see the potential for this struggle when one acknowledges both the human mind and the human capacity for problem solving are both 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, but uses procedural execution. Stream of conscious thought is flighty. The manifestation of action is inherently different between the mind and the machine. 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 permit the translation between a human’s idea and a computer’s behavior to be obvious. Not directly. It may, however, be the most empowering consideration 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 is tasked with modeling the elusive behavior and ensuring its stability. And the hunt itself is consuming. It is important to remember the nature of the human species. We are a living, breathing, imaginative construct. We strive and struggle and cry. We find an obstacle and we have to climb over it. We want to win. By design, we have two lungs, two kidneys, a pair of eyes, and the instincts to eat and to procreate. We are animals. We are proof of life. We have so little in common with computers, our creation.

Even in our intellectual endeavors, we evidence animalistic tendencies. The hunt for a solution produces bio-physical responses. Like a lion trying to feed the pride, our hunt for answers causes our hearts to race, our breathing to change, our senses to sharpen. As he closes in on his prey, the feline can taste success. He can smell the prize. For humans, intellectual inertia torrents through the human mind as fiery blood courses though veins. But in a single moment, a minor misstep can scare off the trophy leaving the human bereft of energy, dissatisfied and hungry. As humans, we might ‘*throw in the towel*.’ A machine will never ‘*call it quits for the day*.’

In order to better secure any target, humans attempt to refine their strategy. The refinement may originate from a desire to reduce requisite effort or to pursue perfection. One such software 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. 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 the related tests first, then when we write the code we’ll know it is correct when it passes the tests.

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 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 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 barrowed pens feverishly scribbling on the back of restaurant napkins. The eureka moment takes place surrounded by eager onlookers and friendly waitresses as the human mind finally solves the puzzle. The hunt ends as the predator crashes down upon his prey. ADD will remind us of our human strengths. And ultimately, ADD will dispose of the unfortunate thought that unfortunately curses the minds of our students.

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

This sentence 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? Never. We are living creatures with inherent strengths. Our abstract thoughts, our ability to ponder in continuous space, our analog conceptualizations are how we naturally process information. We do not think in machine code. We think with symbols and representations. There is no resolution on the mental image of a chair. No refresh rate on our memories. No video card upgrade to make a mental image clearer. It just is. And that is what we are, thinking animals.[12] We are not machines. Our process for creation should be designed with that in mind. We ought play to our strengths.

Discussion now as to the merit of this mechanism is pointless. I will have to show you. I will begin my exploration with a dive into TDD, the nature of inquiry itself, and a discourse on the connection of language and concepts. Next, I will explore how these three venues assisted in the design off what has become Analog Driven Development. I will show the evolution of my test first practice from Beck’s own work. And, I will validate the practice through explorations of code repositories built using ADD. But first, definitions.

*1.2 – Definitions*

Before diving deeper into the substance 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; the nature of a concept that does not yet

exist in a manifested form; *i.e. the idea of a chair versus the mathematical height and weight bearing properties of the chair you are sitting on*

*Digital* : involving discrete space; non-continuous; explicitly manifested version of a concept

*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.

Highlighting the differences between the analog and the digital is important for continuing conversation. Analog representations have meta-connections and conceptual implications that digital representations do not exhibit. Justice, for example. A just action exemplifies what it is to execute justice; however, there is more to the idea of justice than can be fit into a single action. Justice entails a legal system, ideas as to fairness, religious connotation, etc. This is just a common set of American concepts that float to the surface of the mind when justice is thought upon. And this is the analog representation of justice. Thoughts of famous persons who have spoken on justice, things we have been taught about justice. The collection of our knowledge on the topic does not exist in any one story or example. Our human understanding of justice is an amalgamation of experience, time and exposure. And our analog definition of justice is further refined by digital manifestations. Again the distinction in based in the segregation of idea versus example; of rule versus instance. Similar to the design of a class structure and the impending instantiation of a discrete object. Software too has analog and digital portions. Humans were designed to execute action based upon a collection of information. Computers were designed to execute an action based upon what a variable means at a particular moment. Computers view the here and the now only. One single set of instructions to be executed right now under current circumstances with no regard for historical knowledge or future consideration. For the computer, the action has no place in a conscious history. Humans take more into thought. For a human, every action can become a memory, or a fear, or a hobby. Novelty can be attached to growth or self-degradation. For the human, concepts and thoughts are naturally connected. Computers must have a series of actions explicitly pre-determined as no action a machine executes promotes any cognitive inertia. This is the functional difference between analog and digital.

*1.3 – Future Lab : The Venue of Advancement*

Analog Driven Development was developed out of need in connection with the FutureLab Project at Auburn University. The project’s goal was to reversion a piece of educational science software for middle school students. The program contained several experiments that can be worked through; balancing a scale with weights or freefall from a platform. The benefactor, Auburn Engineering alum Walt Woltosz, ’77 Aerospace Engineering, donated the original FutureLab program. FutureLab, the software, would begin to undergo a rebuild from the ground up. The program was being moved C to Java and from mid 1990’s operating systems to the Android OS. FutureLab, the project, would provide a number of new challenges, the most pertinent of which required the design of a homemade physics environment.

The physics designed for FutureLab required that any object to be simulated also be mapped to a location within the designated simulation space, the container where the experiment takes place. This space is an abstract grid system layered on the pixel display of the Android screen itself. Each point within the grid system, a measurably discrete location. Points within the simulation space are combined with metadata into a node structure. These nodes are connected via pointer chain creating enclosed shapes. Shapes congregate into the skeleton of simulation objects: a ball, a standard mass, a cannon, etc. Further metadata is combined with the shape: mass, labels, acceleration vectors. The laws of physics are recapitulated into mathematical functions and managed by event watchers. All of this taking place within an abstract grid system.

ADD evolved into a mechanism for stable software design, but it began as a resource. The development of tests for a system that models physics in an abstract space proved difficult. In order to keep track of the location of objects and their shapes within the simulation space, I began to sketch a blue print for each test. Engineering paper, rulers and a compass became power tools software construction.

**Chapter Two – Background**

Chapter Two explores the foundational elements encompassed by analog development. In terms of primary function, ADD is a test first design paradigm that evolved from Beck’s test driven process. ADD is also a process of intelligent discovery. Analog development, as mentioned above, was originally a product of need. However, as the development tool was refined, ADD itself was intelligently re-designed. It was crafted with acknowledgements as to the strengths and weaknesses of the human mind.

There are several building blocks for the analog process. There are two baselines, each representing a single thread of thought from a major field of study. The first baseline, test driven development, offers the primary goal of this effort: to improve the manner in which software is made. TDD also provides the scaffolding and organization for the analog process. This baseline originates in the field of engineering. The second major building block comes from cognitive psychology. Inquiry ought to be precise and intelligent. The second baseline provides an understanding as to the author of the software. The third piece is a catalyst. Disciplined execution of logic will mix knowledge of engineering practice, software design and intelligent discovery. There are two premises: 1) thought takes place before speech; and 2) humans ‘*think*’ differently than machines ‘*think*’.

Analog development is made of all these pieces. The two premises will serve as a spring board into the mechanism’s very development. TDD will provide the *what* and the *why*. The nature of inquiry will provide the *how*.

*2.1 – Test Driven Development : The First Baseline*

Beck’s desire to overhaul the development process appears to stem from the work environment of a less disciplined era of software engineering. He references a time when testing was not a part of every team’s process. A time when a developer would have to wait through the night to see if the tests passed or failed. Software development was not as comfortable as it is today. One of the primary issues was the human’s confidence in the software artifact itself. Testing and quality assurance practices had not begun to bloom. As such, the industry needed some thought on how to improve quality.

A snapshot of the test driven development mechanism is as follows :

1. Make a test,
2. Run all tests; verify the new one fails,
3. Make a change to the source,
4. Run all tests; verify they all pass,
5. Refactor to remove duplication.[2]

This multi-step process is cyclically repeated. To build a new piece of source, you must have a test that fails because the source is not written; to build that test, you must first pass all other tests. This structure mandates a certain level of stability. If the last function install fails the related tests, that function and the related test must be examined for defects before anything new can be generated. Also, if the last function install has a negative impact on other pieces of the software, running the entire test battery will evidence the problem. This structure provides a very controlled environment. Throughout his exploration of TDD, Beck identifies with the human side of engineering. He goes so far as to say “If you’re upset, take a cleansing breath.” He then goes on to explain how to do so. Beck understands the strain of a negative work environment. So, he hands us two mechanisms for confidence and stability. One related to easing the emotional uncertainty by reducing the number of unknowns in an ongoing project; the other a respiratory mechanism to calm bio-physical tension.

The test driven process itself is not complex. However, it can be tailored. Beck suggests one start small. Test for a class that has yet to be created. Fix the error by installing the class. Then test the constructor of the class to see if a certain value is set. Fix the error by going back and setting the value in the source. Each step has a small test for a small be of code. As the developer becomes more familiar with the process and how they intend to self-tailor, larger tests can be written for much larger installations. These tests can be as small as an assertion on the return of a ‘getter’ function, or as large as the output of a database query.

<*Inustry notes on TDD -- unfinished*>

The process allowed for great strides in the development of quality software, but the mechanism has a greedy heuristic to the design. It is quite powerful, but places do exist for refinement. Most notably, TDD requires a certain amount of duplicative effort, “…speed trumps design, just for a brief moment.”[2] But this brief moment happens once per cycle. Long term design decisions do not receive much conversation. Alterations to existing code are a necessary evil. These tedious changes are the price paid for confidence and stability. Refactoring is a must. Common code spread across same depth in class structure ought be transplanted to higher, more appropriate tiers. Support functionality that can be refactored often requires the generation of related tests. This upkeep is necessary and often temporally random in nature. And as such, emotional flow of development can be turbulent.

When Beck moved test based activities to the front of the process queue, he effectively overhauled the system. Traditionally, production code was written and then tests were created to exercise the source. But this organization also acknowledges a tenant of human behavior. We will always do what we have to do; we will not always do what we should do. The production code has to be finished. The test code does not. If the test code is written first, it cannot become a cut-corner. Also, pre-emptive generation of test code would not be subject to the biases of having already written the source code which in turn would have to pass the tests. Test driven development also began to answer the question, “*When is this done?*” Well, this small piece of code is done when it passes the test over there that is ready to run. By re-structuring the process, Beck gave us the ability to generate small milestones over the course of a large project. Creating an entire database takes time. However, making one table or one query at a time, that removes the emotional gravity of a long term project. That allows for better focus. That is an example of understanding the human

*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 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. As exposure window to the object of inquiry is lengthened, conceptual understand of the object is refined.

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. Examples of obfuscation are: a variable’s behavior over time, and the effect of multi-variable interaction. However, a starting point is needed. 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. (For Blachowicz, this is the second law of inquiry.) This provides a means for intelligent inquiry. Spontaneous discovery and randomized creativity are 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. If a program can be purchased for seventy dollars, it likely took more than seventy dollars of effort and time to produce that program. 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. Metaphysically speaking, 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 is sufficient. Above, I discussed the ability to simultaneously know and not know 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, we are not inquiring into something we are wholly ignorant of. When solving a software problem, one knows what the desired result is. This follows suit with Blachowicz’s second law of inquiry. One also defines 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 to 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 user interface. We have simultaneous knowing and 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, he 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 First Premise*

Any expression that has not spontaneously occurred from the human must first have been a 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 which represent the concept or thought that has occurred. This is a necessary tenant of language; words simply represent ideas and concepts. 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 would 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. The story was not produced by an involuntary set of muscle spasms happening 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 or written 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 animalistic human 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. And we know it. Unbridled and rejuvenated, the human 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.

*2.4 – Conceptual versus Mechanical Representation : The Second Premise*

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. For communication to take place, two humans must be on the same linguistic page. The suggestion here is that the verbal and visual symbols of the language must be the same for both people to communicate.[12] 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 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 forth my definitions, you would be using yours… would you not? And that would be a problem. 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 explicitly designed input for one explicitly 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 a conversation 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. Right now, in your mind, you are visualizing one-third of the original pile in the possession of each person. You can see one-third of that pile. 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.[11]

*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[11]

*Sequences of bits*. This mathematical operation transforms a number, inherently an abstract conceptual mechanism for measurement and counting[12], 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. Safety protocols and numeric representation are both foundational pieces of software. If it is built upon a lie, then corruption permeates. The conceptual resonance of a one-third fraction contains the ideas of division and infinity. The digital representation of a one-third fraction, by definition, is a sequence of bits. There are no conceptual links between 1’s and 0’s. This implies a fundamental difference between machine and human processing.

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. Machines also do not encode ideas with conceptual bridges. A computer will not understand an analogy. A complier cannot say “I bet the human meant to make this Boolean true and not false. Silly human. Let me fix that.” There has been a discussion as to the nature of implication and how it affects the similarity between statements (above in this section). 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. Though, 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 have 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-functional 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. Beck’s test driven development was a major step. It is time for the next step.

We are not going to design a single atomic test. We are going to sketch entire classes, sub-packages, even full packages of source code. Up to seventy percent of time can be spent with a pencil and paper. Not as many long days with eyes bloodshot and tortured by computer monitors.

**Chapter Three – Implementation**

Ultimately this chapter will explain the step by step process of analog development. We will finally have the promised mechanism capable of designing and verifying large piece of software without a computer. First, the baselines for the mechanism must be revisited. The background for each field has been put forth. Now we must grab the specific pieces needed to build the new process. TDD provides the ending goal and the base structure. Cognitive psychology provides acknowledgement as to our inherent strengths and weaknesses. Reasoning will marry the two.

*3.1 – TDD Influences : Similarities and Differences*

Analog development borrows significant knowledge from Beck’s test driven process. A number of similarities and differences can be found between the two methodologies. The primary goal of the each mechanism is the same: to produce stable, reliable and clean code. The aesthetic structures of both processes look similar in their coded forms. However, the step by step details vary to a large degree.

Each test created using both methods contains the same three building blocks. The first is an arraignment of objects. This is what I call ‘the setup’ or ‘the lineup’. This is an organization of all objects and data types that will participate within the test. The second common piece is the behavior being tested. Generally this can be referred to as the ‘target behavior’. This can be a script, a formal function, a database query, anything that encapsulates machine behavior into a custom call of some kind. Lastly, the assertion of the behavior is a part of every test. Ultimately, this is the purposes for the test, to ensure a behavior is being executed and that the resultant is appropriate. This can be ensuring a mathematical calculation produced the proper number or that a database query fetched the proper data entries.

The test suites created from TDD and ADD are also aesthetically similar. The organization of the suites themselves reflects the package structure of the source code in both instances. A database modeling an academic environment will have an individual package for the human classes and tables: students, professors, administrative staff, etc; while another package exists for the modeling of courses, majors and degrees. Each of these packages will have unique test suites that internally organize and breakdown classes into atomic behaviors. The greater test harness for the entire artifact is comprised of these package specific suites.

The primary differences between ADD and TDD begin to evidence when the immediate purpose of each method is explored. TDD does not strive for perfection, and TDD promotes speed and simplicity over performance and accuracy. As such, TDD requires a number of ad hoc repairs to both test and source implementation over time. ADD attempts to answer every question as to every component and behavior before any code is written. As such, ADD requires very little upkeep in terms of modifying code so long as the design appropriately matches the requirements of the project.

Beck suggests that in the natural course of TDD one cannot rely on the idea that appropriate flashes of insight will occur at the appropriate times. This is true. Temporally harnessing insight is not practicable. As such, TDD ventures forth making the best decisions that can be made at the time and writing the line that seems appropriate. ADD writes no lines until the unknowns are resolved. And therein lays a huge drawback to the new method. It can find more excuses for decision making. However, this is why there is less rewriting in ADD.

The measurement of the numerical difference between the two becomes difficult to calculate. The obvious question, which is faster? Firstly, ADD designs, tests and implements in much larger chunks of code and time the TDD. Properly dissecting the time needed for each phase within the larger pieces that ADD tackles, is possible over several experiments, but both tedious and delicate. However, the benefits of in depth design and test documentation are difficult to quantify, as are the benefits of dividing a complex problem into two more distinct problems. In the case of FutureLab, ADD was developed because TDD was insufficient. Building an environment that calculated collisions of organized pixels with individually representative mass values on a synthetic grid system required a test driven process that relied on design work. There were too many variables to keep aware of. So I drew it. For now, that acknowledgement will suffice as the strongest indicator of ADD’s value. It was a tool that successfully allowed an engineer to escape a cognitive limitation: simply keeping all the variables straight.

*3.2 – Influence of Intelligent Discovery : First Order Mapping*

Section 2.2 introduced the idea that the analog solution to the software problem would be a map, a first order map. The first order map, as defined by Blachowicz, is a mechanism that intelligently determines a specific direction to head when traversing a problem. This map is built from the conceptual understanding we have in respect to the problem’s variables. In inquiry, this map is rather dynamic in nature. Information is collected over time. A growing knowledge of the problem’s input variables will change the landscape of the known problem. Second, potentially third order maps are needed for dynamic problems. In this instance, we have a very well defined first order problem. We have a software idea that needs to be manufactured and someone knows what it should do. The problem becomes, “*How do I get that guy’s idea out of his head… and into that phone?”* Unless the desired behavior of software changes, the software map can be created just once.

The requirements of the project are transformed into test cases. This process will be further detailed below. The source code must ultimately pass each test, though the form of the code is currently unknown. This ties back to the first law of inquiry: *knowing while not knowing*. A sketch of the resolved test cases for a sub-set of the code provides the first order map. This sketch is a human encoding of the problem space and gives a *digital* resolution of a singular instance of the greater behavior stream. This ties back o the second law of inquiry: *determining proximity to the target*. If we create an instance of a database that contains pseudo-random data, then we can design a query we would like to have and test its functionality against the test data.

The map’s integrity is verified though implementing the source code itself to a passing state. If the test code is translated from a sketch of the solution space, which is in turn reflective of the desired behavior of the end game software, then one can be confident that the source acts as intended if it passes those tests. Each time a new test is passed, we have garnered ampliative knowledge of the software solution. We now know how to get the desired behavior because we arrived at the desired result which was passing the tests associated with the behavior. We have not only discovered the solution but have also adhered to our own definition of discovery.



Figure 3.1 – test\_collision\_31\_01\_00\_check

*3.3 – ADD : The Process*

The primary question answered by analog development is: *How confident am I that this software package will work as intended?* Software development requires an understanding as to how the individual pieces of the project will interface. Having a collection of code segments that work independantly, but do not work together is unacceptable. The analog process conceptually unites individual modules into a global picture. This is accomplished by creating a visual solution.

Analog development follows the following steps:

1. Escape technology;
2. Intellectually prepare;
3. Sketch the problem and a potential solution;
4. Repair the solution; and
5. Pseudo-Code the solution.

In order to illustrate these steps, an example from FutureLab will be explored. The visualization was drafted to depict the algorithm created to check for collisions of shapes within the simulation grid. In FutureLab, each object within the simulation space has a shape. That shape is used to determine the boundary of the object. Collisions are detected by determining whether or not one shape has touched another. The sketch (Figure. 3.1) shows a number of individual use cases:

1. Homogenous collisions (*i.e. Circle vs Circle, Square vs Square*);
2. Heterogeneous collisions (*i.e. Circle vs Square*); and,
3. Non-collisions.

Figure 3.1 has a significant amount of information. First off, the top right corner of the document contains the test scenario’s date of creation and name, test\_collision\_31\_01\_00\_collision\_check. The top left of the page contains collision types by shape: those involving circles and those involving squares. The central portion of the page depicts a *potential* state of the physics simulator itself. Each shape has a name and a location. And each can be seen to touch, or collide, with various other shapes within the sketch. The formal test created from this visualization can be found at the end of the section in Figure 3.2.

*Stage One – Escape Technology*

Stage one, along with stage two, immediately addresses the nature of software’s author, a human, a thinking animal. Humans are habitual creatures and software developers spend a lot of time in front of a keyboard. It is easy for a developer to jump to a computer to “*check something*” during a design meeting. A natural tendency exists to spontaneously seek verification through a tool humans are familiar using. This stems from an emotional trust in the machine. In order to remove this crutch, it is easiest to move away from the computer altogether.

*Stage Two – Intellectually Prepare*

The second stage can be ritualistic. Psychology has proven that working in a single environment improves recollection. A student taking an exam in the room where lectures were given has a greater capacity to recall pieces of information that are difficult to recollect. Unifying environment with intellectual work or practice can have positive side effects. This stage can be executed as part of a weekly or daily schedule. The purpose here is to begin churning cognitive inertia. This can be accomplished through several means, or a combination thereof: review of pre-existing design documents, review of meeting notes and sketches, or mentally walking through the requirements for the given package. Even a simple phone call to a fellow developer to ask for clarification on a feature can begin to build intellectual steam. This preparation is synonymous with an athlete stretching before a game.

This preparation requires the developer ask two questions. One, “*What am I building?*” In this instance, an algorithm that will check for collisions within a simulated physical environment. Two, “*Why am I building this?*” This particular algorithm was needed in order to collect information regarding which objects were colliding. That information is turned over to another algorithm that executes the necessary calculations of impulse acting on objects found to collide with one another.

*Stage Three – Sketch the Problem and the Potential Solution*

Stage three involves sketching the problem into a solution. In this stage, the developer will take up pencil and paper and draw the answer to the problem. Returning now to figure 3.1. This test scenario attacks the use casing associated with checking for collisions.

As givens, we have a class representing a Circle, a Square and an abstract-class representing a generic event within a physical environment. This is the Event\_Interaction class. Collision has been scheduled to derive off of Event\_Interaction, and as such will inherit a method to check the simulation space for the particular event. In this case, the check method will look for any collision. The Circle class is an aggregation of data: a central point where the object is fixed on a grid system, a shape with a radius, and some miscellaneous fields for user interface display and other administrative purposes. The Square class is quite similar, both being derivatives of a higher level Shape class.

First, the grid system is marked off on engineering or graph paper. Arbitrary values may be applied to the grid. However, having the origin near the center of the paper promotes simpler math by hand. Then the Circles can be drawn on the paper in such a way that they collide (*Alpha vs Beta*). Their positions are determined. Other shapes are drawn to collide (*Delta vs Iota*). *Eta* and *Gamma* are not colliding with any other objects. However, the algorithm will have to be sure not to flag these for collisions. Their presence is necessary.

Once the objects are placed upon the grid, their locations can be explicitly determined. When drawing the sketch, it was most important to evidence as many use cases as possible. The specific location of each shape is now important for tracking purposes. The result of the collisions in terms of the software use casing was a primary goal. However, the focus has turned to the specifics of the test.

*Stage Four – Repair the Solution*

Stage four involves a review of the initial sketch in light of the project requirements through the use cases. Various sketches may be required to depict all use cases. However, each use case must be accounted for within the set of visualizations. If a use case has not been translated into visual form, for instance a collision between a circular object and a square shaped object, then this interaction must be drawn. There are two purposes to stage four. The first is to ensure requirement coverage. The second is perform a moment of self-reconnaissance. Stage three requires the depiction of the problem space as it is currently understood by the developer. Stage four requires that the initial depiction be repaired for any missing considerations. The developer is able to gage their level of comfort with the problem. The transition from phase three to four allows one to determine any discrepancy from their current knowledge base to the one they ought have.

In the instance of figure 3.1, heterogeneous collisions are important. Had the collision between *Alpha* and *Delta* been missing from the visualization, such a collision would need to be created. A review of use cases will determine the completeness of the set of visualizations.

*Stage Five – Pseudo-Code the Solution*

Stage five begins the transition from human thought to machine code. The algorithm crafted to check for collisions within FutureLab must be able to pass every collision instance drafted as a test. The conceptual logic for the algorithm can be deduced directly from the sketches. A collision will not occur between *Eta* and *Gamma* because they do not touch. However, a collision will occur between *Alpha* and *Beta* because they do touch. The properties of the shapes themselves begin to betray the solution into light. In the case of two Circles, if the distance between their central points is less than the combined length of their radii, then a collision has occurred. If a Square is involved in a potential collision, one of its vertices may be inside of another shape (*i.e. the bottom-left corner of Delta is current inside of the area contained by Iota*).

An initial draft of the algorithm might look like:

1. F(x) collision\_check( Shape shape\_one, Shape shape\_two )
2. result = false
3. if shape\_one && shape\_two are Circles :
4. distance = calculate\_distance( shape\_one.location, shape\_two.location )
5. if distance < ( shape\_one.radius + shape\_two.radius )
6. result = true
7. else :
8. for each vertex of shape\_one :
9. if the vertex is inside shape\_two :
10. result = true
11. return result

This first pass of pseudo-code will often be incomplete as a fully functional algorithm. However, it can be amended in time. The purpose of drafting this algorithm is to peek at what support pieces are needed.

Before the first draft of the collision\_check algorithm, the developer may or may not have known that determining the distance between two points was a necessary sub-routine of (*line 3 of the above*). The developer may not have realized the need to have a sub-routine check for the presence of a single point within another shape (*line 8 of the above*). These support functions can be visualized as well. Figure 3.3 shows the depiction of the algorithm which checks whether or not a given point exists within a shape. Figure 3.4 is the digital aspect of 3.3. These can be found at the end of the section.

Within Stage Five, the ampliative nature of analog development can be witnessed. As one problem is solved, others come to light. The generalized function designed to check for collisions needs support the developer was unaware of before exploring the problem. The knowledge gain associated with the functionality of the software itself can take a on a life of its own. As the pseudo-code for a given scenario is written, the human can begin to sketch the support functionality. Or, whatever greater function will use the currently drafted algorithm. Conceptual granularity can take either course.

*The Aftermath*

Aftermath includes the activity of translating the sketches into encoded test scenarios. Nothing of consequence will be added to the *intent* of the sketch, which is to resolve a given instance of a collision. The test will carry the same *intent*, to resolve an encoded version of that very instance. The ‘*digital’* aspect of figure 3.1 can be found in figure 3.2.

Secondly, this involves the creation of production code to pass the newly drafted test scenarios. The purpose of analog development is to create a robust test harness that sufficiently exercises project uses cases. A custom environment exists as a digital scenario to be resolved by source code. Assuming proper encoding of the test sketches themselves, code that properly resolves the digital scenario confidently meets the requirements of the project. This is because requirement has been transformed into explicit cases, each case into a piece of a test. Passing the tests passes the requirements.

Within this post-mortem phase, it is possible that defects in the sketches will evidence. Missing or incorrect considerations require the sketches themselves to be amended along with the encoded test scenarios. This is often due to an error related to project requirements; not having them all, not understanding them correctly, etc.

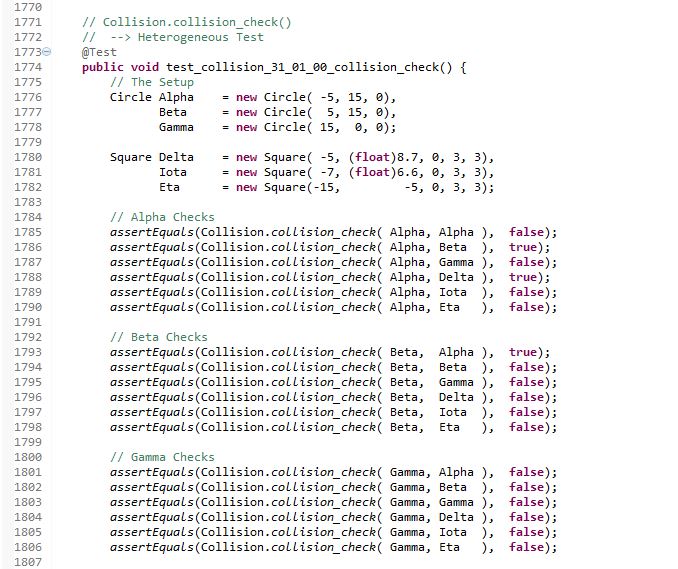


Figure 3.2A – ‘Digital’ aspect of test\_collision\_31\_01\_00\_collision\_check

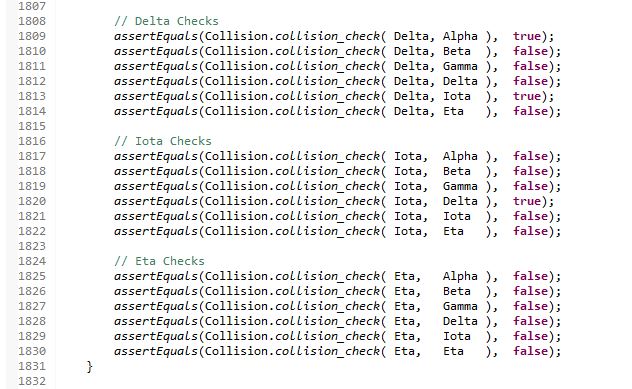


Figure 3.2B – ‘Digital’ aspect of test\_collision\_31\_01\_00\_collision\_check

**Chapter Four – Validation**

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

<Notes>

Creating the map begins by defining the explicit behaviors of a software project. Methods for requirement elicitation are beyond the scope of this discourse. Once collected, the desired functionality can be categorized and broken into similar pieces. Packages and sub-packages begin to define themselves naturally through the human mind’s ability to recognize similar features and themes.

Once the general behavior categories are isolated and inventoried, they can be transformed into ……

<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.

In order to resolve a problem, it must first be defined.

Do we really need to build a bridge, or do we need to find a way to get people to food on the other side of the river? What do I want to do and why do I want to do that?

**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

[11] Oxford University Press, *The Oxford English Dictionary*, Oxford University Press, 2013

15 Jan 2014 < http://www.oed.com/ >

[12] F.M. Cornford, *Plato’s Theory of Knowledge: The Theaetetus and the Sophist,* Dover Publications, Mineola, NY, 2003

[13] website for FutureLab

[14] G.L. Murphy, *The Big Book of Concepts*, MIT Press, Cambridge, MA, 2004

**Appendix Alpha**

**Appendix Beta**

**Appendix Gamma**