7-2019

Developing 5GL Concepts from User Interactions

David Stuckless Meyer
Grand Valley State University

Developing 5GL Concepts from User Interactions

David Stuckless Meyer

A Thesis Submitted to the Graduate Faculty of

GRAND VALLEY STATE UNIVERSITY

In

Partial Fulfillment of the Requirements

For the Degree of

Master of Computer Information Systems

Padnos College of Engineering and Computing

July 2019

“[Fifth generation languages] focus on constraint programming. The constraint programming, which is somewhat similar to declarative programming, is a programming paradigm in which the programmer only needs to specify the solution to be found within the constraints rather than specifying the method or algorithm of finding the desired solution.” (Balagurusamy, 2009, p. 340)

Abstract

In the fulfilling of the contracts generated in Test Driven Development, a developer could be said to act as a constraint solver, similar to those used by a 5th Generation Language(5GL). This thesis presents the hypothesis that 5GL linguistic mechanics, such as facts, rules and goals, will be emergent in the communications of developer pairs performing Test Driven Development, validating that 5GL syntax is congruent with the ways that practitioners communicate. Along the way, nomenclatures and linguistic patterns may be observed that could inform the design of future 5GL languages.

Table Of Contents

INTRODUCTION

Fifth generation, or Constraint Programming (CP) as it is also known, is a concept within the field of computer science, asserting that: given a complete description of a problem space, the computer can be capable of working out its own solution that is no less efficient than a solution that a human practitioner might work out (Freuder, 1996). In practice, the creation of such a solving engine program that can handle very large and complex problem spaces has proven difficult (O'Sullivan, 2012). Furthermore, the specification of the problem space can prove more complex than the problems they contain.

> *"Constraint Programming represents the Holy Grail of programming: the user states the problem, the computer solves it"*
>
> Eugene Freuder, Director of the Cork Constraint Computation Centre

The specification of a problem space, for a 5GL, is achieved with a certain style of syntax. This syntax enables developers to express functional constraints in a way that is understood by the machine. But with a more advanced solving engine, it may be possible to relax the syntactic constraints and allow the humans to use a more human language - dramatically reducing the cognitive load required on the part of the humans.

This raises the question of how developers would express constraints in the absence of such restrictions on syntax.

Lacking such an advanced constraint solver, this part can instead be played by another practitioner. Leveraging this, our study was executed under the guise of a series of pair programming exercises, leveraging Test Driven Development (TDD). In this programming exercise, the test code produced by pairs of programmers forms a set of expectations, or constraints, for the production code similar to 5GL syntax. The code that fulfills those expectations is then capable of producing the same results as the constraint solver, but with more utility given the artifacts that are produced in the process. In an effort to simplify the data collection process, test participants were asked to communicate solely through the code, via an online collaborative code editing application, and were randomly paired for each exercise in an attempt to remove contextual bias[1].

In the fulfilling of the contracts [2]generated in Test Driven Development, a developer could be said to act as a constraint solver, similar to those used by a 5th Generation Language.(5GL) We therefore hypothesize that Fifth Generation linguistic mechanics, such as facts, rules and goals, will be emergent in communications between a pair of developers performing Test Driven Development, validating 5GL syntax as congruent

[1] Bias introduced by the context in which the study is undertaken, if a pair is sufficiently acquainted, then there will be contextual data to their communications that is not captured.
[2] Tests in Test Driven Development can be seen as contracts for anticipated functionality.

with the ways that practitioners communicate (Freuder, 1996). Along the way, nomenclatures and linguistic patterns may be observed that could inform the design of future 5GL languages.

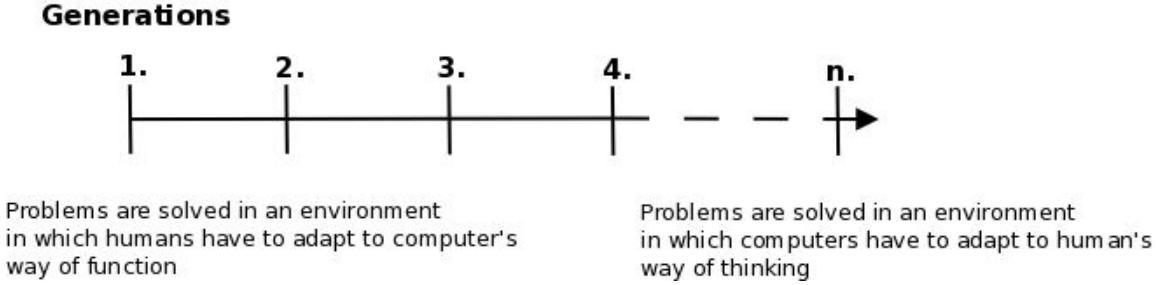
Inspired by Jakob Nielson (2009) in his discount usability technique for user interface design research, this project seeks to further the field of formal linguistics within computer science by applying a human-as-machine-analog concept, to the study of programming languages and developer productivity, in an attempt to find ways in which humans naturally express nuance which could be leveraged by fifth generation languages.

REVIEW OF LITERATURE

Generational Languages

The concept of generational languages was first mentioned at a computer conference in 1963 in reference to higher level languages (Mullery, 1963). The first book that referenced this construct (with the inclusion of the 4GL specification) was *Applications Development Without Programmers* (Martin, 1982). The concept is that progressive generations of computer languages are accompanied by increasing levels of abstraction away from the hardware, freeing the developer to focus more on issues in the customers problem space and devote less effort into translating the problem space into instructions for the machine to execute. (See Figure 1.1)

Figure 1.1



This diagram shows the progression of generational languages. (Sutten, 2010)

First Generation

Direct Instruction: Machine code.

Pure machine code represents the first generation of programming languages (Martin, 1982). With this level of language, code is explicitly non-portable, and no structuring of the code is enforced beyond what the machine itself requires. Programming in machine code requires detailed knowledge of the hardware that is executing the code . A non-trivial amount of effort goes into translating real world requirements into language which the machine can understand (Backus, 65).

Second Generation

Mapped Instruction: Assembly code.

> *"An assembler is a translator that translates source instructions (in symbolic language) into target instructions (in machine language), on a one to one basis."* (Salomon, 1992, p.1)

Assembly languages make up the second generation of programming languages. Theoretically, assembly languages offer cross-platform capabilities, however, because they are mostly a one-to-one mapping of human readable tokens to machine code, very few useful programs can be written that have portability outside of a family of processors. Today assembly language and machine code are most often associated with embedded systems, device drivers, and highly optimized systems like graphics and heavy processing (Meyerovich, 2013). Once support for different architectures is added

to assembly language, resulting in a non-one-to-one mapping from assembly code to machine code, the language in use becomes a third generation language.

Third Generation

Abstract Instruction: most languages. Focus on detail hiding.

> *"As the problems of building computers were gradually understood and solved, the problems of using them mounted. The bottleneck ceased to be the inability of the computer to perform the human's instructions but rather the inability of the human to instruct, or program, the computer." (Sterling, 2010)*

Generic: Procedural, Object Oriented, Functional and Scripting Languages are all Third Generation Languages. (3GL) 3GLs have a syntax that is considered a good compromise between the way that humans and machines think, yet maintain a syntax that still "bears the mark" of the underlying machine code.

> *"Starting from the language understood directly by the computer, the machine language, better notations and formalisms were developed. The main outcome of these efforts was languages that were easier for humans to express themselves in but that still mapped rather directly to the underlying machine language. Although increasingly abstract, the*

languages in the mainstream of development ... all carried the mark of the underlying machine" (Sterling 2010)

The first 3GLs introduced mechanisms for the management of computer resources as well as portability between different platforms. 3GL's still offer a non-trivial amount of control over the internal processes of the machine through the required use of explicit directives.. Most 3GLs allow the creation of complex tools that can represent a problem space of any size, however their basic implementation remains a small, simple, and a largely deterministic process of translating user instructions into assembly code, setting them apart from Fourth Generation Languages.

Fourth Generation

Task Instruction: SQL.

Fourth generation languages (4GL) allow for greater focus on design problem spaces by being tailored to a specific problem domain (Zhao, 2003). In being more specialized, 4GL implementations (often referred to as Domain Specific Languages) are limited in functional scope, allowing for simpler, more natural syntax that focuses more on describing the problem than how to solve it (Deursen, 2000). This limitation in scope however, must not be mistaken for simplicity, as often the implementation details that 4GLs abstract away are handled by a contextually dynamic execution engine, resulting in non-deterministic handling (many-to-many) of directives such as with SQL execution plans relative to table size.

Fifth Generation

Goal instruction: Prolog

Fifth generation languages (also known as Constraint, Declarative or Logic Programming) are focused on describing the problem space, explicitly leaving the algorithmic details to the evaluation engine to figure out (Freuder, 1996). Most fifth generation languages adopt a style syntax that resembles horn clauses[3], (Sterling 2010; Clocksin, 2003) enabling the evaluation engine to approach "solving" the application as an algebraic proof (Ross, 1991). Fifth generation languages are often associated with AI projects (Fuchi, 1993; Freuder, 1996).

At the time that the generational languages nomenclature was coined, with the anticipation of wide scale adoption of fourth and fifth generation languages, the terminology made sense. Today however, the failure of fifth generation languages to gain and maintain widespread adoption (Meyerovich, 2013) in the face of newer technologies shows that the paradigm of *generational language* fell short as a practical classification system. Most modern languages end up in the 3GL category, and the progression represented therein was not as central to the future of software development as originally thought. This does not, however, preclude the initial premise that being further abstracted away from the implementation details will increase developer productivity (Fuchi, 1993; McConnell, 2003).

[3] Horn clauses can be thought of as boolean equations: $u \leftarrow p \wedge q$
Means: if p and q hold, then also u holds (can be considered true)

5th Generation Languages in Detail

> *"[Fifth generation languages] focus on constraint programming. The constraint programming, which is somewhat similar to declarative programming, is a programming paradigm in which the programmer only needs to specify the solution to be found within the constraints rather than specifying the method or algorithm of finding the desired solution."*
> *(Balagurusamy, 2009, p.340)*

Fifth generation languages are focused on constraint programming (CP), in which the programmer only needs to specify the solution to be found within the constraints rather than specifying the method or algorithm of finding the desired solution. (Freuder, 1996). Such programs are said to be solved instead of executed, as the approach to their execution is much more like a search or a proof than the sequential instructions used by other language generations. The problem domain is described to the computer in such a way that the computer is then able to provide answers about the problem domain. Today this is accomplished through a minimalistic syntax, as compared to other language paradigms.

> *“The ability to directly express the domain specific knowledge in models leads to higher efficiency in systems design and implementation”*.
> (Vallecillo, 2012)

Other software development paradigms give the developer more control over the execution of the application, yet simultaneously, they require the developer to exercise that control as explicit instruction forcing the developer to understand any given problem domain twice: once for themselves, and once on behalf of the machine. This preference for efficiency of development over control is the hallmark of fifth generation technologies.

Today 5GL languages are nearly synonymous with declarative languages, based in set theory, lambda calculus, and first-order predicate logic. These languages are typically associated with Natural Language Processing (NLP) and AI projects (Freuder, 1996), where complex and unexplored relationships can exist within the problem domain. Leveraging these languages involves describing the problem domain as a series of facts and relationships in such a manner that the machine can use them to solve queries about the problem domain (Abdelmalek, 2015). This can be a lengthy and computationally intensive process, so while languages like prolog are amongst the oldest languages, their use of resources at one time restricted their widespread adoption.

Once a description of the relationships within the problem domain is available, the machine is able to answer questions about the problem domain in whole, or in part, by leveraging deductive reasoning. CP and functional languages lend themselves easily to the concept of partial execution, implicitly supporting (without additional planning/design) pre-execution of parts of a program before all the required data are known, including the potential for a complete result from incomplete data. (see figure 1.2.) This is not to say that such capabilities are beyond other paradigms, but that they require explicit design in order to support this feature.

Figure 1.2

Consider, for example, the case of a simple date validator

Procedural:

```
Function ValidDate(year, month, day){
    Return year > 0 and
        month > 0 and
        month < 13 and
        day > 0 and
        day <= 31
    }
```

Constraint based:

```
ValidDay(day) :- day > 0, day <= 31.
ValidMonth(month) :- month > 0, month <= 12.
ValidYear(year) :- year > 0.
ValidDate(year, month, day) :- ValidYear(year),
                               ValidMonth(month),
                               ValidDay(day).
```

If we then call

```
ValidDate(unknown1, unknown2, 32)
```

Procedural:

In its present state, the procedural method insists on attempting to validate year first, and must be heavily modified in order to handle every possible incomplete data set.

Constraint:

The constraint based approach however, out of the box supports incomplete data sets, and can operate in such a manner that it will only evaluate the constraints it can based on the data provided. When run, the constraint based program skips evaluation of year and month, then discovers that the date is definitively invalid based on day number alone, so it is able to provide a return value. (False)

Furthermore, if the date is such that its validity cannot be determined by the day alone, some languages allow for functions to return functions that represent the partially solved scenario, allowing for optimized solving of the problem once more data becomes available. In such a system the method call:

ValidDate(unknown1, unknown2, 1)

Could be said to return the following method:

```
Lambda(Year,Month)
    ValidDay(day) :- day > 0, day <= 31.
    ValidMonth(month) :- month > 0, month <= 12.
    ValidYear(year) :- year > 0.
    ValidDate(year, month, day) :- ValidYear(year),
```

ValidMonth(month),
~~ValidDay(day)~~.

that is ready to answer the valid date question once the rest of the data is provided.

Example of Procedural vs. Constraint based programming

CP's minimalistic syntax (consisting of: declaration, assignment, unary operators, binary operators, and comparators) has enabled the creation of constraint solver libraries for most languages that leverage the syntax of 3&4GL languages. Most CP syntax, regardless of language, resembles horn clauses, enabling the evaluation engine to approach "solving" the application as an algebraic proof. To accomplish this, most languages (or languages subsets) leverage 4 primary forms: (Neilson, 2009)

Facts

- *"Facts are a means of stating that a relation holds between objects"* (Sterling 2010)
- Facts are true. They are the most basic element in the system.
- Facts define predicates using literals and unary operators.
- A fact is a piece of information pertaining to a litteral.

Rules

- *"[Rules enable] us to define new relationships in terms of existing relationships."* (Sterling 2010)
- Rules can be looked at as an amalgamation of facts which all must be explicitly met.

- Rules define predicates using predicates, literals, variables, operators, and, together with facts, make up the problem space.

Goals

- A goal is a proposed rule that may or may not conflict with the rules in the system.
- The "execution" of a goal within a problem space will return whether or not that Goal is legal, given the rules that have already been defined.
- Goals can be implicitly met.
- Goals are expressed as rules
- When executed against the problem space, goals return whether a particular relationship holds true, given the facts within the system.

Queries

- *“Queries are a means of retrieving information from a logic program. A query asks whether a certain relation holds between objects.”* (Sterling 2010)
- Queries are expressed using predicates like goals, but also include unknown values.
- When executed, queries return the conditions that must be met by variables in order for the query, expressed as a goal, to evaluate to true.

It is important to note however that while the aforementioned describes today's CP technologies, there is no guarantee that CP languages in the future will stick to this format. While this formation is conducive for approaching the problem space as a formal

logic equation, this is not necessitated by the design of constraint based languages, but by today's compiler and solver technologies.

Given that the definition of a fifth generation programming language is one that intends to allow the developer to focus solely on the problem domain, (Abdelmalek, 2015) it is then ironic that most of the tools of its implementation continue to require that ideas be expressed to the machine in a rigid format based on horn clauses, not on human language *(Sterling 2010).*

Test Driven Development

Test Driven Development (TDD) is a software development practice in which program code is only written in response to tests that outline the desired functionality. In following such a technique, no code can ever be written without an accompanying test that can aid in code development and guard against side-effects.

There are a number of sources attributed with discovering, or rediscovering, TDD (McCracken, 1957) (McIlroy, 1968) (Dijkstra, 1972) (Kent, 2011). This is likely because, as a practice, TDD is closely related to the practices used in the scientific method and engineering (Mugridge, 2003). Practitioners haven't always agreed on what constitutes best practices for TDD (Hammond, 2012). Regardless, the goal of TDD remains the same: *"encourage simple designs and test suites that inspire confidence*" (Beck, 2014, p.14). This is accomplished by utilizing the smallest possible development-review cycle; individual methods and lines of code. Working individually or in pairs, by authoring tests before code is written, and running those tests afterwards, developers are able to review their code as they write it. The tests ensure that the code does what the developer expects it to do. (Beck, 2011)

While not the only field that uses such a process (Peters, 2014), within the engineering disciplines, computer science has, perhaps, used it to the greatest effect due to the low cost of running automated test suites, and the speed with which changes are made to complex codebases that are being worked on by scores of developers simultaneously (Gupta 2015).

Within the field of software engineering, practitioners have also found other ways of leveraging the byproducts of TDD. For example, test code can be automatically run when new code is submitted to code repository. This can help prevent erroneous code from making its way into the main code base (Fowler, 2006). When a developer submits code that causes tests to fail, this is an indication that the submitted code may have caused unintended side effects and may need to be reviewed.

Test suites created as part of TDD can be leveraged by mutation testing which makes changes to the *production code* and then verifies that the change is caught by the test suite (Jorgensen, 2014; Lipton, 1979). This is well suited for software development because machine time is cheap, and it costs almost nothing to intentionally make a bad build of software to test against. The same can not be said for disciplines more closely related to the physical sciences.

The most powerful case for TDD over unstructured unit testing is the nuance of prioritization:

> *“The prior approach to unit testing at IBM RSS was an ad-hoc approach. The developer coded a prototype of the important classes and then created a design via UML class and sequence diagrams [...] In all cases, the unit test process was not disciplined and was done as an afterthought. More often than not, no unit tests were created, especially when the schedule was tight, the developer got side tracked with problems from previous projects, or when new requirements that were not clearly understood surfaced.“* (Maximilien, 2003, p.2)

The writing of software to test software that is assumed good will always have a low priority, and will cause automated testing to get pushed out of most projects. By insisting that unit tests are written first, and can contribute utility by acting as a framework for writing production code, guarantees that the tests can’t get pushed off the project to reduce project time (Maximilien, 2003).

Pair Programming

“Pair programming is a style of programming in which two programmers work side-by-side at one computer, continuously collaborating on the same design, algorithm, code, or test.” (Williams, 2003)

While pair programming is typically associated with extreme programming (XP) its nameless practice dates long before that (Beck, 2005). For instance, Fred Brooks, in his book: *The Mythical-Man Month: Essays on Software Engineering*, was able to recall utilizing strategies similar to pair programming as early as the 1950s. As such, there is a wide range of ideas concerning specific implementation details. Kent Beck’s (2005) book, however, has served as a definitive rallying point for those practicing paired programming and has become the standard for paired programming practices.

Pair programming is often used when practicing TDD where one participant acts as the developer and the other participant acts as the tester (Hulkko, 2005; Goldman, 2010). Overall, pair programming has generally been accepted in the software development community as a net positive (Williams, 2010; Beck, 2005; Vanhanan, 2007; Sun, 2011; Ariesholm, 2007).

Spectrum of linguistic Interactions

We believe that the best code is written when one developer is describing a process to another, in a language that the machine happens to understand (Martin, 2015; MacConnell, 1998).

Software development, being a relatively young field, has undergone some major shifts in popular paradigms. These shifts were partly due to more efficient means to develop software and partly due to the continuous drop in the price of machine time. The result has been a gradual increase in levels of indirection between instructions given by the operator, and the commands that the machine understands, creating a continuum of linguistic modes that are used to communicate with the machine.

Machine Language (and Assembly)

In the beginning of software development, machine time was more expensive than programmer time (Moore, 1965) and thus, the very first computer languages reflected this. “Machine Language” describes languages created for expressing ideas to machines about how to operate, with strict allocation for small structure and consideration only for the internal mechanics of the machine. Machine Language is not designed with any consideration for how humans think, nor for the scale of the application, making it a difficult and tedious language to work in and debug. (Backus, 1957)

Machine Language for Humans (High Level Languages)

Gradually, as machine-time came down in cost, more languages were developed that better suited the way humans think. However, they were still grounded in the mechanics of how the machine operated, with strict requirements pertaining to structure, grammar, typing, and context. This created an environment which eased the effort required to translate the concepts for the machine by an individual once properly trained..

One key component of machine language and machine language for humans is the asymmetry of the communications. While humans give machines directives in the form of programs, In most situations, communication from machine to human is limited to the execution (or non-execution) of the application, and debug messages that are closely tied to the function of the machine.

Human Language (Natural Language Processing)

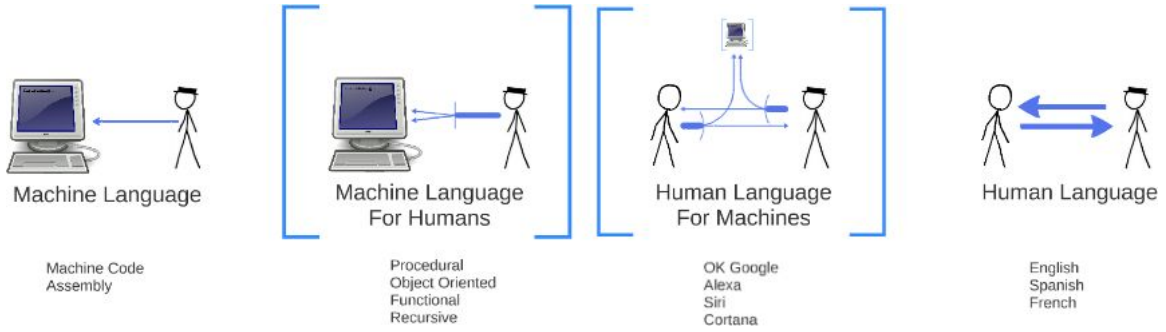
Today machine time is considerably cheaper than developer time, (Raymond, 2008)(Knuth,1968)(Moore, 1965) leading to significant efforts in NLP . The hope in NLP is to create systems that can understand languages intended for communicating between humans. Most of these efforts are aimed at direct interaction with the user addressing the machine, instead of the machine passively understanding communications between humans. There still remains a gap then in achieving a truly

natural computational language. Successful 5GL exploration would help to bridge this gap.

Human Language for Machines

The classification category Human Language for Machines is a gap that currently exists in the realm of structured languages. It exists as a subset of human language with simplified grammar and explicit context for communication between humans that machines can understand. Classically, limited NLP implementations have been seen as incomplete or broken, because they are viewed solely from the perspective of human to machine communications, not task specific subsets of human to human communications.

Figure 1.3



Spectrum of linguistic interactions

From this spectrum of linguistic interactions, it may stand to reason that the human language for machines paradigm could improve expression and ease of use over machine language for humans, given the reduced workload on the developer to put

instructions in terms the machine will understand. Research within this paradigm stands to improve our understanding of how to create languages and nomenclatures that will result in code that is easier to understand, write and maintain.

It is from this perspective then that we approach our subject matter: by observing humans communicating with a limited nomenclature (an established machine language for humans in this case) participating in an exercise with a comparable analog (a 5GL language simulation, aka pair based TDD) allowing us to pick out both familiar (rules, facts and goals) and unfamiliar patterns, thus validating the design of 5GL languages, and simultaneously generating suggestions for future language implementations.

Research

Summary

Our goal was to see if: by writing tests, a tester would be able to guide a developer to implement an assignment. Pairs of test subjects used an online collaborative software development environment to complete a basic programming assignment. Communication between the tester and the developer was only possible via the code written in the development environment.

Hypothesis

Primary

Fifth Generation linguistic mechanics, such as facts, rules and goals, will emerge in the communications between the tester and the developer as they perform Test Driven Development.

Secondary

Nomenclatures and linguistic patterns will be observed in this exercise which could inform the design of future 5GL languages.

Data Collection

To test our hypothesis, the communication between a tester and a developer performing Test Driven Development is analyzed for common patterns and modes of dialogue including CP facts, rules, and goals. In an effort to simplify the data collection process, test participants were asked to communicate solely through the (3GL) code, via an online collaborative code editing application, and were randomly paired for each exercise in an attempt to remove contextual and idiosyncratic interaction bias.

Protocol

Data was collected from 16 pairs of students during February and March of 2019. The data was collected in two different formats:

Scheduled Sessions

Pairs of participants would sign up for a time to participate in the experiment on youcanbook.me (a scheduling website). 5 min before that time, the proctor would send the participants links to participate in the study (calvin.edu/~dmeyer15/TDDResearch). During the study, help was available via the chatroom function of the data collection site (codebunk.com). After participation, these participants were sent a voucher for a free pizza each.

Public Sessions

Public sessions were held as events in computer labs at both GVSU and Calvin College at predetermined times. Attending students were randomly paired based

on programming language, and emailed links (to calvin.edu/~dmeyer15/TDDResearch) to allow them to participate anonymously in the study. During the study, a project administrator was available to answer questions and keep the participants aligned with the intended task. After participation, these participants were served pizza, pop and cookies, and some took home leftovers.

Once at the study landing page (calvin.edu/~dmeyer15/TDDResearch), participants were asked for consent to use their data in the study. Once consent was established, participants were given a brief primer on test driven development, with a specific focus on the workflow that would be used in the study. At this point that participants were informed whether they would be acting as Tester or Developer during the study.

Developers were directed straight from the primer to the collaborative development environment (http://codebunk.com), which was pre-populated with a testing framework and some example tests. Testers were given a specification to review, detailing the functionality they were to guide the developers to create, via tests. Once the specification was reviewed, the testers were directed to the collaborative development environment as well.

The Assignment

Each tester was given a specification for the core functionality of a vending machine for which to write unit tests. The specification included functional requirements for 5 functions for which unit tests could be written for the purpose of guiding the implementation efforts of the paired

developer. The specification included auxiliary functionality for a pricing system that stores the price of every item in the machine, an inventory system that dispenses products, and a cashier system to handle money. The functions for which tester was to write unit tests are described in the following subsections.

MethodP

The first part of the assignment instructed the tester to guide the developer in writing a function that would accept the product coordinates supplied by the user and A reference to the pricing system. The method would need to pass those coordinates to a similarly named function in the pricing system and return the result of that function call.

Of the successful groups most implementations went along these lines:

Figure 2.1

```
Def MethodP(row, col, pricing):
    return pricing.MethodP(row, col)
```

Example Implementation of MethodP

Where MethodP is the assigned method, *`pricing`* is an object implementing the pricing system interface, and pricing.MethodP is the routine from the pricing interface that returns the price of a particular item.

MethodF

The second part of the assignment was for the tester to guide the developer in writing a function that would accept the price of the desired item, a pointer to the cashier system, and would return true if the user had put enough money in the system for the selected

item, and false otherwise. Of the successful groups most ended up with an implementation along these lines:

Figure 2.2

```
Def MethodF(price, cashier):
    return cashier.MethodM() >= price
```

Example implementation of MethodF

Where MethodF is the assigned method, price is the price of the requested item and cashier.MethodM() returns how much money the user has supplied.

MethodC

The third part of the assignment was for the tester to guide the developer in writing a function that would accept the price of the desired item, and a pointer to the cashier system. The method would then query the cashier system for amount of money supplied by the user, and the available change. (an array of quarters, dimes and nickels) A successful implementation would likely fall along these lines:

Figure 2.3

```
Def canMakeChange(change_to_make, coins):
  If (change_to_make < 0):
    Return False
  return
    Change_to_make == 0 or
    coins[0] and
      canMakeChange(
        change_to_make - 25,
        [coins[0] - 1, coins[1], coins[2]]) or
    coins[1] and
      canMakeChange(
        change_to_make - 10,
        [coins[0], coins[1] - 1, coins[2]]) or
    coins[2] and
      canMakeChange(
```

```
        change_to_make - 5,
        [coins[0], coins[1], coins[2] - 1])
Def MethodC(price, cashier):
  return canMakeChange(cashier.MethodM()- price, cashier.MethodC())
```

Example Implementation of MethodC

Where the price is in cents, cashier.MethodM() returns how much money the user has supplied and cashier.MethodC() returns an array with 3 elements representing quantities of quarters, nickels and dimes available for the machine to make change. The last 2 problems were to check if a given product was in the machines inventory, and to put it all together and write the main routine that would perform the whole transaction of buying an item from the machine.

Participants collaborated, via test code (Facts and Rules) and production code (Facts and Goals). The tester wrote tests which used the as-yet-to-be-written code described within the assignment, and the developer then examined those tests and wrote code that those tests depended on. Test subjects took turns writing code (respective of their roles) in a manner that resembled a conversation, where the tester expressed an idea and then the developer responded by parroting back what they thought the tester was asking for, at which point the tester would either move onto a new idea, or try to elaborate further on the current idea.

While the participants were asked to set aside an hour for the experiment, some neglected this courtesy, and some, more enthusiastic participants, continued working on the assignment past the allotted hour. At the conclusion of an hour of working on the

exercise, participants were informed that the allotted hour had concluded, and that while no one was going to ask them to continue past the time allotted, no one was going to ask them to stop, either. At that point, most groups conferred over the collaborative development environment and decided to be done.

Analysis

The coding session for each tester/developer pair was recorded for analysis. This allowed us to view the code written by the tester and the code written by the developer in terms of as a conversation, made up of statements by both the tester and developer. This made ordering and linking the functionality requested by the tester and implementation by the developer easier to follow within the code.

After the code was broken down into statements, it was further processed before analysis:

1. *Transformation:*

   Refactoring equivalent function and variable names to common names.

2. *Classification:*

   Matching the code to the part of the specification it relates to.

3. *Filtering:*

   Evaluating the code for its applicability to the assignment, and whether or not it followed the rules of the assignment.

The communication between the testers and developers were then analyzed for the presence of idioms analogous to CP facts, rules, and goals. Given how open ended the assignment was, and how new to programming many of our test subjects were, the decision was made to identify ALL facts that could be inferred from every piece of code written by the tester or developer, and then backtrack to the relevant facts from the rules and goals, once identified.

To aid in the identification of facts, rules and goals, (now that we have established more parameters surrounding the experiment), we can simplify our qualifications of these terms without risking loss or mis-catorigation of data. We can do this because 5GL concepts, expressed through a 3GL as the experiment requires, share some easy to spot features. To put it another way, because of the linguistic constraints of the experiment, we can greatly simplify our definition of facts, rules and goals without altering the way we look at our data. Our definitions of facts, rules and goals then are as follows:

Facts

- Literal values
- Anything about the code that is constant, unconditional or structural.

Figure 2.4

```
Var a = 1
```

Example Fact 1

- Fact: There is a variable called **a**

- Fact: The variable: **a,** holds the value 1

Figure 2.5

```
Int v = add(1, 2)
```

Example Fact 2

- Fact: There is a variable **v**
- Fact: Variable **v** is an integer
- Fact: There is a method **add**
- Fact: The method **add** takes 2 parameters
- Fact: The first parameter to method **add** is an integer.
- Fact: The second parameter to method **add** is an integer
- Fact: The method **add** returns an integer

Rules

- A rule is any statement that is conditional and made up of facts, variables and other rules.
- Facts and rules make up the problem space as described by the tester.
- At runtime all rule values are resolved to facts. If values cannot be resolved, the program will fail with an undefined variable.
- By expressing a rule as an assertion, the fulfillment of the rule becomes an imperative for making the program function, thus expressing the need for action on the part of the developer.

Figure 2.6

```
assert(3 == add(1, 2))
```

Rule: when passed 1 and 2, ”add” will return 3

Figure 2.7

Java, Session 4, Exchange 1, Tester

`Public Boolean test_P(){`
`    Machine m = new Machine();`
`    IPricing p = new PricingFake(3);`
`    if (m.MethodP('A', 1, p) != 3) return false;`
`    return true;`
`}`

In this case, returning false is the same as failing the test
Rule: Machine().MethodP must return 3, if it is passed `‘A’, 1` and `new PricingFake(3)`.

Goals

- Goals are rules, written by the developer in response to the rules written by the tester, that allow the program to run.
- For our purposes, if a goal is not congruent with the rules set in the problem space, its test(s) will fail.

Figure 2.8

Assertion Example - Multiple Rules, One Goal
`assert(3 == add(1, 2))` `assert(5 == add(2, 3))`
In response to the rules above, a developer would likely write the following Goal:
`Def add(a, b):` `    Return a + b`

An example of a Goal written in response to stated rules

Figure 2.9

Java, Session 2, Exchange 1, Tester
`class PricingFake {` `    private Map<String, Integer> itemPrices;`

```
    PricingFake(Map<String, Integer> items) {
        itemPrices = items;
    }

    int MethodP(char row, int col) {
        return itemPrices.get(
            "" + row + Integer.toString(col)).intValue();
    }
}

class TestFixture {
    public Boolean test_MethodP_Returns_pricing_of_selected_item() {
        PricingFake p =
            new PricingFake(new HashMap<String, Integer>() {{
                put("A1", 125);
                put("A2", 150);
            }});

        Machine m = new Machine();
        assert m.MethodP('A', 1, p) == 125;
        assert m.MethodP('A', 2, p) == 150;

        return true;
    }
}
```

Rules written by a tester for the MethodP problem in the assignment

Figure 2.10

Java, Session 2, Exchange 1, Developer

```
class Machine {
    int MethodP(char row, int col, PricingFake p) {
        return p.MethodP(row, col);
    }
}
```

A goal, written in response to the rules above, for MethodP

In the example above the pair managed to correctly express and interpret the need for both a class named Machine and a method: MethodP the Tester correctly interpreted that MethodP was just a wrapper around the call to pricing fake, MethodP.

Disambiguation: Variables

When a Tester assigns a value (fact) to a variable, that variable acts as a representation of that fact; it is a known quantity[4]. However, when a Developer writes a function that takes parameters, the parameters do not represent known quantities. The facts and rules provided by the tester are examples of what values CAN be passed to the function, but are (typically) only a small subset of the full spectrum of values which the function may be given. For this reason

we say that only the developer writes goals, which relate facts and variables. In this context, a variable is an unknown value.

Fact Breakdown - Example #1

This example shows the facts for a method, `MethodP`, its parameters, return value, and supporting infrastructure such as class name and supporting classes, being conveyed with two unit tests.

Figure 2.11

Java, Session 1, Exchange 1, Tester	
LN	**Code**
1	`Class Pricing {}`
2	
3	`class TestFixture {`
4	`    public Boolean test_1(){`
5	`        Machine m = new Machine();`
6	`        Pricing p = new Pricing();`

[4] Compilers will often times attempt to replace variables with the literal values they represent to simplify the machine code.

7	`        int price = m.MethodP('A' , 1  , p);`
8	`        return true;`
9	`    }`
10	
11	`    public Boolean test_2() {`
12	`        Machine m = new Machine();`
13	`        Pricing p = new Pricing();`
14	`        return 100 == m.MethodP('B', 1, p);`
15	`    }`
16	`}`
LN	**Analysis**
	Breaking down the code above yields the following set of facts:
1	There is a class named Pricing
2	
3	Class TestFixture is part of the testing framework, and is not specifically included in our analysis
4	test_method_p is part of the testing framework, and therefore is not specifically included in our analysis
5	There is a class named Machine
6	There is a variable p which stores an instance of Pricing
7	Class Machine has a method MethodP Machine's MethodP takes 3 arguments Machine's MethodP can accept a character as its first parameter Machine's MethodP can accept “A” as its first parameter Machine's MethodP can accept an integer as its second parameter Machine's MethodP can accept 1 as its second parameter Machine's MethodP can accept a Pricing object as its third parameter Machine's MethodP can accept a p as its third parameter Machine's MethodP returns an integer
8	The return value of test_method_p is part of the testing framework, and therefore is not directly included in our analysis, but true indicates a successful run of the test.
9	
...	
14	Machine's MethodP can accept “B” as its first parameter Machine's MethodP can return 100
	Note: For the sake of brevity, facts have been listed only at their first occurrence

Fact Breakdown #1

Fact Breakdown - Example #2

This example shows the facts for methods `MethodP` and `MethodF`, their parameters, their return values, and their supporting infrastructure such as class name and supporting classes, being conveyed with two unit tests. Each unit test, `test_method_p` and `test_method_f`, contain multiple tests for MethodP and MethodF respectively.

Figure 2.12

Python, Session 1, Exchange 2, Tester

LN	Code
1	class IPricing:
2	def __init__(self, d):
3	self.data = d
4	
5	def method_p(self, a, b):
6	return self.data[(a, b)]
7	
8	class ICashier:
9	def __init__(self, a):
10	self.a = a
11	
12	def method_m(self):
13	return self.a
14	
15	class TestFixture:
	...
18	def test_method_p(self):
19	p = IPricing({
20	('A', 1): 1.00,
21	('B', 2): 2.50,
22	('C', 5): 1.75
23	})
24	

```
25          assert Machine().method_p('A', 1, p) == 1.00
26          assert Machine().method_p('B', 2, p) == 2.50
27          return True
28
29      def test_method_f(self):
30          c = ICashier(2.00)
31
32          assert Machine().method_f(1.50, c) == True
33          assert Machine().method_f(2.00, c) == True
34          assert Machine().method_f(2.50, c) == False
35          return True
```

LN	Analysis
	Breaking down the code above yields the following set of facts:
1	There is a class named IPricing
2	IPricing has a non-default constructor IPricing has a constructor that accepts 1 parameter
3	IPricing's constructor saves its 1st parameter in the variable “data”
4	IPricing has a method method_p
5	IPricing's method_p takes 2 parameters
6	IPricing's “data” variable needs to be subscriptable using a tuple as a key IPricing's 1st constructors parameter needs to be subscriptable using a tuple as a key IPricing's method_p returns whatever is returned from subscripting data by the tuple (a,b)
7	
8	There is a class named ICashier
9	ICashier has a non-default constructor ICashier has a constructor that accepts 1 parameter
10	Cashier's constructor saves its parameter in the variable “a”
11	
12	ICashier has a method method_m ICashier's method_m takes no parameters
13	ICashier's method_m returns the value stored in variable “a”
14	
15	Class TestFixture is part of the testing framework, and is not specifically included in our analysis
...	
18	test_method_p is part of the testing framework, and therefore is not specifically included in our analysis
19	There is an IPricing object that is initialized, and thus stores, with a dictionary that is keyed using tuples, that is saved as variable “p” IPricing can be instantiated with a dictionary of ({('A', 1): 1.00, ('B', 2):

	2.50, ('C', 5): 1.75 }) without crashing
20	
...	
25	There exists a class named Machine Machine has a constructor that takes no arguments Machine has a method method_p Machine's method_p can take 3 arguments Machine's method_p can accept a character as its first parameter Machine's method_p can accept ‘A’ as a first parameter Machine's method_p can accept an integer as its second parameter Machine's method_p can accept 1 as a second parameter Machine's method_p can accept an IPricing object as its third parameter Machine's method_p can accept p as a third parameter Machine's method_p can return a float Machine's method_p can return 1.0
26	Machine's method_p can accept ‘B’ as a first parameter Machine's method_p can accept 2 as a second parameter Machine's method_p can return 2.0
27	The return value of test_method_p is part of the testing framework, and therefore is not directly included in our analysis, but true indicates a successful passing of the test.
28	
29	test_method_f is part of the testing framework, and therefore is not specifically included in our analysis
30	ICashier's constructor can be passed a float value of 2.00 The variable “c” stores an ICashier
31	
32	Machine’s method_f can be passed a float as its first parameter Machine’s method_f can be passed 1.50 as its first parameter Machine’s method_f can be passed an ICashier as its second parameter Machine’s method_f can be passed c as its second parameter
33	Machine’s method_f can return a value that equates to true
34	Machine’s method_f can be passed 2.00 as its first parameter Machine’s method_f can be passed 2.50 as its first parameter Machine’s method_f can return a value that equates to False
35	The return value of test_method_p is part of the testing framework, and therefore is not directly included in our analysis, but true indicates a successful passing of the test. *Note: For the sake of brevity, facts have been listed only at their first occurrence*

Fact Breakdown #2

Looking at the code in this way, all successful convergence on goals by tester and developer exhibit these patterns of expressing facts, rules and goals.

Conversation

Putting all of this together then, we get our final analysis of a statement by a tester, and a response by a developer:

Figure 2.13

Java, Session 4, Statement 6, Tester	
LN	**Code**

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
…
25
…
63
64
65
66
67
68
69
70
71
72
	`interface ICashier {`
	`  public double getBalance();`
	`  public double getPrice();`
	`}`
	
	`class Cashier implements ICashier {`
	`  double balance;`
	`  double price;`
	`  public Cashier(double b, double p) {`
	`  balance = b;`
	`  price = p;`
	`  }`
	`  public double getBalance() {return balance;}`
	`  public double getPrice() {return price;}`
	`}`
	
	`class TestFixture {`
	
	`  public Boolean test_C() {`
	`  ICashier c1 = new Cashier(0, 30);`
	`  ICashier c2 = new Cashier(0, 50);`
	`  Machine m = new Machine();`
	
	`  if ( !(m.MethodC(75, c1)) ) return false;`
	`  if ( !(m.MethodC(40, c1)) ) return false;`
	`  if ( m.MethodC(40, c2) ) return false;`
	`  return true;`
	`  }`

LN	**Analysis**
64 65 66 67 68 69 70	FACT: There exists an object of type ICashier called c1 FACT: c1's balance variable is 0 FACT: c1's price variable is 30 FACT: There exists an object of type ICashier called c2 FACT: c2's balance variable is 0 FACT: c2's price variable is 50 FACT: There exists an object of type Machine called m RULE: IF m.MethodC is passed 75 and c1, THEN it will return True RULE: IF m.MethodC is passed 40 and c1, THEN it will return True RULE: IF m.MethodC is passed 40 and c2, THEN it will return False

A means used by a pair of subjects to Specify the MethodC assignment.

Given the above facts and rules then, the developer of this pair wrote:

Figure 2.14

Java, Session 4, Statement 7, Developer	
LN	**Code**

1
...
14
15
16
17
	class Machine {
	
	public boolean MethodC(int n, ICashier c) {
	return (c.getPrice() < n);
	}
	}

LN	**Analysis**
14 15	FACT: there exists a method: MethodC, which returns a boolean FACT: there exists an integer called n FACT: there exists an ICashier called c GOAL: MethodC will return true if and only if c.getPrice() returns a value that is less than n

A means used by a pair of subjects to fulfill the MethodC assignment.

It is noteworthy that the preceding example represents a large jump in logic, there is no deterministic algorithm, given these facts and rules, to come to the goal intended by the tester. It is a reasonable assumption, however, that the facts, rules and goals expressed will reflect the assignment.

While the two examples given show effective communication, only about 1/3 of the groups managed this feat. Of the other 2/3 of the study participants, there were a number of modes of failure:

- Tester lack of programming knowledge

    - There were a number of groups where a lack of familiarity with the technologies in use or the ideas used in the assignment resulted in the groups producing little to no code.
- Developer lack of programming knowledge
    - These groups produced initial test code, but were unable to produce production code that satisfied those initial tests.
- Tester, lack of Test Driven Development knowledge
    - There were a few groups where there was a clear understanding of programming syntax, but an inability to connect on the idea of making the program fail if the required logic was not present. Without that possibility of failure (often expressed as an assertion or a return value) It was difficult for the developer to determine what was desired.

In the cases we cite as having successful communication, there was at least one round of communication whereby the tester wrote tests that the developer satisfied either to the satisfaction of the assignment OR the satisfaction of the tester. While the accurate interpretation and expression of the assignment by the tester to the developer was the objective given to the tester, our objective is only to identify successful communication from tester to developer. This means that successful communication by the tester to the developer of an inaccurate interpretation of the assignment can still be deemed successful communication for the purposes of our study so long as the tester indicated

that the developers understanding was complete by moving onto another section of the assignment.

Results

1,815 lines of code were collected from 16 pairs of developers. Of those pairs, 1 did not produce a single line of code, and 5 produced code that was unusable within the study, because the group communicated directly about the assignment or they did not produce coherent code. Of the 10 remaining groups, 1,265 lines of code were produced yielding 78 code conversational statements, 11 goals, from: 69 rules, supported by 189 facts. Five of the 10 pairs produced goals intended by the tester. The 5 pairs that produced the 11 goals, also produced 42 rules, while the 5 groups that did not produce any goals only produced 27 rules.

While our sample size is very restrictive, the consistency of the results and the apparent correlation between number of rules written and number of goals produced can suggest that writing tests that are focused on expressing rules may play a role in expressing goals under the circumstances of this study.

Of the 11 goals produced, 6 of them required at least one round of revision or addition to the relevant rules to get them to fully reflect the intent of the tester. Of the 11 goals produced, 2 were not a convergence on the ideas expressed by the Tester and did not use overlapping rules. The other 9 all used overlapping rules to converge upon the goal,

and of those that needed to be refined, the final revision coincided with the addition of overlapping rules.

Putting it all together, we now have an example of convergence on a single goal, through overlapping rules, over the course of 3 exchanges between tester and developer.

Figure 3.1

Java, Session 4, Exchange 3, Tester

```
class ICashier {
    public IChashier() {
    }
}

class TestFixture {
    public Boolean test_F() {
        ICashier c = new Cashier();
        IMachine m = new Machine();
        if (!(m.MethodF(75, cashier))) return false;
        return true;
    }
}
```

The tester defined the basic definition of methodF

Figure 3.2

Java, Session 4, Exchange 3, Developer

```
class Machine {
    public boolean MethodF(int n, Cashier c) {
        return true;
    }
}
```

The developer Responded by writing code that makes the test work.

Figure 3.3

Java, Session 4, Exchange 4, Tester

```
interface ICashier {
    public double getBalance();
}
class Cashier implements ICashier {
    double balance;
    public Cashier(double n) {
        balance = n;
    }
    public double getBalance() {return balance;}
}
class TestFixture {
    public Boolean test_F() {
        ICashier c = new Cashier(100);
        Machine m = new Machine();
        if (!(m.MethodF(75, c))) return false;
        if (m.MethodF(150, c)) return false;
        return true;
    }
}
```

The tester adds another call to MethodF, expanding the definition and adding overlap of one parameter, and no overlap in return value.

Figure 3.4

Java, Session 4, Exchange 4, Developer

```
class Machine {
    public boolean MethodF(int n, ICashier c) {
        if (n < 100) {
            return true;
        }
        else { return false; }
    }
}
```

With only 2 examples, the developer can still do little more than guess at the intention of the tester. With no overlap on the first parameter, and no non-overlap on the second parameter, the developer has little choice but to conclude that the change in return value is purely a function of the first parameter.

Figure 3.5

Java, Session 4, Exchange 5, Tester

```
interface ICashier {
```

```
    public double getBalance();
    public double getPrice();
}
class Cashier implements ICashier {
    double balance;
    double price;
    public Cashier(double b, double p) {
        balance = b;
        price = p;
    }
    public double getBalance() {return balance;}
    public double getPrice() {return price;}
}
class TestFixture {
    public Boolean test_F() {
        ICashier c = new Cashier(100, 0);
        ICashier c1 = new Cashier(50, 0);
        Machine m = new Machine();

        if (!(m.MethodF(75, c))) return false;
        if (m.MethodF(150, c)) return false;

        if (!(m.MethodF(25, c1))) return false;
        if (m.MethodF(75, c1)) return false;

        return true;
    }
}
```

The developer adds 2 more examples. The overlap established by giving 2 examples where the first parameter is 75 with different return values will cause the developer to consider a more sophisticated implementation.

Figure 3.6

Java, Session 4, Exchange 5, Developer

```
class Machine {
    public boolean MethodF(int n, ICashier c) {
        return (c.getBalance() > n);
    }
}
```

The developer, faced with overlapping rules, (correctly) chooses to use the relationship between the first parameter of Machine.MethodF and the first parameter to the Cashier constructor (which is then returned by Cashier.getBalance()) to determine the return

value MethodF should return.

This example of overlapping rules was typical of overlaps found in successful production of goals by the developer and leads to the conclusion that a mix of overlapping and non-overlapping rules play a role in communicating the relationships, and by proxy, convergence. While such convergence is aided by the contextual awareness of the (human) developer, the quality of the developers' responses will improve with more deterministic example sets.

Conclusions

Primary Hypothesis

It is clear from the data collected that facts, rules and goals emerged during successful communication between tester and developer. Therefore, we find support for our primary hypothesis: *Fifth Generation linguistic mechanics, such as facts, rules and goals, will be emergent in communications for pairs of developers performing Test Driven Development.* Through our analysis of the data collected during the course of the experiment, it is reasonable to conclude that we found support for our primary hypothesis, insomuch as can be found with the small sample size. experiment

Secondary Hypothesis

Support was found for our secondary Hypothesis: *Nomenclatures and linguistic patterns will be observed in this exercise which could inform the design of future 5GL languages.* We found support for our secondary hypothesis from the results of the experiment in the form of a pattern of overlapping tests that coincided with convergence of ideas between members of a pair, insomuch as can be found with the small sample size. This can act as a waypoint for future language and AI designers, as a benchmark for what human interactions are like.

Overlapping Rules

Example: No Overlap

Given a black box function declaration $f(x,y) \rightarrow n$ and the following data, Determine $f(x,y)$

Figure 4.1

x	y	n
1	3	7
4	5	6
7	2	0

Example: No Overlap

While clearly the lack of examples is a hindrance, the fact that there is no overlap between values of x and y between the different examples makes it almost impossible to determine the internals of the function or the individual contributions of x or y.

Example: Overlap

Given a black box function declaration $f(x,y) \rightarrow n$ and the following data, Determine $f(x,y)$

Figure 4.2

x	y	n
1	3	7
1	2	6
7	2	0

Example: Good overlap

With the alteration of only one example, creating an overlap between example values of x and y. Allows us to see the individual contributions of x and y and resolve the relationships between x, y and z, determining that $f(x,y) = 5 - x + y$

Example: Full Overlap

Given a black box function declaration $f(x,y,z) \rightarrow n$ and the following data, Determine $f(x,y,z)$

Figure 4.3

x	y	z	n
1	3	3	7
1	3	4	8
7	2	2	0
7	2	3	1

Example: Too Much Overlap

In this example, x and y are fully overlapped, and z is partially overlapped with x and y. z's relationship to (x and y) and n is made clear with these examples, however, because there is a full overlap between x and y, their individual contributions to the outcome cannot be determined.

Partial overlap then is very important in constructing efficient and expressive data sets to communicate functionality.

6th generation language

Throughout the history of software development, language technologies have revolved around finding more efficient and effective means to deterministically communicate the testers ideas to the machine. If we consider this experiment run between a human tester, and an A.I., we will see that through the course of this research an interesting dynamic emerged whereby our AI stand-in made successful, intuitive guesses about what the testers intent was, without the aide of a fully deterministic model having been provided by the tester. This intuitive gap represents a major shift from full specification by the tester [as required by all previous language technologies] to a collaborative definition of the task as defined by both human and AI. We would like to propose that as AI improves, this form of human-computer collaboration is an inevitable step in the evolution of software development. This could be facilitated by something as simple as a predictive database built up form gigabytes of different code-bases off github, [as annoying as clippy, but for code], or as complex (or more) as Neural-nets; tuned to the individual developers coding and problem solving style. As this represents both an improvement in efficiency (doing more with less time and planning) and effectiveness (saying more with less code, without the loss of specificity) we would like to suggest that this type of technology, when backed by a comprehensive, collaborative AI, be considered a defining feature of a 6th generation of software development paradigm.

Lessons Learned: Human research

While some difficulties were anticipated concerning human centered research, the full scope of the challenges was not fully appreciated until they were encountered. The primary impact of these challenges was felt in the projects effective response rate. While response rate is often a challenge for any project, pairing up participants compounds this issue, doubling the number of required participants to achieve a satisfactory response rate.

Beyond this however, skill level played a significant role in determining the potential for success of a pair. Unfortunately, no surplus of ability on the part of one pair member could overcome a lack of skills possessed by their partner. As such, the impact of a lack of skill of one pair member effectively doomed the affected pair to, doubling our losses due to this challenge.

While this study proved the feasibility of this research technique, using professionals as research subjects instead of students, and using a much larger group of participants would give the results that this study hints at true validity.

Lessons Learned: time series data

One hurdle that had to be overcome to accomplish this study was to come up with the tools necessary to collect the data. The fact that the code was being developed actively by 2 people at once, and we needed to be able to associate edits with individuals, and

the fact that we needed to see the intermediate (unfinished) stages of the code, combined with the fact that the pair needed to be able to run the code, proved to be a very unique use case, and a major hurdle for the study.

The discovery (by the researchers) of code interview software (codebunk.com) was a key turning point in the project, as it provided the capability to record the sessions for analysis, but processing the results still proved to be a time consuming process. An initial prototype of the project included a means for tester and developer to maintain better separation between their code, and even a means to indicate whose turn it was to code, locking the pair to one person typing at a time. Such features would be an absolute necessity for scaling up this project, to decrease the amount of effort required to break the code in to conversations.

Continued Research

We believe that this research demonstrated the viability of this line of thinking. Continuing this study with a larger sample size, more stringent screening of test subjects and improved instrumentation would go a long way to increase confidence in the results and allowing for a more detailed analysis.

Prepared Response

Q: *It would appear that it is not possible to define a functioning program without expressing rules and goals. If that is the case, how can this be considered research?*

A: While it may not be possible to successfully code without expressing goals, the strategic use of facts and rules to successfully communicate goals is not guaranteed. It can also be argued that facts and rules are unavoidable in test driven development with good test coverage[5] of the code. However the goal of this exercise was never coverage, but rather communication. Along with that, most of the subjects were unfamiliar with any ideas or principles for achieving good code coverage.

[5] Code coverage is a common metric in test driven development. It is a reference to the percentage of code exercised by tests. For our experiment, successful goal production in the face of poor coverage would represent a lucky guess on the part of the developer.

Appendix

Website

Introduction

Thank you for your interest in advancing the art of Computer Science through participation in this groundbreaking research project. Please note that your participation is entirely voluntary, and that you are in no way required or obligated to begin or complete the exercises presented, and that you may choose to have your results excluded from the study. During collection, exercise results will be anonymized to protect your privacy.

This project is studying interactions between partners in a particular style of pairwise test driven development using the Python language. Basic knowledge of the python language is required, but feel free to look up any information, keywords etc. that you need to complete the requested tasks.

Please, don't hesitate to contact me with any further questions

- David Meyer
    - (616)581-4992
    - dmeyer15@calvin.edu

- Adjunct Faculty
- Computer Science Department
- Calvin College

After hitting Yes or No, you will be paired up with another test participant, and forwarded on to a collaborative online software development environment.

Do you consent to have your exercise results included in the study?

YES | NO

Test Driven Development Primer

Directions

For our test, we will be using pairwise test driven development with a few specific limitations. Typically, pair programming involves a pair of developers working in a highly integrated fashion, often with roles being described as "Driver" and "Navigator" where both are involved in all parts of development. For this study however, we ask that you perform instead in the assigned roles of:

Tester

Has the program specification and only writes test code.

Developer

Only writes code to make the test code function.

Furthermore, we ask that you make every attempt to communicate only through the code, through the writing of tests, and their satisfaction. This is not an assignment, you will not be graded, there are no trick questions, but if you communicate through english (comments, talking, etc.) about the assignment, Your data will not be useful for the study. (Communication like "BRB BIO" and even "wait, what did you change" are OK!)

Looking at it from a different angle...

We are asking that you and your partner play/work as a Charades/Pictionary, team using test code instead of pictures or gestures: one developer will attempt to get the other to write an assigned piece of functionality, but may ONLY describe that functionality through test code, without the aid of (fully) descriptive variable and function names. (count is OK, number_of_dollars_in_account_before_tax is not)

As a part of the experiment, as we are looking to maximize on communication through the code, the object and method names in the assignment have been obfuscated, so that the usage of the code cannot be inferred via naming convention alone.

Example Workflow:

assume the spec requires a function named: divide_it(x, err=err) that will return 1/x unless x<4, in which case, it will return the result of errorModule.err().

The tester could communicate this to the developer by writing the following tests (remember, it's good practice to keep tests small and simple)

How to start

1.The tester writes a simple test method that confirms that there IS a method named divide_id(x)

```
class TestFixture{
    Boolean test_divide_it_exists(){
        var thingToTest = new ThingToTest();
        thingToTest.divide_it(1.0);
        return true;
    }
}
```

After writing this test, the code will not run, and it is up to the developer to write the code that makes the test work

2.The coder then responds by defining a method that makes the test run successfully.

How to test (and communicate) method definitions

```
class ThingToTest{
    public void divide_it(double i) {
    }
}
```

3.The tester tests divide_it's I/O by calling it with parameters and using assert on the return

value.

```
class TestFixture{
    Boolean test_divide_it_exists(){
        var thingToTest = new ThingToTest();
        thingToTest.divide_it(1.0);
        return true;
    }

    Boolean test_divide_its_data(){
        var thingToTest = new ThingToTest();
        Debug.Assert(thingToTest.divide_it(4) == 0.25);
        return true;
    }
}
```

4.The coder then responds by modifying the divide_it method to pass the test.

```
class ThingToTest{
    public double divide_it(double i) {
        return 0.25;
    }
}
```

5.The tester then modifies the the previous test with more data points to express the required functionality.

```
class TestFixture{
    Boolean test_divide_it_exists(){
        var thingToTest = new ThingToTest();
        thingToTest.divide_it(1);
```

```
        return true;
    }

    Boolean test_divide_its_data(){
        var thingToTest = new ThingToTest();
        Debug.Assert(thingToTest.divide_it(4)  == 0.25);
        Debug.Assert(thingToTest.divide_it(5)  ==  0.2);
        Debug.Assert(thingToTest.divide_it(10) ==  0.1);
        Debug.Assert(thingToTest.divide_it(20) == 0.05);
        return true;
    }
}
```

6.The coder then responds by modifying the divide_it method to pass the test.
How to specify implementation

```
class ThingToTest{
    public double divide_it(double i) {
        return 1.0/i;
    }
}
```

7.The tester creates a pair of tests to assert the boundary condition at x=4, asserting that err is called when x<4. Weather or not a method has been called can be accessed through the called property of the mocked method.

```
interface IGenerateErrors{
   void generateError();
}

class TestFixture{
    class FakeErrorGenerator: IGenerateErrors{
        public  bool errorGeneratorWasCalled = false;
        public void generateError() {
            errorGeneratorWasCalled = true;
        }
    }

    Boolean test_divide_it_exists(){
        var thingToTest = new ThingToTest();
        var fakeErrorGenerator = new FakeErrorGenerator();
        thingToTest.divide_it(1, fakeErrorGenerator);
```

```
        return true;
    }

    Boolean test_divide_its_data(){
        var thingToTest = new ThingToTest();
        var fakeErrorGenerator = new FakeErrorGenerator();
        Debug.Assert(thingToTest.divide_it(4, fakeErrorGenerator)  ==
0.25);
        Debug.Assert(thingToTest.divide_it(5, fakeErrorGenerator)  ==
0.2);
        Debug.Assert(thingToTest.divide_it(10, fakeErrorGenerator) ==
0.1);
        Debug.Assert(thingToTest.divide_it(20, fakeErrorGenerator) ==
0.05);
        return true;
    }


    Boolean test_divide_it_out_of_bounds_err() {
        var thingToTest = new ThingToTest();
        var fakeErrorGenerator = new FakeErrorGenerator();
        thingToTest.divide_it(3.9999999, fakeErrorGenerator);

        //return true if error generator was called, passing the test
        return fakeErrorGenerator.errorGeneratorWasCalled;
    }

    Boolean test_divide_it_out_of_bounds_no_err() {
        var thingToTest = new ThingToTest();
        var fakeErrorGenerator = new FakeErrorGenerator();
        thingToTest.divide_it(4, fakeErrorGenerator);

        //return false if error generator was called, failing the test
        return !fakeErrorGenerator.errorGeneratorWasCalled;
    }
}
```

8.The coder then responds by modifying the divide_it method to pass the test.

```
class ThingToTest{
    public double divide_it(double i, IGenerateErrors fakeErrorGenerator) {
        if (i<4)
```

```
            fakeErrorGenerator.generateError();
        return 1.0/i;
    }
}
```

Assignment

Overview

Your company has been contracted to provide the core logic for a new vending machine. This logic will be integrating the sub-systems of the vending machine, whose interfaces are already well defined.The integration will be bringing together the functionality of the Pricing System, Cashier System and Inventory Management System.

Your assignment is to lead your partner to write an object that implements the following "interface" (term used loosely for scripting languages) by writing test code: (feel free to copy and paste the method definitions) Please help the developer implement this interface by writing tests that make calls to, and through (by providing mock object replacements for the VendCo objects) an object that implements this interface.

```
using VendCo;

public interface IMachine
{
    /// <summary>
    ///     This method calls through to the IPricing interface to get the
price of the
    ///     item at location row,column.
    ///     THIS METHOD REALLY IS A SIMPLE PASS-THROUGH METHOD, MEANT TO
GET YOU IN THE
    ///     GROVE OF THE WORKFLOW.
    /// </summary>
    /// <param name="row">
    ///     Character between "A" and "Z" representing the shelf of the
users order.
    ///     You may assume that this input will always be valid
    /// </param>
    /// <param name="column">
    ///     Integer between 0 and 63 representing the column of the users
order.
    ///     You may assume that this input will always be valid
    /// </param>
    /// <param name="pricing">
    ///     An object providing product pricing services for the Vending
Machine
    /// </param>
    /// <returns>
    ///     The price of the item at the specified location in cents. ($1 =
100)
    /// </returns>
```

```
    /// <example>
    ///     IPricing pricing = new Pricing();  //Pricing class does not
exist, you will need to mock this out for testing
    ///     IMachine machine = new Machine();  //This Machine class is what
the developer will be writing
    ///     Console.WriteLine("Cheetos, located in dispenser A,1 cost: $" +
machine.MethodP("A",1, pricing)/100.0);
    /// </example>
    int MethodP(char row, int column, IPricing pricing);

    /// <summary>
    ///     Queries the Cashier system for how much money the consumer
supplied.
    ///     Compares money supplied against cost of item.
    /// </summary>
    /// <param name="itemPrice">
    ///     The price of the item requested by the user in cents. ($1 =
100)
    /// </param>
    /// <param name="cashier">
    ///     An object providing cashier services for the Vending Machine
    /// </param>
    /// <returns>
    ///     True if the consumer has supplied enough money for the
item_price,other wise False
    /// </returns>
    /// <example>
    ///     ICashier cashier = new Cashier();  //Cashier class does not
exist, you will need to mock this out for testing
    ///     IMachine machine = new Machine();  //This Machine class is what
the developer will be writing
    ///     if (!(machine.MethodF(75, cashier)))
    ///         Console.WriteLine("The user has not put in enough money to
buy something that is $0.75");
    /// </example>
    bool MethodF(int itemPrice, ICashier cashier);

    /// <summary>
    ///     Queries the Cashier system for how much money the consumer
supplied,
    ///     and how much change is available by denomination, and
calculates if
    ///     the proper change can be made.
    /// </summary>
    /// <param name="itemPrice">
    ///     The price of the item chosen by the user in cents. ($1 = 100)
```

```
    /// </param>
    /// <param name="cashier">
    ///     An object providing cashier services for the Vending Machine
    /// </param>
    /// <returns>
    ///     True if the vending machine can make change for the item
requested
    ///     given the change on hand.
    /// </returns>
    /// <example>
    ///     ICashier cashier = new Cashier();  //Cashier class does not
exist, you will need to mock this out for testing
    ///     IMachine machine = new Machine();  //This Machine class is what
the developer will be writing
    ///     if (!(machine.MethodC(75, cashier)))
    ///         Console.WriteLine("Error: Insufficent Change in machine for
transaction");
    /// </example>
    bool MethodC(int itemPrice, ICashier cashier);

    /// <summary>
    ///     This method checks to see if a particular product is in
inventory.
    /// </summary>
    /// <param name="row">
    ///     Row is character between "A" and "Z" representing the shelf of
the users order.
    ///     You may assume that this input will always be valid
    /// </param>
    /// <param name="column">
    ///     Column is an unsigned integer between 0 and 63 representing the
column of the users order.
    ///     You may assume that this input will always be valid
    /// </param>
    /// <param name="inventory">
    ///     An object providing inventory management services for the
Vending
    ///     Machine via the IInventory interface.
    /// </param>
    /// <returns>
    ///     True if the item in the specified location is in stock
    /// </returns>
    /// <example>
    ///     IInventory inventory = new Inventory(); //Inventory class does
not exist, you will need to mock this out for testing
    ///     IMachine machine = new Machine();  //This Machine class is what
```

```
the developer will be writing
    ///     if (machine.MethodS("A",1, inventory))
    ///         Console.WriteLine("Yeay! The machine has cheetos! :) ");
    /// </example>
    bool MethodS(char row, int column, IInventory inventory);

    /// <summary>
    ///     Performs the entire transaction from the time that the user
enters
    ///     the coordinates of the desired product to dispensing the
product.
    /// </summary>
    /// <param name="row">
    ///     Row is character between "A" and "Z" representing the shelf of
the users order.
    ///     You may assume that this input will always be valid
    /// </param>
    /// <param name="column">
    ///     Column is an unsigned integer between 0 and 63 representing the
column of the users order.
    ///     You may assume that this input will always be valid
    /// </param>
    /// <param name="pricing">
    ///     An object providing product pricing services for the Vending
Machine
    ///     via the IPricing interface.
    /// </param>
    /// <param name="cashier">
    ///     An object providing cashier services for the Vending Machine
via the
    ///     ICashier interface.
    /// </param>
    /// <param name="inventory">
    ///     An object providing Inventory management services for the
Vending Machine.
    /// </param>
    /// <returns>
    ///     True if the purchase was successful, otherwise, false.
    /// </returns>
    /// <example>
    ///     IPricing pricing = new Pricing();       //Pricing class does
not exist, you will need to mock this out for testing
    ///     ICashier cashier = new Cashier();       //Cashier class does
not exist, you will need to mock this out for testing
    ///     IInventory inventory = new Inventory(); //Inventory class does
not exist, you will need to mock this out for testing
```

```
    ///     IMachine machine = new Machine();       //This Machine class is
what the developer will be writing
    ///     machine.MethodB("A", 1, pricing, cashier, inventory);
    /// </example>
    bool MethodB(char row, int column, IPricing pricing, ICashier cashier,
IInventory inventory);
}
```

Pricing

The pricing system is the index of item prices in our vending machine, and can be thought of as little more than a 2 dimensional array of prices, supplied by the vending machine owner. To aid in usability, this array is accessed through the following interface:

```
namespace VendCo
{
    public interface IPricing
    {
        /// <summary>
        ///     Accepts the row and column entered by the consumer.
        /// </summary>
        /// <example>
        ///     PriInterface pricing = new Pricing(); //Pricing class does
not exist, you will need to mock this out for testing
        ///     Console.WriteLine("Cheetos, located in dispenser A,1 cost:
$" + pricing.MethodP("A",1 / 100.0));
        /// </example>
        /// <param name="row">
        ///     Character between "A" and "Z" representing the shelf of the
users order.
        ///     You may assume that this input will always be valid
    /// </param>
        /// <param name="column">
        ///     Integer between 0 and 63 representing the column of the
users order.
        ///     You may assume that this input will always be valid
        /// </param>
        /// <returns>
        ///     Returns an integer representing the price of the specified
product in cents. ($1 = 100)
        /// </returns>

        int MethodP(char row, int column);
    }
}
```

Cashier

The Cashier system accepts money and dispenses change. Your system will use it to find out how much money the consumer has put in to the machine. Your system will also tell it how much change to dispense, and in what denominations.

```
namespace VendCo
{
    public interface ICashier
    {
        /// <summary>
        ///     Returns an integer representing the current amount of money
that the
        ///     consumer has put in to the vending machine in cents. ($1 =
100)
        /// </summary>
        /// <example>
        ///     ICashier cashier = new Cashier(); //Cashier class does not
exist, you will need to mock this out for testing
        ///     Console.Writeline("The user put in $" + cashier.MethodM() /
100.0 + " before making a selection");
        /// </example>
        int MethodM();

        /// <summary>
        ///     Returns an array with the machines inventory of quantities
of
        ///     Quarters, Dimes and Nickels. ([40, 30, 60] would then be 40
Quarters,
        ///     30 Dimes and 60 Nickels)
        /// </summary>
        /// <example>
        ///     ICashier cashier = new Cashier(); //Cashier class does not
exist, you will need to mock this out for testing
        ///     int[] change = cashier.MethodC();
        ///     int availableQuartersQty = change[0];
        ///     int availableDimesQty =    change[1];
        ///     int availableNickelsQty =  change[2];
        /// </example>
        /// <returns>
        ///     An array with the machines inventory of quantities of
Quarters, Dimes
        ///     and Nickels. (in that order)
        /// </returns>
        int[] MethodC();
```

```
        /// <summary>
        ///     Dispenses the quantity of change to the customer as
directed.
        /// </summary>
        /// <example>
        ///     ICashier cashier = new Cashier(); //Cashier class does not
exist, you will need to mock this out for testing
        ///     cashier.MethodD(1,2,3); //will dispense 60 cents in the
form of 1 quarter, 2 dimes, and 3 nickels
        /// </example>
        /// <param name="quarterCount">
        ///     The number of quarters to dispense to the customer.
        /// </param>
        /// <param name="dimeCount">
        ///     The number of dimes to dispense to the customer.
        /// </param>
        /// <param name="nickelCount">
        ///     The number of nickels to dispense to the customer.
        /// </param>
        void MethodD(int quarterCount, int dimeCount, int nickelCount);
    }
}
```

Inventory

The inventory system dispenses product, and detects when a dispenser is empty.

```
namespace VendCo
{
    public interface IInventory
    {
        /// <summary>
        ///     Determines if a dispenser is empty
        /// </summary>
        /// <example>
        ///     IInventory inventory = new Inventory(); //Inventory class
does not exist, you will need to mock this out for testing
        ///     if (inventory.MethodE("A",1))
        ///         Console.WriteLine("BOO!, The machine is out of cheetos
:( ");
        /// </example>
        /// <param name="row">
        ///     Character between "A" and "Z" representing the shelf of the
users order.
        ///     You may assume that this input will always be valid
```

```
        /// </param>
        /// <param name="column">
        ///     Integer between 0 and 63 representing the column of the
users order.
        ///     You may assume that this input will always be valid
        /// </param>
        /// <returns>
        ///     True if the a dispenser at the inventory location is empty,
and will
        ///     otherwise return false
        /// </returns>

        bool MethodE(char row, int column);

        /// <summary>
        ///     Dispenses the product to the customer from the specified
dispenser.
        /// </summary>
        /// <example>
        ///     IInventory inventory = new Inventory(); //Inventory class
does not exist, you will need to mock this out for testing
        ///     Console.Writeline("Now dispensing Cheetos");
        ///     inventory.MethodI("A",1);
        /// </example>
        /// <param name="row">
        ///     Character between "A" and "Z" representing the shelf of the
users order.
        ///     You may assume that this input will always be valid
        /// </param>
        /// <param name="column">
        ///     Integer between 0 and 63 representing the column of the
users order.
        ///     You may assume that this input will always be valid
        /// </param>

        void MethodI(char row, int column);
    }
}
```

Bibliography

Abdelmalek Amine, Ladjel Bellatreche, Zakaria Elberrichi, Erich J. Neuhold and Robert Wrembel. Computer Science and Its Applications. In proceedings of 5th IFIP TC 5 International Conference, CIIA 2015, Saida, Algeria.

Arisholm, Erik, et al. "Evaluating pair programming with respect to system complexity and programmer expertise." *IEEE Transactions on Software Engineering* 33.2. 2007. Print.

Backus, John W., et al. "The FORTRAN automatic coding system." Papers presented at the February 26-28, 1957, western joint computer conference: Techniques for reliability. ACM, 1957.

Balagurusamy, E. “Fundamental of Computers”. New Delhi: Tata McGraw Hill, 2009. Print.

Beck, Kent. “Extreme Programming Explained: Embrace Change” Second ed. Reading, MA: Addison-Wesley, 2005. Print.

Beck, Kent. Test-driven development by example. Boston: Addison-Wesley, 2014. Print.

Beck, Kent. “Why does Kent Beck refer to the "rediscovery" of test-driven development?”. *Quora.com.* 11 May 2012.Web.

Brooks, Fred. The Mythical-Man Month: Essays on Software Engineering, Anniversary Edition. Boston: Addison-Wesley, 2010. Print.

Clocksin, W. F., & Mellish, C. S. (2003). Programming in Prolog:. New York: Springer-Verlag.

Deursen, A. V., Klint, P., & Visser, J. (2000). Domain-specific languages. ACM SIGPLAN Notices, 35(6), 26-36.

Dijkstra, Edsger W. "The humble programmer." *Communications of the ACM*15.10 1972. 859-866. Print.

Fowler, Martin. "Continuous Integration." *Martinfowler.com*. 6 Aug. 2017. Web.

Fuchi, Kazuhiro, Kowalski, Robert, Furukawa,Koichi, Ueda,Kazunori, Kahn,Ken, Chikayama,Takashi & Tick,Evan. “Launching the New Era”. *Commun. ACM, 36*, 49-100, 1993. Print.

Freuder, E. (1996). In pursuit of the holy grail. ACM Computing Surveys, 28(4es). doi:10.1145/242224.242304

Goldman, Max, and Robert C. Miller. "Test-driven roles for pair programming." Proceedings of the 2010 ICSE Workshop on Cooperative and Human Aspects of Software Engineering. ACM, 2010. Print.

Gupta, R.K., Prajapati, H. & Singh, H. (2015). *Test-Driven JavaScript Development.* Packt Publishing

Hammond, Susan, and David Umphress. "Test driven development: the state of the practice." Proceedings of the 50th Annual Southeast Regional Conference. ACM, 2012. Print.

Hulkko, Hanna, and Pekka Abrahamsson. "A multiple case study on the impact of pair programming on product quality." Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on. IEEE, 2005.

Jorgensen, Paul C. “Software testing: a craftsman’s approach”. CRC press, 2016. Print.

Knuth, Donald Ervin. The art of computer programming. Reading Mass.: Addison-Wesley Publ., 1968.

Lipton, R. J. "Fault diagnosis of computer programs." *Student Report, Carnegie Mellon University*. 1971. Print.

MacConnell, Steve. “Code complete: a practical handbook of software construction”. Chapter.2.3 Common Software Metaphors: Software Penmanship: Writing Code. Microsoft Press, 1993. Print.

Martin, James. “Application development without programmers”. Prentice Hall PTR, 1982. Print.

Martin, Robert C. “Clean code: a handbook of agile software craftsmanship”. Chapter 5: Formatting: The Purpose of Formatting. Pearson Education, 2009. Print.

Maximilien, E. Michael, & Williams, Laurie. "Assessing test-driven development at IBM." *Software Engineering, 2003. Proceedings. 25th International Conference on*. IEEE, 2003. Print.

McCracken, Daniel D. “Digital computer programming”. *Program Checkout.* John Wiley & Sons, 1957. Print.

McIlroy, M. D. "Software Engineering: Report on a conference sponsored by the NATO Science Committee." 1968. 138-155. Print.

Meyerovich, L. A., & Rabkin, A. S. (2013). Empirical analysis of programming language adoption. ACM SIGPLAN Notices, 48(10), 1-18. doi:10.1145/2544173.2509515

Gordon E. Moore, Cramming More Components onto Integrated Circuits," Electronics, pp. 114–117, April 19, 1965.

Mugridge, Rick. "Test driven development and the scientific method." *Agile Development Conference, 2003. ADC 2003. Proceedings of the*. IEEE, 2003. Print.

Mullery, Alvin P., Schauer, Ralph F., & Rice, R. "ADAM: a problem-oriented symbol processor." *Proceedings of the May 21-23, 1963, spring joint computer conference*. ACM, 1963. Print.

Nielsen, Jakob. "Usability engineering at a discount." *Proceedings of the third international conference on human-computer interaction on Designing and using human-computer interfaces and knowledge based systems (2nd ed.)*. Elsevier Science Inc., 1989.Print.

O'Sullivan, Barry. Opportunities and challenges for constraint programming, Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, p.2148-2152, July 22-26, 2012, Toronto, Ontario, Canada

Peters, H., Knieke, C., Brox, O., Jauns-Seyfried, S., Krämer, M., & Schulze, A. (2014). A Test-Driven Approach for Model-Based Development of Powertrain Functions. Lecture Notes in Business Information Processing Agile Processes in Software Engineering and Extreme Programming, 294-301.

Raymond, Eric Steven. The art of UNIX programming:. Boston: Addison-Wesley, 2008.

Salomon, David. “Assemblers and loaders”. Ellis Horwood, 1992. Print.

Sun, Wenying. “The true cost of pair programming: Development of a comprehensive model and test”. Diss. University of Kansas, 2011. Print.

Sterling, L., Shapiro, E. Y., & Warren, D. H. (2010). The art of Prolog: advanced programming techniques. Cambridge, Mass: MIT Press.

Sutinen, Erkki. "Introduction to Computer Science." *cs.joensuu.fi/~appro/ics/03-01.php*. University of Eastern Finland. 2010. Web.

Vallecillo, A., Tolvanen, J., Kindler, E., Störrle, H., & Kolovos, D. (2012). Modelling foundations and applications 8th European conference ; proceedings. Berlin: Springer.

Van Gelder, Allen, Ross, Kenneth A., & Schlipf, John S. "The well-founded semantics for general logic programs." *Journal of the ACM (JACM)* 38.3 (1991): 619-649. Print.

Vanhanen, Jari, & Lassenius, C. "Perceived effects of pair programming in an industrial context." *Software Engineering and Advanced Applications, 2007. 33rd EUROMICRO Conference on*. IEEE, 2007. Print.

Williams, Laurie. “Encyclopedia of Software Engineering”. *Pair Programming*. CRC Press. 2010. Print.

Williams, Laurie, & Kessler, Robert. *Pair programming illuminated*. Addison-Wesley Longman Publishing Co., Inc., 2002. Print.

Zhao, Wei. "A language based formalism for domain driven development." *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM, 2003. Print.