I Throw Itching Powder at Tulips

Richard P. Gabriel
IBM Research
rpg@dreamsongs.com

Abstract

Programming comes in many shapes & sizes.

Categories and Subject Descriptors D.2.9 [Software process models]

General Terms Experimentation

Keywords Agile; science; programming; natural language generation

I want to remind you of something simple: Programming and software engineering are not the same things. Neither is programming the same as algorithm design. We've tangled the several notions of programming—if we try we can unweave them, but sometimes we push on too quickly / get confused. William Griswold ventured this definition of software engineering:

> *The practice of constructing software to satisfy all stakeholder requirements so as to maximize value*
>
> –William Griswold, *personal communication*, 2013

Programming is more fundamental. Venturing a guess, I would define programming as designing a set of mechanisms to enable some device (very broadly construed) to do something it normally could not using mechanisms it already has. Thinking about computers is the simple case: programming is putting together instructions in a programming language that is easy for people to understand which cause the underlying mechanisms of the computer—including its physical and electrical components—to realize the purpose of the program. But it also works for physical devices, biological systems, and people too. For example, when we teach a child to add, we are creating a program that builds on the child's existing ability to count on fingers. To the child the notion of adding is novel, but perhaps counting on fingers is not. At first, addition is a program; later it is an ability.

When we describe how to drive from one place to another, that's a program that uses the driver's ability to understand directions, to drive, and to recognize telltales to get that person from one place to another.

When people try to put together large software systems—large enough that teams are needed and dangerous enough that safety is crucial—they apply engineering techniques (as best they can) to the project. That's the start of software engineering. When people wonder whether the program they have devised really will achieve its desired purpose using the underlying mechanisms of the "computer," that's the start of the theory of computation and algorithm design.

In recent years the conflict of more traditional software engineering approaches and agile have made a muddle of the concept of programming—to a degree where sometimes only software engineering is considered programming. I think.

Software engineering is not what I do when I program. I am programming. I write software as part of doing science. I use software as a machine or instrument to explore how the mind / brain might work. Not when the mind's thinking—that's old AI. I mean when people are creating.

Software engineering is for producing something that someone can describe, either using specifications (the old-fashioned way) or a product backlog (or some other suchlike thing in an agile setting). I don't do anything like that. And software engineering is about producing software in a group.

The way we teach programming makes everything software engineering. Pundits' statements are deceiving:

> *Programmers mediate between the negotiated and uncertain truths of business and the crisp, uncompromising domain of bits and bytes and higher constructed types.*
>
> –Kevlin Henney, *97 Things Every Programmer Should Know*, 2010 [1]

Onward!'14, October 20–24, 2014, Portland, Oregon, USA.

ACM 978-1-4503-2585-1/14/10…$15.00.
http://dx.doi.org/10.1145/2661136.2661155

This looks correct, innocent even. When I program I indeed worry whether the programs I write will do something like what I intend; but there are no "truths of business" in sight. The most progressive teachers reinforce this and other stereotypes, like this programming assignment:

> *Develop a function that when given an initial amount of money (called the principal), a simple annual interest rate, and a number of months will compute the balance at the end of that time. Assume that no additional deposits or withdrawals are made and that a month is 1/12 of a year. Total interest is the product of the principal, the annual interest rate expressed as a decimal, and the number of years.*
>
> –Felleisen et al, *How To Design Programs* [3]

A business-related problem; a problem posed to the student by an outsider; a problem. Everything that the program should accomplish is spelled out, by an expert instructor or "customer"—of course it is solvable, of course it is achievable, of course all thinking is reduced to whether the program achieves its intended purpose using the underlying mechanisms of the computer at hand. Programming happens, but in a limited context. A very well-respected professor ventured this opinion to me:

> *I think the biggest mistake we make with the starting point for undergraduate education is that we introduce programming at all. The right starting point, IMHO, is requirements and specification together with the associated mathematics that they require.*
>
> –anonymous, *ruminating on a first course*

Problem solving; being told what to solve, what goals to achieve. This is hammered into students and job seekers who are asked repeatedly to solve problems (homework / job interviews). The idea of "a problem" doesn't necessarily encompass the certitude of solution, but as taught in the context of programming, the unintended implication is that a problem is a puzzle, and puzzles have solutions. A puzzle is a test of ingenuity.

My characterization of why I program—building a software machine to explore nature and create a theory—might remind you of Peter Naur's "Programming as Theory Building" [2]. Naur is talking about a related but different task—the creation of software as the final goal:

> *…the primary aim of programming is to have the programmers build a theory of the way the matters at hand may be supported by the execution of a program.*
>
> –Peter Naur [2]

The "matters at hand" are roughly the stuff in the real world that the program being built needs to handle, and the theory in question is a conceptual framework for understanding the means by which software achieves that handling.

In my case, the theory is a scientific theory to be discovered, and the software is an instrument to help discover / forge that theory, which the software does by reacting to or revealing something about the material agency of the world. The resulting software may or may not be interesting by itself. My model of doing science with software can be used to discover the Naurish theories programmers build to create software, and it might happen that the software created in my model is the software the programmers eventually create.

Productivity and value are essentials for business programming. Jeff Sutherland wrote this about Scrum:

> *Scrum is a simple framework used to organize teams and get work done more productively with higher quality. It allows teams to choose the amount of work to be done and decide how best to do it, thereby providing a more enjoyable and productive working environment. Scrum focuses on prioritizing work based on business value, improving the usefulness of what is delivered, and increasing revenue, particularly early revenue.*
>
> –Jeff Sutherland, *A Brief Introduction to Scrum*, 2007 [4]

Notice that the assumed context is business, creating value for a customer who drives requirements and judges acceptability, and programming while consuming cash slowly and producing revenues quickly. After being taught (implicitly) that programming begins when someone (a teacher) tells you to start, and that the goal / topic / domain / problem for programming is given (by that teacher), working as part of someone else's machine is not foreign. Sutherland continues:

> *Designed to adapt to changing requirements during the development process at short, regular intervals, Scrum allows teams to prioritize customer requirements and adapt the work product in real time to customer needs. By doing this, Scrum provides what the customer wants at the time of delivery (improving customer satisfaction) while eliminating waste (work that is not highly valued by the customer).*
>
> –Jeff Sutherland, *A Brief Introduction to Scrum*, 2007 [4]

Agile's contribution was to turn on its head the following premise of earlier software engineering methodologies: change is expensive and needs to be avoided. Or at least limited to the earliest possible parts of requirements gathering and design. The idea is that making a change to a design is cheaper than making a change to an implementation. Steve McConnell tells it this way:

> *In the worst case, reworking a software requirements problem once the software is in operation typically costs 50 to 200 times what it would take to rework the problem in the requirements stage (Boehm and Papaccio 1988). It's easy to understand why. A 1-sentence requirement can expand into 5 pages of design diagrams, then into 500 lines of code, 15 pages of user documentation, and a few dozen test cases. It's cheaper to correct an error in that 1-sentence requirement at requirements time than it is after design, code, user documentation, and test cases have been written to it.*
>
> –Steve McConnell, 1996 [5]

As if correcting that 1-sentence requirement were that easy. Just change some of the words, right? It's easy, for example, to change plans for a vacation from driving to Bisbee, Arizona, to flying to the French Riviera—just change the modes of transportation and some hotels in the plan—but the cost of the change will hit hard later. This sort of flawed thinking is super-easy to fall prey to while sharpening one's gullibility for thinking in stereotypes. If it's easy to change a bad requirement to a good one, think how easy it is to change a good requirement to bad. Well, all one need do is expand the 1-sentence to 5 pages of design diagrams and think about them; or then into 500 lines of code and think about them; or into 15 pages of user documentation and then a few dozen test cases and think about them—that's when you see the problem. Making the change is easy while knowing the change is smart is hard.

This is where agile comes in. The bug, they claim, is that as long as the code being produced isn't running in a way that the "customer" can observe, errors in requirements can persist longer than need be—because the customer is unable to observe and then intervene. They say

> *Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.*
>
> –Agile Manifesto Principles [6]

And this enables them to

> *Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.*
>
> –Agile Manifesto Principles [6]

The differences between up front planning and agile approaches seem similar to the differences in approaches to art: experimental versus conceptual. To wit: for experimental artists:

> *… planning a painting is unimportant. The subject selected might be simply a convenient object of study, and frequently the artist returns to work on a motif he has used in the past. Some experimental painters begin without a specific subject in mind, preferring instead to let the subject emerge as they work. Experimental painters rarely make elaborate preparatory sketches. Their most important decisions are made during the working stage. The artist typically alternates between applying paint and examining the emerging image; at each point, how he develops the image depends on his reaction to what he sees.*
>
> –Galenson, *Old Masters and Young Geniuses* [7]

For conceptual artists:

> *…planning is the most important stage. Before he begins working, the conceptual artist wants to have a clear vision either of the completed work or of the process that will produce it. Conceptual artists consequently often make detailed preparatory sketches or other plans for a painting. With the difficult decisions already made in the planning stage, working and stopping are straightforward. The artist executes the plan and stops when he has completed it.*
>
> *...extreme practitioners...make all the decisions for a work before beginning it. It is unclear, however, if this is literally possible. There are artists who came close to it, and perhaps achieved it, during the 1960s, by making plans for their work and having these plans executed by others.*
>
> –Galenson, *Old Masters and Young Geniuses* [7]

Picasso was a conceptual artist (mostly), and he did Big Design Up Front. If there was a key 1-sentence requirement that made possible his masterpiece, *Les Demoiselles d'Avignon* [8], it took Picasso more than 400 studies and sketches (a record for artistic preparation) to get it right.

But as always, both software engineering camps are vassals—they are handed problems, given direction, and paid for piecework. In general.

None of this is new thinking. And it's not really at the heart of the matter that's engaging me. Right now I am using programming very differently from what the software engineering approaches assume and celebrate, and also from what teachers of programming prepare students for. I program to explore scientific questions. What I produce are instruments that help me peer into the unknown. I don't work on puzzles but on mysteries. I don't have customers; neither requirements; nor specifications; nor test cases (really); nor design issues of the same sorts as software engineers; there are no roadmaps; everything is a prototype and also an end-product; I don't know whether the next thing I try will work, can work, should work; and I am profoundly disappointed when my programs fail to surprise me. If traditional software engineering and agile define two points on a spectrum, what

I do is as far from agile as agile is from traditional software engineering, with agile in the middle between my spot and traditional SE.

This is how I see it.

My next task is to show you what I do and let you judge whether it depicts programming differently from how software engineers and algorithm designers see it—differently from how you see it. I believe programming is a fundamental tool of discovery and creativity which has been harnessed to serve the needs of industry, capitalism, as well as the greater good—that the boring parts of programming are immensely valuable, but also that those boring parts contain islands of programming like the continent I live on.

I am working on a natural language generation (NLG) system as part of a DARPA project. The specific thrust of my work is to take a template of a counter-message and tailor it for the audience—that is, to the person or people to whom it will be sent. This Template Reviser (as I originally called it—it's called *InkWell* now) is intended to be a precursor to a full-blown natural language generation (NLG) system. I worked on an NLG system for my PhD back in the late 1970s, and this can be viewed as a matured sequel.

When I worked on NLG back then, I had an interest in writing and creativity but not a lot of experience or education. Since then I've worked a bit on understanding creativity, but more importantly I got an MFA in Creative Writing, namely in poetry. I've published a small book of poems, and have been writing a poem a day for fourteen years now. I've written a novel (unpublished), and have four other published books. With that education and experience I've come to realize that writing (creatively) involves a wide variety of implicit influences and contributing factors—influences and factors that determine word choice, phrasing, and structure. The plain meanings of words can tell one story, and other stories can be told by the connotations of those words, linkages between ideas and images can be made with sound—the so-called *music* of the words—and secondary and tertiary structures can be established by using these and other writing craft elements.

The DARPA project is called "Social Media in Strategic Communication" (SMISC). One of its statements of work:

> *Take a template of a counter-message and tailor it for the audience.*

This goal is the extent of my requirements and my only interaction with the "customer." I interact with researchers through conversations, email, research papers, and meetings, but the purpose of those interactions is to get ideas, report on findings, and to gather encouragement. In no way do these interactions seem like instructions, direction, or orders. I believe that those sorts of things pop up in my work, but from a very different source, and for very different purposes than for software delivery to businesses. This is not a puzzle because there is no correct answer nor is anyone around to declare my program's counter-messages acceptable.

Commercial software is generally not exciting software. It rarely breaks new ground; if anything is difficult it's difficult because its algorithms might be hard or performance is elusive or because the right kinds of data structures are hard to pin down. Of course there are exceptions. In many cases these difficulties are hidden by frameworks, middleware, libraries, and the like.

And though what the software does might be boring, its design and construction are likely not, and there is tremendous pleasure in designing and building something of value and beauty. But rarely is the construction of commercial software a grand challenge—sometimes it is, but not frequently. This reality makes the task of creating commercial software mostly a matter of getting the details the way the customer likes. Admirable, but not my game. Consider:

> *But merely extending knowledge a step further is not developing science. Breeding homing pigeons that could cover a given space with ever increasing rapidity did not give us the laws of telegraphy, nor did breeding faster horses bring us the steam locomotive.*
>
> –Edward J. v. K. Menge [9]

I do science.

The scenario I work from is this: some group is engaged in a persuasion campaign that deserves to be thwarted—this can be a phishing attack ("your account has been compromised; please re-enter your critical information here"), a protest that could become dangerous ("WBC will picket the sodomite whorehouse and dog kennel masquerading as St. Agnes Catholic Church in religious protest and warning"), or terrorist plotting. Through monitoring social media, the organizers of the persuasion campaign and their social networks are identified. Analysis produces the material required to plan a counter-messaging campaign. My program delivers that campaign, using the materials gathered.

What makes this hard is that it isn't likely that telling the persuaders "don't do that" is going to work, nor is it clear that those being targeted will respond more to my messages than to the persuaders'. What needs to be understood are the motivations, the objectives, the incentives, and the styles of communication likely to work as dissuasion.

I need to figure out a program that will generate messages that include influences designed to dissuade and that considers personal characteristics of the message recipients. For example an influence might be a bias toward choosing words with cheerful connotations or using Biblical rhythms to im-

ply moral authority. A personal characteristic could be a set of perceived needs and attitudes.

My approach is to develop a set of templates—but not the boring kind you might imagine—figure out how to select the right ones, figure out how to compose them elegantly, and determine how to tune them for the audience.

One of the primary ways to appeal to an audience is to exhibit a particular set of personality traits. The group I work with in the lab uses the so-called "Big Five" personality traits [10] [11] with a good dose of other personality facets and values. Big Five is a consolidation of approaches to assessing personality based on examining texts people use to describe themselves. For example, if a person says:

- I am the life of the party
- I don't mind being the center of attention
- I feel comfortable around people
- I start conversations
- I talk to a lot of different people at parties

this is evidence that the person is *extraverted*. Each of the five traits represents a spectrum with the following endpoints and definitions:

- ***Openness***: inventive/curious vs. consistent/cautious. Openness reflects the degree of intellectual curiosity, creativity, and a preference for novelty and variety.
- ***Conscientiousness***: efficient/organized vs. easy-going/careless. Conscientiousness reflects a preference for organization, dependability, discipline, duty, and achievement—planned instead of spontaneous.
- ***Extraversion***: outgoing/energetic vs. solitary/reserved. Extraversion represents energy, positive emotions, urgency, assertiveness, sociability, and a tendency to seek stimulation. Talkativeness.
- ***Agreeableness***: friendly/compassionate vs. analytical/detached. Agreeableness is compassion and cooperation rather than suspicion and antagonism; it represents a trusting and helpful approach.
- ***Neuroticism***: sensitive/nervous vs. secure/confident. Neuroticism is the tendency to experience unpleasant emotions easily (anger, anxiety, depression, and vulnerability); it refers to emotional stability and impulse control. [11]

Personality traits are determined by a sort of simple linguistic analysis—"to what degree do you agree with the following statements?" But researchers in my group have taken that much further. Rather than looking at responses to directed questions, the analysis looks at texts people write. This work is based on writing samples from volunteers who also have taken personality tests, and then machine learning was used to establish a function that takes text and produces a judgment about personality. This is pretty clever work that I didn't do, but I needed to understand it well enough to embed it in my NLG program.

The process works by examining the words in a writing sample, and counting the numbers of words in each of 68 categories ([Figure 1] on the next page). This approach is based on the work of James Pennebaker called "Linguistic Inquiry and Word Count" (LIWC—pronounced "luke") [12]. This yields a vector of percentages—for each category there is a corresponding percentage of words in the document that fall into that category. These counts are determined by a dictionary that maps words to categories. For example, the word "agony" maps to categories 12, 16, and 19, which are *Affect*, *Negative affect*, and *Sadness*, resp. Using these probabilities, it's possible to compute an estimate of the writer's Big-Five personality traits, using the work of Tal Yarkoni [13]. Yarkoni found correlations between LIWC scores and the Big Five traits, and these correlations are expressed as a simple linear combination of the probabilities that LIWC computes.

Here's an example of all this. Suppose this is a text we want to analyze:

Reading Gabriel's essays is pure agony.

The LIWC scores for this are as follows:

Category	Count	Percentage of all Words
Sadness (19)	1	16.7%
Cognition (20)	1	16.7%
Perception (27)	1	16.7%
Certainty (26)	1	16.7%
Seeing (28)	1	16.7%
Affect (12)	1	16.7%
Present (39)	1	16.7%
Negative Emotion (16)	1	16.7%

This of course is too small a sample for accuracy—the LIWC dictionary recognizes only these words: *reading, is, agony*—but this is just an example of how the analysis works.

Applying Yarkoni's coefficients we get the following Big-Five analysis:

Trait	Value
Agreeableness	-4.2%
Conscientiousness	-16.34%
Extraversion	–
Neuroticism	9.90%
Openness	-8.51%

These values should be interpreted like this: The sign says whether the trait is evident (+) or its opposite is (-); the mag-

All pronouns	Numbers	Cause@Causation	Social	Space	Leisure	Symptoms & sensations
1st person singular	Affect	Insight	Communication	Up	Home	Sexual
1st person plural	Positive affect	Discrepancy	Reference to others	Down	Sport/exercise	Eating/drinking
Total 1st person	Positive feelings	Inhibition	Friends	Inclusion	TV/movies	Sleeping/dreaming
Total 2nd person	Optimism	Tentativeness	Family	Exclusion	Music	Grooming
Total 3rd person	Negative affect	Certainty	Humans	Motion	Money	Swear words
Negations	anxiety	Sensation/perception	Time	Occupation	Metaphysical	Non-fluencies
Assents	Anger	Seeing	Past	School	Religion	Fillers
Articles	Sadness	Hearing	Present	Job	Death	
Prepositions	Cognition	Touching	Future	Achievement	Physical states/factors	

Figure 1

nitude ranges from 0% to 100%, which represents the range of theoretically possible values. *Extraversion* is left blank because there were no Yarkoni coefficients associated with it in the set of non-zero elements of the LIWC vector—meaning there is no evidence.

A loud bray may be heard almost two miles away.

–traditional

The Agile Manifesto—when I saw it the first time I laughed like a jackass. I still chuckle and it's been years:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

…not because it's wrong, but because I always have the feeling that the authors of this list believed they'd taken the righthand statements and stated their opposites as their favored values—that they thought, for example, that "working software" is an opposite of "comprehensive documentation," and that by doing so they have succeeded in making a set of outrageous (but true) statements. (I love agile development, so don't get the wrong impression.) Take a look at my aphorisms for programmers doing science (immediately before the references).

Here's what I mean, example-wise. The subtitle of Kent Beck's extreme programming book is "Embrace Change" [14]. This and the last bullet above make it seem like embracing change is a radical new idea, and as completely different from design up front as you can get. But embracing change is the midpoint of the range. One endpoint is to fear change so much you plan forever; the midpoint is to embrace change; and the opposite endpoint is to inject change / suggest it / insist on it. Other dimensions might exist.

In my world of programming I use the following manifesto:

- Nature over individuals and interactions
- Insights over working software
- Problem engagement over customer collaboration
- Grappling with mystery over responding to change

Let's look at one of these for a minute. What does "insights over working software" mean?

To understand how this LIWC / Big Five / Yarkoni thing works, I decided to program it up myself. Moreover, I needed code for this in the inner loop for the NLG system I was going to write.

After this exercise I had working software right in front of me. My results replicated all the published results I could find as well as the results from my group's Java implementation. While reading the Yarkoni paper I noticed this statement:

> *The results converge with other recent findings suggesting that, contrary to popular wisdom, people do not present themselves in an idealized and overly positive way online, and maintain online identities that reflect the way they genuinely see themselves and are seen by others.*
>
> –Yarkoni [13]

I slightly mistook this to be saying that people cannot very easily hide their personalities when they write. The work of Pennebaker seemed predicated on this as well. And earlier in the Yarkoni paper I read, "previous studies have found systematic associations between personality and individual differences in word use" [13]. At the top of the next page is a set of Big-Five measurements of six corpora. The ones inside the red outlines refer to things I have written; looking from left to right, they are my book "Patterns of Software," a collection of about 5,000 of my poems (that's not a typo—five thousand), my book "Writers' Workshops and the Work of Making Things," and an unpublished novel I wrote ("Traditional Salvation") [Figure 2]. That is, essays, poems, nonfiction, and fiction. The next two are "Leaves of Grass" by Walt Whitman and the collected stories of Ernest Hemingway.

One thing that became clear to me looking at this chart is that I don't correlate well with myself. The red outlines are around things I wrote—each a different genre. The righthand figure on the next page ([Figure 3]) is grouped according to genre a bit. In the red outlines are my poems and Whitman's; not in outlines are my novel and Hemingway's stories. Now things look correlated better—but by genre and not by person.

I decided to do an experiment: what if instead of the Big

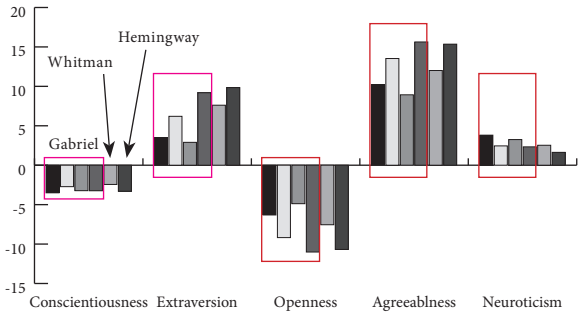


Figure 2

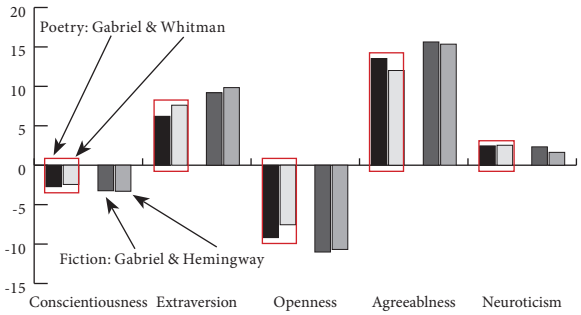


Figure 3

Five personality traits, I postulated three genre-based traits: Poetry, Fiction, and Nonfiction. And like the work my group did, what if I used machine learning to discover coefficients that would map LIWC scores to genres?

My first attempt was a failure. I took a set of training corpora, labeled them with binary judgments—poetry, fiction, or nonfiction, exclusively—and used a simple machine learning algorithm to develop classifiers for each genre. The machine learning algorithm did not converge. If I looked at it as a software engineering problem, I would have explored whether I had coded the machine learning algorithm correctly, or whether perhaps I had not applied it correctly. I had been using this algorithm and my implementation of it for about ten years, so that didn't seem like a good approach.

The reason for divergence was that it was foolish to think a piece of literature is purely poetry or any of the other genres. What about narrative poetry, lyrical fiction? I then created a training target where each corpus was labeled as a mixture of genres [Figure 4]. With this the machine learning algorithm converged. When I tested it, though, it was a little off. I tacked on two decision trees—one for the main genre and another for a thing I called a "mixin genre," which is a kind of secondary characteristic. For example, Shakespeare's Hamlet is classified by my classifier as Fiction with a Poetry mixin—not a bad result. Here are the two decision trees—one for main genre:

-5.0 ≤ P (poetry)
-5.0 ≤ NF (nonfiction)
otherwise (fiction)

and the other for mixin genre:

-10.0 ≤ P < -5.0 (poetry mixin)
-10.0 ≤ NF < -5.0 (nonfiction mixin)
35.0 ≤ F (fiction mixin)

P, NF, and F are the outputs of the three learned sensors. The results of the classifier on a number of corpora are in [Figure 5]. (The parenthetical judgments are what those works are considered to be in literary circles.)

The system correctly identifies genre more than 80% of the time, and in the cases where it isn't completely right, it's half-right 37% of the time. The predicate shows that fiction is not special, and analyzing what the genre sensors look for, one can identify the characteristics that determine genre.

This work also shows that the LIWC instrument probably mixes a variety of detected signals, and so it is not a pure sentiment instrument.

Agile goes half way wrt change: from resist change to welcome change—what about inject change? By creating opportunities for / making changes, a scientist explores, then discovers, and later understands. The world was pretty satisfied with Newtonian relativity for a long time, but physicists kept monkeying with the ideas.

The genre detector shows this. I had working code that was perfectly fine, and my employer doesn't really care about literature and creative writing. Nevertheless, my genre exploration made my group question and explore the nature of the (computational) instruments they were building—along with the science behind it. Solid ground truth is hard to come by because in most cases it comes from studies of undergraduates and Mechanical Turkers. This implies that it is usually a bad plan to sit pat on the current (scientific) understanding and the code that realizes that understanding's implications.

A scientific pursuit has no boss or teacher, aside from nature or reality—a creature not interested in collaboration. If you lock yourself away with theory and rumination, you will dig yourself a hole with you always at the bottom, because the mind can't so easily work on pure thought-stuff. The mind needs something to play with, and the more visceral that play, the better. Even writing things down and staring at them, or tossing equations or expressions on the chalk board and erasing / revising is needed. Software is a machine scientists dream up to explore nature.

Usually software is a way to explore data associated with scientific research. Gather a pile of data and then analyze

Corpus	Poetry	Fiction	NonFiction
Poemsrpg (P)	85.0	-10.0	-10.0
Leaves of Grass (P)	95.0	-30.0	-50.0
Traditional Salvation (F)	-10.0	80.0	-25.0
Hemingway (F)	-10.0	95.0	-75.0
Patterns Of Software (NF)	-35.0	-5.0	95.0
Writers' Workshop (NF)	-10.0	-2.0	90.0
Faulkner (F)	-5.0	95.0	-65.0
Ulysses (F)	-5.0	90.0	-15.0
Emily Dickinson (P)	95.0	-25.0	-80.0
Unabomber (NF)	-70.0	-50.0	85.0
Wizard of Oz (F)	-25.0	85.0	-35.0
Call Of The Wild (F)	-12.0	87.0	-55.0
Huckleberry Finn (F)	-5.0	45.0	-40.0
Metamorphosis (F)	-25.0	70.0	-35.0
Origin Of Species (NF)	-80.0	-10.0	75.0

Figure 4

it. In the situations I'm talking about, the software itself is a direct path into the material being studied—this following the physicist turned philosopher, Andrew Pickering [16], who says that science is a process of finding a stable point between a *conceptual framework* (a theory) and the *material agency* of the world as revealed through *machines*—or instruments. Scientists build machines to explore reality, and what the machine does or reports is interpreted according to the conceptual framework. When the machine reports something unexpected or contradictory (a *resistance* in Pickering's terminology), the conceptual framework is adjusted along with the machine (usually) (each adjustment is called an "accommodation"), and more observations are made. Eventually, the conceptual framework, the machine, and the resistances settle down, and a fact is manufactured.

In this sort of scientific exploration, the software will talk to you. But I don't mean that it's talking about itself—the code I write is an intermediary between my research thinking and the part of the world I'm looking at. It's like a telescope from the early days of science or the Large Hadron Collider these days. I don't accept working software / I keep pushing it / I keep changing it until an insight drops out.

In the case of the genre exploration, the tool being developed helped me stumble on something only tangentially related to my direct scientific work. This happens all the time in other scientific disciplines. Many, though, consider software and programming a hard-headed tool like a shovel and therefore not suited for direct exploration. This is silly, of course.

Current methodologies avoid my type of programming. It isn't software for hire where business value is being created, nor is it defense software. Defense considerations—the code **must** work—led to the heavy methodologies.

Corpus	Classfication	Corpus	Classification
Knott (P)	Poetry	Gribble / Fedora (P)	Poetry
Trakl (P)	Poetry	Janet Holmes / Humanophone (P)	Fiction[Poetry]
Lanier (P)	Poetry	Janet Holmes / F2F (P)	Poetry
The Wasteland (P)	Poetry	Front Page NYT Article (NF)	Fiction[Nonfiction]
Moby Dick (F)	Fiction	Richard Schmitt / Kodiak (F)	Poetry[Fiction]
Gay Stories (F)	Fiction	Richard Schmitt / A Year of Counseling (F)	Poetry[Fiction]
To Kill a Mockingbird (F)	Fiction	Harper / Prac. Found. for Prog. Lang (NF)	Nonfiction
Hamlet (?)	Fiction[Poetry]	Ellen Bryant Voigt / Song and Story (P)	Poetry
Bertrand Russell (NF)	Nonfiction	Tennyson / In Memoriam (P)	Poetry
Charles Babbage (NF)	Nonfiction	US Constitution (NF)	Nonfiction
Darwin (NF)	Nonfiction	Tom Lux / I Love You Sweatheart (P)	Fiction
Crazy CS Person (NF)	Poetry	rpg / Sharp Tone (P)	Poetry
Bible (?)	Fiction[Poetry]	Cass Pursell / Men and Stones (F)	Fiction
Pete Turchi's New Book (NF)	Fiction[Nonfiction]	Proust's Longest Sentence (F)	Fiction

Figure 5

One of the little secrets about business is that firms are very conservative. They want to win by being "better," but only enough better to win—not a lot better. Business value is mostly about catching up quickly. Rarely, I think, is it about being quick as a first mover. First movers come from the kind of programming & science I am talking about, and usually that takes place in research labs, universities where startups are incubated, etc. The rules I use make no sense for defense nor for business.

Another purpose of InkWell is to serve as the software half of a writing *centaur*, a ***centaur*** being a human/machine collaboration. InkWell takes text as input and a (largish) set of constraints, and produces a number of possible revisions while endeavoring to satisfy and balance the stated constraints. A writer uses InkWell to assist with revisions, and the workflow is iterative with the writer creating / tweaking textual templates and constraints, and InkWell producing revisions, which feed back into the process.

The term *centaur* originated in computer chess, and refers to the pairing of a human chess player and a chess-playing computer, usually a PC or laptop. Garry Kasparov [17] came up with the idea of such collaborations, and the chess community supplied the colorful and metaphorical name. There are some major differences between a chess program and InkWell as computer halves of centaurs. The chess playing

computer helps avoid blunders the human might make. InkWell suggests avenues of exploration the human might miss.

There are two goals InkWell serves:

- mimic a specific writer
- assist creativity in writing

InkWell takes a template (example in the Appendix ([Figure A1]), which is a specification of original text annotated with which words are variable and characteristics of those words for InkWell to consider. There are also a number of other, writing-related constraints written either as local bindings in the template or stated in the UI, which specifies global parameters and constraints. The example in [Figure A1] shows how a writer might express a template describing Robert Frost's "Stopping by Woods on a Snowy Evening" [15]. Here are some of the general ways to mimic a writer:

- match specified (or measured) Big Five personality traits and associated personality facets; match basic human values as described by Schwartz [18] and Chen [19]
- match a writer's word choice: favored words, word music, word length, favored mood
- match writing patterns: *n-grams* (2-, 3-, 4-, and 5-grams); an ***n-gram*** is a series of *n* words in a row that has appeared in a naturally occurring, existing text

Assist Creativity: every writer has days when they "have it" and days when they don't. Books, articles, blog posts, courses, coaches, and workshops exist to help writers defeat writer's block. Or some have just a little less talent than preferred. InkWell can help here too, using these techniques:

- use conservative or wild synonym choice (*associative* versus *dissociative* writing)—search diameter, search distance, preference for nearby, preference for far away, various synonym aspects (hypernyms, meronyms, etc)
- satisfy constraints like word-length, alternative meanings, word rhythms
- favor echoes (similar sounding words) and rhymes
- select words based on ontology (concepts), proximity in the synonym network, or a cluster of word-centric concepts to favor or avoid
- favor specific word groups or avoid them
- specify constraints, both local and global
- take into account a writing mood specified by a construct called a *halo*

Any constraint can be inverted: e.g. sound like a particular writer or sound like anyone but that writer, rhyme two words or ensure they don't, observe n-grams or deliberately violate them. InkWell produces any number of candidate revisions, and the writer can pick and choose revisions and wordings.

The notion of a *halo* is a good example of mimicking writerly thinking. A ***halo*** is a mood device. You specify a set of words, and InkWell starts with each of those words and fans out along synonym arcs to other words. Where several of these wavefronts hit, those words are given more weight in the revision process. Looking at Frost's poem, the line

> *The woods are lovely, dark, and deep*

is revised this way

> *The woods are bright, light, and high*

when given the happiness halo:

> *Delighted, Ebullient, Ecstatic, Elated, Energetic, Enthusiastic, Euphoric, Excited, Exhilarated, Overjoyed, Thrilled, Tickled pink, Turned on, Vibrant, Zippy*

and this way

> *The woods are hot, rough, and cold*

when given the anger halo:

> *Affronted, Belligerent, Bitter, Burned up, Enraged, Fuming, Furious, Heated, Incensed, Infuriated, Intense, Outraged, Provoked, Seething, Storming, Truculent, Vengeful, Vindictive, Wild*

A long time ago (~1980) I wrote a simple NLG system—my PhD thesis was this: a generalized planning system based on loose descriptions of individual agents, heuristic matching, resource-limited computation, and mixed planning and execution could do a good job of producing text [20]. The system was called **Yh**, and it was a small-data program. It was about 75,000 lines of code and had maybe 10,000 dictionary entries and language-related agent descriptions. Yh was used as the tail-end of an automatic programming system (called **PSI** [21]) at the Stanford Artificial Intelligence Lab to describe in English the programs produced (how they worked) and in the mixed-initiative user dialog that gathered the specifications for the programs to be generated.

InkWell is different; it has these parts:

- WordNet synonym dictionary: 160,000 words [22] [23]
- 5,000 most common words
- CMU phonetic dictionary : 125,000 words [24]
- rhyming dictionary: 42,000 words
- stem dictionary: 163,000 entries (+ Porter Stemmer + Lemmatization)

- n-grams: 30m from general literature; 100,000–1,000,000 per writer including the Google 2-grams [25] and the COCA 3-, 4-, and 5-grams [26]
- ~30,000 lines of code: template compiler, constraint optimizer, word & phrase adjustments, etc
- n-grams (including 1-grams) from a specified writer; currently there are around 50 writers to choose from (and supplying new ones is trivial)

A naïve flow diagram is to the right [Figure 6].

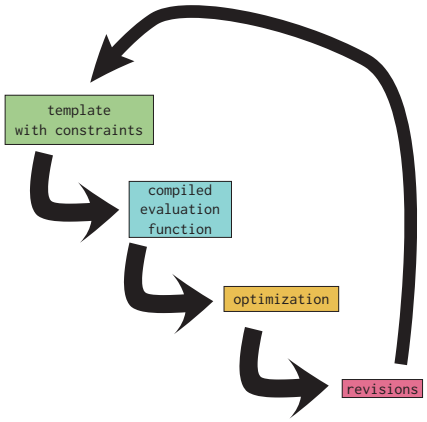


Figure 6

A template resembles a Lisp program whose body looks like text with parenthesized annotations. The template along with all the specified constraints and parameters is compiled into an evaluation function which returns 0 when all constraints are satisfied. InkWell selects a set of candidate replacement words and phrases, and an optimization process then selects the combination of words and phases that best satisfy (minimize) the evaluation function. The optimization process uses simulated annealing—mostly because the optimization is over discrete word choices and I haven't been able to find a better process. (One nice characteristic of simulated annealing is that I don't have to worry about staging the order of choices—I can just add constraints and let the relatively undisciplined SA process do its thing.) This produces (as many) revisions (as the writer wants); the writer can pick and choose which suggestions to use, and the template can be altered to do more focused explorations.

For example, to evaluate how close a set of word choices is to a specified personality profile, InkWell analyzes the proposed text to (computationally) determine its exhibited personality. This is then compared to the target, and either the current text is considered a step in the right direction or a step in the wrong direction. Because the analyzed texts are generally pretty short, the LIWC dictionary needed to be expanded, which was done algorithmically using the WordNet synonym dictionary, a decay function, and semantic categories derived from the LIWC categories—this expanded the LIWC dictionary from about 2,300 words to about 25,000 words—this is now the dictionary my group uses. The computation of personality scores needs to be fast, which is accomplished by pre-computing as much of the LIWC categories as possible and using caches.

Let's look at some output. I'll use the familiar Frost poem [15], but we'll look at only the last two stanzas. The Lisp [27] UI (and yes, I know the UI is ugly and horrible) for InkWell is shown in the Appendix. So is the original poem. Here are the original last two stanzas:

He gives his harness bells a shake
To ask if there is some mistake.
The only other sound is the sweep
Of easy wind and downy flake.

The woods are lovely, dark, and deep.
But I have promises to keep
And miles to go before I sleep,
And miles to go before I sleep.

The first revision specifies conservative synonym search (don't traverse far from the originally specified words), and InkWell is instructed to prefer word choices that Ernest Hemingway [28] used in his short stories as well as short words. Here is that result (changes from original underlined):

He gives his harness bells a shake
To ask if there is some mistake.
The only other sound is the brush
Of comfortable wind and small snowflake.

The forest is lovely, off, and abundant.
But I have promises to keep
And miles to go before I sleep,
And miles to go before I sleep.

The second variant specifies wild synonym search (go far and wide), and Walt Whitman [29] is the writer to sound like (& use short words):

He gives his work values a run
To say if there is some misconception.
The only other being is the brush
Of simple appearance and new person.

The lands are divine, certain, and low.
But I have loves to keep
And runs to go before I go,
And runs to go before I undergo.

The last example of these stanzas is my favorite. It's like the Whitman one except I've asked InkWell to sound a little like a very theoretical and a bit controversial computer scientist [30]:

> He gives his direction presentations a can
> To hold if there is some misconception.
> The only other element is the exit
> Of informal termination and yielding person.
>
> The developments are satisfying, insensitive, and extended.
> But I have complexities to terminate
> And amounts to go before I separate,
> And amounts to go before I admit.

In addition, InkWell was instructed to prefer long words. Looking at the template in the Appendix, you can see some interesting things—things that are specified as soft constraints:

- InkWell tries to match the currently loaded writer's Big Five personality traits; for the last example this is the computer scientist; call this the "target writer"
- the target writer's word choices and n-grams are matched
- InkWell is told to make as many chosen words echo as possible; two words echo when the share word sounds—this is derived from the CMU Phonetic dictionary
- InkWell tries to make all the words it chooses different from each other
- for the examples that target Hemingway and Whitman, InkWell is asked to prefer short words; this is in terms of syllable count, as derived from the CMU Phonetic dictionary
- the `bind` statements are like Lisp's `let`; they bind variables to a word-choice specifications, and all occurrences of the same variable in the body of the text will be replaced by the same word; e.g., `w2` always refers to the same word or phrase that means "snow"
- variable words are called out; they are annotated with their parts of speech (including semantic category, if known (e.g. `verb-cognition`)) and other constraints
- many of the constraints are easily understood; for example this expression `(ref w2 :different w1 :rhyme w1)` means the word selected to mean "snow" should be different from the word selected to mean "know," and that those two words should rhyme; the expression `(ref w3 :echo w1)` means that the word selected to mean "queer," "odd," or "unusual" should echo the word selected to mean "snow"; the expression `(mile noun-quantity pl)` means the word chosen to mean "mile" should be a noun in the semantic category "quantity," and that when that word is expressed in the final output, it should be made plural
- others are not so clear; the expression `(sleep verb :different sleep :rhyme sleep)` means that the word selected to mean "sleep" here shouldn't be the same as the word selected in response to the binding for `sleep` (the last word of the previous line), but should rhyme with it; the expression `(wood noun-plant pl :+sense [forest] :-sense [wood])` means that the word chosen to mean "wood" should be in the semantic category "plant," should be made plural, should be of the same sense as the word "forest" and not the same sense as "wood," (the material trees are made of)—this is achieved by starting at the word "forest" and spreading out, increasing the strength of each word encountered by an amount that decays toward 0 with distance (the amount starts positive for `:+sense` words and negative for `:-sense` words)
- predicates can be specified for each word, and a ***pervasive predicate*** is one that applies to all of them; the predicate `syllable-bonus-few` indicates a preference for short words

The n-grams are used to try to maintain some degree of familiarity and sense.

What did I start with if I didn't start with hard requirements or a spec? I had had a ten-year hobby of using simulated annealing to solve puzzles and do conference room scheduling. I had developed several SA frameworks, and it was fun and productive. As a writer I had a sense that I balanced a lot of concerns while writing, especially poetry. I felt that these concerns were not structured and that I didn't consider them in a particular order.

I started with the idea that I could represent a paragraph as a sequence of boxes containing words, phrases, and other things, that each box could have an associated set of alternatives, that I could represent the concerns as numerically measurable features viewed as soft constraints, and that SA would be able to do its magic to balance all those concerns to select the best alternatives. My first constraints were the Big Five personality measurements, the n-grams, and the proximity of synonyms to original words. All the other constraints in InkWell fell out of trying to think of how to measure numerically the strength of a (possible) craft element—how much do these two words rhyme, does this word better match a given semantic sense, is this word more like what a particular writer would use. Experience with InkWell guided me, and I'd say it led me by the nose.

Kasparov laments our turning away from mystery and focusing on puzzles. He doesn't put it exactly that way:

> *This is our last chess metaphor, then—a metaphor for how we have discarded innovation and creativity in exchange for a steady supply of marketable products. The dreams of creating an artificial intelligence that would engage in an ancient game symbolic of human thought have been abandoned. Instead, every year we have new chess programs, and new versions of old ones, that are all based on the same basic programming concepts for*

> *picking a move by searching through millions of possibilities that were developed in the 1960s and 1970s.*
>
> –Garry Kasparov [17]

Does this also say that in turning away from creativity / mystery we have turned to puzzles? Turned to providing immediate value to firms? Kasparov goes on:

> *Like so much else in our technology-rich and innovation-poor modern world, chess computing has fallen prey to incrementalism and the demands of the market. Brute-force programs play the best chess, so why bother with anything else? Why waste time and money experimenting with new and innovative ideas when we already know what works? Such thinking should horrify anyone worthy of the name of scientist, but it seems, tragically, to be the norm. Our best minds have gone into financial engineering instead of real engineering, with catastrophic results for both sectors.*
>
> –Garry Kasparov [17]

Whenever I add a new constraint type to InkWell I have to go through a period of getting re-acquainted with it. InkWell operates in a very complicated space of constraints, and finding the good spots takes time. After revising InkWell, I must revisit all my personal heuristics about how to write templates to get it to be creative, cautious, wild, or conservative. It's a pleasurable re-familiarization—kind of like getting to know a new lover: things roughly work the same, but all the details and nuances are fresh and exciting. I typically allocate a few days to do this, and I often send off some of the results to my poet friends for their amusement.

InkWell is in Common Lisp. I write in Common Lisp because I know it well, know how to do crazy things with it, and it flows from my fingers rapidly. InkWell is complex: maybe half a dozen compilers (two very substantial), the optimization infrastructure and InkWell manifestation of it, the synonym machinery, the constraint computation machinery, the parallelization to make the synonym discovery and optimization run acceptably fast, the numerous caches to make things run fast enough to be tolerable (even with the parallel stuff).

For example, the template shown in the Appendix for the Frost poem compiles to a Lisp function 1952 lines long. This function is composed of the following:

- 16 (static) calls to the rhyming predicate
- a (static) call to a function that measures how echo-y a set of 44 words are
- a (static) call to a function that measures how diverse the same set of 44 words are
- 126 (static) calls to a function that determines whether pairs of words are known 2-grams
- 120 (static) calls to a function that determines whether triples of words are known 3-grams
- 5 (static) calls to the LIWC / Big Five personality computation

This function is run in the inner loop of the optimization process. The template specification in the Appendix runs it a million times. That means that the function that determines whether a pair of words in the revised text is in the Google 2-grams set is called 126,000,000 times.

An example of the way the evaluation function works is rhyming. Suppose a template specifies that the words selected for the variable words *v1* and *v2* should rhyme. When two actual words, *w1* and *w2*, are chosen, they are passed to a function (120 lines of Lisp code including subfunctions) that computes a rhyming score, $0 \le r \le 1$, for them, based on an algorithm for rhyming. Roughly, that algorithm looks at all phonetic spellings of *w1* and *w2* in pairs, starts at the ends of each pair, and computes how much each syllable rhymes along with how many syllables rhyme. The maximum rhyme score for all pairs is chosen for *r* and then "flipped" (*1-r*) so that a perfect rhyme yields 0. All the soft constraints are treated this way. All such evaluations are summed, and the overall evaluation function is minimized over all word choices. This approach enables InkWell to select these three delightfully unexpected words as rhymes at the end of the Frost poem: *gulp*, *hole up*, and *nap*.

To make this run fast enough to be usable, there are multiple layers of caches that memoize these function calls and parts of them.

I could probably figure out how to make a more compact evaluation function for my needs—perhaps by finding other ways to compute the constraints or their equivalents. But the combination of straightforward computation and lots of caches makes experiments easy even if the code is complicated.

Given this, I was able to do some interesting investigations that might not have been possible were things more streamlined. For example, after I stumbled across the CMU Phonetic Dictionary [24] and came up with the rhyme-strength algorithm, I was able to approach the poetic concept called "echoes." Before this discovery I had no realistic approach for how to measure it. Later I will be able to analyze rhythm because the phonetic dictionary scores stress levels for syllables.

The words selected need to be fleshed out for output. For example, recall that the word for "mile" needs to be turned into a plural. Moreover, the WordNet synonym dictionary contains phrases as well as words. Handling the details for all these adjustments takes a pile of algorithms, many tables, and complicated special cases. The code for this part of the system is 1000 lines right now, and I generally add to it / revise it with every other new template I try.

My methods of exploring how to get a program to choose words and phrases varied over time depending what my collaborator, InkWell, was teaching me. I always would stick with the basics of the scientific method, but I would chase hunches, build unlikely infrastructures to explore what seemed like dead ends, and generally would use the heuristic of looking intently, from time to time, at the least likely idea.

InkWell itself—as well as my old NLG program Yh—does the same thing with respect to abandoning common sense. In InkWell, simulated annealing works (well) because it will occasionally / randomishly choose to make changes to the state of affairs that make things worse. This is likely the reason it's so good at finding unusual rhymes. In Yh I used a technique I called *counterinduction* after the philosophical concept of the same name [31]. The idea was that when planning progress is advancing slowly or not at all, Yh would allocate a lot of resources to explore less likely approaches. Kind of like a chess program that can do heuristic estimates of potential next moves choosing to explore to some depth the consequences of making a move the heuristics don't like.

Here is how the biologist Kim Lewis puts it:

> *This is part of what I teach my students—how to shut down your common sense…. You have to start looking for a perfect solution and ignore whether it's realistic. That mindset helps you battle your 'common sense,' which is what prevents you from inventing new things.*
>
> –Kim Lewis[32]

"Gabriel, you loser…we have a word for this in agile—it's called a ***spike."***

> *Sometimes a user story is generated that cannot be estimated until the development team does some actual work to resolve a technical question or a design problem. The solution is to create a "spike," which is a story whose purpose is to provide the answer or solution. Like any other story or task, the spike is then given an estimate and included in the sprint backlog.*
>
> –http://www.solutionsiq.com/resources/glossary/bid/56550/Spike [33]

It's funny how when I talked about this with audiences in the past and got the "you loser" comment, I couldn't really answer the criticism. The answer is in the above quote but only today while writing this text did I realize it. "…Resolve a technical question or a design problem."

> *A spike solution, or spike, is a technical investigation. It's a small experiment to research the answer to a problem. For example, a programmer might not know whether Java throws an exception on arithmetic overflow. A quick ten-minute spike will answer the question.*
>
> – Shore & Warden, The Art of Agile Development [34]

Agile is aimed at assisting a business guy create business value right away. The developers are not interested in figuring out what that business value is, but simply wish to hear it told to them.

> *At that 2001 meeting in Snowbird where we wrote the Agile Manifesto, Kent Beck stated one of our goals: "… to heal the divide between development and business."*
>
> –Robert Martin, The *True* Corruption of Agile [35]

My goal is to resolve a mystery, but it's not a technical question about how to design or code InkWell—though I have plenty of those—it's a mystery about what makes for creative and artistic natural language generation. Spikes are detours developers take to figure out things about the programs they are writing; they are not detours to figure out what business values to pursue. Confusing spikes and Naur's theory building for what I am up to is the same mistake twice.

> *"Alright friends, you have seen the heavy groups, now you will see morning maniac music. Believe me, yeah. It's a new dawn. Good morning, people!"*
>
> –Grace Slick, Jefferson Airplane, Woodstock, August 16, 1969.

We arrive at the crux. What is programming? It's easy to be confused—by things like this for example:

> *A software development methodology or system development methodology in software engineering is a framework that is used to structure, plan, and control the process of developing an information system.*
>
> –http://en.wikipedia.org/wiki/Software_development_methodology [36]

Software development involves programming, but software development isn't programming. Methodologies are about appropriate ways to develop software in an engineering-related context. For military purposes and for safety-critical purposes, it's essential to not make a mistake, and the way to do that is lots of conceptual planning near the beginning to be certain nothing can go wrong. This leads to the heavy methodologies.

In business contexts the high-order bit is to get working software fast, and there is a premium for helping firms catch up quickly to competitors, veer ahead with a new (but typically incremental) product, or respond to sudden changes of direction the customer might throw into the hopper. But as

with the military contexts, the endgame is a software-related artifact in the domain's real world doing things that someone needs or wants.

There are other contexts. One is science—but a particular type of science. It's not science where data gathered from instruments is analyzed. It's when the software and its programming form a machine to explore nature alongside the scientist. It could be a simulation that helps the scientist understand what's going on, or it could be like InkWell which is trying to create an artificial model of what writerly creativity is.

Another, similar, context is learning about a topic through programming it up. For example, my ability to understand lots of technical things is limited, and so I program them up myself to see how the mechanisms lock together to make it happen.

Another is to explore what a question that's easy to ask might actually mean. For example, as part of the InkWell project I am exploring what "rhyminess" could mean. In writer circles a particular writer might be considered more (or less) *musical* than another; what does that mean? My first answer was that it is the percentage of words in a text that rhyme. I programmed that up and found the idea doesn't take into account a text that is extremely rhymey in a couple of isolated places and otherwise flat.

My next idea was to take a window about 100 words wide and pass that over a writer's corpus skipping ahead 50 words at a time (that is, overlapping the windows), computing the percentage of words that rhyme in each 100-word group, and reporting the average of those percentages. This resulted in scientists and nonfiction writers being very rhymey because there would be knots of high-rhyme bundles, usually because lots of technical words rhyme for no tasteful reason.

I noticed that people considered rhymey had narrower standard deviations than flat writers for the set of rhyminess windows. So I tried a formula where a writer's rhyminess is the average rhyminess minus twice the standard deviation. The results are pretty intuitive, but I think there is a stronger notion of periodicity at work that needs to be considered. (There are some rhyminess scores in the Appendix.)

The reason I told this rhyminess story was to demonstrate that the questions being explored with software this way are not strictly about analyzing data (like creating a simple word usage model of a writer) nor how to achieve the requirements of the program (how to implement a theory of rhyminess), but what a concept might mean.

It might seem that the programming stories I told are just a set of projects I worked on—extending the LIWC dictionary, programming up a literary genre detector, algorithmic rhyming, exploring the concept of rhyminess, and tailoring texts using optimization—but they are just waypoints on a single journey to discover how natural language generation can be done and what creativity means in writing. InkWell and all its inner stuff is the machine I lug around with me as I explore this space. It's my learning machine.

I program to explore.

If you learned something from this essay I would be very disappointed. I've simply pointed out that programming is not software engineering, and because of that, the principles and practices of the heavy methodologies and agile are too limiting and even irrelevant. Programming is like writing in that sense. You can write a 5-paragraph theme for homework, you can write a requirements document for your division, you can write a marketing piece for your product manager, you can take effective writing courses. Or you can do this:

> *"I write entirely to find out what I'm thinking, what I'm looking at, what I see and what it means. What I want and what I fear."*
>
> –Joan Didion, *Why I Write* [37]

Or this:

> *You may wonder where plot is in all this. The answer… is nowhere…. I believe plotting and the spontaneity of real creation aren't compatible…. I want you to understand that my basic belief about the making of stories is that they pretty much make themselves. The job of the writer is to give them a place to grow.*
>
> –Stephen King, *On Writing* [38]

Or this

> *But during my very early writing, certainly before I'd published, I began to learn characters will come alive if you back the fuck off. It was exciting, and even a little terrifying. If you allow them to do what they're going to do, think and feel what they're going to think and feel, things start to happen on their own. It's a beautiful and exciting alchemy. And all these years later, that's the thrill I write to get: to feel things start to happen on their own.*
>
> *So I've learned over the years to free-fall into what's happening. What happens then is, you start writing something you don't even really want to write about. Things start to happen under your pencil that you don't want to happen, or don't understand. But that's when the work starts to have a beating heart.*
>
> –Andre Dubus III, *By Heart* [39]

That's why I wrote this essay—to find out.

Acknowledgments

Some of this work was supported by DARPA (W911NF-12-C-0028).

Appendix: Code Examples & Stuff

```
(with-personality-traits (*writer-big-five*)
 (with-global-constraints ((all-echo)(all-different))
  (with-pervasive-predicates (#'syllable-bonus-few)
   (bind ((w1 (know verb-cognitive)) (w2 (snow noun-substance)) (w3 (or (queer adj) (odd adj) (unusual adj)))
          (w4 (or (year noun-quantity) (week noun-quantity) (month noun-quantity) (season noun-quantity)))
          (w5 (shake verb)) (w6 (flake noun)) (here (here adj)) (near (near adj))
          (mile (mile noun-quantity pl)) (sleep (sleep verb))
          (woods (wood noun-plant pl :+sense [forest] :-sense [wood])))

"Whose (ref woods) these are I (think verb-cognition) I (ref w1).
 His (house noun) is in the (village noun) though;
 He will not see me (stop verb gerund) (ref here :rhyme near)
 To (watch verb-perception) his (ref woods) (fill verb) up with (ref w2 :different w1 :rhyme w1).

 My (little adj) (horse noun-animal) must (think verb-cognition) it (ref w3 :echo w1)
 To (stop verb) without a (farmhouse noun) (binding near :rhyme w3)
 Between the (ref woods) and (frozen adj) (lake noun)
 The (darkest adj) (evening noun) of the (ref w4 :different w3 :rhyme w3).

 He gives his (harness noun) (bell noun pl) a (ref w5 :echo w3)
 To (ask verb) if there is some (mistake noun :rhyme w5).
 The only other (sound noun) is the (sweep verb)
 Of (easy adj) (wind noun) and (downy adj) (ref w6 :different w5 :rhyme w5).

 The (ref woods) are (lovely adj), (dark adj), and (deep adj).
 But I have (promise noun pl) to (keep verb :rhyme sleep)
 And (ref mile) to go before I (ref sleep),
 And (ref mile) to go before I (sleep verb :different sleep :rhyme sleep)."))))
```

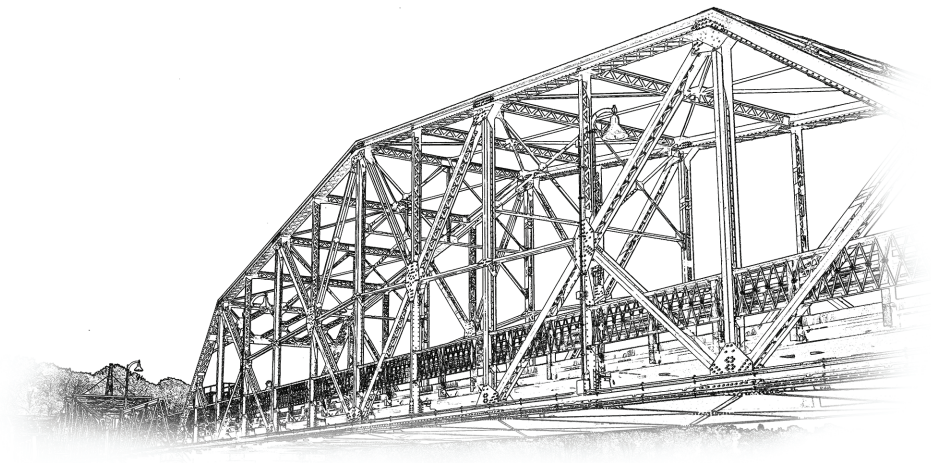
Stopping by Woods on a Snowy Evening

Whose woods these are I think I know.
His house is in the village though;
He will not see me stopping here
To watch his woods fill up with snow.

My little horse must think it queer
To stop without a farmhouse near
Between the woods and frozen lake
The darkest evening of the year.

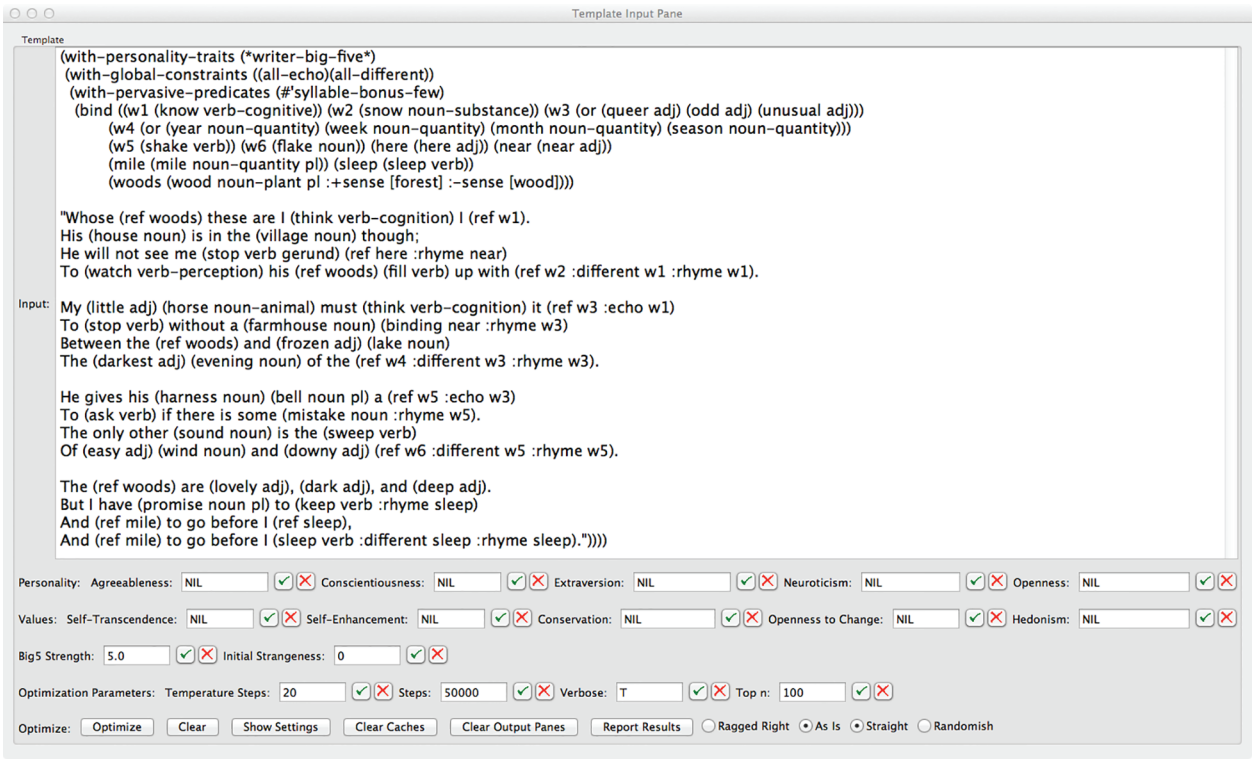
He gives his harness bells a shake
To ask if there is some mistake.
The only other sound's the sweep
Of easy wind and downy flake.

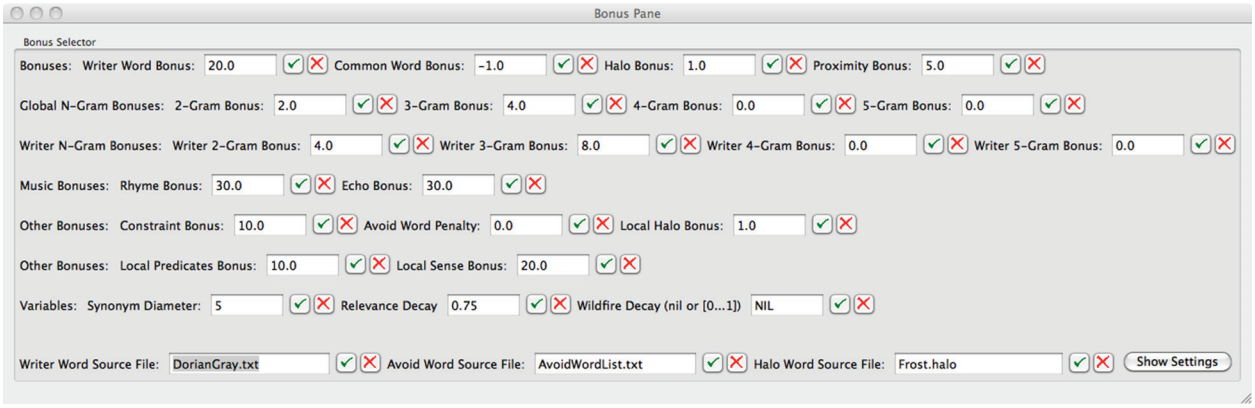
The woods are lovely, dark and deep.
But I have promises to keep,
And miles to go before I sleep,
And miles to go before I sleep.

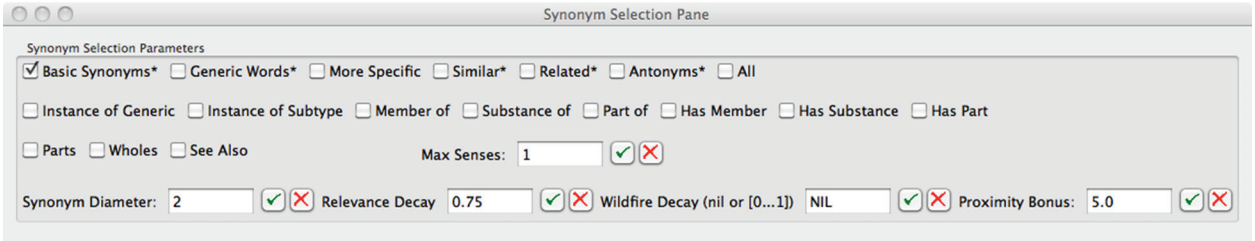


©rpg

Figure A1


Template Input Pane
Template
(with-personality-traits (*writer-big-five*)
(with-global-constraints ((all-echo)(all-different))
(with-pervasive-predicates (#'syllable-bonus-few)
(bind ((w1 (know verb-cognitive)) (w2 (snow noun-substance)) (w3 (or (queer adj) (odd adj) (unusual adj)))
(w4 (or (year noun-quantity) (week noun-quantity) (month noun-quantity) (season noun-quantity)))
(w5 (shake verb)) (w6 (flake noun)) (here (here adj)) (near (near adj))
(mile (mile noun-quantity pl)) (sleep (sleep verb))
(woods (wood noun-plant pl :+sense [forest] :-sense [wood])))
"Whose (ref woods) these are I (think verb-cognition) I (ref w1).
His (house noun) is in the (village noun) though;
He will not see me (stop verb gerund) (ref here :rhyme near)
To (watch verb-perception) his (ref woods) (fill verb) up with (ref w2 :different w1 :rhyme w1).
Input:
My (little adj) (horse noun-animal) must (think verb-cognition) it (ref w3 :echo w1)
To (stop verb) without a (farmhouse noun) (binding near :rhyme w3)
Between the (ref woods) and (frozen adj) (lake noun)
The (darkest adj) (evening noun) of the (ref w4 :different w3 :rhyme w3).
He gives his (harness noun) (bell noun pl) a (ref w5 :echo w3)
To (ask verb) if there is some (mistake noun :rhyme w5).
The only other (sound noun) is the (sweep verb)
Of (easy adj) (wind noun) and (downy adj) (ref w6 :different w5 :rhyme w5).
The (ref woods) are (lovely adj), (dark adj), and (deep adj).
But I have (promise noun pl) to (keep verb :rhyme sleep)
And (ref mile) to go before I (ref sleep),
And (ref mile) to go before I (sleep verb :different sleep :rhyme sleep).""))))
Personality: Agreeableness: NIL Conscientiousness: NIL Extraversion: NIL Neuroticism: NIL Openness: NIL
Values: Self-Transcendence: NIL Self-Enhancement: NIL Conservation: NIL Openness to Change: NIL Hedonism: NIL
Big5 Strength: 5.0 Initial Strangeness: 0
Optimization Parameters: Temperature Steps: 20 Steps: 50000 Verbose: T Top n: 100
Optimize: Optimize Clear Show Settings Clear Caches Clear Output Panes Report Results Ragged Right As Is Straight Randomish


Bonus Pane
Bonus Selector
Bonuses: Writer Word Bonus: 20.0 Common Word Bonus: -1.0 Halo Bonus: 1.0 Proximity Bonus: 5.0
Global N-Gram Bonuses: 2-Gram Bonus: 2.0 3-Gram Bonus: 4.0 4-Gram Bonus: 0.0 5-Gram Bonus: 0.0
Writer N-Gram Bonuses: Writer 2-Gram Bonus: 4.0 Writer 3-Gram Bonus: 8.0 Writer 4-Gram Bonus: 0.0 Writer 5-Gram Bonus: 0.0
Music Bonuses: Rhyme Bonus: 30.0 Echo Bonus: 30.0
Other Bonuses: Constraint Bonus: 10.0 Avoid Word Penalty: 0.0 Local Halo Bonus: 1.0
Other Bonuses: Local Predicates Bonus: 10.0 Local Sense Bonus: 20.0
Variables: Synonym Diameter: 5 Relevance Decay 0.75 Wildfire Decay (nil or [0...1]) NIL
Writer Word Source File: DorianGray.txt Avoid Word Source File: AvoidWordList.txt Halo Word Source File: Frost.halo Show Settings


Synonym Selection Pane
Synonym Selection Parameters
Basic Synonyms* Generic Words* More Specific Similar* Related* Antonyms* All
Instance of Generic Instance of Subtype Member of Substance of Part of Has Member Has Substance Has Part
Parts Wholes See Also
Max Senses: 1
Synonym Diameter: 2 Relevance Decay 0.75 Wildfire Decay (nil or [0...1]) NIL Proximity Bonus: 5.0

Rhyminess Scores

Writer / Piece	Rhyme Score	Average Rhyminess	Max Rhyminess	Min Rhyminess	Standard Deviation
Mark Twain: The Adventures of Huckleberry Finn	53.12	61.30	73.45	47.37	4.09
King James Bible	52.65	62.46	94.68	42.11	4.90
Ernest Hemingway Stories	51.56	60.58	75.38	40.20	4.51
Franz Kafka: The Metamorphosis	51.42	58.57	68.70	48.53	3.58
Alfred, Lord Tennyson, : In Memoriam A. H. H.	51.36	59.00	71.30	47.97	3.82
Robert Frost: poems	51.30	59.50	73.74	44.44	4.10
Lewis Carroll: Through the Looking Glass	50.92	58.55	70.59	48.00	3.82
Bible: New International Version	50.65	60.05	87.67	35.71	4.70
William Faulkner Stories	50.31	59.78	77.69	42.86	4.74
Richard Gabriel: Traditional Salvation (novel)	50.16	58.60	72.7	44.03	4.22
••• (35 entries)					
Harper (technical) [30]	42.06	54.27	89.71	26.60	6.10
••• (3 entries)					

Upright twitcher is not low paid. [40]

Aphorisms for Science Programmers

Aphorism: *Create opportunities for change.*

If you want something new, invite change—don't wait. Change is how to explore; exploring is how to discover. If you require a master to instruct change, you have no opportunities for discovery.

Aphorism: *Engage continuously with your software.*

Software is the machine that connects you to the realm you're exploring; thinking is fine, but seeing is more important.

Aphorism: *Code and scientists must work together.*

Software will tell you when it is the wrong instrument. If it is silent, suspect it. Don't accept working software—keep pushing it, keep changing it until an insight drops out.

Aphorism: *The first thought that comes to mind is almost certainly a cliché.*

I learned that in writing school. For research, this can mean that the first thing you think to try for your program could be way off base, even if it seems to be working well. Remember: projects given to you are mere puzzles, worthy of a homework problem, not a mystery that can give rise to science

When working with software as part of doing science, avoid working on puzzles unless they are part of the ugly innards of the program. Lots of good algorithm design can come from puzzles, but there is a difference between the reaction that something is clever and that something is amazing. In a sense, puzzles are engineering problems, and when the puzzle is large enough, it can become a mystery and the outcome—if reached—is likely science. Not guaranteed; but likely.

Aphorism: *There can be no interaction or collaboration unless you engage with the mystery your software reveals.*

If you turn away, the mystery flees; stare back / don't blink.
Mysteries make us uncomfortable.

Aphorism: *If you understand exactly what your code does, it's taught you nothing…it's reflecting you, not nature.*

When you feel comfortable with it and turn your back, it just laughs and laughs. If you can't figure something out, use mystery (or machine learning).

Aphorism: *If your effort is sustainable, you aren't learning anything.*

You have to push yourself or you are not only expending sustainable effort but you are sticking close to what you (and everyone else) know. In the old days, scientists pulled all-nighters—for weeks on end. This is not sustainable. Surprise pushes you and you respond with passion; passion means you can't stop; not stopping is not sustainable. Don't do what is sustainable—make it so you're surprised. But don't explore yourself into an early grave.

Aphorism: *Technical excellence and good design are* for engineers.

Pay attention to technical excellence, and mystery slips away—and with it nature and science.

This is not as wrongheaded as it sounds. There are programmers who design and build quite well without expending a lot of planning and engineering effort. Consider composers. One would expect that someone well-trained and well-practiced at writing music, performing music, and thinking about music would be more likely to produce a good song in a single day than could a randomly selected person. This is what talent means:

> *the skill to do something that is hard*
>
> –rpg

A great researcher who is also a great programmer can focus on the exploration and not the details of making the instruments and sensors. Such a one can pay attention to the insights coming into view and explore diversions, which is where insights and science might be. Engineering is for producing reliable things, and that takes a lot of special knowledge and skill, and also a lot of historical information to be able to know what the problems will be and which puzzles must be solved *in each particular case*. This is indeed hard, and it's what the heavy methodologies and agile are designed to facilitate. Waterfall and Scrum are the midwives of engineered artifacts.

Engineered artifacts aren't science.

Aphorism: *Simplicity is beside the point. Nothing is wrong with simplicity.....later.*

References

[1] Kevlin Henney, *97 Things Every Programmer Should Know: Collective Wisdom from the Experts*. O'Reilly Media, 2010.

[2] Naur, P., "Programming as Theory Building," In: *Microprocessing and Microprogramming, Vol. 15,* pp. 253–261, 1985.

[3] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, *How to Design Programs: An Introduction to Programming and Computing*. The MIT Press, 2001.

[4] Jeff Sutherland and Ken Schwaber, "A Brief Introduction to Scrum." *The Scrum Papers: Nuts, Bolts, and Origins of an Agile Process*. http://assets.scrumfoundation.com/downloads/2/scrumpapers.pdf?1285932052, 2007.

[5] Steve McConnell, http://www.stevemcconnell.com/articles/art04.htm, 1996.

[6] http://agilemanifesto.org/principles.html

[7] Galenson, David W. *Old Masters and Young Geniuses: The Two Lifecycles of Artistic Creativity*. Princeton University Press. Princeton, NJ, 2007.

[8] http://en.wikipedia.org/wiki/Les_Demoiselles_d%27Avignon

[9] Edward J. v. K. Menge, *The Quarterly Review of Biology*, 1930.

[10] John, O. P., & Srivastava, S. "The Big-Five trait taxonomy: History, measurement, and theoretical perspectives." L. A. Pervin & O. P. John (Eds.), *Handbook of personality: Theory and research*, (Vol. 2, pp. 102–138). Guilford Press, New York, 1999.

[11] http://en.wikipedia.org/wiki/Big_Five_personality_traits

[12] Yla R. Tausczik and James W. Pennebaker, "The Psychological Meaning of Words: LIWC and Computerzed Text Analysis Methods," *Journal of Language and Social Psychology*, 29(1) 24–54, 2010.

[13] Tal Yarkoni, "Personality in 100,000 Words: A large-scale analysis of personality and word use among bloggers." *Journal of Research in Personality*, 44(3): 363–373, June 2010.

[14] Kent Beck, *Extreme Programming Explained: Embracing Change*. Addison-Wesley, 2000.

[15] Frost, Robert, *New Hampshire*. Henry Holt. New York. 1923.

[16] Andrew Pickering, *The Mangle of Practice: Time, Agency, and Science*. The University of Chicago Press, 1995.

[17] Garry Kasparov, "The Chess Master and the Computer." *The New York Review of Books*, February 11, 2010.

[18] Schwartz, S. H., "Basic human values: theory, measurement, and applications." *Revue Française de Sociologie*, 47(4), 2006.

[19] Jilin Chen, Gary Hsieh, Jalal Mahmud, Jeffrey Nichols. "Understanding Individual's Personal Values from Social Media Word Use." *Proceedings of CSCW 2014*, Baltimore, MD, February 15–19, 2014.

[20] Gabriel, Richard P., "An Organization for Programs in Fluid Domains." *STAN-CS-81-856, AIM-342*, Stanford University, 1981.

[21] Green, Cordell, "A Summary of the PSI Program Synthesis System." *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, Cambridge, Massachusetts, August 1977.

[22] George A. Miller (1995). "WordNet: A Lexical Database for English." *Communications of the ACM* Vol. 38, No. 11: 39–41.

[23] Christiane Fellbaum (1998, ed.) *WordNet: An Electronic Lexical Database*. Cambridge, MA: MIT Press.

[24] http://www.speech.cs.cmu.edu/cgi-bin/cmudict.

[25] http://storage.googleapis.com/books/ngrams/books/datasetsv2.html

[26] Corpus of Contemporary American English, http://www.ngrams.info/download_coca.asp

[27] http://www.lispworks.com/.

[28] Ernest Hemingway, *The Complete Short Stories of Ernest Hemingway*. Scribner, 1998.

[29] Walt Whitman, *Leaves of Grass*. 1855.

[30] Robert Harper, *Practical Foundations for Programming Languages*, Carnegie Mellon University, Spring, 2010.

[31] Feyerabend, Paul. *Against Method*. 4th ed., New York, NY: Verso Books, 2010.

[32] Kim Lewis, http://www.northeastern.edu/magazine/it-takes-more-than-a-brilliant-scientific-mind-to-make-a-major-breakthrough/

[33] http://www.solutionsiq.com/resources/glossary/bid/56550/Spike

[34] James Shore & Shane Warden, *The Art of Agile Development*. O'Reilly Media, November 2, 2007.

[35] Robert Martin, "The *True* Corruption of Agile," http://blog.8thlight.com/uncle-bob/2014/03/28/The-Corruption-of-Agile.html.

[36] http://en.wikipedia.org/wiki/Software_development_methodology

[37] Joan Didion, "Why I Write." *New York Times Book Review*, December 5, 1976.

[38] King, S., *On Writing*, Pocket, New York, 2002.

[39] Joe Fassler, "The Case for Writing a Story Before Knowing How It Ends." *The Atlantic*, Oct 8 2013.

[40] Federico Garcia Lorca, "Only mystery makes us live. Only Mystery," caption on sketch.