From Software Creationism to Software Evolutionism

François-René Rideau
TUNES Project
fare@tunes.org

Abstract

The lives we live are woven around the stories we tell. This is true of programmers as of all humans. Now the greatest of all stories are *origin stories*. In a first part, I will examine the origin stories of software, from simple tales of software creation to elaborate theories of software evolution. As I do, I will relate these stories to the tools they explain and the technological realities we bring about by following them. In a second part, I will conclude by reflecting on storytelling, on the progression of the above stories, and on what lies beyond. Stories are fun! And they subtly inform us. Let me tell you a good story...

Keywords Creationism, Evolutionism, Software Engineering, Tools, Storytelling, Logogony, Anthropodicy

1. Introduction and Disclaimer

1.1 Logogonies and Anthropodicies

If you have to structure software along informal design patterns rather than formal abstractions, you have *run out of language* (to quote Rich Hickey). But if you can't recognize and discuss informal patterns in the structure of software *development*, you have not yet entered the realm of language with respect to software engineering.

Now the most powerful patterns about how humans behave are *stories*: they explain the behavior of a protagonist in terms of purposes and challenges, with causes and consequences for success and failure. Human brains are attuned to stories, whether implicit or explicit, and humans are wont to cast themselves into roles defined by stories. Furthermore, the most powerful stories are *cosmogonies* and *theodicies*: a *cosmogony* tells the origin of the existence of the Universe, and the role of God (or Gods) in this origin; a *theodicy* is a trial of those Gods, to explain the existence of Evil. This essay explores the most powerful stories about Software, *logogonies* and *anthropodicies*: a *logogony* tells the origin of the existence of Software, and the role of Man (or Men) in this origin; an *anthropodicy* is a trial of Man, to explain the existence of Bugs.

1.2 Claims and Non-Claims

This essay should by no means be construed as either insulting or vindicating anyone's religious or irreligious beliefs regarding cosmogonies. Indeed this essay makes no claim regarding the origins of Man — it only discusses the origins of Software! This essay doesn't even make any specific claim as to which story best explains the origins of which software — it only tries to make each logogony understood, as well as its implications. Each reader can take it from there. If the presented story arc does promote a thesis about the origins of software, in the end it is that, as far as the emergence of software goes, *Man is no God* — which should hardly offend anyone.

Still, this essay does make a claim: that *stories matter* — that you should examine the stories you live by, be careful which you choose to accept or to reject, and constantly refine them. For they will shape you. So yes, question my beliefs; and question your own — that's the point, whichever your or my beliefs are.

1.3 From Über-God to Underdog, and Beyond

In the next two parts of this essay, I explore a gamut of increasingly elaborate logogonies, and accompanying anthropodicies — stories about how software, and accompanying bugs, come into existence. First, creationist stories; then, evolutionist stories. To each story, we associate tools that humans use while writing software, that each fit into the narrative yet not the previous ones.

In a second part of this essay, I step back from the stories themselves, and discuss stories as a tool: what we can tell about them, how they shape our behavior. I then conclude with some remarks about what to expect from future stories.

2. Software Creationism

2.1 The Simplest Logogony

How does software come into existence? Isn't it obvious? If you ask a layman or a newbie, his explanation might be as follows:

At first the machine was uninitialized and blank; then Man said: "Let Software be such" — and so Software was.

This simplest of stories is *Software Creationism*: it casts the programmer as a God Almighty outside and above the machine; the software is His creation, His proxy, embodied in the machine.

This belief not only comes naturally to non-programmers when confronted with the apparition of software; it was also historically the first logogony assumed; and it is still the tacit logogony underlying most undergraduate computer science education: in exercises and tests, assignments and projects, students are expected to produce a perfect solution to a perfect specification, from what is defined arbitrarily as "scratch", whether on paper or on a restricted test machine. Their code is evaluated once by the teacher; it must stand alone and neither rely on any code by anyone else nor contribute to such.

No programming tools are necessary in this logogony; just pen and paper to write the perfect program, and a switchboard to insert the program into the machine. A perfect Programmer-God does not need tools: He transmits His perfect ideas to the machine directly in its memory in binary (or base ten, etc.). Programming the machine is best done directly in the machine's language, for optimum representation of the God's perfect idea.

Indeed, that's not how software is written these days — which only shows that this naive logogony has long lost its dominance.

2.2 The Simplest Anthropodicy

Software creation by a superior God is a beautiful story; however, anyone who ever endeavors to program soon realizes that programs seldom run perfectly at the first try, or even the second. Bad things happen: bugs, tyops, mismanipulations, cosmic rays, malfunctions, errors, even outright mistakes.

If the Programmer-God is perfect (at least once trained) — then whence do Bugs come?

Sophisticated philosophers will claim that while a perfect Programmer has a perfect program in mind, its rendition onto an imperfect finite machine is necessarily limited in form by physical constraints — a mere shadow of its platonic ideal.

But most people readily rush to the simplest explanation: *the Devil*. A devil modifies software in a way counter to God's intent. Whether this Devil is an opposing force outside God, a personality defect within God, or a necessary artefact of the laws of Nature that God created — is unclear and might not really matter. What clearly matters is that bad things are symptoms of the presence of an Evil force.

2.3 The Birth of Tools

To fight the Devil, the Programmer-God invents tools: blinkenlights and punched cards (and adhesive tape) are used so the Programmer-God may monitor as well as write the code. Thus programs can be double checked, fixed, retried, stored, despite any failures introduced by the Devil. Programs must be read as well as written, decoded as well as coded, and thus are born programming languages — starting with Assembly, to recompute label addresses and jump offsets when code is modified. Software development practices are developed, to be followed religiously.

From a better (or less bad) theory of programming, we thus get better (or less bad) tools — these tools improve programming by coping with its identified imperfections. This will continue to be true as we improve our logogonies.

Interestingly, anytime we find a new and hopefully better logogony, we will always be able consider a variant of it where some dark forces conspire to undo or corrupt what the creative forces strive to achieve. Thus, the idea of such opposing forces is a universal "mixin" for logogonies, the *devil mixin* — our first storytelling meta-tool.

2.4 Layered Creation

While naive software creationism can adequately explain small programs, the theory quickly reaches its limits: large working programs just do not spring fully armed from the head of Man. In a more refined logogony, Man still has an essentially perfect idea of the Program, but imperfections of the Machine require creation in multiple, neatly organized *layers*:

On the first day, Man separated requirements from bugs. On the second day, Man divided the program into routines... On the seventh day, Man rested as the demo ran flawlessly.

This logogony leads to new conceptual tools: top-down design, software architecture in nice layers, flow-charts, the waterfall process. With it come divide-and-conquer algorithms. In terms of languages, it layers a FORmula TRANslator on top of Assembly.

2.5 Iterated Creation

Now, as an anthropodicy, the *devil mixin* applies to this story of Layered Creation of Software: at each of the steps, the Devil may cause mistakes. This leads to a slightly modified approach to layereed creation: iterating the waterfall process, until the code is stable enough, then start a new layer of work.

At first, Man wrote a single-file prototype. Afterwards, Man worked on an alpha version, then a beta. Eventually, Man released v1, v2, v3... At last Man produced a stable version — although, He keeps issuing patch releases. (Be worried when He stops)

Each version is limited in resources and has to make tough choices and compromises that leave behind a layer of code *fossils* that you discover when you dig in the source code.

This suggests a new logogony, in which Man cannot create entire software systems in one act, if only because of the sheer amount of work required. Instead Man creates software in many steps, starting from foundations, building layers upon layers, bootstrapping complex structures from simpler ones, shaping tools and tool-making infrastructures, replacing parts with better ones as the need and opportunity arises, building scaffolding that is destroyed later possibly leaving fossils along the way — all according to a carefully designed master plan.

In other words, *Iterated Creation* is but another name for...

2.6 Intelligent Design

Intelligent Design is the most common logogony among software engineers, indeed implied by their very claimed title—because it flatters them: it recognizes enough of the difficulty of programming to set professionals above students and amateurs who can only write small programs; yet it affirms that professionals tame big complex software issues through the systematic endeavor of their Manly brainiac powers. They are intelligent and in control.

With this logogony are elaborated such tools and concepts as algebraic data structures and algorithms, operating systems, source code, compilers, compiler-compilers, build systems, modelling tools, hierarchically layered systems, the iterated waterfall process, release cycles, and all kinds of neat engineering practices.

Now of course, this logogony can be enhanced with the devil mixin, at which point new tools are engineered to counter the chaos introduced by the Devil: error messages, loggers, tracers and single-steppers to help locate bugs; line-editors to modify the Program; acceptance testing to validate It.

2.7 Polytheism

Another useful mixin for logogonies is *the polytheism mixin*. In this story modifier, there isn't one Man, with one Master Intent and consequent actions, but a lot of Men (including Women), each having Their part in creating the Software, each with Their own intent and actions.

In some variants, it may be that these many Men are but facets of a same unique Man, who takes multiple roles to address the multi-faceted endeavor of Software design; or it may be that Man's intent changes with time, or that Man is moody and has tantrums.

Man's ways are impenetrable to Software, but enhanced theories of what Man is lead to the introduction of new tools. To address multiple programming Men, files are invented; as Men and Their work get organized in hierarchies, so are files hierarchically organized in (sub)directories. Source code comments and formal documentation serve to convey intent and content between Men. Machines are time-shared, operating systems grow to manage multiple users, and eventually multiple users at the same time, each running multiple processes. Communication protocols are developed to exchange data between machines, between machines and Men, between Men.

The devil mixin can also be combined with the polytheism mixin. Each Man could have his Devil — failures in his personality traits. The Devil could be chaos in way that Men try to work with each other. The Devil may be a Man himself — a malicious programmer. Each of these explanations for errors in Man's Design leads to new techniques to address the identified sources of error. Better management techniques are developed; programmers review each other's code; user accounts are protected by passwords; resources have usage restrictions; files are backed up; redundancy checks are added to communication; errata complete documentation, and pages are intentionally left blank to prepare for them.

2.8 Limits to Intelligence

Intelligent Design was a much improved logogony compared to its predecessors; but sooner or later, it too reaches limits in its ability to describe reality accurately: the design of most software

is just really *bad*. Whether you consider the end result or the process to get there, you find that it shines neither by its efficiency nor by its elegance. Not only are most prototypes no good at all, most software projects are cancelled before they are shipped, or scratched shortly thereafter — for good reason. A lot of work is wasted without any positive result to show for it. It isn't rare that good ideas were discarded in favor of bad ideas. Even when things work, it is often for the wrong reasons; and pieces of code that survive are used in ways they were not intended.

The issue isn't that errors creep in that corrupt the implementation of a perfect idea; the issue is that the idea was far from perfect to begin with. As far as creator gods go, Programmers are only so bright. Men have "bounded rationality". In plain words, we are plain stupid. Thus, the next stepping stone on the way towards better logogonies is: Unintelligent design.

2.9 Unintelligent Design

The Programmer-God may have an intent, but He's a blind idiot who doesn't quite know what it is he wants or how to achieve it. He not only makes gross mistakes, he goes on wild goose chases that lead nowhere, and sets impossible goals while ignoring obvious truths.

Tools to help Him design programs will thus include helpful messages from his compilers for error diagnostic and recovery. Their role is not to tell an intelligent programmer "the devil crept in while you weren't looking, just have a look here, you can obviously see him and chase him", it is to tell the unintelligent programmer "what you did was stupid, here is the explanation why", for it would be hard for his limited intellect to figure it out all by himself. Syntax checking, type checking and various kinds of advanced semantic checking are invented to catch the more or less obvious errors and converge more quickly towards what the programmer would mean if only he were capable of forming coherent intent.

Interactive help, manuals and hints constantly remind Man of things that He should know better. Integrated development environments help Man play with the code and get faster answers as to whether or not his ideas make sense. Software interfaces are made idiot-proof by making languages more abstract and completing them with ample compile-time and run-time checking. Work is divided into "modules", so that what limited intelligence there is can be focused in module implementation, whereas using modules through their public interfaces requires much less intelligence; complexity is thus managed away from stupid users. Good design ensures all choices that Man makes can be taken based on a shallow limited view of the world, the only kind that fits the programmer Man's tiny brain. There is no shortage of imaginable tools and prosthetic devices to help Man cope with his mental disabilities; and these tools are themselves limited mainly by the inability of their own Manly programmers.

When a devil adds machine malfunction to operator dysfunction, testing becomes something to take seriously and systematically. When multiple gods are involved, the many resulting processes running at the same time must be protected from each other; the software is divided in many parts, that are tested separately; and contracts for what happens at their interface are attemptedly defined and enforced. Because the programmer gods cannot be trusted to remember all the issues with the software, some software must be used to systematically track those bugs and issues. When some of the programming Men are malicious, you're glad they are idiots, too, and you bury them under the weight and complexity of security features that will catch each of the more obvious malicious types of behavior.

2.10 Progress through Humility

One trend can already be observed in all these stories: they each chip away at the supposed greatest of Man, identify more of His imperfections; yet, acknowledging these imperfections is precisely what enables the invention of tools to cope with these imperfections, which themselves enable the creation of greater software. Greater humility is what makes progress possible.

Also note that the tools tell the story, even when we don't make the story explicit. Why do you use a typechecker? Because you make type errors. What if you don't use a typechecker. You still make type errors and pay the price. Whatever stories you tell others to impress them, or tell yourself to boost your own ego, the tools you choose to use (or fail to use) tell a more honest story about yourself.

But how far does Man have to humiliate Himself before a truthful story emerges about the origins of Software? If Software isn't the *Triumph of the Will* of Man, then what really is the driving Force behind Software? Before we can give answers to these question, we have to change our point of view...

2.11 Lamarckism

Whether software is designed by intelligent or stupid Men, or by something else altogether, we may importantly understand that software *changes* to adapt to new circumstances: new goals, new programmers, new customers, new insight, new technologies, etc. And so we come to focus on the nature of this change through time, rather than just on its product at a given moment. Such is *Software Lamarckism*.

Filesystems may remember many versions of the files they hold, each with a different version number. Software releases are numbered too. Because many Men may be working at a time, a piece of software may exist along many different versions; moreover, these versions are not in a strict linear order, but may come in branches that sometimes diverge from each other, and sometimes merge back together. "Ports" from one language to another, and "inspiration" from one project to the next, are other ways that information is copied from one Software project to another.

To understand the differences introduced, whether they are intelligent, stupid or malicious and what to do of them, new tools compute differences between files. To merge the intelligent changes and the fixes to the stupid and malicious ones along the many different branches, tools are created to apply computed differences to branched files. Revision control and change management is born, and continuous backup remembers all previous versions of tracked files.

Lamarckism is not a complete theory of why and how change happens, but it introduces a useful focus on change. It is a Great Mixin that can be applied to all the previous logogonies, a starting point for more elaborate theories that will explain the development of software in the terms of this incremental process of change.

3. Software Evolutionism

3.1 Supernatural Selection

Unintelligent Design, while acknowledging Man's stupidity at writing software in the small, still posits His grand design for building software in the large. How is this position defensible?

Lamarckism, by shifting the spotlight towards the change process, leads to asking why and how programmers lacking complete understanding choose to keep or change some or some other parts of the software. The immediate answer is that as Men write, they stumble upon good or bad features that they winnow by propagating the good and by eliminating the bad. The software writing process is thus some kind of artificial selection, under the careful, intelligent guidance of the Programmer-God. God impresses upon

the process a definite direction, Progress, and otherwise lets software evolve organically in this divine order. Judging software being much easier than writing software, it is defensible for Man to be good at the former even though he's bad at the latter. This logogony is *Supernatural Selection*.

With this logogony, new tools are selected into prominence. Prototyping tools help Man flesh out as many ideas as possible as quickly as possible, so he may select the correct ones. Formal specifications help define *what* software should be doing, without worry about *how* it will be doing it. Heuristic search algorithms use intelligently designed strategies to systematically explore spaces of potential solutions too large to be explored by the programmer themselves. The combination of these two approaches leads to *declarative programming*, whereby Man focuses on intent, and delegates implementation to the machine.

From one evaluation phase to the next, programs are transformed through systematic metaprograms. To prevent the devil from corrupting software, formal proofs are developed that perfectly exclude undesired behavior. To coordinate multiple Men, software modules separate interface from implementation, allowing for experimentation and adaptation separately in each part; rational developer communities are created, conferences are given, journals are published.

This whole approach has also been called the *First Wave of Cybernetics*. It combines an understanding of the natural dynamics of software with a faith in the ultimate power of an intelligent and purposeful Programmer-God, culminating with expert systems using explicit knowledge representation in an attempt to solve complex real-world problems.

3.2 Teleological Evolution

The logogony of Supernatural Selection obviously suffers from the same shortcoming as did the theory of Intelligent Design before it, in that it supposes that the Programmer-God (or at least some of them) are supremely intelligent as regards judging the quality of software change. This shortcoming is obvious once the logogonies are articulated as clear theories, rather than adopted without a thought by mimetism or what seems to work. An immediate improvement over that logogony is thus to stop believing in Men as supremely intelligent Programmer-Gods. Men may guide the evolution of software, but their contribution to the process is hardly an overall intelligent coherent purpose; rather it is through a number of interventions based on partial knowledge, intuition, randomness, towards a progress that can be felt but not defined. Such is the theory of *Teleological Evolution*.

With the transition from intelligent guidance to unintelligent guidance, we are led to the appearance of new tools, that roughly correspond to the *Second Wave of Cybernetics*. Genetic Algorithms, supervised learning through neural networks, probabilistically approximately correct learning methods allow to mine information from large databases without any explicitly designed representation of knowledge. Weakly structured computations enable data manipulation despite limited understanding. At a smaller scale, programmers are satisfied with randomized algorithms that have good enough performance in practice despite having dreadful worst case guarantees. To protect from the Devil, checksums and probabilistic proofs can be more useful than unattainable formal proofs. To synchronize multiple Programmer-Gods, user communities come to prominence: users, though less proficient than developers, are those who possess the most direct distributed knowledge of what makes the software useful or not.

The logogony of Teleological Evolution loosens the strictures of Design or of Supernatural Selection, and opens the space for practical software solutions to problems beyond the full grasp of programmers. While it reckons the importance of reasonable endeavor, this importance is also de-emphasized; indeed, even reason can be seen as but a fast-track internal process of random production and selection inside the programmer's mind, as guided by his godly intuition. In the end, Teleological Evolution embraces an unfathomable mystical intuition as the ultimate divine source of creation.

3.3 Natural Selection

As far as logogonies go, the notion of evolution under manly guidance was an improvement over that of direct design by purposeful Men, which was itself an improvement over the notion of direct creation. But in each case, this was only pushing back one level the assumption of a driving intent external to the world. Real evolutionary theory does away with this assumption. Survival of the fittest does not suppose an external criterion of fitness to which living creatures are submitted; rather, survival itself is the only criterion for survival, tautological and merciless. Survival is its own purpose: those programs that survive—survive; those that don't—don't. Changes that improve the odds that their host software should survive and propagate, thereby statistically tend to propagate themselves and colonize their respective niches. Changes that decrease the odds that their host software should survive and propagate, thereby statistically fail to propagate themselves and eventually disappear. Changes that best fit a niche and not others—survive in that niche and not others. How software changes help fit a niche decides whether the changes survive and spread, not whether Man explicitly and correctly anticipated and intended those changes to be successful in particular ways.

The cumulative result of this natural selection is an evolutionary process that favors bundles of traits that tend towards their own reproduction. This freewheeling evolution necessitates no godly intervention, neither by an intelligent conscience, nor by madmen. More remarkably, programmers are no gods above it, and their actions are no such interventions. Programmers are but machines like others, bundles of self-reproducing traits competing to exploit the resources of the universe. As compared to other machines in this programming universe, certainly programmers are unique and different — everyone's all unique and different; that doesn't exempt them from the laws of natural selection. Programmers are machines among others, trying to survive in a wild machine-eats-machine world; their actions are their attempts to survive and reproduce by gaining an edge in the race for ever more efficient acquisition and use of reproductive programming resources. Their failure means their code stops being used and is forgotten. If God exists, then he is not Man; and ever since Man created Software, God has just been relaxing, sitting back and enjoying the show. Software Evolution is not directly controlled by Man and not actively guided by God, it is God's Spectator Sport, and Man is a competitor among others. Such is the logogony of *Natural Selection*.

3.4 Software Darwinism

Natural Selection, unlike Supernatural Selection or Teleological Evolution, is what (software) darwinists mean when they speak of (software) "evolution".

With this perspective on software development, we gain new mental models for development processes. We think of software in terms of self-sustaining systems, that evolve and compete based on their ability to survive and spread. We understand that the hosts and actors of this memetic competition are men as well as machines, or even more so. We may then notice that systems are never born big, and that the only big systems that work are those that were born small and evolved and grew in a way that they were kept working at every step [2]. We relate the spread of ideas to the demographics generations of humans and machines passing on their forking and mingling traditions—relating memetics to genetics. We understand

that pieces of hardware, software and wetware survive as part of ecosystems, with cycles of development and use by various humans, where economic and legal aspects have their importance as well as technical and managerial aspects. We realize that these systems compete on a market ultimately driven by economic costs, of which technical aspects are but one part, often not the most decisive one, though they are what the technicians obsess about.

Models such as above mostly serve to filter out doomed business models and self-defeating attitudes when the model can explain how they go against reality. But they also lead to a few positive tools that actually help. A *Third Wave of Cybernetics* attempts to re-create artificial life and life-like phenomena through the emergence of behavior from many software agents. Unsupervised learning and tournament competitions yield results unreachable by supervised learning and explicit fitness functions. Understanding that the forces opposing creation act not through supernatural means but through the action of malicious or misguided humans, we achieve security through a mix of computer cryptography, growing networks of human trust, retaliating against bad behavior, and educating new people.

Software Darwinism provides a big picture that puts haughty programmers down from their godly pedestal and back into the muddy real world. It doesn't offer direct solutions to design problems so much as it dispels our illusions about fake solutions and unearned authorities. No one is a god, above the others, to predict what will work and dictate what to do; our experts' dreams are often but vain obsessions, whereas some rare amateurs' successful experiment may start a revolution. Life is the ultimate judge—accept no substitute, and respect its sanction.

3.5 Evolution is an Inside Job

Natural Selection may appear to look down on the world as a soulless marketplace. But it will only appear soulless if you imagine yourself in the seat of that laissez-faire God above the world. Face it: you're no god, you're not outside the world and above it. There *might* be a God (or Gods), who might or might not be intervening in this world—but you have to come to the realization that *you* are definitely neither Him nor any of Them. You're one of us earthworms, trying to make the best out of what you have (or not trying, and thus probably failing and promptly disappearing into irrelevance). Evolution is not something for you to enjoy watching from above, it is something you are part of, willy nilly. You can't just let nature decide, you're part of the nature that will decide. Whichever genes and memes you carry may or may not survive—-it is largely up to your actions whether they will succeed or fail. You're either in the experimental set of changes that may or may not work out well, or you're in the control set of the obsolete that will surely be replaced. Such is the view from *Inside Evolution*.

The tools that matter are those that are available to you. Your resources are limited, and you should invest them wisely. Which tools will make you most productive personally? Opportunities are there to be seized; if not by you now, by someone else later. On the other hand, it may be too soon to invest in some ideas, and too late to invest in others; timing is key. Specialization will help, and can be a long-term investment that provides compound interests. As for cooperation with other non-gods, you can only go so far with your own efforts, and success lies in being able to leverage the efforts of other people. Which tools allow you to reuse as much as possible of these people's efforts? Tools can be technical, or can be social. Not just software libraries, but software communities, software market niches, software business contracts. Of course, you always need some kind of exclusive resource to ensure a revenue stream; your combined proficiency, trustworthiness and time are ultimately the only such resource you have, and ample enough to live well if you can market it, though it will probably not make you super rich. On the devil side, intellectual frauds will try to have you adopt their bad ideas, and other scammers will try to divert your resources in their favor; you must learn to avoid them.

Evolution as an Inside Job restores the soul in the marketplace for software: yours. You're the entrepreneur of your own life.

3.6 Social Evolution

As you fully grasp the fact that all actors are individuals, not just yourself, you start taking into account incentive structures. Incentive structures will put you and your associates in a position to productively cooperate at your full potential, or to work at only a fraction of it; so carefully watch both your legal and business arrangements.

With a systematic view of incentives, you stress the importance of contracts and accountability as a way to structure human interaction, re-uniting liberty of means and responsibility for results in complex software arrangements. For instance service-level agreements will allow to robustly build larger, more complex structures than direct command chains. You may recognize the value of free markets as a way to organize people and to evaluate ideas, rewarding those able to invest their resources in the good ones rather than the bad ones. You may celebrate startup companies as light innovation structures with highly motivated personnel.

You may also consider long-term effects of licensing issues on development ecosystems. For instance, proprietary software has a definite short-term advantage over free software in capitalization, focus, coherence — and in the ability to use the latter when the latter can't use the former. But in the long run, proprietary software destroys incentive from anyone who doesn't fully trust the software owner; and that trust can last but until the eventual catastrophe inevitable in any centralized management. All proprietary software has a suspended death sentence. Only free software can be immortal and has a chance at maintaining long-term ecosystems that keep evolving long after any particular strain of momentarily superior proprietary software has come and gone.

Thinking in terms of social evolution, of arms races from positive feedback, equilibria from negative feedback, invariants from limited resources, variants from energy sources and entropy sinks, can make you see what is invisible to those ignorant of the perspective.

4. Stories Programmers Tell

4.1 Stories Evolve!

From naive Instant Creationism to the darwinist view from Inside Evolution, we can observe a *story arc* in these logogonies themselves: Man is taken down from his pedestal as an Über-God above the machine, until he becomes an underdog competing as part of processes that vastly surpass him, at the same level as machines. Yet, each time Man's image is humbled, Man himself is elevated, as tools are then invented to address his identified weaknesses — until he is one with the machines that augment him ever closer to godhood.

The *evolution* of these stories can indeed be seen as an elaboration, whereby each step replaces an overly simplistic story that *assumes* power with a more accurate one that *creates* power. This *teleological evolution* towards more darwinistic stories being more accurate and more empowering could be a divine truth about our universe; or it could be merely a claim by the author of this essay, the validity of which could be less than universal; yet, even without being universal, its relevance to your own life could make it adaptive for you (for "us"?) to take it seriously.

In any case, we just stepped back from what the stories say to talking about the stories themselves. We *went meta* on the stories. Whereupon the topic of our story is storytelling.

4.2 Stories Matter

Now, the tools previously described already exist; they have been created, engineered or selected, or they have emerged, without any of these logogonies being explicitly stated, much less used as conscious guides. Do these stories have any relevance, or are they but nice-sounding post-hoc rationalizations? Even if relevant, don't they come too late to help invent those tools?

Daniel Dennett wrote: "There is no such thing as philosophy-free science; there is only science whose philosophical baggage is taken on board without examination." In software programming as in any other human endeavor, not stating your assumptions won't save you from the consequences of following them when they are erroneous—not anymore than putting your head in the sand will save you from predators you can't see. The overarching stories you follow—or paradigms, as they are commonly called— embody the assumptions you implicitly make, often without a conscious decision; they determine the horizon of phenomena you can comprehend. What exists beyond these stories, you cannot see. And each step in the evolution of these stories adds relevant phenomena to which you were previously blind, of which you were a helpless victim, that you can now see and master.

Diagnosing how a "series of unfortunate events" is no accident but the necessary consequence of some previously invisible phenomenon is a necessary first step to properly addressing an issue. "Failing to plan is planning to fail." If you assume an inadequately simplistic paradigm, you will keep attacking your problem with inadequate tools because you cannot even see how better tools apply to your case. You will waste significant human resources at unrewarding repetitive tasks at which humans are unreliable compared to the cheaper machines that you could have used; despite these sacrifices, or rather because of them, you will keep failing, until competition people using better paradigms drives you out.

4.3 Stories as Tools

The stories we live by are seldom told in so many words—precisely because uttering them would make the "plot holes" in the stories uncomfortably obvious: in addition to the pain of living a bad story then comes the shame of being the story's sucker. So these stories remain implicit, official lies that go unsaid but are well internalized as the logical justification for the processes followed and the tools used. They are *Games People Play* [1].

However, once you realize the stories are themselves subject to change, then making them explicit, uncovering their plot holes and feeling their discomfort can become the means, opportunity and motivation for quitting (or wholly avoiding) a bad game and finding a better one. *Going meta* about stories is then a tool for the group therapy of dysfunctional software development teams, of dysfunctional software ecosystems—as well as for other dysfunctional human behaviors. Leaving stories implicit imprisons us; making stories explicit liberates us.

4.4 The Proper Role of Stories

A story may good by itself—if it brings original insight; it may be bad by itself—if it is logically inconsistent; but most stories are only good or bad in context—as useful or misleading ways to describe a situation.

If your problem is so simple that you grok it all and can write down a software solution in one breath, by all means, do it! Don't follow a 12-step plan to software development to be rinsed and lathered along 30 iterations. If on the other hand you can't fully understand the problem you're tackling, if designing a solution is too hard for structured methods, then keep escalating the methods you use until you hopefully solve your problem. Generating random programs until a solution is found should be a last resort. Yet given proper biases in generation, cleverness in detection, and raw power, this resort might be made affordable, and find solutions where other methods fail.

Thus, when more elaborate stories are invented, older, simpler stories don't die out: they each find their niche of validity, where the cost of improving on them is higher that the return on the improvement; meanwhile, new stories cover new ground more so than claim known territory. Interestingly, as stories inspire tools that increase man's reach, the domain of validity of older stories is expanded rather than narrowed. Modern languages and IDEs indeed make it possible to instantly create or systematically engineer programs that would previously have been monumental endeavors—if they could have been imagined at all.

The question therefore isn't to find a one-size-fits-all story, but to identify the story that best fits the situation at hand, which will make you most effective if you play the games it suggests.

4.5 Leveraging Stories

Programmers could automate away a lot of the issues they face today if only they graduated from the paradigm of (Un)Intelligent Design to that of Supernatural Selection or better: Just giving up on having humans write software directly, and instead adopting more declarative and generative programming approaches could deal with better results and fewer efforts with (de)serialization, persistence, schemas and schema upgrades, replication, performance autotuning, representation selection, or maintenance coherence between multiple data representations, code layers, specifications, documentation and test files, etc. What more if they adopted the Inside View of Evolution, they would have a healthier approach to negotiating what the software should or shouldn't be doing in the first place.

As the profession matures, software libraries and programming practices will spread that can be anticipated in terms of examining which logogony underlies some unsatisfactory practice, and imagining how a more elaborate logogony could enhance it.

For instance, unintelligent design brings us manual tests; lamarckism makes testing incremental; supernatural selection inspires generator-driven property checks; teleological evolution suggests whitebox fuzz testing; natural selection suggests bug bounties; and the inside view suggests paying attention to how management creates incentives towards better or worse code.

Intelligent design suggests type declarations; unintelligent design, type-checkers; lamarkism, type-based refactoring and schema versioning; supernatural selection, generic types and type inference; teleological evolution, automated learning of probabilistic type schemas for data extraction or activity monitoring; natural selection, learning with competing models and automated translation between type hierarchies; inside evolution, taking human factors, responsibilities and incentives into account in the modular design of class hierarchies and type systems.

The explicit application of logogonies to any particular topic immediately suggests a fertile research program to analyze existing practices and create new ones based on more elaborate logogonies.

4.6 Beyond These Logogonies

Your software activities are unlikely to follow the "perfect" logogony; but that doesn't mean that the story bringing the most tragic dysfunction in your life is your logogony. There are many stories about Software and about Man, that define your current behavior, the roles you play. If you work in a team (and you do), stories also define the way this team is organized, sets its goals and its means, the way each of its members sets their goals and means, the way people interact with each others inside and outside the team, etc. And the one story that you get most wrong, that is causing you the greatest pain or waste in your life, that you could benefit most from fixing... is for you to discover, examine, and replace. Logogo-

nies were just the most spectacular way of illustrating the notion of stories and games we play; they are by no means the only interesting stories, or the stories most relevant to whichever important issue you're dealing with at the moment. So, make the stories of your life explicit, identify those you're living, and rewrite them, better!

One the other hand, if progress in how software is developped can be related to stories about software development, a question naturally arises: what are the next logogonies? Is the above "view from Inside Evolution" the be-all, end-all of programming paradigms? Is there nothing left but incremental refinement of existing concepts and tools? Or are there paradigms as radically advanced compared to software darwinism as software darwinism is compared its predecessors? Can one identify and adopt this paradigm early on, and thus get an edge over competition?

5. Stories Future

5.1 Present Optimism

The simplest view about logogonies is that there will be no new ones, at least none that works. Our understanding of software development is mature; though there may be myriads of minor details to get right, the big picture is complete. This is *Present Optimism*: the "end of history" was reached, from there it's smooth sailing and/or decline.

Of course, assuming the big picture about software will ever contain but finite understandable information, there *will* be a point when there is no room left for further paradigm improvements. Therefore Present Optimism will *some day* be true, about logogonies as about many things.

On the other hand, considering how new the field of software development is and how fast it has changed in just the last few years, it seems premature to declare that we fully understand how software is developed and will never find new deep insights. If our understanding of software development were to remain stagnant for, say, five to ten years, and all developers were to settle towards a finite set of well understood unchanging methods, then we could assert with much more confidence that indeed we have reached the acme of software development. But this hasn't nearly happened yet, and the case for Present Optimism is rather slim.

5.2 The Singularity

A different kind of optimism, and a common idea regarding future logogonies has always been that computers will somehow surpass men in intelligence and take over the menial task of programming: they will be genies, who will grant your every software wishes, the details of which they can anticipate better than you can specify. This is Extreme Future Optimism, or *Singulatarianism*: the theory that soon(er or later), we'll reach a Technological Millenium, or *Singularity*, when all our worries are taken away.

However, this Millenarianism is based on a misunderstanding, that is best dispelled by contrasting its equal and opposite misunderstanding: Millenarian Luddism, the claim that technology unless stopped will bring a bleak future where humans are reduced to misery as machines take all their jobs away. Hopefully the errors will cancel each other in a collision from which light will emerge.

5.3 No Escape from Evolution

Computers have replaced humans in many ways, and will keep replacing them in more ways. Computerized tools that replace humans while programming are the heart of our story so far. But automation does not destroy human jobs, it only displaces jobs towards new areas not covered by tools. Successful tools provide more satisfactions than before while reducing efforts. More for less—progress . The human resources previously expended toward those satisfactions are not destroyed but liberated; they are made available, to be redirected to new useful endeavours that couldn't previously be afforded, together with the increased riches with which to pay them. Furthermore, the *law of comparative advantage* ensures that *even at absolute disadvantage*, the tasks relatively better done by meatware than by software will remain a domain of human activity.

Of course, nobody prevents from using or sponsoring a human-intensive way of programming that declares some forms of machine assitance taboo. There will always be a cottage industry of "brain made" software just like there now is a cottage industry of "hand made" pottery (still using tools, just neolithic ones). But just like automation in other industries increased productivity and made mankind vastly wealthier, so will automation in programming increase productivity and serve mankind—it already does through all the software development tools previously mentioned. And with further progress, programming in today's hip tools will become as obsolete as programming in COBOL or Assembly has become: a waste effort, and guaranteed ultimate failure, for the Luddites who refuse automation.

The laws of Economics still apply. Which includes the laws of Evolution. Indeed, the ideas behind Evolution were discovered by historians as applied to economics, long before they were ever applied to biology—or software.

5.4 Sentient Agency

Machines can turn feats into chores, chores into menial tasks, menial tasks into assumable commodities. But they can neither create nor destroy the desire for ever more, ever greater satisfactions, the ability to adapt and work towards these satisfaction, and the individual accountability for the results: in other words, human drive, spirit and agency. As our past worries are cared for, we focus on loftier worries. Ultimately, only human parents create human jobs, and only illness and death destroy them; the rest is a matter of organizing existing human resources. The fear of Artificial Intelligence and claim of Human Supremacy is a lifeformist stance wrapped in the usual protectionist fallacies, and its narrowmindedness should inspire the same spite as racist or nationalist arguments before it.

Conversely, blind faith in Artificial Intelligence is but another millenarian superstition—people dreaming of being saved from having to live their own lives. This blind faith is a cop out, in that it wishes away the very nature of life and its intrinsic challenges. Even if "intelligent" machines were to replace humans in the activity of programming, said machines won't be able cop out of having a logogony: software issues will have to be addressed, the buck will have to stop at *someone*. And that someone is necessarily a sentient agents, whether electronical or biological, equally constrained by the laws of "Human" Action, i.e. purposeful action: the competition for scarcity of resources, the power of incentives, the benefits of cooperation, the law of supply and demand, the importance of property rights, etc., and ultimately, *evolution*, will apply to them as to they apply to us.

However brighter or gloomier than today a future with artificial intelligences will be, bridging the gap between today and that future, if possible, won't be achieved by hand-waving. It will require a paradigm shift that the cop out precisely aims at blanking out.

The legitimate cop out is not to assume knowledge but to admit ignorance: "my previous investigations didn't lead to any firm conclusion to this question, and I don't have enough combined care for the matter and trust in the remaining venues available for investigation to afford further investigation."

6. Open Conclusion

6.1 Educated Guesses

The future promises many revolutions in how software will be written: between "artificial intelligence" and cyber-security arms race, mind-defying heuristics and automated formal methods, tomorrow's technology is likely to obsolete today's programming practices as hopelessly primitive and insecure. Can we make more precise predictions? *It is difficult to make predictions, especially about the future* (Karl Kristian Steincke). If you could provide an accurate functional description of what the future would bring, then this future would already be there, by the simple execution of this description as a plan. But we can make educated guesses. Here are mine...

6.2 Bootstrapping Intelligence

Activities involving human intelligence have already started being replaced by computer automation. All of the programming tools we mentionned above are indeed forms of this replacement. The purpose of computers is to *automate that which can be automated*. This applies to all human activities, including "intellectual" activities. In particular this applies to software development itself.

One may hope that applying intelligence automation techniques at improving themselves may be self-catalytic: cumulative progress in machine intelligence leads to ever more progress in machine intelligence; bias towards systems that mutate better and faster leads to systems with even stronger bias toward mutating better and faster; pressure for higher value and lower costs leads to more valuable, cheaper automation that itself increases this pressure. Intelligence at writing machine learning software may be the most general kind of intelligence computers may have; and series of meta-level mutations can quickly bootstrap such intelligence in a positive feedback loop, until a phase transition is reached.

It is not immediately relevant whether a "Singularity" is ever reached whereby mankind is transcended, or whether mankind is as elaborate as intelligence gets, whether artificially intelligent autonomous agents ever emerge, or the only agents ever are humans and machines remain their props. That is not for any of us to decide, only to observe; maybe partake. Now, any progress we make is quantum, made of irreducible bits of information that take us closer to this goal, whether by big strides or tiny steps; and the amount of progress to reach whatever "intelligence" machines can embody is finite; therefore, if humans keep at it, they'll reach their destination eventually, whether in decades, centuries, millenia or stranger aeons. What matters is that some road will be taken, and that those who stay behind will become irrelevant; what matters is that inasmuch as there is an inflexion point where we'll reach quickly diminishing returns on investment in machine intelligence, this point seems to be well ahead of us; what matters is that we are within the part of the curve that accelerates.

Now is a time that you, personally, can have an impact. To maximize it, you may ask questions such as: What tools can you develop today that will best increase automation intelligence in the long run? Which generally applicable software methods are not currently applied to improving software development itself? What kind of architecture makes it easier to combine such methods and apply them to the improvement of software development itself? What essential aspects of more intelligent software are currently left unresearched?

6.3 As we grope for the future...

All of us each have to make choices. I can see many opportunities: in architecting software that combines induction (machine learning) and deduction (algorithms); in recognizing that interaction with computers is *dialogue not command*, between unequals that have specific comparative advantages; in understanding of how the social organization of programmers and the architectural organization of programs are related via the feedback and incentive structures they induce.

But in the end, with this and other articles, my choice will have been to try to reach out a happy few programmers to open their minds to improving how they may think about programming, and becoming assumed entrepreneurs of their own software life; that they may control the internal evolution of their identity to adapt to the world, rather than be the unconscious victims of an external evolution they can't fathom — and creating software ecosystems that *reflect* this ability to improve oneself from within.

Bibliography

[1] Eric Berne. Games People Play: The Psychology of Human Relationships. 1964.

[2] John Gall. Systemantics: How Systems Work and Especially How They Fail. 1975.