CANNIBAL CODE

If technology advances in cycles, you might assume the best legacy modernization strategy is to wait a decade or two for paradigms to shift back and leapfrog over. If only! For all that mainframes and clouds might have in common in general, they have a number of significant differences in the implementation that block easy transitions. While the architectural philosophy of time-sharing has come back in vogue, other components of technology have been advancing at a different pace. You can divide any single product into an infinite number of elements: hardware, software, interfaces, protocols, and so on. Then you can add specific techniques within those categories. Not all cycles are in sync. The odds of a modern piece of technology perfectly reflecting an older piece of technology are as likely as finding two days where every star in the sky had the exact same position.

So, the takeaway from understanding that technology advances in cycles isn't that upgrades are easier the longer you wait, it's that you should avoid upgrading to new technology simply because it's new.

Alignable Differences and User Interfaces

Without alignable differences, consumers can't determine the value of the technology in which they are being asked to invest. Completely innovative technology is not a viable solution, because it has no reference point to help it find its market. We often think of technology as being streamlined and efficient with no unnecessary bits without a clear purpose, but in fact, many forms of technology you depend on have vestigial features either inherited from other older forms of technology or imported later to create the illusion of feature parity.

For example, most software engineering teams maintain 80-column widths for lines of code. It is easier to read short lines of code than long lines of code; that much is true. But why specifically 80 columns? Why not 100 columns?

Amazingly, an 80-column width is the size of the old mainframe punch cards that were used to input both data and programs into the room-sized computers built during the 1950s and 1960s. So right now, solidly in the 21st century, programmers are enforcing a standard developed for machines most of them have never even seen, let alone programmed.

But, why are mainframe punch cards 80 columns wide? Punch cards used by the forebears of the earliest computer companies—back when they were mechanical "tabulating machines" used primarily for things like the census—were ad hoc and incredibly inefficient. They were designed to tally, not calculate, so they were modeled after what a railroad conductor might use for tickets, rather than for storing data.[1] The cards needed to be fed into machines in batches and then sorted and stored. To avoid having to re-invent everything, the cards themselves were designed to be approximately the same size as the paper currency of the United States at the time: 3¼ by 7⅜ inches. This meant companies could repurpose existing drawers, bins, and boxes to acquire necessary accessories.

---

1. Geoffrey D. Austrian, *Herman Hollerith: Forgotten Giant of Information Processing* (New York: Columbia University Press, 1982), 124.

By the 1920s, customers were leaning on IBM to get more data storage out of a single card. IBM's innovation was to change the shape of the holes themselves, making them more rectangular so that they could be placed closer together on the card.[2] That meant 80 columns of possible holes.

Now, let's go even deeper. What about the punch card itself? Why were the first computers designed to take input from stiff cards with holes punched into them? Keyboards have existed as long as typewriters, and the first modern typewriter was patented by Christopher Latham Sholes, Carlos Glidden, and Samuel W. Soule in 1868, nearly a century before some of these mainframes were developed. Telegraphs were experimenting with different types of keyboards even earlier than that. Why would people prefer to punch holes in a thick piece of stock paper when they could just type their information on a keyboard?

The problem with keyboards, or similar input devices, is that it's easy for human operators to mistype things, especially if those human operators get no visual confirmation that what they think they typed is actually what the machine received. Think about typing a password into a field on a website that hides what you type. One disadvantage to such password-masking fields is if you hit the wrong key, you might not notice until the system rejects your input. How many times have you mistyped a password like this? Now imagine inputting an entire message without being able to see what you typed. Operator error was a big concern for telegraphs, especially when they started to play a larger role in ferrying critical messages around the globe.

The solution was to have a keyboard, but instead of interfacing directly with the telegraph, the keyboard would produce a record that could be checked for errors before the machine tried to send the message. Many different variations on this concept were developed, and the one that eventually stuck was punching holes in paper tape.

---

2. US Patent 1,772,492, Record Sheet for Tabulating Machines, C. D. Lake, filed June 20, 1928, *http://ibm-1401.info/Patent1772492.pdf*.

What's curious about the era of tabulating machines in the late 19th century and the era of early computers in the 20th is that they arrived at the same solution in different ways. The punch cards of tabulating machines were developed from railroad tickets, but the punch cards of telegraphs were developed from the textile industry.

More than a century earlier, French weavers had been automating the pattern designs of elaborate rugs by printing out a design in the form of a series of punched holes on cards and feeding those cards into their looms. This allowed weavers to produce high-quality products much faster, with more artistry and greater accuracy.

The telegraph further refined the system by introducing the concept of encoding. When the goal is to manipulate the threads in a giant loom to create a complex pattern row by row, there's no point in overcomplicating things. One hole per raised thread is perfectly effective.

However, when the goal is to send messages long distances, that kind of literalism is inefficient. Telegraph operators were already accustomed to using code to represent different letters, but those codes were optimized to reduce operator error. In Morse code, for example, the most common letters have shorter codes. This keeps transmission fast and minimizes the strain on the operator. Once telegraphs started producing a physical record that the operator could double- or triple-check before sending the message, the most significant gains in performance were to be had by optimizing the encoding for the machines themselves. Letters that were expressed in code length anywhere between one to five units were not easy for machines to deal with. Machines do far better when every letter is equal in length. The best codes now were ones that were a bit more complex, had a fixed length, and ultimately stored more data.

A few different systems were developed. The first one to stick was developed by Emile Baudot in 1870. The so-called Baudot code, aka International Telegraph Alphabet No. 1, was a 5-bit binary system.

Fast-forward to the early computer age when people were developing massive room-sized machines that also were using binary systems.

They needed a way to input data and instructions, but they had no visual interface. Computers wouldn't be developed to work with monitors until 1964 when Bell Labs incorporated the first primitive visual interface into the Multics time-sharing system. We had no way of seeing the input the computer was receiving, so we borrowed an interface from the telegraph, which, in turn, was borrowing one from 18th-century French weavers.

Technology is like that. It progresses in cycles, but those cycles occasionally collide, intersect, or conflate. We are constantly borrowing ideas we've seen elsewhere either to improve our systems or to give our users a reference point that will make adopting the new technology quicker and easier for them. Truly new systems often cannibalize the interfaces of older systems to create alignable differences.

This is why maintaining technology long term is so difficult. Although blindly jumping onto new things for the sake of their newness is dangerous, not keeping up to date is also dangerous. As technology advances, it collects more and more interfaces and patterns. It absorbs them from other fields, and it holds on to historic elements that no longer make sense. It builds assumptions around the most deeply buried characteristics. Keep your systems the way they are for too long, and you get caught trying to migrate decades of assumptions.

Unix Eats the World

A common piece of advice for building successful software is to keep what you are trying to do simple. But what exactly makes one design feel simple and another design feel complicated? Why is a line of code 80 characters long simpler and easier to read? It is short, but what if I told you that user experience research actually puts the ideal number at 50 to 60 characters wide? This means 80 characters is a good 50 percent longer than what we know works best from actual testing.

The human machine is strongly biased toward the familiar. We perceive concepts and constructs we know as simpler, easier, and more

efficient just because they are known and comfortable to us. We don't need to be experts in a construct or even necessarily like it in order for familiarity to change our perception of it. In the 1960s, psychologist Robert Zajonc conducted a series of experiments documenting how even a single exposure to something increased positive feelings about it in later encounters. He found this effect with languages, individual words, and images. Later researchers have observed similar preferences in how financial professionals invest,[3] how academic researchers evaluate journals,[4] and what flavors we enjoy when we eat.[5] In psychology, the term for this is the *mere-exposure effect*. Simply being exposed to a concept makes it easier for the brain to process that concept and, therefore, feels easier to understand for the user.

Developing new technology or revitalizing an old system is, therefore, most likely to be effective when building on familiar concepts. Reference points create alignable differences that help us assess the value of something new, but those same reference points make the new technology feel simple and easy, lowering the barrier to entry and increasing the odds it will be adopted as well as the speed of adoption.

Consider the Linux operating system. It's easily one of the most popular operating systems for web servers if not computers in general. Hundreds of variants currently exist that are available to install freely, and there are any number of professional versions. Linux was the uncontested victor to emerge from a mad race to develop an operating system that was both portable to many different types of computers and free of restrictive licenses.

---

3. Gur Huberman, "Familiarity Breeds Investment," *The Review of Financial Studies* 14, no. 3 (June 2001): 659–680, *https://doi.org/10.1093/rfs/14.3.659*.

4. A. Serenko and N. Bontis, "What's Familiar Is Excellent: The Impact of Exposure Effect on Perceived Journal Quality," *J. Informetrics* 5, no. 1 (January 2011): 219–223.

5. Patricia Pliner, "The Effects of Mere Exposure on Liking for Edible Substances," *Appetite* 3, no. 3 (September 1982): 283–290.

Linux is often described as the most popular version of the Unix operating system, except the two OSes share very little when it comes to implementation.

The story of Linux kicks off with the breakup of Bell Systems in 1982, nearly a decade before its creation. A 1956 consent decree against AT&T had forbidden the telecom giant from "any business other than the furnishing of common carrier communications services." This meant that when Bell Labs computer scientists Dennis Ritchie, Ken Thompson, and Rudd Canaday began developing Unix in the 1970s, no one was sure whether AT&T was allowed to sell it. The lawyers at AT&T decided to play it safe and allow it to be sold to academic and research institutions with a copy of its source code along with the software.[6]

Having the source code made it easy to port Unix to different machines as well as modify and debug it. People printed it out and annotated it with their own commentary. Unix became an easy option for teaching students how operating systems worked. It spread like wildfire across a wide variety of different institutions, including universities, museums, governmental organizations, and at least one all-girls private school in the early days.

Users began putting their modified versions of Unix on magnetic tape and making copies to distribute among each other. These essentially were forks and pull requests long before the infrastructure for such things existed. The principal motivation for sharing was to distribute bug fixes and patches.

Meanwhile, AT&T's lawyers were trying to figure out what to do with Unix, and they were waffling between their original determination and a more traditional restrictive approach to intellectual property. Unix historian Peter Salus tells the story of how AT&T's developers actively participated in the piracy of their own intellectual property:

---

6. Peter H. Salus, *The Daemon, the Gnu, and the Penguin*, Reed Media Services, September 2008.

> *A large number of bug fixes was collected, and rather than issue them one at a time, a collection tape was put together by Ken [Thompson]. Some of the fixes were quite important. . . . I suspect that a significant number of the fixes w ere actually done by non-Bell people. Ken tried to send it out, but the lawyers kept stalling and stalling and stalling.*
>
> *Finally, in complete disgust, someone "found" a tape on Mountain Avenue [the address of Bell Laboratories was 600 Mountain Avenue, Murray Hill, NJ] which had the fixes.*
>
> *When the lawyers found out about it, they called every licensee and threatened them with dire consequences if they didn't destroy the tape… after trying to find out how they got the tape. I would guess that no one would actually tell them how they came by the tape (I didn't). It was the first of many attempts by the AT&T lawyers to justify their existence and to kill UNIX.*[7]

When the university students who studied Unix as part of their computer science degrees graduated and got jobs, they brought Unix with them. AT&T's licensing became more restrictive with every new version, as the company tried to figure out what it legally could do to leverage this thriving community it had accidentally created.

Then in 1982, the US Department of Justice settled its second antitrust case against the telecom and broke up "Ma Bell." AT&T was suddenly free from the consent decree that kept it from treating Unix fully as a product, and it wasted no time in cracking down hard on the community that had grown over the course of a decade.

If you lived through similar attempts to stop sharing other forms of intellectual property, like music and movies, you can understand how once people became accustomed to having Unix as a free and modifiable operating system, they didn't want to give it up and go back to the way things were before. Taking away access to Unix's source code sent the

7. Ibid.

community on the hunt for a replacement that was open sourced and ideally free.

An early contender was a variant of Unix developed at Berkeley called Berkeley Software Distribution (BSD). BSD had a growing community, but it had used part of Unix's source code as its base, so it was quickly bogged down in litigation. The heir to Unix needed to present itself as Unix-like while not including any intellectual property from AT&T.

Enter Linux, which was developed as a pet project by computer science student Linus Torvalds. There was never any intention to create a full operating system from Linux; it was intended to be only a kernel for the specific chip architecture to which the creator happened to have access. The Linux operating system, therefore, was pieced together from a variety of software from other groups. Most of its Unix-like interfaces came from Richard Stallman's GNU project, and GNU itself contained no Unix code by design.

So in a way, Linux is a descendant of Unix that involves no code directly from Unix. But, why hold on to the Unix look and feel at all? Once the decision to start writing something completely new was made, what was the value of wrapping things up to look like Unix? For Stallman, the situation was clear: free software was a moral mission. The goal was not to build a free alternative to Unix, but to build a free *replacement* for Unix that would completely overtake and drive Unix out of business. He did not hesitate to describe the strategy of the GNU project in extremes:

> *As the GNU Project's reputation grew, people began offering to donate machines running Unix to the project. These were very useful, because the easiest way to develop components of GNU was to do it on a Unix system, and replace the components of that system one by one. But they raised an ethical issue: whether it was right for us to have a copy of Unix at all.*
>
> *Unix was (and is) proprietary software, and the GNU Project's philosophy said that we should not use proprietary software. But, applying the same reasoning that leads to the conclusion that violence in self*

> *defense is justified, I concluded that it was legitimate to use a proprietary package when that was crucial for developing a free replacement that would help others stop using the proprietary package.*
>
> *But, even if this was a justifiable evil, it was still an evil. Today we no longer have any copies of Unix, because we have replaced them with free operating systems. If we could not replace a machine's operating system with a free one, we replaced the machine instead.*[8]

Stallman used Unix's interfaces because he understood that if GNU's interfaces matched those of established pieces of software, the users of the proprietary pieces of software would have a bigger incentive to switch.[9]

Let's go down one more level: why did Unix have the interface it had in the first place? Most Unix commands are two-letter abbreviations for words that don't seem to need abbreviating. The authors of *The UNIX-HATERS Handbook* attribute this interface to the hardware available to Unix's creators:

> *The novice Unix user is always surprised by Unix's choice of command names. No amount of training on DOS or the Mac prepares one for the majestic beauty of cryptic two-letter command names such as cp, rm, and ls.*
>
> *Those of us who used early 70s I/O devices suspect the degeneracy stems from the speed, reliability, and, most importantly, the keyboard of the ASR-33 Teletype, the common input/output device in those days. Unlike today's keyboards, where the distance keys travel is based on feedback principles, and the only force necessary is that needed to close a microswitch, keys on the Teletype (at least in memory) needed to travel over*

8. Chris DiBona, Sam Ockman, and Mark Stone, ed., *Open Sources: Voices from the Open Source Revolution* (Champaign: O'Reilly Media, 1999).

9. Ibid.

half an inch, and take the force necessary to run a small electric generator such as those found on bicycles. You could break your knuckles touch typing on those beasts.

If Dennis and Ken had a Selectric instead of a Teletype, we'd probably be typing "copy" and "remove" instead of "cp" and "rm." Proof again that technology limits our choices as often as it expands them.

After more than two decades, what is the excuse for continuing this tradition? The implacable force of history, AKA existing code and books. If a vendor replaced rm by, say, remove, then every book describing Unix would no longer apply to its system, and every shell script that calls rm would also no longer apply. Such a vendor might as well stop implementing the POSIX standard while it was at it.

A century ago, fast typists were jamming their keyboards, so engineers designed the QWERTY keyboard to slow them down. Computer keyboards don't jam, but we're still living with QWERTY today. A century from now, the world will still be living with rm.[10]

Just as programmers are now writing lines of code that would fit on a punch card, they also use operating systems whose interfaces were designed to best fit teletype keyboards. Leveraging familiar constructs to boost adoption can create strange traditions.

Inheritance Paths

If people will more quickly adopt technology that follows an already familiar pattern, even one they hate, it's worth exploring how people become exposed to certain patterns in the first place. From the very beginning, computing has been a cross-functional industry. Networks of people are formed around the development of computers and the

10. S. Garfinkel, D. Weise, and S. Strassmann, *The UNIX-HATERS Handbook* (San Mateo: IDG Books, 1994), 18–19.

professions most likely to use computers to do other work. In the early days of computers, this meant computer users were both the computer scientists who built applications, developed languages, and designed architectures *and* the professionals such as scientists, mathematicians, and bankers. Even today, these groups have a tendency to silo themselves, limiting their exposure to interfaces created for other use cases.

Consider the following: one of the most successful early programming languages is COBOL, and yet modern programming languages have inherited very little of COBOL's design patterns. For example, we do not section code off into divisions, nor do we use periods to end lines of code. Few programmers would guess that PIC is a variable character string. Some of COBOL's features have reappeared in other languages, but very little of its syntax and interface was retained. Instead, COBOL itself has adopted many constructs from later languages in an effort to clean up its act.

On the other hand, ALGOL60 has profoundly shaped the structure and syntax of virtually every modern language, but you'd struggle to find a programmer today who has ever even heard of it.[11]

When we examine the accomplishments of various programming languages, COBOL is the obvious winner. COBOL programs still shuffle millions of transactions and trillions of dollars from point A to point B. It's hard to name a single thing of significance that was ever implemented in ALGOL60. The language BCPL, a similarly influential and obscure descendent of ALGOL60, survived just long enough to become the grandfather of C. So how on Earth did the patterns of failed languages become more familiar to early computer scientists than the patterns of the first truly successful, cross-platform high-level programming language?

The answer is that COBOL was a language built for people who did not want to understand how the computer worked; they just wanted

11. History buffs and recovering anthropologists do not count.

to get the job done. When the Committee on Data Systems Languages (CODASYL) was developing COBOL, the attitude among those devoted to the study and development of computers was that you should learn the flavor of Assembly relevant for your particular machine. Making programming more accessible and code human-readable was considered an anti-pattern, dumbing down the beauty of programming for an unworthy audience.

This audience, however, was made up of people who actually used computers for practical purposes, and many of them were largely unamused by the idea that they should rewrite their programs every single time they upgraded their machines. This group of people didn't care about being "real programmers." They cared about getting stuff done, better and faster than the competition if possible. Technical correctness didn't matter. Elegance didn't matter. Execution mattered, and anything that lowered the barrier to using computers to execute their goals was preferable to more powerful tools that were harder to learn.

Computer scientists during this period had opposite incentives. While COBOL users were judged and rewarded based on their ability to get nontechnical things done faster with computers, ALGOL60 users were judged and rewarded based on their ability to expand the functionality of what was even possible to do with the machines in the first place. Typically, there were two types of accomplishments in this space: get the machine to do something new or get the machine to do something more efficiently than before. For computer scientists, the programming language *was* the output. After it was developed, the next step was not to write programs, but to write papers about the language and share them with other academics for feedback and study.

Roughly three networks of people were programming computers between the 1950s and 1970s: scientists and mathematicians, data processors, and academics or computer researchers.

Scientists and mathematicians used computers for calculations, and they preferred languages that reflected scientific and mathematical

notation as much as possible. This community popularized FORTRAN.[12] When two math professors at Dartmouth wanted to create a language to make programming more accessible to students, they borrowed heavily from the syntax of FORTRAN II to develop BASIC. BASIC went on to spawn hundreds of variants, many of which are still in use today.

Data processors used computers to read data from one source and either run calculations or transform that data in some way before saving it to another source. These were the COBOL users, and that language proved so effective, it is still being used today.

If you want proof that adoption is influenced by shared knowledge among networks of people and not strictly merit, consider this: the organizations that are trying to replace their old COBOL applications today are not migrating them to what would be the first choice for data processing among modern programming languages, which is Python, but to the language that has inherited COBOL's market of a common language for businesses, which is Java.

The design of the language is never what's important; it's the people. The type of people who would have become COBOL programmers before are now becoming Java programmers, making Java the natural choice, despite that it was not designed to handle the use case for which COBOL was optimized.

Perhaps that's why so much COBOL remains in place, having resisted all attempts to eliminate it.

Academics and computer researchers focused on the development of computers. When they finally moved off Assembly, it was onto languages specifically for documenting and implementing algorithms. ALGOL60 may not have been used to build many applications, but it was what the Association for Computing Machinery (ACM) used to describe

12. FORTRAN is itself an abbreviation of Formula Translation.

algorithms in textbooks and academic sources for more than 30 years. This made it a powerful influence on the languages researchers later developed.

The University of Cambridge developed the Cambridge Programming Language (CPL) based on ALGOL60. CPL led to BCPL, which was stripped down to create B, which was further modified to create C. Next, C became the programming language of choice for this group of users, and it led to the development of a huge number of languages used by all kinds of programmers: Java, Go, PHP (via Perl), Ruby, Python, and Swift.

Also popular with this group were the Lisps. Because the original LISP was only a theoretical design document, to this day, waves of different implementations spring up quickly followed by futile attempts to standardize. During the 1960s and 1970s, Lisp was strongly associated with AI research and largely was relegated to that niche. Ironically, our own era of computing has seen much more progress in AI, but Lisp hardly plays a critical role. Instead, today's Lisps are seen as a family of general programming languages that occasionally inject ideas and structures into more mainstream languages.

So this pivotal moment of computer science history had two groups of people who programmed in order to achieve some practical purpose not related to the computers themselves and one group that worked with computers to push the boundaries of what the computers themselves could do. The bulk of languages that exist retain the constructs that were familiar to this third group of programmers, even though COBOL, FORTRAN, and BASIC had a much wider community of users.

Overall, interfaces and ideas spread through networks of people, not based on merit or success. Exposure to a given configuration creates the perception that it's easier and more intuitive, causing it to be passed down to more generations of technology. The lesson to learn here is the systems that feel familiar to people always provide more value than the systems that have structural elegances but run contrary to expectations.

Leveraging Interfaces When Approaching Legacy Systems

When I'm working on a legacy system, I always start off by evaluating the prospective users. Who will be maintaining this system long term? What technologies are they comfortable with? Who will be using this system the most? How do they expect the system to work?

That doesn't mean things can't be changed or new concepts can't be introduced. Particularly if the system is a couple decades old, the interfaces are probably tied to processes and associations that don't make sense anymore, just like the way 80-character lines come from punch cards, two-character Linux commands come from teletype machines, and the save icon on desktop applications is a floppy disk. Sometimes changing interfaces to get rid of requirements that are no longer relevant is a good thing. Defining what the requirements of a minimum viable product (MVP) would be today if the system were brand new is a great thought experiment to run when formalizing a plan of attack.

However, even when the result of change is net positive, changing interfaces is not free. Making people think adds friction and increases the odds of failure, even if the new interface is better and more consistent with the overall vision of the product.

Engineers tend to overestimate the value of order and neatness. The only thing that really matters with a computer system is its effectiveness at performing its practical application. Linux did not come to dominate the operating system world because it had been artfully designed from scratch; it scraped together ideas and implementations from a number of different systems and focused on adding value in one key place, the kernel.

The incentives that reward individual software engineers for their uniqueness, their ability to do new things, or to do old things in innovative ways are still present, even if the desire to publish papers in academic journals has been supplanted by the desire to write popular blog

posts. Yet technology is more likely to be successful when it builds on common things. These two forces are always in tension with any software project, but legacy systems are particularly vulnerable.

We know, for example, that iterating on existing solutions is more likely to improve software than a full rewrite. The dangers of full rewrites have been documented. Joel Spolsky of Fog Creek Software and Stack Overflow described them as "the single worst strategic mistake that any software company can make."[13] Chad Fowler, general manager of startups at Microsoft, describes it this way:

> *. . . almost all production software is in such bad shape that it would be nearly useless as a guide to re-implementing itself. Now take this already bad picture, and extract only those products that are big, complex, and fragile enough to need a major rewrite, and the odds of success with this approach are significantly worse.*[14]

Fred Brooks coined the term *second system syndrome* in 1975 to explain the tendency of such full rewrites to produce bloated, inefficient, and often nonfunctioning software. But he attributed such problems not to the rewrites themselves, but to the experience of the architects overseeing the rewrite. The second system in second system syndrome was not the second version of an existing system, it was the second system the architect had produced. Brooks' feeling was that architects are stricter with their first systems because they have never built software before, but for their second systems, they become overconfident and tack on all kinds of flourishes and features that ultimately overcomplicate things. By their third systems, they have learned their lesson.

---

13. Joel Spolsky, "Things You Should Never Do, Part I," Joel on Software, April 6, 2000, *https://www.joelonsoftware.com/2000/04/06/things-you-should-never-do-part-i/*.

14. Chad Fowler, "Software as Spec," December 28, 2006, *http://chadfowler.com/2006/12/28/software-as-spec.html*.

Unfortunately, when confronted with the troubles of existing systems, engineering teams tend to build the most momentum around starting from scratch. Initiatives to repair and restore operational excellence gradually, much the way one would fix up an old house, tend to have few volunteers among engineering teams. That's because Zajonc's mere-exposure effect has an upper bound. There's a point where familiarity breeds contempt.

From an economic perspective, there's a difference between risk and *ambiguity*.[15] Risks are known and estimable threats; ambiguities are places where outcomes both positive and negative are unknown. The traditional school of thought tells us that human beings are averse to ambiguity and will avoid it as much as possible. However, ambiguity aversion is one of those decision-making models that test well in laboratories but break down when brought into the real world where decisions are more complex and probabilities less clearly defined. Specifically when the decision involves multiple attributes, a positive framing of the problem can flip people's behavior from ambiguity-avoiding to ambiguity-seeking.[16]

The incentives of individual praise aside, engineering teams tend to gravitate toward full rewrites because they incorrectly think of old systems as specs. They assume that since an old system works, all technical challenges and possible problems have been settled. The risks have been eliminated! They can add more features to the new system or make changes to the underlying architecture without worry. They either do not perceive the ambiguity these changes introduce or they see such ambiguity positively, imagining only gains in performance and the potential for greater innovation.

---

15. Frank H. Knight, *Risk, Uncertainty, and Profit*, Boston: Houghton Mifflin Company, 1921.

16. Vicki M. Bier and Brad L. Connell, "Ambiguity Seeking in Multi-attribute Decisions: Effects of Optimism and Message Framing," *Journal of Behavioral Decision Making* 7, no. 3 (September 1994): 169–182, *https://doi.org/10.1002/bdm.3960070303*.

Meanwhile, the existing system has little ambiguity left. It is what it is, hypothetical potential exhausted. We know that past the upper bound of mere exposure, once people find a characteristic they do not like, they tend to judge every characteristic discovered after that more negatively.[17] So programmers prefer full rewrites over iterating legacy systems because rewrites maintain an attractive level of ambiguity while the existing systems are well known and, therefore, boring. It's no accident that proposals for full rewrites tend to include introducing some language, design pattern, or technology that is new to the engineering team. Very few rewrite plans take the form of redesigning the system using the same language or merely fixing a well-defined structural issue. The goal of full rewrites is to restore ambiguity and, therefore, enthusiasm. They fail because the assumption that the old system can be used as a spec and be trusted to have diagnosed accurately and eliminated every risk is wrong.

Beware Artificial Consistency

In the next chapter, I'll go into detail about how to balance these tensions to develop a strategy around when to reinvent and rewrite and when to leverage existing and familiar interfaces. But for now, the takeaway from this exploration of how traits are passed down should be that perception of simplicity is influenced by what your use case for technology exposes you to. Things seem easier when they are familiar. Familiarity is determined by what you are doing with technology and who you are doing it with.

But familiarity has downsides as well. While working with legacy systems, you'll find yourself fielding many proposals that claim to improve the system largely by establishing artificial consistency. *Artificial consistency* means restricting design patterns and solutions to a small pool

[17] Michael Norton and Jean Frost, "Less is More: The Lure of Ambiguity, or Why Familiarity Breeds Contempt," *Journal of Personality and Social Psychology* 92, (January 2007): 97–105, *https://doi.org/10.1037/0022-3514.92.1.97*.

that can be standardized and repeated throughout the entire architecture in a way that does not provide technical value. What's important to understand about artificial consistency is that it focuses on consistency of form and classification over functionality. As an example, Node.js and React.js are both forms of JavaScript. These two technologies look consistent, but they do different things and are built upon different abstractions. The fact that they are both forms of JavaScript doesn't give Node.js an edge when interacting with React.js over any other backend language that an engineering team might choose instead. An engineer's skill in one does not necessarily translate to the other.

Artificial consistency can bring value to nontechnical processes. For example, standardizing on one programming language makes recruiting, hiring, and, ultimately, sharing engineering resources much easier. But when the principal purpose of a modernization effort is to provide technical value, be careful not to be seduced by the assumption that things that look the same, or that we use the same words to describe, actually integrate better.

Another place where artificial consistency comes into play is with databases. The top choices for databases 10 years ago are not the top choices today, so senior leaders sometimes will ask that legacy databases be migrated to another option more consistent with whatever newer systems are using. As with the previous example, there are legitimate nontechnical reasons to do this, such as not wanting the expense of supporting two different databases that essentially behave the same way, but the issue quickly can get out of hand when the engineering team is being asked to remove the key value store they're using for a cache in favor of a relational database.

Figuring out when consistency adds technical value and when it is artificial is one of the hardest decisions an engineering team must make. Human beings are pattern-matching machines. The flip side of finding familiar things easier is that we tend to over-optimize, giving in to artificial consistency when better tools are available to us.