GLOSSARY

WAR & PEACE & IT

Business Leadership, Technology, and Success in the Digital Age

MARK SCHWARTZ
author of *A Seat at the Table*

25 NW 23rd Pl, Suite 6314
Portland, OR 97210

First Edition
Printed in the United States of America
24 23 22 21 20 19 1 2 3 4 5 6 7 8 9 10

Cover and book design by Devon Smith

For information about special discounts for bulk purchases or for information
on booking authors for an event, please visit our website at ITRevolution.com.

WAR AND PEACE AND IT: GLOSSARY

GLOSSARY

When my publisher asked me to write a glossary for *War and Peace and IT*, I happened to be re-reading Ambrose Bierce's *The Devil's Dictionary*, his cynical and amusing re-thinking of the dictionary that included definitions like "BORE, n: A person who talks when you wish him to listen" and "BACK, n: That part of your friend which it is your privilege to contemplate in your adversity." Well, you can imagine what happened when I sat down to write my glossary. My publisher later convinced me that there was a need for a serious definition of each term, but my initial definitions have survived in the blue text below.

A/B testing: A technique that involves trying several alternative designs for a feature and seeing which works best. It is often used for designing user interfaces for web applications and is a great way to settle disputes about what design is best. For example, if you want customers to click on a button and are not sure whether the button should have a picture of a bobblehead or an actual baseball player on it, you can try both. Set it up so that some users get the bobblehead and some the baseball player, and measure which gets more clicks. In the digital world where small changes can be made quickly and cheaply, and where A/B testing can easily be automated, this has become a very effective way to make design decisions.

> A technique for proving to co-workers that their design ideas are wrong. Simply set up your website so that some users see your colleague's design and some see your superior design, see what success metric yours does better at, and settle on that metric as the target.

Agile: A way of organizing software delivery efforts, first formalized around the early 2000s, that emphasizes constant learning and adjustment. It is often contrasted with "plan-driven" approaches, in which a plan is made in advance and the project team tries its best to stick to the plan. Agile approaches, on the other hand, start from the premise that there are many unknowns and a high likelihood of changing circumstances, and therefore emphasize constant adjustment of the plan to get the best results. The principles of Agile software delivery were articulated in the *Agile Manifesto* of 2001. Agile has proven to be extremely effective in the digital world, although its principles may seem counterintuitive to those used to classical project management techniques such as Gantt charts and requirements documents.

> The generic name for software delivery frameworks that emphasize constant adjustment and flexibility yet demand conformance to rigid practices. See, for example, Scrum, which insists on employing participants too busy to show up for meetings, called product owners, and an endless list of features called a backlog that can constantly be put in different orders. A variation on Scrum called SAFE, or "Sphinxlike Agile Framework something-with-an-e" promotes the use of a baffling and obscure—but impressive—diagram to achieve flexibility.

Ambassadorship: Selling ideas across an enterprise. I use the term as it is used in Deborah Ancona and Henrik Bresman's book *X-Teams*, where it is the responsibility of autonomous teams to influence the broader organization with their ideas; that is, to advocate for their efforts. I contrast this with traditional Agile techniques like Scrum, where teams are told what is valuable to the enterprise by someone like a product owner or onsite customer, and is insulated from the rest of the enterprise by, say, a Scrum Master.

> The ability to sell Girl Scout cookies to all of one's co-workers, not just one's own team. The term is used as an analogy to a nation's ambassador to a foreign state, whose responsibility is to communicate his or her nation's wishes to the other country before invading it anyway. Within a business organization, teams need to sell their ideas to the rest of the enterprise until they give up and get someone high in the org chart to agree with them and impose their ideas on everyone.

Budget model: The basis for choosing the numbers that will appear in the budget; that is, the assumptions and formulas that are used to justify a budget request. My argument is that given the uncertainties of IT and the seeming difficulty of controlling it, enterprises have often chosen targets in a way that does not consider the implications of an IT budget on the overall financial picture of the company, for example, limiting spending that will result in a positive marginal return.

> Last year's budget plus a little more. Alternatively, the amount of spend that will result in hitting Earnings-Per-Share targets.

Business value: A difficult-to-define term that is often used incorrectly in discussions of IT investments. In my earlier book, *The Art of Business Value*, I discussed how misuse of the term has led to poor IT decision-making and ineffective IT delivery. In theory, when choosing among IT investments, the one that adds the most business value should be preferred. In practice, different investments deliver different types of value which cannot practically be compared. Furthermore, because of uncertainty, business value forecasts verge on the meaningless. There are better ways to make investment decisions.

> The justification for whatever someone wants at the moment. As in "adding a screen that will let me watch my cat while I'm at work will deliver tons of business value, and I'm from the business and therefore IT's customer, so IT had better do it to keep me happy." Easily confused with ROI, NPV, and LOL.

Call option: A financial call option allows the holder to buy a particular security at a particular price (the strike price) on or before a particular date but does not impose the obligation to do so. In other words, the purchaser of the option is betting that the option to buy will be valuable because the underlying security's price will have gone up above the strike price. In many ways, Agile IT investments are like call options. The company invests money to build a subset of the features, which then gives it the option to continue investing and building more of the features—but it is not obliged to do so. If the additional features have become less valuable, for example, or if the initial features accomplish enough of the goal, then the company might decide not to make the incremental investment. That is powerful in an environment of uncertainty.

The right to buy something for less than it is worth. A fundamental human right granted to the enterprise by the gurus of Snowbird. Refusing to exercise that right is called "being risk averse" by many enterprises.

Chaos Engineering: IT systems have grown so complex that it is often difficult to even imagine what can go wrong or what the consequences of different unexpected circumstances will be. Chaos Engineering is a new discipline in the IT world that tries to make systems fail in unexpected ways, both to build confidence in their resilience and to help improve it. In 2011 Netflix released to the broader community something called Chaos Monkey—a script that deliberately caused random servers to crash. If a system is well-engineered by today's standards, it should patch itself up and continue operating despite the crash. As crazy as it sounds, the only way to test the consequences of a failure in live operational systems is to make the failure actually occur and watch what happens. Chaos Engineering is based on this pioneering work and explores the unpredictable effects that may follow when something goes wrong in an IT system.

What software engineers do for a living. For the last five decades or so, IT practitioners have been designing more and more intricate ways for their systems to create chaos.

Commander's intent: A military concept that allows for team autonomy and empowerment. Special forces teams operate in unpredictable environments, and it can be deadly for them to strictly follow a pre-ordained plan. Instead, they operate with a knowledge of the commander's intent and then adapt their actions based on the course of events with the goal of accomplishing that intent. This is a useful model in the digital world as well, since it is also dominated by uncertainty. Autonomous teams can be given the authority to make fast decisions and to be innovative, as long as they are guided by the business intent.

The missing element of top-down management. If your manager actually told you why he or she wants you to do something, then you'd know as much as he does and would be the manager, which would obviously create a conflict. Instead, many managers prefer to figure

out exactly what you need to do, which of course means that they are doing your job and are actually the subordinate. Better two subordinates than two managers, I guess.

Complex adaptive system: A complex adaptive system is one that we can't understand even if we understand its parts, because the interactions between those parts are so complex that the behavior of the system is unpredictable even knowing its current state. There are more formal definitions in the scholarly articles on the subject, but this one will do for understanding how I use the term in the book. Complex adaptive systems include feedback loops that result in non-linear behaviors and components that adapt and change with the circumstances. Biological organisms are examples of complex adaptive systems. The collection of essays in Clippinger's *The Biology of Business* shows that business enterprises also have the characteristics of complex adaptive systems. In the digital world, the implication is that we can best learn about the behavior of the system by "provoking" it and observing the result, because it is in practice impossible to predict how it will respond.

A useful term for things we don't understand, like the weather, the stock market, or biological systems such as German philosophers. Businesses, apparently, are evolutionary systems, and it therefore might pay for you to eat your co-workers before they eat you, especially the lawyers.

Cost of delay: A concept that can be used instead of ROI to make and prioritize business investment decisions. There is a cost to not having a particular technology capability—both in cash costs and opportunity costs—and it varies with time. If we have to prioritize two projects, each of which will take six months, we can calculate the cost of delaying each project six months, and whichever has a higher cost should be done first.

Assumed by an organization to be zero when the delay is caused by bureaucracy, drawing Gantt charts, fighting organizational political battles, or waiting for Oracle to release its next update. Assumed to be infinite when the boss wants something. Somewhere between these two are market-changing innovations that the enterprise's survival depends on.

Creative destruction: The economist Joseph Schumpeter proposed that economic cycles occur as large enterprises introduce innovations that displace what they had offered before, in a continual process of creative destruction. His economic theory was largely based around entrepreneurial activity by large firms. It is interesting in today's digital economy, in that large enterprises that want to survive disruption by new entrants have to find ways to continually put aside what has worked for them in the past and innovate anew.

> Another name for Chaos Engineering. Dadaist art briefly created a fashion for creative destruction, but it really came into its own as management artists produced priceless works like Blockbuster Video, Palm Pilot, and America Online. Alas, poor COBOL! I knew him.

Data asset: I have defined the data asset as the intangible, off-the-books asset that represents the enterprise's ability to extract value from its data. It is a complex asset that includes the data itself, its fitness-to-purpose and cleanliness or quality, its accessibility to employees who want to analyze or use it, the tools that are available to analyze it, the privacy and security safeguards that surround it, and the organizational policies and structures that affect its use. We often speak of data as having a value, but really it is not the data per se, it is the agility with which the data can be used to accomplish strategic objectives.

> The vast amount of information that the company has collected which no one knows about or has access to, and which spells each customer's name three different ways.

Data lake: In traditional IT, one stored data in a way that facilitated the way one planned to use it. For analytics, this meant creating a well-defined and rigidly structured data warehouse based on the types of queries expected. But today, we collect a wide variety of data before we know how we will use them, and value flexibility and agility in their use. A data lake is a collection of data in all of its varieties and forms which can be used flexibly with the broad range of analytic tools now available. In other words, data we collect can be dumped into the data lake with little or no attempt to manipulate them for analysis because today's analytic tools are more flexible. For example, if your company acquired another company with different IT systems, it used to take time to integrate systems to be able to combine their data for reporting. But with a data lake, you

pour in the data from both companies' systems and can often begin combining them immediately (for analysis, not necessarily for transaction processing).

> A stress-producing way of collecting so much data that you will never be able to figure out what to do with it, so-called because of your fear of drowning in its depths.

DevOps: Today's best model for delivering and maintaining IT systems. Traditionally, development and operations were two siloed organizations within IT, with the development team responsible for creating new software or making changes to existing software, and the operations team managing that software as people used it—among other things, responding to emergency outages. Unfortunately, in that model the two organizations had conflicting incentives: operations wanted systems to be stable, development wanted to deliver code and changes as quickly as possible—which causes instability. In the DevOps model, a single team is responsible for both development and operation, incentivized both to deliver quickly and maintain stability. With the help of automation and this combination of skills, conflict between these two objectives can eliminated.

> With DevOps, the IT community discovered the *portmanteau*, a word made by gluing other words together. This very useful technology has resulted in constructions like DevSecBizFinOps. We have only begun to tap the potential of this linguistic technology, which someday may create a FrankensteinMonsterLike sewn together word that actually walks off on its own, starts turning servers on and off randomly like a ChaosMonkey, and forever haunts the IT department.

Discovery-driven planning: In a very predictable environment, a plan made in advance can be adhered to and can lead to the expected results. The more unpredictable the environment is, the less likely that a plan made in advance will actually lead to its desired outcomes. Instead, the plan must constantly be adjusted as new information is acquired—in fact, one of the goals of execution is to acquire useful new information that can be used to adjust the plan. In discovery-driven planning, the team tries to identify the key unknowns that will affect the plan, to organize execution so as to gather information to reduce those unknowns, and then to adjust the plan accordingly.

Remember all that money you spent getting your employees trained on classical project management techniques? I hope not.

Escaped defects: "Defect" is meant as a somewhat broader concept than "bug." It includes, for example, problems like security vulnerabilities, pieces of code that won't scale well if the number of transactions increases, and features that aren't usable by people with disabilities. In today's software development practices, developers write automated test scripts to test their code, and they run those tests constantly as they are developing the software. As a result, the automated tests are constantly finding defects and the developer is constantly adjusting the code. In other words, finding defects is a good and expected part of the process. The only bad defects are the ones that *don't* get detected—the ones that make it through testing and remain in the code that is released to users, a.k.a. escaped defects.

We use the term "defects" to avoid the terrifying-sounding idea of escaped bugs. Nevertheless, an escaped defect is one that bites the user of a system and that is seemingly as inevitable as mosquitos in the Alaskan summer. Once an escaped defect is caught it can safely be returned to the system, over and over again.

Feature bloat: The undesirable property of IT systems that have unnecessary or under-used features. IT systems have a tendency toward bloat as new features are constantly added to them, while old features are almost never removed. In the book, I make the case that writing and signing off on a requirements document in advance of building an IT system invariably leads to feature bloat, and that every feature that does not contribute to the business goal of a system should be considered feature bloat. Feature bloat is the enterprise's enemy: it increases costs, provides more opportunity for defects, increases the "attack surface" available to hackers, and requires that those extra features continue to be maintained over time. We used to think that scope creep was the danger—that is, additions to the feature list after development commenced—but in truth it is feature bloat, even if that bloat is in the initial set of requirements, that we should worry about.

An increased opportunity for escaped defects. After a full meal of requirements engineering, many projects experience feature bloat,

which is—to put it indelicately—primarily just wind. Feature bloat is also known in Scrum as the backlog.

Functional debt: I have used this term in analogy to the term "technical debt," which is in wider usage. Functional debt, in my definition, refers to the places where legacy IT systems don't really do what the company needs them to do today. Often the company will have found workarounds because the system doesn't do what they need; or, they may have had to constrain their business processes or forego moving into new markets because of this functional debt. In that sense it is "debt"—it has a cost, as if the company is paying interest on it until it is retired by replacing or enhancing the system.

A disturbing lack of frustration when employees or customers have gotten used to the parts of an IT system that make no sense. New employees have to be taught to do things the accepted way rather than the obvious way. Modernizing an IT system is the technical name for declaring bankruptcy.

Growth hypothesis: In Eric Ries's book *Lean Startup*, he talks about the product development process as guided by two hypotheses: a value hypothesis and a growth hypothesis. At any given moment, the organization is proceeding based on an idea about how the new product or service will create value for its customers (the value hypothesis) and how it will get customers to adopt the product (the growth hypothesis). The important point is that these are hypotheses, not certainties; a goal in the product development process is to conduct experiments and gather information that will confirm or refute them, in which case other hypotheses can be substituted. In the book I argue that "requirements" given to IT should be thought of only as hypotheses about what will effectively accomplish the business goal, and these hypotheses should be tested before full commitment is made to investing in them.

The assumptions one makes to convince venture capitalists to fund one's startup; or, the misguided belief that customers will find your product irresistible because you do.

Hypervisor: A piece of software that makes a single computer act as if it is many computers; that is, act as if many "virtual machines" are available rather

than just the single "physical" machine. It is a core component of datacenters today and of the cloud; an AWS datacenter has many *many* "racks" of servers, and each of those servers has a hypervisor that—in effect—multiplies the number of servers available to customers.

> A supervisor who is everywhere at once or tries to be. See *Commander's intent*. Similarly, the software that supervises a computer's operations to ensure that all defects can be enacted simultaneously.

Impact mapping: A technique described by Gojko Adzic in his book *Impact Mapping* that I have found extremely useful when applying the principles I present in the book. With Impact Mapping, a team begins with a business objective, maps it into what needs to change in order to accomplish that objective, and then maps each of those changes into possible ways to cause the change. These become the hypotheses that can be tested and used to drive the delivery of new IT services or business process changes. I present it as a way to support the *Objective Model*.

> A technique for identifying all the people who will resist your changes and plotting ways to trick them into cooperating.

IT asset: I have defined the IT asset as the intangible, off-the-books asset that represents the total of the enterprise's IT capabilities, inasmuch as they can be used to derive future revenues and manage future costs. It is a complex asset that includes not just the capabilities of the IT systems, but their internal quality, their architecture, and their flexibility for change. An important point of the book is that this asset is more valuable—literally—when it is more agile; that is, if the IT asset is amenable to fast, low risk, and inexpensive change, then it actually has more financial value because it supports business agility, which allows the company to seize opportunities and respond to disruption.

> The total of all the IT capabilities you have now that will soon be disrupted if you do not read and follow the advice in my book. As Napoleon says in the foreword, it is "recommended and required reading," but since you've already paid for it, why not start using it to groom your IT asset now?

Lean: The set of techniques and mental models behind Lean manufacturing and other Lean disciplines, highly influenced by the Toyota Production System. Lean focuses on shortening lead times by eliminating waste from processes. Lean theory identifies certain typical categories of waste and advocates mapping each step in a process, finding instances of those kinds of waste in the steps, and eliminating it. It has been applied to the software delivery process and is a fundamental part of the theory behind DevOps.

> Desirable in IT and manufacturing, where it designates a lack of waste; catastrophic in meat, where it designates a lack of taste. In IT and manufacturing it is the domain of black belts, while in food it is presumably in the domain of sumo wrestlers. You can make an IT process lean by dispensing with parentheses and computers [see entry for *Serverless*], defects, cubicles, statefulness, users, Gantt charts, politics, jackets and ties, and . . . *ahem*, feature bloat. You can make food lean by dispensing with taste buds.

Legacy estate: The IT systems that an enterprise has in place because of actions it has taken in the past (development or purchase). Because businesses are constantly changing (and technology is changing as well), at any given moment this legacy estate is likely to be somewhat out of date. In other words, it is carrying *functional debt* and *technical debt*. The legacy estate often acts as a drag on the enterprise's nimbleness and business agility.

> The vast, crumbling technology home of your current business processes, possibly sold to you by a realtor who stood to benefit.

Loose coupling: An important concept in IT system design with a large impact on agility. In general, parts of an IT system depend on other parts, and a change in one part can have effects on those other parts. Loose coupling is the design concept of reducing these dependencies. It is a design ideal, difficult to realize completely in practice. But the more IT components are coupled loosely, the less cost and risk there is in making a change—in other words, the more agile the architecture is.

Not, as it sounds, a frowned-upon sexual behavior, but yet a way to proceed without knowing much about your partner. Loose coupling often breeds a multitude of microservices.

Marginal cost: In economics, marginal cost is the cost of producing one incremental unit at a given level of production. So if you are currently producing 1,000 bobbleheads a year, what is the additional cost of producing the 1,001th? If it will result in more marginal revenue than marginal cost, you should generally do so. In the book I show that in the old IT world, many decisions made by considering total cost should now in the digital world be made by comparing marginal costs to marginal returns. The tools of the digital world allow us to make decisions at the margins, and doing so adds economic value.

The incremental cost that might break the camel's back—but which you'll keep spending until the accountants notice or the camel objects. In an IT budget, the cost of your 1,001th instance of SAP.

Metered funding: More or less equivalent to staged investment, an investment technique where funding is provided in increments based on the results of preceding funding increments. Instead of buying $100 worth of candy believing you can eat it all, you buy $10 worth of candy, see if you are sick yet, then buy another $10 worth of candy, and continue the process, stopping when you feel sick. This makes it more cost-effective to reach your goal of eating until you feel sick, if such be your goal, as you might not need to spend the full $100. In business investments, it is a good way to reduce the risk of an investment by continually checking on the results of your investment before committing more money.

For a startup, the practice of increasing losses by taking more money from the VCs, thereby making your company seem more valuable in the long term, earning respect and admiration and therefore compelling VCs to invest additional funds. This process lowers the risk VCs face, because they are able to confirm the startup's ability to lose money before they agree to provide more.

Microservice: Software today consists of a number of small software components that interact with one another, passing messages back and forth and

delegating tasks to each other. Today's best practices suggest architecting software as a number of small, independent, service-providing components, each of which accomplishes a piece of the total functionality and which delegate work to each other—microservices. One advantage of microservices is that each service can be scaled independently—in other words, if a particular function does a high volume of processing, you can add hardware power to it without also adding hardware power to the rest of the system, resulting in cost efficiencies. That is one advantage among many for microservices.

> Something smaller than last year's services but larger than next year's lambdas. An improvement to yesteryear's Service Oriented Architecture in that smaller services more easily fall through the cracks in the budget. The child of loose coupling that will one day grow into a full-fledged deployment, once incubated in a repository. If not killed by a Chaos Monkey or ChaosMonkey.

Minimal viable product (MVP): See *Growth hypothesis*. In order to test a *value hypothesis* or *growth hypothesis*, the company builds the smallest possible product it can give to customers that will provide information to confirm or refute the hypothesis. In other words, the company mitigates risk and designs a better product by building it incrementally, starting with an absolute bare minimum usable version that provides the company information on the most important unknowns it faces. The more "minimal" the product, the less the company is risking, and the less it is paying to get the information it needs. Contrast this approach with the traditional approach of writing a requirements document defining a product that the company thinks customers want, then building the product, and only then finding out whether customers really want it.

> A version of the product that can quickly be used to put senior executives into a panic. The features of the MVP can be defined by making a list of the features everyone thinks should be in the ultimate product and burning it.

Objective Model: I have posited the Objective Model as a way a company can govern projects while at the same time leaving teams autonomous and allowing them to innovate. The idea is simply to pass a business objective to a cross-functional team to fulfill, without the intermediate step of developing

"requirements." Because with today's techniques the team can immediately begin delivering capabilities that help accomplish the goal, their work can be overseen simply by measuring goal accomplishment against cost, and since investments can be staged (metered), risk is very low.

> The radical idea that an enterprise should decide what it wants to accomplish and then go about accomplishing it. The even stranger idea that success can be measured by figuring out how successful you've been, rather than by making the lines on a Gantt chart line up. Unlikely to be adopted by the enterprise.

Organizational asset: Along with the *IT asset* and the *data asset*, I have defined the organizational asset as the intangible, off-the-books asset that represents the total of the enterprise's human capabilities, inasmuch as they can be used to derive future revenues and manage future costs. It is a complex asset that includes not just the skills of the people in the organization, but also the company's bureaucracy and pre-defined processes that determine what those people are allowed to do, the training that has been provided to them, the organizational structure and how work is distributed, and the cultural norms that affect their behavior. An important point of the book is that this asset is more valuable—literally—when it is more agile; that is, if the employees are prepared to move quickly and nimbly, for example, because the company has hired generalists whose skills remain valuable even if the environment changes around them.

> The people who can thwart a transformation and the business processes and organizational structures that allow them to do so. An asset that easily becomes part of the salvage value when the company goes out of business.

Process agility: The agility of standardized processes in use across the organization; how easily they can change. This is largely affected by the way the organization handles its bureaucratic requirements. Is it a learning bureaucracy, where rules can be adjusted frequently and quickly? Is it a Lean bureaucracy, where rules are enforced in the least wasteful way and as a result impose the fewest costs on change? Process agility is an important part of what I have defined as the *organizational asset*.

The ability to move off the train tracks when the train is coming at you, without first filling out a Request for Transfer.

Product model: See *objective model*. The product model is another way to decentralize authority. Teams are associated with specific products or services that the company offers and are free to innovate, prioritize, and make decisions within the bounds of that product's success as defined by the company. At AWS, small product teams are fully responsible for their products and constantly take input from customers to drive their product roadmaps. They are authorized to act based on that customer input without having to seek approvals (generally) at higher levels of the organization.

A way of allowing teams to make decisions even when senior executives are playing golf or Fortnite; considered a good practice in DevOps and golf clubhouses.

Refactoring: Refactoring is the process of improving the technical internals of a system without changing its outward functionality that users of the system see. It is done to reduce technical debt, keeping the system nimble and less likely to have defects. Often it involves simplifying code or making it fit standard patterns or conform to design standards. It is a normal part of IT system delivery, as technical debt develops naturally over time and must be addressed or else it will lead to longer delivery cycles, more defects, or potential security vulnerabilities.

Retrofitting the legacy estate to make it earthquake-proof, thereby increasing its value for people who want to move to San Francisco and become software gurus.

Reference data problem: Data that standardizes the choices available to users for a data field or that provides common standards for a data item is referred to as reference data. For example, when you are asked to enter your address in a web form, you might be given a drop-down that lets you choose from a list of states or countries. The data in that drop-down is reference data. Reference data seems very straightforward but turns out to have subtle challenges. For example, consider the list of countries that are available to choose from. The country list you can choose from when entering your current address might

be different from the country list you see when asked to enter the country in which you were born, as some countries might have ceased to exist or changed their names (perhaps Macau, for example). These types of issues can lead to many unexpected complexities in an IT system.

> When my list of possibilities does not match your list of possibilities and I can't convince you that mine is right, and then it turns out that the list that is built into our software is different and can't be changed anyway.

Scope creep: The waterfall approach to software delivery required that all of the requirements for the software be specified before development began. Any additions to the requirements after that point were labeled scope creep and were strongly discouraged. The problem with this is that it is normal during a software development process to find out that requirements need to change. As Jeff Patton says in his book User Story Mapping "scope does not creep—understanding grows." Outlawing scope creep incentivizes employees to pad the initial requirements with everything they think they might need, resulting in feature bloat.

> A sinister plot to deliver everything the enterprise needs. The plot can only be defeated by spending years listing all the things the enterprise doesn't really need and doing those things.

Servant leadership: A leadership style that views the leader's job as helping "subordinates" to be successful in their work. A servant leader sees his or her employees as the ones who are actually adding value, and his or her job as supporting them in doing so, often by removing impediments that they face in accomplishing their initiatives. Servant leadership works well with Agile approaches, where cross-functional teams are autonomous and empowered, rather than continually taking orders from leaders. Servant leadership is contrasted with "command and control" leadership styles, where the "boss" tells the employees what to do and makes sure they do it.

> A leadership framework that advocates the leadership techniques of "bringing the donuts to the meeting" and "making sure a conference room is available" rather than "securing a big, comfy corner office"

and "condescending." Still considered experimental in many large enterprises.

Serverless computing: A relatively new capability of the cloud that eliminates the need for IT teams to provision and configure infrastructure to run their code. Instead, they can simply specify the conditions under which they want a piece of code to be run, and the cloud does the work of providing the computing power. The enterprise only pays for the amount of computing power used, rather than the cost of having a computer that sits around between tasks.

What used to be called LISP programming, made much more efficient by eliminating the parentheses. It saves money by not only eliminating the parentheses, but eliminating the computers too.

Staged investments: See *metered funding*.

Technical debt: The discrepancy between the internals of your system and the ideal. Given a particular set of functions that code implements, the internals of the code and infrastructure can be better or worse in terms of maintainability, quality, scalability, security, and so on. Any differences from the ideal have a cost to the company, although the cost might not be apparent on the surface: technical debt might cause later changes to the system to take longer, may limit what the system can be made to do, may have hidden security vulnerabilities, or may provide a comfortable hiding place for bugs. I distinguish technical debt from *functional debt* (see above), which is the gaps from the ideal functionality; technical debt refers only to the hidden internals of the system, not the functionality that users see.

What we all owe to software vendors like Oracle until we replace them with open-source software or any new software created in this millennium. Note: they will still claim that we continue to be in debt to them. Also refers to code written by anyone who is now older than 25.

Value hypothesis: See *growth hypothesis*.

Waterfall: The traditional way to organize IT delivery projects, by dividing them into phases with one phase following another linearly. When represented

on a Gantt chart, which it typically was, such a project resembled a waterfall. A waterfall project is carefully planned out in advance, with the requirements defined up front, and a series of milestones planned out to gauge the project's progress. In general, the IT system was delivered at the end of the project, in a single large delivery. Success in a waterfall project is measured by adherence to the project plan and schedule. Although this is the traditional way to organize IT projects, it has proven to be largely ineffective and has been replaced by Agile delivery approaches.

> An ingenious way to avoid delivering anything until it is too late and supporting that pace with status reports. Delivery can be pushed off further by spending time constantly adjusting Gantt charts. Because it is ineffective, it is the preferred approach of companies that plan to be disrupted.

ABOUT THE AUTHOR

Mark Schwartz is an iconoclastic CIO and a playful crafter of ideas, an inveterate purveyor of lucubratory prose. He has been an IT leader in organizations small and large, public, private, and nonprofit.

As an Enterprise Strategist for Amazon Web Services, he uses his CIO experience to advise the world's largest companies on the obvious: time to move to the cloud, guys. As the CIO of US Citizenship and Immigration Services, he provoked the federal government into adopting Agile and DevOps practices. He is pretty sure that when he was the CIO of Intrax Cultural Exchange, he was the first person ever to use business intelligence and supply chain analytics to place au pairs with the right host families.

Mark speaks frequently on innovation, change leadership, bureaucratic implications of DevOps, and using Agile practices in low-trust environments. With a BS in computer science from Yale, a master's in philosophy from Yale, and an MBA from Wharton, Mark is either an expert on business value and IT or just confused and much poorer.

Mark is the author of *The Art of Business Value* and *A Seat at the Table*, and the winner of a *Computerworld* Premier 100 award, an Amazon Elite 100 award, a Federal Computer Week Fed 100 award, and a *CIO Magazine* CIO 100 award. He lives in Boston, Massachusetts.

ADDITIONAL TITLES BY THE AUTHOR