The Tower of Babel Did Not Fail

Paul Adamczyk
paul.adamczyk@gmail.com

Munawar Hafiz
University of Illinois
mhafiz@illinois.edu

Abstract

Fred Brooks' retelling of the biblical story of the Tower of Babel offers many insights into what makes building software difficult. The difficulty, according to common interpretations, comes from the communication and organizational problems in software development. But the story contains one more important lesson that people tend to miss: one cannot accomplish *impossible* goals, which programmers are often asked to do. Software engineering, as a discipline, can overcome poor communication; but as long as we attempt to live up to impossible expectations, we will always fail.

Categories and Subject Descriptors K.6.3 [*Management of Computing and Information Systems*]: Software Management—Software development; K.7.m [*The Computing Profession*]: Miscellaneous—Codes of good practice

General Terms Design, Documentation, Economics, Human Factors, Languages, Management

Keywords communication, documentation, engineering, programming, science

Introduction

The story of the Tower of Babel comes from the book of Genesis, but software developers, regardless of their metaphysical views, know it as a story of failing software projects – a tradition that dates back to Fred Brooks [6].

The main takeaway from this story is the confusion of the language that led to the failure of communication between men. God became afraid that men working in unison might accomplish impossible things, so he used his supernatural powers to confuse their language. Interpreting this part of the story literally implies that the Almighty was considering (if only for a brief moment) men to be his equals and then resorted to a trick to reclaim his superior position. But more than a story about a power struggle between mortals and God, the Tower of Babel contains lessons about human endeavors. It suggests, for example, a historical explanation of how large ancient cities evolved – it was a communal effort, not just the work of powerful rulers. More to the point, the builders of the tower had failed due to a technical reason – they were trying to accomplish an *impossible* task: to build "a tower that reaches to heavens." Once the failure became apparent, the tribes separated, migrating in different directions, because their ambitious project didn't hold them together anymore. They lost contact and, over time, their languages began to grow apart. Had they tried to re-assemble later to build the tower again, they would not be able to communicate effectively.

I[1] have heard the story of the Tower of Babel told many times, as a quote from Brooks, to illustrate why software fails. The main point made by the storytellers is that programmers can't communicate effectively among themselves and with their stakeholders. This fact should not be surprising: an industry built by introverts for introverts is bound to experience some communication problems. But there is more to consider: much too often, a software project fails, because it tries to accomplish an impossible goal, something that can't be done, by definition. What are some impossible goals of software engineering? To build perfectly malleable software. To overcome the physical limitations of the domains to which software is applied. To make software development project management easily predictable. To guarantee consistent return on investment in software by adding new features. Some expectations come from non-technical stakeholders, who don't understand which tasks can and can't be accomplished with software. (This is not surprising: God in the biblical story was also unable to gauge the feasibility of building the tower of Babel.) Other expectations originate within our discipline. But, in spite of the constant pressure to achieve impossible goals, our discipline is continuously growing. To explore this apparent contradiction, this essay examines the expectations placed on software engineering.

OOPSLA/SPLASH'10, October 17–21, 2010, Reno/Tahoe, Nevada, USA.

[1] A note on the authors' voice is in order. Even though we are two authors, our opinions are presented with a collective *I*. I use the term *we* when referring to the community (i.e. programmers, *or* software developers, *or* software engineers). These three terms will be used interchangeably for reasons that are explained later in the essay.

1. On Impossibility

impossible *(adj) 1 a : incapable of being or of occurring b : felt to be incapable of being done, attained, or fulfilled : insuperably difficult*

—Merriam-Webster's Online Dictionary

Brooks notes that the goal of the builders of the tower was "naively impossible" [6], but he doesn't follow up on this observation, because he is focusing on communication and organization. Software development today is facing many impossibilities. Granted, many other disciplines constantly confront impossibilities. Medicine, for instance, deals with incurable diseases and ultimately always loses to death. The difference is that people outside of the medical field *understand* that not all diseases have cures. But people don't have an intuitive understanding of the limitations of software, which gives rise to many impossible expectations.

It's possible to show that some expectations are incorrect. Theory of computation proves many results about the limits of computability, which show that computers can't do everything. Others, like the impossibility of building defect-free software, are slowly becoming understood. Dijkstra's assertion that testing can't prove the absence of bugs [10], coupled with Pareto's 80/20 rule [2] that predicts the increase in cost of removing each additional bug, help to highlight what makes this expectation unrealistic. Some other expectations are also impossible to achieve, but they are not obvious to people outside of our discipline.

Perfect Flexibility

Some blame for the unrealistic expectations from software can be attributed to its flexibility and scalability. Software doesn't behave like physical materials, because it has no physical presence, which makes it a unique medium. As such, there are no good analogies that would help people who don't have experience with software to grasp intuitively which tasks are easy to accomplish with software and which are impossible.

Software is malleable and deceptively easy to change. A modularized system makes it easy to make sweeping changes that affect the entire product, if the software was designed to handle that type of change. But it's impossible to build software that can easily accommodate every type of change – no modeling or decomposition technique can do that, regardless of what research in MDA or AOP might suggest. Not all changes can be planned for at the design time and other, seemingly trivial changes break systems or take long time to implement. These, admittedly capricious, restrictions baffle non-technical stakeholders. One day the programmer is a magician, another day the same whiz can't add a simple button to the user interface.

Another form of flexibility in software is scalability. There are many ways to build systems out of existing systems. Once implemented, a program will always perform its well-defined tasks without failure. However, this property isn't guaranteed – combining two seemingly flawless programs into a new one doesn't guarantee a flawless result. To a non-technical stakeholder, this constraint is also difficult to comprehend.

Software can achieve excellent reliability, even in highly distributed environments. For example, telecommunications systems, both wired and wireless, operate at five nines of availability (up 99.999% of the time which translates to about 5 minutes of downtime per year). Telecommunications software is very flexible. Over time, cellular networks have been extended to support a wide range of capabilities of cell phones, e.g. accessing the Internet, downloading and streaming music and movies. But there are limits to this flexibility; when the limit is reached, adding new features becomes increasingly difficult as software turns brittle.

Although nobody realistically expects perfect flexibility from software, different people have experience with software that was perfect in one -ility or another. Given enough people with different experiences, their collective expectations amount to perfection.

No Physical Limitations

Applying software to a new domain helps in automating manual tasks: thus improving productivity. These gains encourage further application of software, with the intent of overcoming physical limitations of the domain.

Unfortunately, software can't remove limitations inherent to the domain for which it's built. Embedded systems, even though they are built according to the well understood laws of the physical world, are bound by the physical limitations of mechanical systems they control. Financial software is limited by the accuracy of economic models of how money circulates. (The fate of the exotic financial derivatives that brought about the current economic downturn shows that these rules apparently still aren't understood very well.) Social Web applications are limited by the attention span of people (more precisely: teenagers) who use them. We build software that extends the known limits, but we're bound to run into new limitations at some point.

Another limitation software helps to overcome is how much information and knowledge one can access. But software is only a tool, and the information is only as good as its understanding. One can't expect that the quality of information will improve by the virtue of being implemented in software. Nor can the users of software become more intelligent just by using it.

The expectation of overcoming physical limitations comes from users and system analysts, who understand the needs of the market, but not the limitations of software. While no sane person would ever consider using an abandoned landing strip as a base of an aircraft carrier, software sometimes goes through even more unbelievable metamorphoses. The manner in which software is created (i.e. the software development process) also raises impossible expectations.

Predictable Project Management

Project managers expect that the quality of project scheduling and estimation in software engineering will match those of other related domains. Unlike the previous two expectations, which are based on the observation of current trends, nothing in the state of practice in software engineering supports this expectation.

Historically, management of software projects has been poor. "The Chaos Summary 2009" from the Standish Group reports that only 32% of all projects they studied were successful. They define a successful project as delivered on time, on budget, with required features and functions [32] Critics have been calling it the software crisis. Parnas [29] points out the fallacy of this claim: it's not possible to have a crisis that lasts for decades, yet this simplistic explanation resonates with many non-technical stakeholders. The Standish Group recognizes that some improvement occurred over the years. They attribute the improvements of project results to more formalized project management, but also point to the adoption of iterative software development as a key factor [31]. The Standish Group studies only IT projects, so we can't compare their findings to other fields.

In my experience, the reason why some projects *are* delivered on time and within budget has little to do with the quality of schedule and estimates of project management, but rather indicates the perseverance of programmers. Given a choice between working long hours or being let go because the project failed, we choose to face the familiar technical challenges, hoping that if this project succeeds, the next one will have more realistic deadlines.

It's easier to estimate the time and effort required if a project is very similar to something that was done before [7]. In civil engineering, all bridges share the underlying physics, materials science, and mathematical principles, so building a new bridge is an exercise of "routine design" [8], which means following predefined steps. There's no such common denominator for software, which requires extensive exploration (or "original design" [8]) nearly every time a new product is built. Software estimates fail, because (application frameworks and software product lines notwithstanding) the products being built are one of a kind. Even if new software is built solely to replace an existing system, it's still delivered with new features. Otherwise, the existing software would be used instead, thus eliminating all the risks of developing a new product.

Software project management is failing, because it focuses mostly on *controlling* the software development process according to the unreliable estimates. Tom DeMarco points out that the focus on people is important, but instead of putting so much effort on control, we should focus more on the main tasks: on transforming the world with software and fostering exploration [14]. The focus of what we do should not be on the artifact, the program, but rather on the environment in which the program will be deployed. Exploration is harder to measure than the execution of predefined steps, but this is the only way to make real progress. Unfortunately, even good development processes that improve programmer productivity, such as agile development, morph into something different when project managers try to apply them in practice. Mitchell observes that many organizations that claim to practice agile development selectively follow the management practices (iterative planning and daily stand-up meetings), but not the engineering ones (test-driven development, user stories) [26]. As a result the exploratory quality is completely abandoned. Improvements can come only from wholehearted attempts to change.

Market conditions, business contract negotiations, and many other factors that influence project schedules are beyond programmers' control. Good project managers understand these factors and are able to set realistic goals, achievable by the development team. But often people without technical expertise, who understand only the management side make all the important decisions. The result is impossible deadlines, forcing developers to deliver an unfinished or incompletely tested product, which inevitably fails at the hands of real users. Often this is the only time when non-technical management communicates directly with developers. But fixing the reported defects quickly is not enough to regain their respect. Business people don't appreciate workers who can't meet schedules or deliver quality products. The past performance discredits developers from providing reliable input later. So developers find themselves continually trying to meet deadlines assigned by people who don't know how to set realistic expectations.

The lack of understanding of the difficulties of software development by the management precludes predictable project management. It's impossible to successfully manage a task without having some understanding of what the task entails.

Guaranteed Profits

Project management is not a goal; it's a means to accomplish the main goal of any product, which is to make money. Brian Foote summarizes it well:

Programming is building a bigger abacus for man to count his money.

Non-technical stakeholders are not familiar with the constraints that make building software difficult. They tend to focus on the bottom line. The expectation of guaranteed profits is a major source of unrealistic expectations that plague our profession. Owners, investors, and other stakeholders want to control/influence what we do and how. They are looking for consistent return on their investment, because their main goal is to continue to make profits. But such a goal has no end in sight – regardless of how much money one has made already, there's always more to be made.

The drive for profits affects other factors of software development, such as good quality and reuse. Software is often being extended past its prime. To maximize profits, non-

technical stakeholders insist on adding more features, even if the software is becoming bloated. In the quest to capture new markets, old software quickly reaches its limits – it becomes costly to maintain and eventually profits dwindle. Alternatively, if certain software captures most of its market before it becomes unmaintainable (e.g. SAP in accounting and business operations), its users are stuck with software that cannot be changed.

Software doesn't age or wear out the way physical materials do. Rather than through use, software ages when it's modified, especially when modified hastily [29]. Since maximizing profits requires adapting existing software to new problems by adding more features, the changes cause software to age.

Software has many limitations, but it opens many possibilities and gives people the illusion that anything can be done and (because it's done by computers) it can be done well. This illusion is spread by advertisers, vendors, technology *evangelists* and mis-informed business people. Software vendors build up unbelievable expectations, even among software practitioners. They offer productivity-increasing software platforms, libraries, tools, and services. These solutions rarely match the overblown promises, but by the time the customers take note, the vendor is already marketing the *next* version of their product that *will* solve the problem. Similarly, well-known experts offer new methodologies and processes aimed to solve the same types of problems. Too often, it's the fame of the expert rather than the quality of ideas that prompts organizations to implement these inventions. Non-technical people get caught up in the hype, convinced that software engineering is just one step away from a breakthrough that will allow them to create perfect software; they want to be the first ones to accomplish this feat.

2. Self-imposed Impossibilities

So far, I've discussed four types of impossible expectations. That software is perfectly flexible, so making any changes to software is trivial. That software can overcome physical limitations of the domain it serves. That software can be managed in an easily predictable way. That growing software by adding more features will guarantee continuing financial success. These expectations have a common characteristic – they originate outside of our community, from non-technical stakeholders. This suggests that perhaps software development would not be restrained by impossible expectations if we simply ignored the expectations of non-technical people.

"As We May Think" [9]

To entertain this hypothesis, let's consider one very successful endeavor in software development, the World Wide Web. Tim Berners-Lee built the initial version of the Web for the physicists to help them organize their vast information with the use of hypertext. He believed that the Web could connect humans and machines so "that once the state of our interactions was on line, we could then use computers to help us analyze it, make sense of what we were doing, and where we individually fit in, and how we can better work together" [4]. (This great idea goes back to Vannevar Bush [9].) Initially, in the early 1990s, the Web was free from unrealistic expectations. As it grew, it incorporated academic institutions and then commercial traffic.

Then the Web became a huge success, giving rise to new industries such as eCommerce and social networking. Soon, the Web began to face same types of impossible expectations of perfection and financial success. New types of software systems, most notably search engines, appeared and raised the expectation of bringing disparate sources of information together to produce new knowledge. Unfortunately, petabytes of data available on the Web can't be searched effectively. Few improvements have been made in the quality of Web search in the last decade (Google's page rank algorithm still remains the biggest breakthrough idea) but searching is nowhere near true understanding. The expectations of predictable project management is also present, and equally impossible, because the technologies for building new applications on the Web change very often, making it difficult to estimate schedules. And, like every other successful endeavor, the Web faces the same impossible expectations of continuing profits by adding new capabilities on top of the base Web technologies.

More deserves to be said about the first impossible expectation – the perfect flexibility of software – because a lot of progress has been made in this area. The software running the Web is very flexible. The systems available on the Web today showcase incredible capabilities, especially since they are built on the original HTML, HTTP, and URI protocols designed in the early 1990s, and have to account for all their limitations. A great invention of the Web is the status code "404 Not Found," which solves the problem of presenting unavailable data – if there is no resource to show, the system handles the error gracefully. Another example of extending flexibility is the upcoming HTML5, which includes multimedia and text editing support, which promises to make Web software behave the same as desktop software, in all browsers. The flexibility of the Web is constantly being pushed to its limits, but it's not getting any closer to reaching the idealistic goals set up by Berners-Lee.

The history of the Web brings forth two other impossible goals of software engineering. First, it's not possible to design a massive system like the Web up front. The Web came into existence in a highly organic, non-designed, self-organized manner, and recognizing this fact challenges the way we think about building software systems. Second, although Web is a qualitatively as well as quantitatively different system from anything previously experienced in software engineering, we still try to think of it as just another "machine", a system that we can manage, and set expectations for.

Whether it's Berners-Lee designing a worldwide network to better understand how people think, or companies like Google using the Web to deliver free services for their users – the end result of trying to achieve great, but impossible goals is the same: increased expectations. These idealistic expectations originate within the software engineering community. But regardless of their source, the impossible expectations affect the core of software engineering; they can't be avoided. Every successful software will run into them eventually, by the virtue of its success.

Metaphors of Our Discipline

Another source of unrealistic expectations that originate within our community is the definition of our discipline. More precisely: lack of a single definition. Software engineering encompasses so many tasks that it's impossible to do all of them well. Before explaining why the self-image of software engineering is a source of impossible expectations let me summarize some of the more prevalent views (or metaphors).

Without distinguishing between computer science and software engineering, Fred Brooks says that building software is like a **craft**, because the programmer is a toolsmith [5]. Programmers serve others by fixing their problems and derive their rewards and self-worth from solving problems. Brooks emphasizes that building software is not a science, even though the research methods we practice are modeled on sciences.

Another direct analogy for software development is the **assembly line**. It's a very old perspective that comes from the true engineering disciplines [13]. Rather than building software from scratch, it's better to assemble it from existing artifacts: reusable programs and libraries, but also programming languages, frameworks, tools [18]. This approach also advocates developing more documentation artifacts and more focus on repeatability of tasks.

One view of the **relationship between computer science and software engineering** is that the former provides the theoretical basis for the latter. For example, Parnas notes that "using science and mathematics to build products for others is what engineers do" [30]. But since programmers think of themselves as scientists, the problems exhibited by typical software "are exactly those to be expected when products are built by people who are educated for other professions and believe that building things is not their *real job*" [30]. His point was that many people from other fields were working as programmers (or managers), but today having a computer science degree is not sufficient either, because many programs don't teach enough about software engineering.

Michael Jackson [20] observes that software engineering is **too broad a discipline** to be meaningful. He equates its scope to "physical engineering," an imaginary discipline that puts the concepts of physics into practice. As different engineering fields have corresponding branches of natural sciences, Jackson wants to see similar specialization in software engineering. He lists compiler construction, operating systems and GUI as examples of individual "engineering" disciplines that have branched out of software engineering, even though they are all based on computer science. One observation to add here is that some of these disciplines have developed a corresponding "science." Compilers have elaborate mathematical notations and systems for manipulating the symbolic representation of the data, but operating systems have none. Instead they follow a standard architecture with well-understood decomposition of the kernel into modules and well understood algorithms to implement these tasks.

Programming revolves around people. Chuck Connell notes that the difference between computer science and software engineering is that the former has lasting results, but the practices in the latter change constantly [12]. He points out that software engineering "directly involves human activity," which makes the results of tasks such as maintainability, safety, or requirements engineering dependent on the people who work on them. "Tried-and-true methods that work for one team of programmers do not work for other teams" [12]. This means that software development is a lot like a **social science**, because it deals with people, especially with controlling and managing people. For example, studies of agile practices such as pair programming by Laurie Williams [11] show importance of observing programmers at work. However, observation is only a part of social sciences; the other is experimentation. But it's hard to conduct large-scale programming experiments, and it's impossible to perform double-blind studies (procedures in which neither the subjects of the experiment nor the persons administering it know which aspects of the experiment are critical).

Some (non-trivial) portion of software development borders on **art**, i. e. work that produces aesthetically pleasing, yet useful, results. Donald Knuth considers programming "an art form in an aesthetic sense" [21]. He adds style and taste as important ingredients of good programs. Programming languages with small number of constructs, like Smalltalk and Python, appear to have this quality. But even "hard core" programming languages, like C++, have art. I like Qt, many people like Boost, two collections of libraries and language extensions that simplify many programming tasks in C++. What makes a language or a library esthetically pleasing? A small number of concepts that mesh well together in a uniform manner (in accordance with Brooks' conceptual integrity [6]), are simple to use, and extensible. Every software developer has his or her personal favorites that fit this category. Unfortunately, being an art form is not a prevalent state: when such software shows good potential, it is extended and over-extended until it reaches its limit. As its size explodes, the good qualities – simplicity, extensibility, uniformity – slowly erode. This seems to be the fate of all successful systems and languages.

All these views represent some truth about software development, because programming touches a wide variety of tasks. James Herbsleb points out that our discipline is an **interdisciplinary science** [19] that is more than computer science. He notes that "understanding the underlying mechanisms in play as people tackle software engineering tasks – mechanisms rooted in human cognition, social practices, and culture – is critical to the progress of our field." I haven't discussed cognition or culture, but these concepts also come into play when trying to define the proper scope of our discipline.

It seems that programmers are living in a world of metaphors. To describe our discipline, we explain how it relates to other, better defined disciplines and occupations, which are a little *like* programming: like a craft, like an art, a social science, an assembly line. The term "software engineering" is itself a metaphor; so is "computer science."

The sheer number of metaphors makes it obvious that our discipline has difficulty with defining its proper scope. No individual can be a toolsmith of *all* tools and Jack of *all* trades. A discipline without well-defined boundaries can't produce good results consistently. If we keep on trying to incorporate too many skills into the definition of our profession, we will continually find ourselves chasing what we know are unrealistic and unattainable goals.

(Sadly) Not Rocket Science

Every discipline is likely to face some unrealistic expectations, challenges that drive its progress. What is so special about software? The expectations from software appear to be more extreme. Having looked at them in detail, let me highlight what makes our discipline unique.

Programming is not as difficult as engineering. A lay person is likely to consider the established engineering disciplines to be difficult, because they require the application of mathematics and physics. Most software is not that hard to write. Scientists write their own programs using environments like Matlab; they feel they don't need dedicated programmers. The proliferation of Web authoring tools makes it easy for non-technical people to create their own Web pages. Since software development is easier than, say, rocket science, it's no surprise that people expect software professionals to deliver quality products.

In contrast, the products of engineering work are much easier to understand than software. Even though most people would not claim to know how to build a bridge, a plane, or a rocket, they would be able to recognize these products. If a rocket were missing a major component, like a propulsion system, a non-engineer would be able to point that out, even without knowing anything about propulsion. Not so with software – an average person cannot tell apart a completed product from a prototype that provides only 5% of the advertised functionality, but has a slick user interface. This lack of transparency makes it more difficult to manage expectations in software than in disciplines that create tangible products.

Ironically, not only the users, but also the programmers fail to recognize that expectations from software are different than from engineering. What we do is both easier *and* harder than typical engineering fields: the engineering part is often easier, while the non-engineering aspects are often harder than in other disciplines. However, programmers want to be like engineers and scientists. These are our dominant metaphors, even though our work is different. In attempts to show that we belong, we sometimes "over-engineer" our solutions by building in extra functionality for future changes that never materialize, by introducing optimizations prematurely, or by building and using bloated software platforms and processes. Some of our science is whimsical (e.g. formal methods define elaborate proof techniques for toy-size programs that fail to scale to 1KLOC – the amount of production-quality code that a competent programmer can produce in a month). All this effort goes to show that programming is a scientific discipline. This explains why alternative approaches, such as agile-style continuous development, still tend to be ignored, because they aren't recognizable as engineering. They are seen as inferior, because "nobody could build a rocket that way."

This interplay of ideals and realities of software development amplifies the impossible expectations from software. We would probably understand them better if we finally admitted that what we do is more than merely building (or "engineering") software.

3. Good Communication: A Difficult Goal

In his story of the Tower of Babel, Brooks discusses communication and organizational problems of software engineering. Effective communication is still one of our biggest challenges, but it's not an impossible goal; it's merely difficult. One reason that makes it difficult is that we don't focus on the essential problems of communication, but rather focus on tasks like documentation.

How Documentation Hinders Communication

It is sometimes suggested that software engineers can streamline communication by preparing more documentation artifacts. While this approach works well in other engineering fields that produce written documents in order to explore ideas and to refine the design, it doesn't work in software engineering. When a project plan calls for many documentation deliverables, developers get bogged down with writing documents (that add little value to the development process), rather than writing the software.

Large documentation manuals, produced to communicate the behavior of a software system to its users, suffer from many problems.

First, maintaining up-to-date external documentation separately from code is hard to enforce, because it violates the

"say it once and only once" rule [1]. The code defines the actual state of the system, and trying to duplicate it without tracing the documentation back to the code will cause problems when the code changes. Often documentation is obsolete by the time the software is implemented.

Naur goes farther and claims that documentation cannot convey the essential information necessary to understand a program, because software development is not about text production [27]. Documentation, even if produced to the best of our abilities, is likely to mislead a reader who is not familiar with the software. Naur claims that the only way to learn about the software is to work with other programmers who know it. While I disagree with Naur's Theory Building View, which he proposes as an alternative approach, his distrust in text as a primary medium of software development is well-reasoned and convincing.

Second, programmers have little motivation to keep documentation up to date – the programmer likely won't be around when the code needs to be changed, and there are always more pressing tasks to complete. Parnas suggests "imposing standards on structure and documentation, making sure that products that are produced using *short cuts* do not carry the industry *seal of quality*" [29]. Although this is a commendable stance in theory, quality documentation is not a required deliverable on most projects, because there is no good business case for it.

Third, there are no good examples of software documentation. Should one document software with textual descriptions? with pictures and UML diagrams? using formal notation? Brooks spends most of his Tower of Babel chapter discussing a documentation technique called workbook. It was a thick manual that included all the documentation produced for the project. While Brooks still advocates the use of workbook in the second edition of his book, in 1995 [6], this approach is too cumbersome for most projects. An automated process would be ideal. Mahoney suggests documenting software evolution using the moving picture metaphor [24], i.e. recording the process. While this idea has the advantage of obtaining the data for free while programmers are working, it faces the problem of editing the raw data to recover the important lessons.

Finally, it's not easy to determine which parts of the system are more important and should be documented first. And even if the order is known, it's impossible to write the documentation without repetition, once again violating the "say it once" rule. The only good documentation I'm familiar with comes with open source libraries and frameworks (like J2EE frameworks or Qt). These systems offer many forms of documentation: high-level text overview of main concepts, fully functional demos, cookbook code, discussion board, wiki. The effort expanded to produce and maintain these artifacts is justified by large user base. A typical application doesn't benefit from most of these artifacts, because they are costly (in time and effort) to produce.

Better Communication

Documentation is only one of many examples of misdirected focus in software development. It doesn't help that programmers tend to not be good writers. We are much better at problem solving and creative thinking.

In order to improve communication in software engineering, we ought to use these skills to our advantage. We need to learn how to better manage the expectations, both from internal and external stakeholders, to be more agile about what we do, and try to understand better which goals are feasible and which are impossible. To improve communication, we need to look for better ways to understand and explain what we do. Let's consider in some detail two examples that take advantage of programming skills – defining good abstractions and improving the readability of code.

Communicating through Abstractions

Abstraction identifies a general concept by effectively hiding details. The resulting view is easier to understand and communicate. For example, formal methods work by identifying and eliminating unnecessary details. Without these simplifications, the amount of details would make it impossible to scale any machine-verifiable proofs. But that is one of the reasons why formal methods are ineffective in practice, because eliminating details leads to loss of information and a *lossy* abstraction.

In contrast, good software abstraction is *loss-less*: it hides details without eliminating them. A loss-less abstraction has multiple levels and its users can select the level at which to view, use, or extend it. While continually raising the level of abstraction helps in identifying general concepts, it's effective only if it doesn't discard details. For example, modeling and visual assembly languages built on top of 4th generation programming languages [18] are only as good as their ability to handle modeling errors. Most non-trivial problems have special cases to be considered, so it must be possible to recover these details when necessary.

To illustrate a lossy abstraction, consider the programming model where distributed and local objects are treated uniformly. Jim Waldo and colleagues [33] analyze four fundamental concepts that break this model. First two concepts are latency of procedure calls and the use of pointers for memory access. Since the local latency is at least 5 orders of magnitude smaller than the distributed one and implementing distributed pointers requires much more overhead, accommodating these two concepts requires significant effort and will make local computing inefficient. The two other concepts, partial failure and concurrency, can't be implemented locally, because these don't exist in local computing. The only way to implement them is to make all computing distributed, which is obviously the wrong thing to do. In order to accommodate both programming models, programmers must be able to distinguish between distributed and local objects when necessary. The abstraction of unified

programming model will work well until there is a serious failure – then the errors caused by distribution must be handled as special cases.

Information hiding [28] is a good example of applying a loss-less abstraction. This technique identifies a system decomposition that enables the hiding of internal *changes*. Today, information hiding is often simplified to mean encapsulation and access control (with public, protected, and private visibility modifiers) in object-oriented languages. But there is more to this concept. In his paper, Parnas compares two decompositions of a program into modules – one where each step of the program's algorithm is a module, the other based on isolating design decisions. The second decomposition is more maintainable – it enables the hiding of design decisions (not just data) from other modules so that it's easier to change them independently. It also groups together code and decisions that are likely to change at the same time (in the spirit of the "say it once" rule).

Brooks criticized information hiding in 1975 [6]. He argued that it's impossible to completely and precisely define all interfaces between modules and that it's easier to find errors between interfaces if the internals are not hidden. In the second edition, he admitted that he was wrong. When information hiding is applied correctly, the clients only need to consider the interfaces, which greatly aids the understanding of a system. The hard part of implementing information hiding is figuring out how to divide responsibilities between modules. Very little research has been done on this topic (for one example see Lamb et al [23], who propose a theory of modular information hiding design methods). By better understanding how to divide responsibilities in the code, we can improve communication through the use of better abstractions.

Abstractions are not limited to programming concepts; they appear also in written documentation, e.g. in design patterns [16]. A pattern begins with a descriptive title and a one-sentence summary. An expert can recognize the idea by reading the opening sentence. A reader who doesn't have that experience is led to discover it in steps. A pattern description explains the context where the pattern applies, followed by the forces that make it a non-trivial solution, then the description of the solution – again, from overview to detail. A pattern description typically ends with a list of known uses that mention actual software that uses the pattern. This gives readers references to other documents that describe the concept captured in the pattern. That's the beauty of loss-less abstractions – provide an overview, then add more details to unveil the complexity behind the basic idea that represent the abstraction (in this case: the pattern).

Although it's not a black art, understanding abstractions can't be taught as a prescribed sequence of steps. The key to abstractions is good metaphors; they require creative thinking and continuous refinement. Abstractions are best learned by doing.

Communicating via Code

The way programmers communicate with computers and with one another through code, is also imperfect. My colleague Scott noted recently:

Each line of code could be either arbitrarily capricious, sloppy, or extremely carefully crafted. As a reader, you can't tell which it is, but it makes a lot of difference.

This is probably the saddest fact about the state of practice in software engineering. Readable code is the obvious goal to strive for in order to improve communication. The irony of the word *code* should not escape us; code is, as the name indicates, cryptic. It's easier to write code than to read it later. Many of the decisions made while writing code are not preserved in the code because they can't be expressed with programming languages. But why haven't programmers devised a good way of indicating which code is critical? Certainly not to protect our code from others. The code I write turns into a riddle for me if I don't use it for a few weeks.

Programming languages lack good mechanism for effectively tagging code with additional metainfo. Defining annotations for human readers is rarely used in practice. Similarly, code comments are seldom considered when code needs to be changed. If one were to add a comment to a line of code, stating "This line was implemented this way because...," the comment is likely to stay in the code forever, untouched. Other developers have no problems making changes the code, but nobody ever edits such comments, not even to mark them as "obsolete".

Literate programming [22] is sometimes suggested as a known solution for this problem. Knuth motivated it "Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do." He combined a programming language and documentation into a single system that would improve communication. Today, literate programming is a thing of the past. Perhaps users were put off by the idea of programs as works of literature. Or perhaps the need to combine two languages unnecessarily deterred programmers from focusing on writing software.

One modern technique that helps to improve communication via code, albeit indirectly, is writing unit tests: documenting code by writing more code. Automated unit testing coupled with continuous integration are key practices in agile development, which de-emphasizes the importance of documentation artifacts [3]. Test cases do not make it easier to read code, but they help to clarify which code is important (the important code has more tests). They help developers rephrase what the application says, using code. Tests are run continuously to check that both interpretations match. Test cases are effective, because they are code (so they can be executed, which makes them "real") and documentation at the same time.

It's easier to see source code as a good documentation medium if most of the program's code is accessible. The rise of open source software has been instrumental in giving application developers access to the code of frameworks and libraries. Simple techniques like extractable code comments (think Javadoc) and annotations make it easier to understand code.

The best and worst of communication between programmers occurs via code. Still, code comprehension remains a difficult task.

4. From Impossible to Realistic Goals

Finding better solutions for improved communication is an ambitious goal, but at least it is not an unrealistic one. Perhaps the realism kicks in by acknowledging the fact that there is no unique and perfect way to communicate; instead communication occurs through various artifacts, each playing its part. This subtlety distinguishes an impossible goal from an ambitious, but realistic one.

There is a big difference between impossible and ambitious goals. Building faster stock-trading software is a realistic goal, while building bug-free stock-trading software is impossible. It's important to be able to recognize which goals are unattainable, and to focus on achievable substitutes instead.

Over time, some impossible goals can become achievable. Two factors contribute to this: *technological improvements* and *rationalization of goals*. Technological improvements act as a game changer. With constant technological advances and better understanding of the challenges faced in software engineering, some impossible expectations of the past have become realistic goals. Online upgrades make it possible to update software while it's running, something that couldn't be done earlier. Modular design makes it possible to build increasingly bigger software systems. The communication protocols of the Web enable communication between heterogeneous systems. Multicore processors enable executing parallel programs on even the smallest computers. These and many other technological advances explore new ways of breaking down the incidental complexity [6] that can make a difference between facing impossible, or merely difficult, goals.

Another way to eliminate an impossible goal is to rationalize the expectation, i.e. to constrain the degrees of freedom of the original impossible goals. Typically, this comes from prior experience (first hand or from engineering history), or from better understanding of the problem itself. One of the biggest challenges of software development is dealing with changing requirements. Is trying to keep up with changing requirements an impossible goal? In a way, yes, because requirements changes will always precede changes in the software, so they force programmers to constantly play catch-up. A rational goal is to leverage improved technology to make it easy to play catch-up. At the process level, this could be done by following agile practices. At the code level, refactoring code and making it modular helps bringing in the changes when needed. With good change management and open communication, keeping up with changing requirements becomes a realistic goal.

But fast-paced software projects can't be halted to wait for technological improvements, and rationalization of goals sometimes might feel like a bad compromise. In those cases, setting realistic goals requires open communication to educate the stakeholders who expect too much. Comparison with competing systems show what capabilities are currently available. Techniques like Domain Driven Design [15] show how to grow a common language from experiences of various stakeholders. Realistic goals also come from thinking like designers [8], not merely software designers, but informed creators of systems.

5. The Tower of Babel Did Not Fail

Current research in archeology indicates that the story of the Tower of Babel was inspired by a real edifice, "a staged temple-tower [ziggurat] that the Babylonians knew by the Sumerian ceremonial name of E-temen-anki (House of the Foundation Platform of Heaven and Underworld)" [17]. Throughout the 2nd and 1st millennium BC, virtually every invader of Babylon destroyed the tower, and every successful ruler dreamt of rebuilding it. Several Babylonian kings started the rebuilding process and, although it's not certain how many succeeded, *the tower was completed* at least once, after almost a century of work, around 590 BC.

The final destruction of the tower came at the hands of the Persians. Alexander the Great or one of his successors in Babylonia tried to rebuild it again. Their short-lived effort began and ended with clearing out the debris; it's not clear why. A different, smaller building was erected in place of the tower later, but the extensive foundations of the tower have survived under ground until the modern times.

Although the biblical story says that the tower could only have been built by people communicating with one language, history indicates otherwise. Men speaking many languages worked there, because ancient Babylon was the multilingual center of the civilized world. Less civilized people came (attracted by its fame), conquered the land and destroyed the city, but then were slowly assimilated by the locals who taught them how to rebuild it.

The historical Tower of Babel *did not fail*. In contrast to the biblical story, the tower was completed successfully, and by people speaking many languages. The construction succeeded, because in place of an impossible goal of reaching the heavens, their actual goal was to build the largest tower possible. Today, even as a ruin, the tower teaches many important lessons. Humans are capable of achieving great things, and we enjoy the challenge of completing difficult tasks. We should always think big, but recognize and stay away from the trap of the impossible.

An Epilogue

McCullough's history of the Panama Canal [25] presents a modern study of facing impossibilities. What is impossible for one person or group, need not be impossible for another. Expectations, as well as time, can change the difficulty of a problem.

The first attempt to build the Panama Canal was by a French company, in 1880. The task was led by Ferdinand de Lesseps, who had built the Suez Canal earlier. That success has led to high expectations, even though the conditions in Panama were different. Lesseps underestimated the amount of dirt that needed to be moved, and ignored the effects of tropical diseases. This attempt was abandoned in 1889, after over 20,000 workers died, mostly from disease.

Soon after, from 1904 to 1914, the Americans succeeded at building the canal. They focused first on eliminating mosquitoes so that disease was less of a problem, then on building a good railroad to carry away the dirt (the French one was too small).

One of the advantages that helped the Americans had was that medicine had advanced during that time. People knew what caused malaria and yellow fever; earlier they did not. As a result, the project consumed "only" 5000 lives.

Another advantage was that they didn't try to build a sea-level canal. They used a lake as part of the canal, which meant that every ship has to be raised to the lake level, and then brought down, using locks. The French tried to dig the entire length of the canal (48 miles) by themselves, which would have been even more work.

The story of the French in Panama is a tragic tale of man trying to do the impossible.The American effort benefited from understanding the problems that plagued the French. Technological improvements and successful rationalization of goals (aided by better understanding of the prior impossibilities) have made it possible for Americans to achieve the same goal, only a quarter century later.

6. ACKNOWLEDGEMENTS

The authors would like to thank Baris Aktemur, Nicholas Chen, Maximiliano Davids, Ralph Johnson, Jason Kahn, Maurice Rabb, Linda Rising, Roger Whitney, Arturo Zambrano, the anonymous reviewers, and especially our shepherd, David West.

References

[1] —, *Once And Only Once*. Available at: `http://c2.com/cgi/wiki?OnceAndOnlyOnce`.

[2] —, *What is 80/20 rule?* Available at: `http://www.80-20presentationrule.com/whatisrule.html`.

[3] Beck, K. *Extreme Programming Explained: Embrace Change*, Addison Wesley, 2000.

[4] Berners-Lee, T. *The World Wide Web: A very short personal story.* Available at: `http://www.w3.org/People/Berners-Lee/ShortHistory.html`.

[5] Brooks, F. P. The Computer Scientist as a Toolsmith II, *Communications of the ACM, March 1996*.

[6] Brooks, F. P. *The Mythical Man-Month, Anniversary Edition*, Addison Wesley, 1995.

[7] Brooks, F. P. Three Great Challenges for Half-Century-Old Computer Science, *Journal of the ACM. January 2003*.

[8] Brooks, F. P. *The Design of Design: Essays from a Computer Scientist*, Addison Wesley, 2010.

[9] Bush, V. As We May Think. *The Atlantic Monthly*, July 1945.

[10] Buxton, J. N. and B. Randell, eds. *Software Engineering Techniques*, April 1970, p. 21. Report on a conference sponsored by the NATO Science Committee, Rome, Italy, 2731 October 1969.

[11] Cockburn A. and L. Williams. The Costs and Benefits of Pair Programming, *Proceedings of eXtreme Programming and Flexible Processes in Software Engineering, XP2000.*

[12] Connell, C. Software Engineering $\neq$ Computer Science, *Dr. Dobb's High Performance Computing*, June 04, 2009. Available at: `http://www.drdobbs.com/hpc-high-performance-computing/217701907`.

[13] Cusumano, M.A. *Factory Concepts and Practices in Software Development: An* [sic] *Historical Overview.* Working Paper #3095-89 BPS. Sloan School of Management, MIT, 1989.

[14] DeMarco, T. Software Engineering: An Idea Whose Time Has Come and Gone? *IEEE Software, July/August 2009*.

[15] Evans, E. *Domain Driven Design: Tackling Complexity in the Heart of Software*, Addison Wesley, 2004.

[16] Gamma, E., R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1994.

[17] George, A. R. The Tower of Babel: Archaeology, History and Cuneiform Texts *Archiv fűr Orientforschung*, 51 (2005/2006). pp. 75-95. Available at: `https://eprints.soas.ac.uk/3858/`.

[18] Greenfield, J., K. Short, S. Cook, and S. Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, 2004.

[19] Herbsleb, J. Beyond Computer Science, *Proceedings of the 2005 (27th) International Conference on Software Engineering*.

[20] Jackson, M. Problem Analysis Using Small Problem Frames, *South African Computer Journal 22; Special Issue on WOFACS'98, pp47-60, 1999*.

[21] Knuth, D. E. Computer programming as an art. *Communications of the ACM. Volume 17, Issue 12 (December 1974).*

[22] Knuth, D. E. Literate Programming. *The Computer Journal. Volume 27(2), pp 97–111, 1984*.

[23] Lamb, D. A. and K. A. Schneider. Formalization of information hiding design methods, *CASCON '92: Proceedings of the 1992 conference of the Centre for Advanced Studies on Collaborative research*.

[24] Mahoney, M. Software Evolution and the Moving Picture Metaphor, *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2009)*.

[25] McCullough, D. *The Path Between the Seas: The Creation of the Panama Canal, 1870-1914*. Simon & Schuster, First Edition, 1978.

[26] Mitchell, C. *Guest View: The half-agile path leads nowhere.* SD Times, March 15, 2010. Available at: `http://www.sdtimes .com/GUEST_VIEW_THE_HALF_AGILE_PATH_LEADS_NOWHERE/ By_CYNDI_MITCHELL/About_AGILE/34197`.

[27] Naur, P. *Programming as Theory Building*, North Holland Publishing Company, Micropocessing and Microprogramming 15 (1985) 253-261.

[28] Parnas, D. L. On the Criteria To Be Used in Decomposing Systems Into Modules, *Communications of the ACM, Dec. 1972, Vol. 15, No. 12*.

[29] Parnas, D. L. Software Aging, Invited Talk, *International Conference on Software Engineering, 1994*.

[30] Parnas, D. L. Software Engineering: An Unconsummated Marriage, *Communications of the ACM, Sept. 1997, Vol. 40, No. 9*.

[31] The Standish Group, *CHAOS Summary 2004*. Available at: `http://www.softwaremag.com/L.cfm?doc=newsletter /2004-01-15/Standish`

[32] The Standish Group, *CHAOS Summary 2009*. Available at: `http://www1.standishgroup.com/newsroom/chaos_ 2009.php`.

[33] Waldo, J., G. Wyant, A. Wollrath and S. Kendall. *A Note on Distributed Computing*, Sun Microsystems Technical Report SMLI TR-94-29, November 1994.