Computer Science: Creating in a Fallen World

Russ Tuck (Gordon College)

Russ Tuck (B.S., M.S., Ph.D., Duke University, with dissertation research at the University of North Carolina) has been a professor of computer science at Gordon College since a career change in 2015. Before that, he spent 20+ years in Silicon Valley as a Systems Architect, Software Engineer, Site Reliability Engineer, and Engineering Manager at four companies, most recently Google. He enjoys time with his wife and two grown children, travel, reading, and exercise.

1 Introduction

God, the creator of the universe and everything in it, created every person in his image. One of the many amazing gifts this implies is the gift of sub-creation: our ability to make things in and from God's creation. Just as God repeatedly declared the goodness of his creation, we can take joy in creating things with the abilities God has given us. However, as we exercise this gift, we need to be careful of both what and how we create. The responsibility to create for good, and not for evil, is obvious, although how to do that is sometimes not. In particular, it took decades to appreciate the full impact of our imperfect, sinful nature on how we create software.

The question of how best to exercise the gift of creating is particularly important for Christians involved in Computer Science, whether as teachers, learners, or practitioners, because our field offers so many tools and opportunities for creating things. This is such a rich, broad, and deep question that a single paper cannot possibly answer it. So the present paper aspires only to elucidate and highlight some parts of the answer that I find particularly interesting and compelling in my own career and in teaching my students.

This paper has three main sections. The first section, "Sub-Creation," explores the intersection of sub-creation and computer science, not in the narrow sense of creating virtual worlds, but in the broader sense of making things within God's created world. Our creations can have both obvious and more subtle effects, and all of these are part of creating. The second section, "Creating for Christ," considers what Christians should seek to create, and how we should decide. And the third section, "Creating in a Fallen World," considers how the creative process, and the resulting creations, reflect and respond to our sinful nature and fallen world. Agile software development is an important example, because it embraces and works with our imperfect knowledge and understanding, in contrast with the discredited waterfall (or rational) model which long strove to document complete and detailed requirements and design before most coding begins.

2 Sub-Creation

Sub-creation is the God-given gift of creating, or making things, within God's created universe. People have been making things since Adam and Eve sewed fig leaves together (Genesis 3:7) after their original sin, but Dorothy Sayers argues persuasively for an earlier origin:

> [W]hen we turn back to see what [the author of Genesis] says about the original upon which the "image" of God was modelled, we find only the single assertion, "God created". The characteristic common to God and man is apparently that: the desire and the ability to make things. [22]

J.R.R. Tolkien famously used the term "sub-creation" in a more limited sense to describe the creation of vivid and self-consistent imaginary, or virtual, worlds like "middle earth" in his Lord of the Rings trilogy [27]. I follow Fred Brooks [5], my dissertation advisor, in using Sayers' more general usage. In both cases, the "sub" part recognizes we are limited to creating within, and from the materials available in, the world God created us in. As the joke goes, when an engineer challenged God by saying he could create a man from dirt, God responded, "Not so fast. You get your own dirt." [28]

There are many motives and metrics for our creative acts. According to David Downing, Tolkien viewed "sub-creation as a form of worship, a way for creatures to express the divine image in them by becoming creators." [7] C.S. Lewis takes this a step further by seeking to simultaneously serve a higher purpose, as in the deep truths and Christian metaphors conveyed in The Chronicles of Narnia. [16] Downing notes a similar difference between composers J.S. Bach, who fulfilled his sense of Christian vocation simply by writing music, and Isaac Watts, who wrote hymns. [7] Brooks [5] emphasizes a different distinction, between natural sciences, which "take the discovery of facts and laws as a proper end in itself", and disciplines of design, including computer science, which more properly measure creations by their usefulness and cost.

Interestingly, Lewis, in the context of literature, also disagreed with the tendency to value originality and innovation for its own sake. Instead, "an author should never conceive himself as bringing into existence beauty or wisdom that did not exist before, but simply and solely some reflection of Eternal Beauty and Wisdom." [17] Downing quotes Lewis as concluding that "of every idea and of every method the Christian writer will ask not 'Is it mine?' but 'Is it good?' " [7] This question of "is it good, by God's metrics?" is indeed critical, and reflects our role as creators within God's creation.

Tools and Their Effects

Much of computer science is concerned not just with sub-creation, but more specifically with creating tools [5]: software and computer systems that serve people and solve problems. So it is important to understand the full impact of those tools. A tool's first kind of impact is what it helps the user do, and the second is the effect it has on the user. John Dyer illustrates this with a shovel: it helps the user dig holes wider, deeper, and faster. And if it is used enough, the user is likely to develop calluses and stronger muscles [8] (chapter 2). In fact, whole classes of tools, like exercise machines and language learning apps, are designed primarily for their effect on the user. These effects are clearly linked to the tool and its use.

The third kind of impact is more subtle, like a "nudge." The decisions people make are predictably influenced by the way the choices are presented, sometimes even to their own detriment. For example, people save more for retirement if they are enrolled by default (and even more if the amount saved is automatically increased). Richard Thaler described this in Nudge [26], and won the 2017 Nobel Prize in Economics "for his contributions to behavioral economics." [20] This effect in web pages and other computerized tools is so important that designers can reasonably be called "digital choice architects." The middle-option bias, the scarcity effect, the decoy effect, and the status quo

bias are just some of the mechanisms for influencing choices, and there is a well-documented process for designing and testing the nudge [23]. The goals, motivations, and decisions of myriad "digital choice architects" are clearly significant, particularly since users making choices are commonly unaware of these influences.

These first three kinds of impacts can be combined to create or change habits, and to cause addiction. Nir Eyal documents how companies like Google, Amazon, and the YouVersion Bible app encourage user habits around their products, and why various techniques work. He cautions against creating and knowingly sustaining addictive behavior, pointing out that software knows how it is being used and should at some reasonable point switch from encouraging use to discouraging it. And he proposes an ethical framework based on two questions: do the creators use their own product, and do they believe it can materially improve people's lives? [10]

The fourth kind of impact is yet more subtle, and is sometimes not even fully understood or predicted by a tool's creator. It is suggested by the old saying, "if the only tool you have is a hammer, everything looks like a nail," but it is really the tendencies and values built into the tool. By choosing what to make easy, by the sum of all the nudges, by the choice of feedback (since measurements are often influential), and even by omission, the tool creator helps shape the ecosystem in which the tool is used, including its users. Sometimes, the resulting impact is more apparent in aggregate, in a community, either because the effect on an individual might be small and obscured by individual differences, or because the effects are on interactions [8] (chapter 11).

Marshall McLuhan proposed in 1977 that all human creations, including language, ideas, tools, and clothing, influence humans and society in four ways. Each one enhances something, obsolesces something, retrieves something previously obsolesced, and "flips" into something when pushed to the extreme [19]. Dyer adapts and rephrases this tetrad into a Biblical context for evaluating technology. He proposes evaluating technology by asking four kinds of questions:

- Reflection - how does it reflect God's nature and help people obey God's commands?
- Rebellion - how could it help or tempt people to disobey or rebel against God? [1]
- Redemption - how does it help overcome effects of the fall?
- Restoration - what unintended problems does it bring, and how can they be avoided?

He provides a larger set of questions for each area [8] (Appendix: Technology Tetrad).

In short, software tools have obvious, subtle, and sometimes hidden and even unexpected impacts. All of these effects must be considered in evaluating our work.

3 Creating for Christ

Let us consider what guidance the Bible gives about what tools to build. While "all Scripture is God-breathed and is useful for teaching, rebuking, correcting and training in righteousness" (2 Tim 3:16, NIV), there are a few passages that are particularly relevant for guiding sub-creation in computer science.

[1] Jesus particularly cautions against this in Luke 17:1, Matt 18:6-7, and Mark 9:42.

Start with what Jesus said were the most important commandments: "'Hear, O Israel: The Lord our God, the Lord is one. Love the Lord your God with all your heart and with all your soul and with all your mind and with all your strength.' The second is this: 'Love your neighbor as yourself.' There is no commandment greater than these." (Mark 12:29-31, NIV) If we are loving God with all our being, we certainly will not want to create anything displeasing to him. Rather, we will seek both to live lives pleasing to God, and by our tool building to help others do the same. Paul and Timothy's advice, "Finally, brothers and sisters, whatever is true, whatever is noble, whatever is right, whatever is pure, whatever is lovely, whatever is admirable–if anything is excellent or praiseworthy–think about such things," (Philippians 4:8, NIV) applies both to ourselves and our tools, since they often influence our users' thoughts and attention.

Jesus' second command, "love your neighbor as yourself," particularly as he restated it in Matthew 7:12, "do to others what you would have them do to you," (NIV) provides very practical guidance, including a Biblical foundation for Eyal's two questions (do the creators use their own product, and do they believe it can materially improve people's lives?).[2]

It is notable that God gave work to Adam before sin entered the world: "The Lord God took the man and put him in the Garden of Eden to work it and take care of it." (Genesis 2:15, NIV) It was a blessing to have purposeful, productive activity. It is only after Adam and Eve sinned that God made their work painful and exhausting. (Genesis 3:16-19) So it is reasonable to interpret the joy and satisfaction of creating software as part of God's gift of work, and the frustrations of bugs and other failures as part of the curse that came from sin.

Much later, Paul and Timothy were inspired to write, "whatever you do, work at it with all your heart, as working for the Lord, not for human masters, since you know that you will receive an inheritance from the Lord as a reward. It is the Lord Christ you are serving." (Colossians 3:23-24, NIV) This has long guided believers in our attitude toward work. But it can also provide more specific guidance about creating for Christ. Consider three examples: influencing speech ("The Tongue"), being stewards of users' time ("Stewardship"), and seeking to do avoid and correct bias ("Justice").

3.1 The Tongue

One potential example is the challenging and important area of tools for communication, such as social media. James 3 has very strong warnings about the dangers of the tongue. While these certainly apply to what users say through these tools, tool builders should consider how to help users speak responsibly, or at least not harm them by encouraging the opposite. Since conflict and outrage can be strong drivers of social media "engagement" (short-term usage), simple usage-driven metrics can easily lead developers to make changes that encourage harmful speech, rather than helping forestall it. In fact, there may be a strong financial incentive, at least in the short to medium term, to encourage harmful speech.

But a Christian software engineer or product manager should consider how to help avoid, rather than encourage, harmful speech, in light of Jesus's warning: "Things that cause people to stumble are bound to come, but woe to anyone through whom they come. It would be better for them to be thrown into the sea with a millstone tied around their neck than to cause one of these little ones to stumble." (Luke 17:1-2, NIV) The ideal would be to help users think about "whatever is true,

[2]Though there's no mention in Eyal's work that he was influenced by scripture.

whatever is noble, whatever is right, whatever is pure, whatever is lovely, whatever is admirable" (from Philippians 4:8, NIV). It is not always clear how to provide nudges in this direction. But it is a worthy area for thought and investigation.

3.2 Stewardship

Stewardship is another important Biblical mandate, with multiple implications for computer scientists. Stewardship stems from the fact that "The earth is the Lord's, and everything in it," (Psalm 24:1) and "The Lord God took the man and put him in the Garden of Eden to work it and take care of it." (Genesis 2:15, NIV) In Matthew 25, Jesus defines good stewardship in the parable of the talents (bags of gold), where the master praised servants who invested the money well and punished those who ignored that responsibility. In short, everything we have, including the ability to create software, is really God's. So we must use these gifts to please God – a broad mandate requiring prayer and careful thought to apply in each situation.

An important consequence, and another application of stewardship, is that Christians have a responsibility to write software that helps the user be a good steward of their time. Because mobile devices are becoming ubiquitous in our lives, and are entrusted by their users with considerable time and attention, they have the potential to powerfully shape our habits. There is a strong potential for conflict between the system creators' interests, since they make more money when their systems are used more, and the users' interests. Encouraging, or even just enabling, unhealthy or addictive behavior is clearly bad stewardship of the user's time and attention. It also fails the Luke 17 standard of not causing someone to stumble. In this light, as stewards of God-given software abilities and remembering that Jesus commanded both "love your neighbor as yourself" (Mark 12:31, NIV) and "in everything, do to others what you would have them do to you," (Matt. 7:12, NIV) it is clear that stewardship of users' time should take priority. The opposite, choosing selfish monetary gain at the expense of others' well being, is greed.[3]

Interestingly, at least one large, profitable, secular company believes this is also good business in the long term. Google's "Ten Things We Know to be True" begins with "1. Focus on the user and all else will follow," and goes on to express the unselfish corollary, "we take great care to ensure that [our products] will ultimately serve you, rather than our own internal goal or bottom line" [13]. Nir Ayal agrees, and explains why, summarizing "With very few exceptions, when a product harms people, they use it less or look for alternatives." [9] [4]

3.3 Justice

Righteousness and justice are fundamental to God's character, and this is reflected throughout the Bible. Looking back on the whole Old Testament, Micah 6:8 summarizes "He has shown you, O mortal, what is good. And what does the Lord require of you? To act justly and to love mercy and to walk humbly with your God." (NIV) Currently, machine learning is one of the most important, interesting, and challenging areas of Computer Science in which to apply this

[3] Mark 7:20-23 makes it clear that greed is evil.

[4] Google may be an example of this in another way. One could easily argue that selling porn ads is profiting by helping to harm its users, yet for years Google considered porn "not evil" and sold ads for porn searches. But it stopped in June 2014, in "an effort to continually improve users' experiences." [12]

admonition. Machine learning systems process large amounts of past "training" data in order to discover mathematical "rules" that enable software to imitate past decisions on new data (though it is commonly described as predicting classifications). A significant danger of machine learning is that it can observe discriminatory behavior and learn to do it automatically. It would be a horrible injustice to automate decisions and actions that unfairly harm people or discriminate against them, and to make these actions even more prevalent and hard to prevent. It could camouflage evil with a veneer of impartiality, a quality which people often attribute to computer systems. In addition, it is often impossible to understand the reasons for a machine learning system's actions, making it harder to root out and fix such problems. Christians must work to create systems that help us do justice, and never the opposite.

While I am not a machine learning expert, I wonder if a system can be trained to help find examples of discrimination in training data, allowing a more just system to be trained on data with fewer unjust examples to learn from. While justice is getting increasing attention in the design and use of machine learning systems, both from observers [21] and researchers [2], much more work is still needed.

4 Creating in a Fallen World

Although computer scientists exercise the gift of sub-creation, we do so as imperfect people in a fallen world. This affects not only what we build but, more fundamentally, how we build it. Four aspects of this problem will be explored, in successive subsections.

- Bugs, Tools, and Testing: We are inherently imperfect and mistake-prone, which means our software inevitably contains bugs (mistakes). So we have to test software to find the bugs, then figure out how to fix them.
- Agile Development: More fundamentally and less obviously, our knowledge and understanding are imperfect, so we do not even know exactly what to build. Agile development addresses this by making iterative discovery and refinement of requirements a central part of the development process.
- Diversity: Human bias and discrimination too easily corrupt our process and products, making it important and necessary to increase the diversity of teams and groups throughout our field.
- Robustness: Because we build in an imperfect world where devices fail and bad things happen, we design systems with redundancy and fault-tolerance, so they will (mostly) keep working anyway.

4.1 Bugs, Tools, and Testing

One of the first experiences of anyone learning to write software is that their code sometimes does not work. It does something wrong and unexpected, and it takes careful and often frustrating and time-consuming "debugging" to find and fix a mistake (or multiple mistakes) that caused the problem. Sadly, this is not just the experience of beginners. Every programmer, no matter how experienced, makes mistakes and introduces errors ("bugs") into their programs. It is part of the

human condition: we are imperfect, fallible, mistake-prone. It is a practical example of both the curse of Genesis and the spiritual struggle with sin that Paul describes in Romans:

> [15]I do not understand what I do. For what I want to do I do not do, but what I hate I do. [16]And if I do what I do not want to do, I agree that the law is good. [17]As it is, it is no longer I myself who do it, but it is sin living in me. [18]For I know that good itself does not dwell in me, that is, in my sinful nature. For I have the desire to do what is good, but I cannot carry it out. [19] For I do not do the good I want to do, but the evil I do not want to do -—- this I keep on doing. (Romans 7:15-19, NIV).

Just as we are unable to live perfectly, we are unable to write software perfectly.

Because we inevitably make mistakes writing software, new programming languages are periodically designed to help prevent, or at least detect, more and more types of errors. The fact that, over time, these languages become widely used, is an indication of the seriousness of the problem, because changing languages involves a major cost in rewriting or replacing code written in the previous language.

It is also necessary to test software to know if it works as expected. At first, testing was done manually, often by a dedicated group of test engineers. But this can easily become very slow and expensive, because a large software system may need extensive retesting even if only a small fraction of its code has been changed. The work of testing is roughly proportional to the size of the program times the number of releases. This prevents, or at least discourages, frequent releases (which have benefits discussed further on), and the resulting long times between tests makes it harder to fix problems. There are several reasons for this:

- Programmers have had more time to forget details of what they were trying to do;
- More bugs accumulate before testing, and they sometimes interact in ways that make them harder to detect and harder to fix;
- This approach tends to focus on system-level tests, which are often many layers removed from the errors that need fixing.

As a result, many developers and software organizations invest the time to write additional "unit test" code to test what they just wrote. This requires extra work, and extra debugging since the tests can have bugs. But because the tests are programs, they can be re-run quickly and easily (with a testing framework, also developed out of necessity). This makes it practical to retest a whole program regularly as new features are added. In fact, it makes the human test work much more nearly proportional to the work of creating the program.

In summary, the inevitability of human mistakes has led computer scientists to develop tools, techniques, and practices to prevent, detect, and correct bugs (mistakes in writing software). This is an important point to make clear to beginning programmers, who are often surprised and discouraged by their mistakes, and incorrectly assume that there is something wrong with them.

4.2 Agile Development

While bugs and the need for testing are a reality at every scale of software development, another kind of problem becomes apparent in the effort to build large systems: we do not know what to build. This is an illustration of Paul's statement, "For now we see only a reflection as in a mirror; then we shall see face to face. Now I know in part; then I shall know fully, even as I am fully known." (1 Corinthians 13:12, NIV) His point, that our understanding and knowledge this side of heaven are inevitably flawed and incomplete, is profound. And it has many more direct applications than to software development methods. And yet, as God's word frequently does, it explains human nature and difficulties in many contexts, including this one.

Still, the implications for software development have been slowly discovered at great cost. So let us start with some background. When writing a very small program, it is natural just to think about what to do, quickly write the code, and then test and correct it. And it is natural to try to follow the same general process for a larger project, recognizing that there is more work at each step. This pattern of doing each phase in sequence is called the "rational model" or, more vividly, the "waterfall model" (because it is hard to go back upstream past a waterfall). Many large projects following the waterfall model got bogged down and failed. Some failed completely, but most simply took much more time, work, and money than expected. Some of the problems with large projects were attributable to poor planning and poor communication. So more formal processes were put in place to carefully and precisely document the needs, thoroughly work out and document the design of the desired system, and communicate details and changes between the people working to implement and test it. This helped some, but major problems persisted.

Various models were proposed and attempted as improvements on the waterfall model, and gradually an iterative model called "agile development" has been widely adopted as a best practice [4]. While agile development is practiced in a variety of ways, its core is incremental development with frequent input from the users (or "customer"). The team focuses on getting a useful prototype working as soon as possible, and then refines it in short (often 2-week) iterations. In each iteration, some improvements are made and immediate feedback is obtained from users. This process helps refine both the developers' and the users' understanding of the requirements and possibilities [18].

When developing something new, it is often impossible to anticipate exactly how it will be used, and how that will impact related (often human) processes and systems. In other words, we often do not know quite what to build, or how to build it, until we have built and tried it. The iterative process at the heart of agile development seeks to learn quickly and immediately apply that learning to continued development. This is an effective solution for our inherently limited knowledge.

Agile's iterative development works particularly well with unit testing [1]. By writing sets of small, simple tests for each new piece of code, as it is written, the process of testing is included in each iterative step. Those tests also help ensure existing code is not broken in subsequent iterations.

Other engineering disciplines also deal with uncertainty by building and testing prototypes. The main difference is that software is much more flexible than physical building materials, so an early prototype can often be more or less continuously refined into a useful production system. Unlike most physical creations, a good software system is rarely "final": if it is used successfully, opportunities for further improvements are almost always found, and these are often implemented in further iterations as priorities and resources allow.

The point of this discussion of agile programming methodology is that it enables software development organizations to cope with not only the imperfection of humans which leads to bugs, but also our inherently limited knowledge of our users and how to help solve their problems. Since we do not know exactly what to build, it helps to build iteratively so we can test as early and frequently as possible how our creation fits (and does not fit) the need, and refine our plans appropriately.

While God could use the "waterfall" method to speak the world into creation according to his perfect plan, humans do not have the perfect knowledge required for a perfect plan. Since we are doomed to trial and error, it works best to embrace that and do it as efficiently as possible.

4.3 Diversity

Throughout history, human societies have shown a strong tendency to mistreat and discriminate against "others" unlike the locally dominant group. This seems to be part of our sinful nature, since scripture teaches:

- "The foreigner residing among you must be treated as your native-born. Love them as yourself, for you were foreigners in Egypt." (Leviticus 19:33, NIV)
- "Cursed is anyone who withholds justice from the foreigner, the fatherless or the widow." (Deuteronomy 27:19, NIV)
- "For we were all baptized by one Spirit so as to form one body – whether Jews or Gentiles, slave or free – and we were all given the one Spirit to drink." (1 Corinthians 12:13, NIV)
- "Therefore go and make disciples of all nations, baptizing them in the name of the Father and of the Son and of the Holy Spirit, ..." (Matthew 28:19, NIV)

The United States has a particularly bad history of racial discrimination [29], which contributes to and exacerbates the problem within computer science. Even when we disavow discrimination, we find that implicit bias remains and causes harm [15]. Past discrimination, implicit bias, and other factors combine to create systemic discrimination, which is very difficult to overcome [14] [25].

This is very serious, but what does it have to do with computer science? Computer science is a predominantly white male field. Women and every ethnic group other than white and Asian men are under-represented. Since computer science jobs are generally very well paid and have desirable working conditions, it is unfair that others are left out. This can be a self-reinforcing problem, since members of those groups who try computer science may easily be dissuaded by the discomfort of being such a small minority (and the implicit bias that often happens in this situation). In addition, products designed primarily by one group are often biased toward users in that group and do not work as well for others [30]. Finally, software is often a part of the discriminatory system. Sometimes this is because it was designed by those in the majority and contains their biases (whether explicit or implicit). It can also result from machine learning, which is prone to learning and repeating biases present in the past data it is learning from [21]. All of these effects combine to multiply the difficulties for people who are members of more than one disadvantaged group, like black women [6].

In short, the predominance of white males in computer science, combined with human bias and discrimination, causes a wide variety of serious problems. It is therefore important for Christians

in computer science to work to diversify the field and mitigate the damage. We should work hard to recruit, welcome, support, and encourage women and members of underrepresented minorities into the field.[5] In the meantime, we should also strive to reduce and eliminate bias in the systems we create. Both of these are very important, difficult challenges.

4.4 Robustness

Human nature is not the only source of problems in our post-fall world. Physical things also fail. A lot of work and creativity are required to create a reliable system. I experienced this extensively as the first manager of Gmail's Site Reliability Engineering (SRE) team. Every level of the system had to be designed to work despite hardware, software, and human failures. At the infrastructure level, this included the Google File System (GFS), which accommodated both disk and computer failures, the job scheduling system, which would restart servers (programs) on different computers when a computer running them (or its network connection) failed, and the network, which routed packets around failures in the datacenter network as well as the links between datacenters. It also included code we wrote to take broken machines out of service for repair, notify technicians (who used their own software to schedule the repair work efficiently) and put computers back in service after they were fixed.

At a higher level, the Gmail server software and Google's authentication servers were written to tolerate hardware failure without losing data and with as little disruption to service as possible. When a server failed for any reason, another server would quickly take over its work. The Gmail storage servers were written with special care not to lose user data, even if a bug was introduced. All new data, both email received and sent and other actions like "archive" and "mark spam", was immediately logged in multiple places. Then, if either software bugs or hardware failure caused corruption or loss, the database could be reconstructed from a copy of the log.

In order to know if the service was working, and to be able to investigate and fix problems, we had monitoring systems observing and recording behavior of the system at multiple levels. We wrote rules so the system could alert us immediately to problems that needed human attention, and had an on-call rotation schedule so there was always someone to respond to these alerts quickly (and someone else to help them if needed). We generated graphs of various errors, and studied them regularly to find problems, prioritize them by user impact and risk, and consider how to improve the system to eliminate or reduce them.

When there was a major user-visible outage, the engineer most familiar with it wrote a post-mortem describing in precise detail the sequence of events leading up to the problem, how it was discovered, how it was fixed, and the impact on users. This was followed by the most important part, a prioritized list of changes needed to keep it from happening again. Since some outages were caused or exacerbated or extended by human error, it was critical for everyone on the team, as well as higher management, to understand that the post-mortem's purpose was to prevent future problems, not to assign blame. It was equally important to recognize that human errors are inevitable, so a conclusion that we should "be more careful" or add an "are you sure?" prompt was generally useless. Instead, the system needed to be changed so it was not so easy to make mistakes that

[5]The Grace Hopper Celebration of Women in Technology and the ACM Richard Tapia Celebration of Diversity in Computing seek to encourage and support people in these groups. In addition, the sponsors, AnitaB.org and the Center for Minorities and People with Disabilities in IT (www.cmd-it.org), respectively, seek to identify and publicize best practices for recruiting, welcoming, and helping them thrive.

caused an outage. Often, this meant working to automate a task (by writing a program to do it). Since this took time, we often did it in stages, first automating a safety check or a few steps in the process, until the whole thing was done. When that was not practical, another solution was to redesign part of the system so a mistake did not have such bad consequences. This helped the system tolerate more types of human errors, in addition to various types of hardware and software failure.

This extended discussion of Gmail and its SRE team is just one example of how systems are routinely created and operated to work reliably despite the myriad failures endemic to our fallen world. Google SREs later wrote a whole book about the principles and practices involved [3].

5 Conclusions

There is great joy and frustration in creating and improving computerized systems, reflecting God's gift of sub-creation and the reality of failure in our sin-stained world. This gift, like life itself, comes with God-given instructions and responsibilities. For example, our systems should help our users tame their tongues, not entice them to harmful speech, and our systems should be good stewards of their users' time, never exploiting them for greedy or self-serving ends. Justice should be a prime focus, particularly when using machine learning.

Understanding our sinful nature and fallen world leads to a richer and deeper understanding of many important software development techniques. Debugging, testing, agile development, and redundancy are all productive and appropriate responses to our human nature and fallen world. Instead of asking "why does this work?", we can wonder "why did it take so long to figure this out?" and "what else should we try?" Similarly, this understanding helps explain why we struggle with bias and need a strong focus on increasing diversity in computer science.

Finally, as a personal extension of the theme of stewardship, I am keenly (yet, sadly, only intermittently) aware that nothing I have is really mine. My abilities, my knowledge, my possessions, my time, and my family are all gifts from God. So my true call is to be a faithful steward.

References

[1] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, second edition, 2004. The first edition was published in 1999.

[2] Rachel K.E. Bellamy, Kuntal Dey, Michael Hind, Samuel C. Hoffman, Stephanie Houde, Kalapriya Kannan, Pranay Lohia, Sameep Mehta, Aleksandra Mojsilovic, Seema Nagar, Karthikeyan Natesan Ramamurthy, John Richards, Diptikalyan Saha, Prasanna Sattigeri, Moninder Singh, Kush R. Varshney, and Yunfeng Zhang. Think your artificial intelligence software is fair? think again. *IEEE Software*, 36:76–80, July-Aug 2019.

[3] Betsy Beyer, Chris Jones, Jennifer Petoff, and Nial Richard Murphy, editors. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, 2016.

[4] Frederick P. Brooks, Jr. *The Mythical Man-Month: Essays on Software Engineering*, pages 254–273. Addison-Wesley, anniversary edition edition, 1995.

[5] Frederick P. Brooks, Jr. The computer scientist as toolsmith II. *Communications of the ACM*, 39:61–68, March 1996. http://www.cs.unc.edu/~brooks/Toolsmith-CACM.pdf. This wise and perceptive paper, summarizing decades of thought by a fervent Christian and respected Computer Scientist, has stood the test of time. It's worth reading now, before finishing the present paper.

[6] Kimberly Crenshaw. Demarginalizing the intersection of race and sex: a black feminist critique of antidiscrimination doctrine, feminist theory and antiracist politics. *University of Chicago Legal Forum*, 1, 1989. Article 8, https://archive.org/details/DemarginalizingTheIntersectionOfRaceAndSexABlackFeminis.

[7] David C. Downing. Sub-creation or smuggled theology: Tolkien contra Lewis on christian fantasy, undated. www.cslewisinstitute.org/node/1207.

[8] John Dyer. *From the Garden to the City: The Redeeming and Corrupting Power of Technology*. Kregel Publications, 2011.

[9] Nir Eyal. The real reason Apple and Google want you to use your phone less, June 2018. https://www.nirandfar.com/2018/06/google-apple-less-phone-use.html#more-3348.

[10] Nir Eyal and Ryan Hoover. *Hooked: How to Build Habit-Forming Products*. Portfolio Penguin, 2014. Eyal's work was influenced by Stanford Pofessor BJ Fogg; see [24]. BJ Fogg summarizes his theory of behavior at http://www.behaviormodel.org/index.html.

[11] Verilyn Flieger. On fairy-stories, undated. www.tolkienestate.com/en/writing/translations-essays/on-fairy-stories.html.

[12] Google Inc. Adult content. In Advertising Policies Change log, https://support.google.com/adspolicy/answer/4271759?hl=en&ref_topic=29265.

[13] Google Inc. Ten things we know to be true, 2004. https://www.google.com/about/philosophy.html, accessed Aug. 6, 2018. (Google has certainly been criticized for not always living up to this lofty goal.) An earlier version, from 2004, https://web.archive.org/web/20040603020634/http://www.google.com/corporate/tenthings.html, says "by always placing the interests of the user first, Google has built the most loyal audience on the web." A version in similar form existed before Sept. 2002.

[14] Daniel Hill. *White Awake: An honest look at what it means to be white*. Intervarsity Press, 2017.

[15] Kirwin Institute for the Study of Race and Ethnicity, The Ohio State University. Understanding implicit bias, undated. http://kirwaninstitute.osu.edu/research/understanding-implicit-bias/.

[16] C.S. Lewis. Sometimes fairy stories may say best what's to be said. *New York Times*, November 1956. www.nytimes.com/1956/11/18/archives/sometimes-fairy-stories-may-say-best-whats-to-be-said, also found at apilgriminnarnia.com/2014/01/27/sometimes-fairy-stories/.

[17] C.S. Lewis. *Christian Reflections*, chapter Christianity and Literature, page 7. Wm. B. Eerdmans Publishing Co., 1967.

[18] Robert C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, Pearson Education, 2003.

[19] Marshall McLuhan. Laws of the media. *ETC: A Review of General Semantics*, 34(2), June 1977. https://www.jstor.org/stable/42575246.

[20] Nobel Media. The prize in economic sciences 2017, July 2018. www.nobelprize.org/nobel_prizes/economic-sciences/laureates/2017/.

[21] Cathy O'Neil. *Weapons of Math Destruction*. Crown Publishing Group, Penguin Random House, 2016.

[22] Dorothy L. Sayers. *The Mind of the Maker*. Harcourt, Brace, 1941.

[23] Christoph Schneider, Markus Weinmann, and Jan Vom Brocke. Online user choices through interface design. *Communications of the ACM*, pages 67 – 73, July 2018.

[24] Simone Stolzoff. The formula for phone addiction might double as a cure. *Wired*, February 2018. https://www.wired.com/story/phone-addiction-formula/.

[25] Beverly Daniel Tatum. *Why Are All the Black Kids Sitting Together In the Cafeteria: And other conversations about race*. Basic Books, Hachette Book Group, 1997, 2017.

[26] Richard Thaler and Cass Sunstein. *Nudge: Improving Decisions about Health, Wealth, and Happiness*. Yale University Press, 2008.

[27] J.R.R. Tokien. On fairy stories, 1947. brainstorm-services.com/wcu-2004/fairystories-tolkien.pdf. According to [11], this paper started as a lecture in 1939, was published in *Essays Presented to Charles Williams* (1947), was included in *The Tolkien Reader* (1968), and was eventually published stand-alone (2008).

[28] Reddit.com user 'Jonzell'. Get your own dirt. https://www.reddit.com/r/Jokes/comments/b0faut/get_your_own_dirt/.

[29] Jim Wallis. *America's Original Sin: Racism, White Privilege, and the Bridge to a New America*. Brazos Press, 2016. I found this book to be particularly enlightening and convicting.

[30] Gayna Williams. Are you sure your software is gender-neutral? *Interactions*, page 36ff, January-February 2014. http://interactions.acm.org/archive/view/january-february-2014/are-you-sure-your-software-is-gender-neutral.