13 Afterword

Alan Kay (1940–) received the 2003 Turing Award for pioneering many of the ideas at the root of contemporary object-oriented programming languages. He led the team that developed the Smalltalk language, from which Ruby inherits its approach to object orientation. He also invented the "Dynabook" concept, the precursor of today's laptops and tablet computers, which he conceived as an educational platform for teaching programming.

The best way to predict the future is to invent it.

—Alan Kay

Prerequisites and Concepts

Concepts:

- Agile is not the right fit for every project, nor is SaaS the right architecture for every project. But because of the "virtuous triangle" of SaaS, Agile, and Cloud Computing, this particular combination of ingredients has benefited all three areas, revolutionizing the future of software and making software development easier to learn.
- In this book you've used a successful distributed architecture (SaaS) and frameworks (Rails and jQuery). As an advanced software engineer, you'll likely need to create such frameworks or extend existing ones. Paying careful attention to principles that made these frameworks successful, and the lessons from the past that informed their design, can help.
- Some of these lessons can be found in the wisdom captured in books about the software world. As George Santayana said, "Those who do not know history are condemned to repeat it." We provide some suggestions of "classics in the field" that we believe all software engineers would benefit from reading.

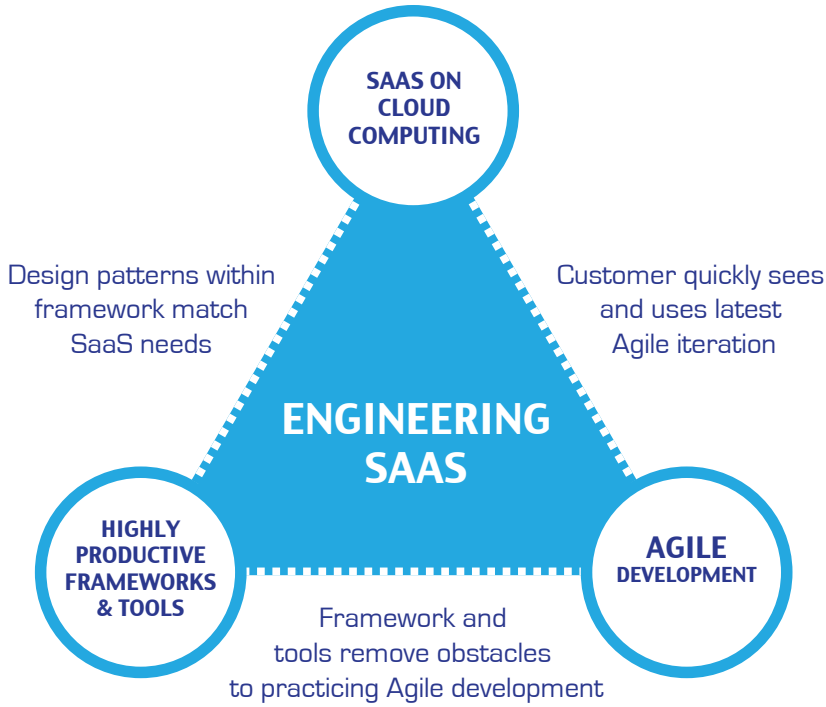

Figure 13.1: The Virtuous Triangle of Engineering SaaS is formed from the three software engineering crown jewels of (1) SaaS on Cloud Computing, (2) Highly Productive Framework and Tools, and (3) Agile Development.

13.1 Looking Backwards

Figure 13.1, first seen in Chapter 1, shows the three "crown jewels" on which the material in this book is based. Each pair of "jewels" forms synergistic bonds that support each other, as Figure 13.1 shows. In particular, the tools and related services of Rails makes it much easier to follow the Agile lifecycle. Figure 13.2 shows our oft-repeated Agile iteration, but this time it is decorated with the tools and services that we use in this book. These 14 tools and services support *both* following the Agile lifecycle *and* developing SaaS apps. Similarly, Figure 13.3 summarizes the relationship between phases of Plan-and-Document lifecycles and their Agile equivalents, showing how the techniques described in detail in this book play similar roles to those in earlier software process models.

Rails is very powerful but has evolved tremendously since version 1.0, which was originally *extracted* from a specific application. Indeed, the Web itself evolved from specific details to more general architectural patterns:

- From static documents in 1990 to dynamic content by 1995;
- From opaque URIs in the early 1990s to REST by the early 2000s;
- From session "hacks" (fat URIs, hidden fields, and so on) in the early 1990s to cookies and real sessions by the mid 1990s; and

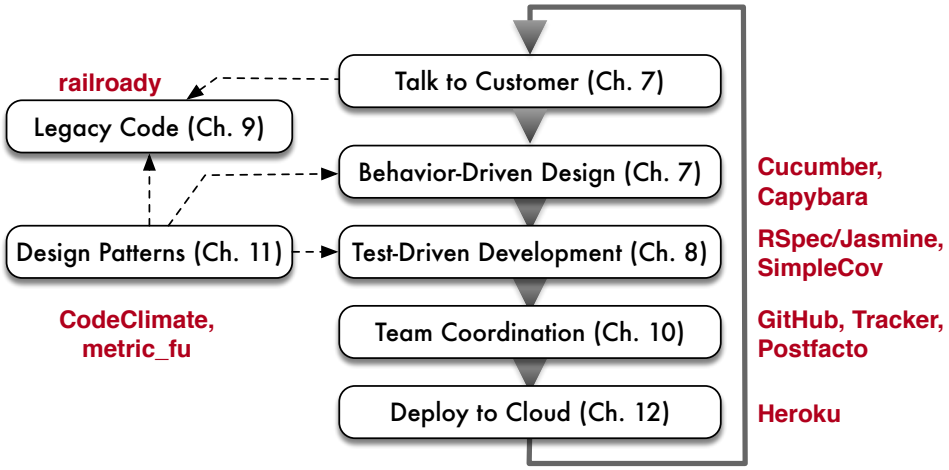

Figure 13.2: An iteration of the Agile software lifecycle and its relationship to the chapters in this book, with the supporting tools and services (red/bold letters) identified with each step.

Waterfall/Spiral	Agile	Chapter
Requirements gathering and analysis	BDD with short iterations so customer participates in design	7
Periodic code reviews	Pair programming (pairs constantly reviewing each others' code)	10
Periodic design reviews	Pull requests drive discussions about design changes	10
Test entire design after building	TDD to test continuously as you design	8
Post-implementation Integration Testing	Continuous integration testing	12
Infrequent major releases	Continuous deployment	12

Figure 13.3: While Agile methods aren't appropriate for all projects, the Agile lifecycle does embrace the same process steps as traditional models such as Waterfall and Spiral, but reduces each step in scope to a single iteration so that they can be done repeatedly and frequently, constantly refining a working version of the product.

- From setting up and administering your own ad hoc servers in 1990 to deployment on "curated" cloud platforms in the 2000s.

The programming languages Java and Ruby offer another demonstration that good incremental ideas can be embraced quickly but great radical ideas take time before they are accepted. Java and Ruby are the same age, both appearing in 1995. Within a few years Java became one of the most popular programming languages, while Ruby remained primarily of interest to the programming languages literati. Ruby's popularity came a decade later with the release of Rails. Ruby and Rails demonstrate that big ideas in programming languages really can deliver productivity through extensive software reuse. Comparing Java and its frameworks to Ruby and Rails, (Stella et al. 2008) and (Ji and Sedano 2011) found factors of 3 to 5 reductions in number of lines of code, which is one indication of productivity.

13.2 Looking Forwards

I've always been more interested in the future than in the past.

—Grace Murray Hopper

Given this history of rapidly-evolving tools, patterns, and development methodologies, what might software engineers look forward to in the next few years?

One software engineering technique that we expect to become popular in the next few years is *delta debugging* (Zeller 2002). It uses divide-and-conquer to automatically find the smallest input change that will cause a bug to appear. Debuggers usually use program analysis to detect flaws in the code itself. In contrast, delta debugging identifies changes to the program *state* that lead to the bug. It requires two runs, one with the flaw and one without, and it looks at the differences between the sets of states. By repeatedly changing the inputs and rerunning the program using a binary search strategy and automated testing, delta debugging methodically narrows the differences between the two runs. Delta debugging discovers dependencies that form a cause-effect chain, which it expands until it identifies the smallest set of changes to input variables that causes the bug to appear. Although it requires many runs of the program, this analysis is done at full program speed and without the intervention of the programmer, so it saves development time.

Program synthesis may be ready for a breakthrough. The state of the art today is that given incomplete segments of programs, program synthesis tools can often supply the missing code. One of the most interesting uses of this technology is in Microsoft Office Excel 2013, called the *Flash Fill feature*, which does programming by example (Gulwani et al. 2012). You give examples of what you want to do to rows or columns of code, and Excel will attempt to repeat and generalize what you do. Moreover, you can correct its attempts to steer it to what you want (Gantenbein 2012).

This split between Plan-and-Document and Agile development may become more pronounced with the advances in practicality of formal methods. The size of programs that can be formally verified is growing over time, with improvements in tools, faster computers, and wider understanding of how to write formal specifications. If the work of careful specification in advance of coding could be rewarded by not needing to test and yet have thoroughly verified programs, then the trade-offs would be crisp around change. For formal methods to work, clearly change needs to be rare. When change is commonplace, Agile is the answer, for change is the essence of Agile.

While Agile works better than other software methodologies for some types of apps today, it is surely not the final answer in software development. If a new methodology could simplify including a good software architecture and good design patterns while maintaining Agile's ease of change, it could become more popular. Historically, a new methodology comes along every decade or two, so it may soon be time for a new one.

This book itself was developed during the dawn of the ***Massive Open Online Course*** (MOOC) movement, which is another trend that we predict will become more significant in the next few years. Like many other advances in this modern world, we wouldn't have MOOCs without SaaS and cloud computing. The enabling components were:

- Scalable video distribution via services like YouTube.
- Sophisticated autograders running on cloud computing that evaluate assignments immediately yet can scale to tens of thousands of students.
- Discussion forums as a scalable solution to asking questions and getting answers from both other students and the staff.

These components combine to form a wonderful, low-cost vehicle for students around the world. For example, it will surely improve continuing education of professionals in our fast changing field, enable gifted pre-college students to go beyond what their schools can teach, and let dedicated students around the world who do not have access to great universities still get a good education. MOOCs may even have the side effect of raising the quality bar for traditional courses by providing viable alternatives to ineffective lecturers. If MOOCs deliver on only half of these opportunities, they will still be a potent force in higher education.

13.3 Essential Readings

Software tools change rapidly: languages and frameworks go in and out of vogue every few years. Software engineering methodologies change over time as well: Agile wasn't the first methodology and won't be the last, and variations of Agile continue to evolve. It may therefore seem perilous to recommend a list of readings that *all* aspiring software engineers should read, much less a list of online sources. Nonetheless, some of the field's bedrock ideas and acquired wisdom has stood the test of time, and we believe all software engineers would benefit by reading them. With some trepidation, we offer suggestions here, reminding the reader that we have *no formal or financial connection* to any of these works, although we do have professional or academic relationships with some of the authors.

Software design and architecture. We have mentioned Unix numerous times in this book; it is arguably the most influential production operating system ever created. While many of our readers were probably first exposed to it as Linux, that is only the latest and most widely adopted implementation of the original ***kernel*** or "core" of Unix, which chose and refined some of the best ideas from pioneering experimental systems such as ***Multics*** while greatly simplifying and streamlining some of its other aspects. Multics was an acronym for Multiplexed Information and Computing Service, with Multiplexed indicating that it was designed to serve multiple users simultaneously; the designers of Unix joked that their much smaller operating system might only be suitable for a single user at a time, so they named it Unics, later shortened to Unix.

The structure of Unix, and its approach to program design and to the management of processes and machine resources, are pervasive. We can suggest no better book than the one written by two of its designers: *The Unix Programming Environment* by Brian Kernighan and Rob Pike. Even many non-Unix operating systems borrow heavily from Unix's models of process and resource management, and from a practical perspective, strong Unix toolsmithing skills are vital when you need to quickly produce some shell scripts to automate an otherwise tedious task.

Software project management. When Turing Award winner Frederick P. Brooks Jr. wrote *The Mythical Man-Month* (Brooks 1995), there was no such thing as "the software industry." Software was generally written by programmers working for the companies that made the hardware, but the processes for estimating effort, coordinating the work of multiple team members, and performing quality control were far less evolved than they were for hardware design, which had a multiple-decade head start. Brooks's account of managing the OS/360 project—the operating system for the groundbreaking IBM System/360, and far and away the most complex piece of commercial software ever written up to that time—still holds valuable lessons for software project management, even if the economics of the industry have changed.

If OS/360 was the face of software development in the 1960s, then collaboratively-

authored open-source development, exemplified by projects such as Linux, can be said to be at least part of the face of software development today. Developer Eric S. Raymond's *The Cathedral and the Bazaar*[1] (Raymond 2001), while controversial, is a good starting point for understanding how collaborative open-source development came about and how it compares to traditional in-house closed-source (proprietary) development. As of this writing, both models are vital to the software industry, with some companies embracing both. For example, Facebook and Twitter do not generally release the source code to their products, but they have released open-source tools such as React and Bootstrap originally developed for internal use.

The history of software. The evolution from proprietary early software to shrink-wrapped consumer software to SaaS is beautifully described in Martin Campbell-Kelly's *From Airline Reservations to Sonic the Hedgehog: A History of the Software Industry* (Campbell-Kelly 2003). For those interested in the corresponding history of hardware and the computing field generally, Paul Ceruzzi's *A History of Modern Computing* (Ceruzzi 2003) provides an outstanding overview; for the impatient, we recommend the much shorter and less detailed *Computing: A Concise History* by the same author (Ceruzzi 2012).

The modern business of software. Today, software is a business, and as Chapter 10 emphasized, most often a team effort. Building and running a successful software enterprise requires balancing technical expertise with great strategies for recruiting, hiring, team building, and retention. Joel Spolsky, creator of the project management tool Trello and co-creator of StackOverflow, for several years wrote a blog called Joel On Software[2], virtually every article of which is worth reading. You can read them online for free, or purchase the two books that collect and organize many of the posts by topic, *Joel On Software* and *More Joel On Software* (Spolsky 2004a,b). And if you're going to be managing software engineers (or anyone else for that matter), the actionable advice in *The One Minute Manager* (Blanchard and Johnson 1982) is hard to beat for clarity and conciseness.

Software as craftsmanship. Throughout the book we've affirmed the value of beautiful, well-tested code. Code that is hard to understand or poorly covered by tests is resistant to enhancement, and so is likely to be short-lived. Steve McConnell's *Code Complete* (McConnell 1993) justifiably remains a classic on practical software construction techniques. Robert C. "Uncle Bob" Martin's *Clean Code: A Handbook of Agile Software Craftsmanship* (Martin 2008) is an excellent and more recent companion to *Code Complete* that emphasizes taking a craftsperson's pride in beauty and elegance in the code you write. Both should be on every developer's bookshelf. Reading and rereading these books periodically will help keep their suggestions at top of mind when you're actually at work.

Finally, to recommend a particular book is not to devalue any other particular book; no list of suggested readings can be definitive, complete, or fully objective. Nonetheless, we believe that these suggested "classics of the genre," which combine historical perspective with modern best practices, form a great starting point for software engineers wishing to really polish their skills while being fully aware of the work of those on whose shoulders they stand.

13.4 Last Words

> *Ultimately, it comes down to taste. It comes down to exposing yourself to the best things that humans have done, and then try to bring those things into what you're doing.*
>
> —Steve Jobs

Software helped put humans on the moon, led to the invention of lifesaving CAT scans, and enables eyewitness citizen journalism. By working as a software developer, you become part of a community that has the power to change the world.

But with great power comes great responsibility. Faulty software caused the loss of the Ariane V rocket[3] and Mars Observer[4] as well as the deaths of several patients due to radiation overdoses from the Therac-25 machine[5].

While the early stories of computers and software are dominated by "frontier narratives" of lone geniuses working in garages or at startups, software today is too important to be left to any one individual, however talented. As we said in Chapter 10, software development is now a team sport.

We believe the concepts in this book increase the chances of you being both a responsible software developer *and* a part of a winning team. There's no textbook for getting there; just keep writing, learning, and refactoring to apply your lessons as you go.

And as we said in the first chapter, we look forward to becoming passionate fans of the beautiful and long-lasting code that you and your team create!

K. H. Blanchard and S. Johnson. *The One Minute Manager*. William Morrow, Cambridge, MA, 1982.

F. P. Brooks. *The Mythical Man-Month*. Addison-Wesley, Reading, MA, Anniversary edition, 1995. ISBN 0201835959.

M. Campbell-Kelly. *From Airline Reservations to Sonic the Hedgehog: A History of the Software Industry*. MIT Press, Cambridge, MA, 2003.

P. Ceruzzi. *A History of Modern Computing*. MIT Press, Cambridge, MA, 2003.

P. Ceruzzi. *Computing: A Concise History*. MIT Press, Cambridge, MA, 2012.

D. Gantenbein. Flash fill gives Excel a smart charge, Feb 2012. URL `http://research.microsoft.com/en-us/news/features/flashfill-020613.aspx`.

S. Gulwani, W. R. Harris, and R. Singh. Spreadsheet data manipulation using examples. *Communications of the ACM*, 55(8):97–105, 2012.

F. Ji and T. Sedano. Comparing extreme programming and waterfall project results. *Conference on Software Engineering Education and Training*, pages 482–486, 2011.

R. C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008. ISBN 9780132350884.

S. McConnell. *Code Complete (Microsoft Programming Series)*. Microsoft Press, 1993. ISBN 1556154844.

E. S. Raymond. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revoluationary*. O'Reilly Media, Inc., 2001.

J. Spolsky. *Joel On Software*. Apress, 2004a.

J. Spolsky. *More Joel On Software*. Apress, 2004b.

L. Stella, S. Jarzabek, and B. Wadhwa. A comparative study of maintainability of web applications on J2EE, .NET and Ruby on Rails. *10th International Symposium on Web Site Evolution*, pages 93–99, October 2008.

A. Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pages 1–10, New York, NY, USA, Nov 2002. ACM. doi: 10.1145/587051.587053. URL `http://www.st.cs.uni-saarland.de/papers/fse2002/p201-zeller.pdf`.

Notes

[1] `http://www.catb.org/~esr/writings/cathedral-bazaar`
[2] `https://joelonsoftware.com`
[3] `http://en.wikipedia.org/wiki/Ariane_5_Flight_501`
[4] `http://en.wikipedia.org/wiki/Mars_Observer`
[5] `http://en.wikipedia.org/wiki/Therac-25`