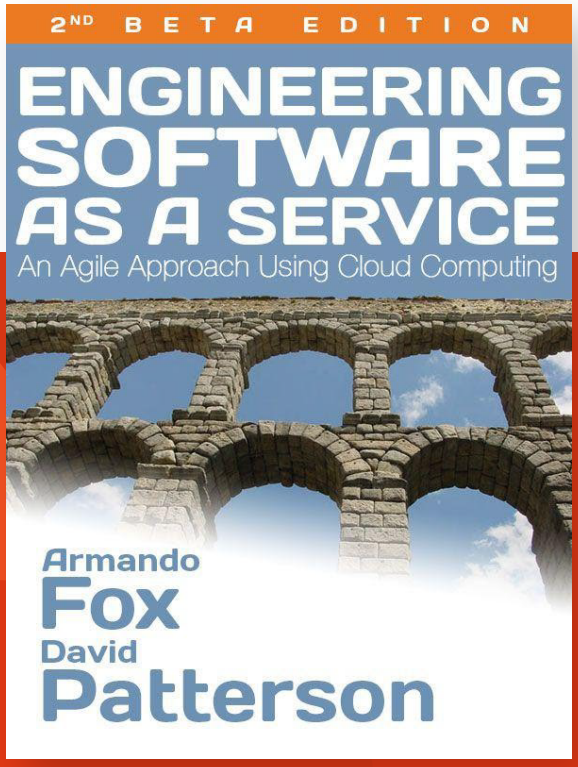
YOUR COMPLIMENTARY CHAPTER



CHAPTER 1:
Introduction to SaaS and Agile Software Development

Engineering Software as a Service: An Agile Approach Using Cloud Computing
by Armando Fox and David Patterson

www.progress.com

PROGRESS

1 Introduction to Software as a Service and Agile Software Development

Sir Maurice Wilkes (1913–2010) received the 1967 Turing Award for designing and building EDSAC in 1949, one of the first stored program computers. The Turing Award[9] is the highest award in computing, which the Association for Computing Machinery (ACM) has bestowed annually since 1966. Named after computing pioneer Alan Turing, it is known informally as the "Nobel Prize of Computer Science."
(This book uses sidebars to include what your authors think are interesting asides or short biographies of computing pioneers that supplement the primary text. We hope readers will enjoy them.)

It was on one of my journeys between the EDSAC room and the punching equipment that "hesitating at the angles of stairs" the realization came over me with full force that a good part of the remainder of my life was going to be spent finding errors in my own programs.

—Maurice Wilkes, Memoirs of a Computer Pioneer, 1985

Concepts

Each chapter opening starts with a one-page summary of that chapter's big concepts. For this introductory chapter, they are:

- ***Software as a Service*** (***SaaS***) deploys software at a single site but makes it available to millions of users over the Internet on their personal mobile devices, which provides benefits to both users and developers. The single copy of the software and the competitive environment for SaaS products leads to more rapid ***software evolution*** for SaaS than for shrink-wrapped software.
- ***Service Oriented Architecture*** (***SOA***) creates apps from components that act as interoperable services, which allows new systems to be built from these components with much less effort.
- ***Cloud Computing*** supplies the dependable and scalable computation and storage for SaaS by utilizing ***Warehouse Scale Computers*** containing as many as 100,000 servers. The economies of scale allow Cloud Computing to be offered as a utility, where SaaS providers need only pay for the resources that they *actually* use rather than what they could *potentially* need.
- ***Software quality*** is defined as providing business value to both customers and developers. Software ***Quality Assurance*** (***QA***) comes from many levels of testing: ***unit***, ***module***, ***integration***, ***system***, and ***acceptance***.
- ***Clarity via conciseness***, ***synthesis***, ***reuse***, and ***automation via tools*** are four paths to improving ***software productivity***. The programming framework ***Ruby on Rails*** follows them to make SaaS developers productive.
- ***Don't Repeat Yourself (DRY)*** warns not to use repetition to achieve reuse, as there should a single representation of each piece of knowledge.
- ***Plan-and-Document*** software development processes or ***lifecycles*** rely on careful, up-front planning that is extensively documented and carefully managed to make software development more predictable. Prominent examples are ***Waterfall***, ***Spiral***, and the ***Rational unified process*** (***RUP***) lifecycles.
- In contrast, the ***Agile*** lifecycle relies on incrementally developed prototypes that involve continuous feedback from the customer on each ***iteration***, each of which takes one or two weeks.

Since change is the norm for Agile, it is an excellent SaaS lifecycle, and the one on which the book focuses.

1.1 Introduction

To me programming is more than an important practical art. It is also a gigantic undertaking in the foundations of knowledge.

—Grace Murray Hopper

Grace Murray Hopper (1906–1992) was one of the first programmers, developed the first compiler, and was referred to as "Amazing Grace." She became a rear admiral in the US Navy, and in 1997, a warship was named for her: the USS Hopper.

We're excited to have the honor of introducing you to major advances in software development. This field is not some dreary academic discipline where few care what happens. Successful software projects can create services that millions of people use every day, leading to companies like ***Amazon***, ***Facebook***, and ***Google*** that are household names. Google is even used as a verb, as a synonym for search. Everyone involved with such services is proud to be associated with them.

The downside is that software projects can also fail so disastrously that they live in infamy. Programmers have heard these sorry stories of the ***Ariane 5 rocket explosion***, the ***Therac-25*** lethal radiation overdose, and the FBI ***Virtual Case File*** project abandonment so frequently that they are clichés. No software engineer would want these projects on their résumés.

Ariane 5 flight 501. On June 4, 1996, an overflow occurred 37 seconds after liftoff in a guidance system, with spectacular consequences[10], when a floating point number was converted to a shorter integer. This exception could not occur on the slower Ariane 4 rocket, so reusing successful components without thorough system testing was expensive: satellites worth $370M were lost.

This book is *not* just the traditional well-intentioned survey of do's and don'ts for each phase of conventional software development. This book makes the concepts concrete with a hands-on demonstration of how to design, implement, and deploy an application in the cloud. The virtual machine image associated with this book comes pre-loaded with all the software you'll need to do it. In addition to reading what we wrote, you can see our demonstrations and hear our voices as part of the 27 screencasts in the following chapters. Moreover, you can even watch us teach this material, for this book is associated with a free ***Massive Open Online Course*** (***MOOC***) from EdX.org[11]. CS169.1x and CS169.2x offer 8 to 10 minute video segments that generally correspond one-to-one with all the sections of this book, including this one. These MOOCs offer quick autograding of programming assignments and quizzes to give you feedback on how well you've learned the material plus an online forum to ask and answer questions.

Finally, this book is inspired by the opportunity to explain two major innovations that have revolutionized software in the past decade, which led us to write this book. To start us on our journey, we begin with a breakthrough in the distribution of software.

1.2 Software as a Service

Software as a Service (SaaS) delivers software and data as a service over the Internet, usually via a thin program such as a browser that runs on local client devices instead as an application binary that must be installed and runs wholly on that device. Examples that many of us use every day include searching, social networking, and watching videos. The advantages for the customer and for the software developer are widely touted:

1. Since customers do not need to install the application, they don't have to worry whether their hardware is the right brand or fast enough, nor whether they have the correct version of the operating system.

2. The data associated with the service is generally kept with the service, so customers need not worry about backing it up, losing it due to a local hardware malfunction, or even losing the whole device, such as a phone or tablet.

SaaS Programming Framework	*Programming Language*
Active Server Pages (ASP.NET)	Common Language Runtime (CLR)
Django	Python
Enterprise Java Beans (EJB)	Java
JavaServer Pages (JSP)	Java
Rails	Ruby
Sinatra	Ruby
Spring	Java
Zend	PHP

Figure 1.1: Examples of SaaS programming frameworks and the programming languages they are written in.

3. When a group of users wants to collectively interact with the same data, SaaS is a natural vehicle.

4. When data is large and/or updated frequently, it may make more sense to centralize data and offer remote access via SaaS.

5. Only a single copy of the server software runs in a uniform, tightly-controlled hardware and operating system environment selected by the developer, which avoids the compatibility hassles of distributing binaries that must run on wide-ranging computers and operating systems. In addition, developers can test new versions of the application on a small fraction of the real customers temporarily without disturbing most customers. (If the SaaS client runs in a browser, there still are compatibility challenges, which we describe in Chapter 2.) SaaS companies compete regularly on bringing out new features to help ensure that their customers do not abandon them for a competitor who offers a better service.

6. Since only developers have a copy of the software, they can upgrade the software and underlying hardware frequently as long as they don't violate the external application program interfaces (API). Moreover, developers don't need to annoy users with the seemingly endless requests for permission to upgrade their applications.

SaaS: Innovate or Die? Lest you think the perceived need to improve a successful service is just software engineering paranoia, the most popular search engine used to be Alta Vista and the most popular social networking site used to be MySpace.

Combining the advantages to the customer and the developer together explains why SaaS is rapidly growing and why traditional software products are increasingly being transformed to offer SaaS versions. An example of the latter is Microsoft Office 365, which allows you to use the popular Word, Excel, and PowerPoint productivity programs as a remote service by paying for use rather than pre-purchasing software and installing it on your local computer. Another example is TurboTax Online, which offers the same deal for another shrink-wrap standard-bearer.

Unsurprisingly, given the popularity of SaaS, Figure 1.1 lists programming frameworks that claim to help. In this book, we use Ruby on Rails ("Rails"), although the ideas we cover will work with other programming frameworks as well. We chose Rails because it came from a community that had already embraced the Agile lifecycle, so the tools support Agile particularly well.

Ruby is typical of modern scripting languages in including automatic memory management and dynamic typing. By including important advances in programming languages, Ruby goes beyond languages like Perl in supporting multiple programming paradigms such as object oriented and functional programming.

Useful additional features that help productivity via reuse include ***mix-ins***, which collects related behaviors and makes it easy to add them to many different classes, and ***metaprogramming***, which allows Ruby programs to synthesize code at runtime. Reuse is also enhanced with Ruby's support for ***closures*** via ***blocks*** and ***yield***. Chapter 3 is a short description of Ruby for those who already know Java, and Chapter 4 introduces Rails.

In addition to our view of Rails being technically superior for Agile and SaaS, Ruby and Rails are widely used. For example, Ruby routinely appears among top 10 most popular programming languages. A well-known SaaS app associated with Rails is Twitter, which began as a Rails app in 2006 and grew from 20,000 tweets per day in 2007 to 200,000,000 in 2011, during which time other frameworks replaced various parts of it.

If you are not already familiar with Ruby or Rails, this gives you a chance to practice an important software engineering skill: use the right tool for the job, even if it means learning a new tool or new language! Indeed, an attractive feature of the Rails community is that its contributors routinely improve productivity by inventing new tools to automate tasks that were formerly done manually.

Summary: ***Software as a Service (SaaS)*** is attractive to both customers and providers because the universal client (the Web browser) makes it easier for customers to use the service and the single version of the software at a centralized site makes it easier for the provider to deliver and improve the service. Given the ability and desire to frequently upgrade SaaS, the Agile software development process is popular for SaaS, and so there are many frameworks to support Agile and SaaS. This book uses Ruby on Rails.

Self-Check 1.2.1. *Which of the examples of Google SaaS apps—Search, Maps, News, Gmail, Calendar, YouTube, and Documents—is the* best *match to each of the six arguments given in this section for SaaS, reproduced below.*

◇ While you can argue the mappings, below is our answer. (Note that we cheated and put some apps in multiple categories)

1. No user installation: Documents
2. Can't lose data: Gmail, Calendar.
3. Users cooperating: Documents.
4. Large/changing datasets: Search, Maps, News, and YouTube.
5. Software centralized in single environment: Search.
6. No field upgrades when improve app: Documents.

■

Self-Check 1.2.2. *True or False: If you are using the Agile development process to develop SaaS apps, you could use Python and Django or languages based on the Microsoft CLR and ASP.NET instead of Ruby and Rails.*

◇ True. Programming frameworks for Agile and SaaS include Django and ASP.NET. ■

1.3 Service Oriented Architecture

> *SOA had long suffered from lack of clarity and direction... SOA could in fact die - not due to a lack of substance or potential, but simply due to a seemingly endless proliferation of misinformation and confusion.*
>
> —Thomas Erl, *About the SOA Manifesto*, 2010

SaaS is actually a special case of a more general software architecture where all components are designed to be services: a ***Service Oriented Architecture (SOA)***. Alas, SOA was one of those terms that is so ill defined, over used, and over hyped that some think it is just an empty marketing phrase, like ***modular***. SOA actually means that components of an application act as interoperable services, and can be used independently and recombined in other applications. The contrasting implementation is considered a "software silo," which rarely has APIs to internal components. If you mis-estimate what the customer really wants, the cost is much lower with SOA than with "siloed" software to recover from that mistake and try something else or to produce a similar-but-not-identical variant to please a subset of users.

SOA confusion At the time of this writing, Wikipedia has independent entries for ***Service-oriented Architecture***, ***Service Oriented Architecture Fundamentals***, and ***Service-Oriented Architecture Types***. The Wikipedia authors can't even agree on the capitalization and hyphenation!

For example, Amazon started in 1995 with siloed software for its online retailing site. According to former Amazonian Steve Yegge[12], in 2002 the CEO and founder of Amazon mandated a change to what we would today call SOA. He broadcast an email to all employees along these lines:

1. *All teams will henceforth expose their data and functionality through service interfaces.*
2. *Teams must communicate with each other through these interfaces.*
3. *There will be no other form of interprocess communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. The only communication allowed is via service interface calls over the network.*
4. *It doesn't matter what technology they use. HTTP, Corba, Pub/Sub, custom protocols—doesn't matter. [Amazon CEO Jeff] Bezos doesn't care.*
5. *All service interfaces, without exception, must be designed from the ground up to be externalizable. That is to say, the team must plan and design to be able to expose the interface to developers in the outside world. No exceptions.*
6. *Anyone who doesn't do this will be fired.*
7. *Thank you; have a nice day!*

A similar software revolution happened at Facebook in 2007—three years after the company went online—when ***Facebook Platform*** was launched. Relying on SOA, Facebook Platform allowed third party developers to create applications that interact with core features of Facebook such as what people like, who their friends are, who is tagged in their photos, and so on. For example, the New York Times was one of the early Facebook Platform developers. Facebook users reading the New York Times online on May 24, 2007 suddenly noticed that they could see which articles their friends were reading and which articles their friends liked. As a contrasting example of a social networking site using a software silo, Google+

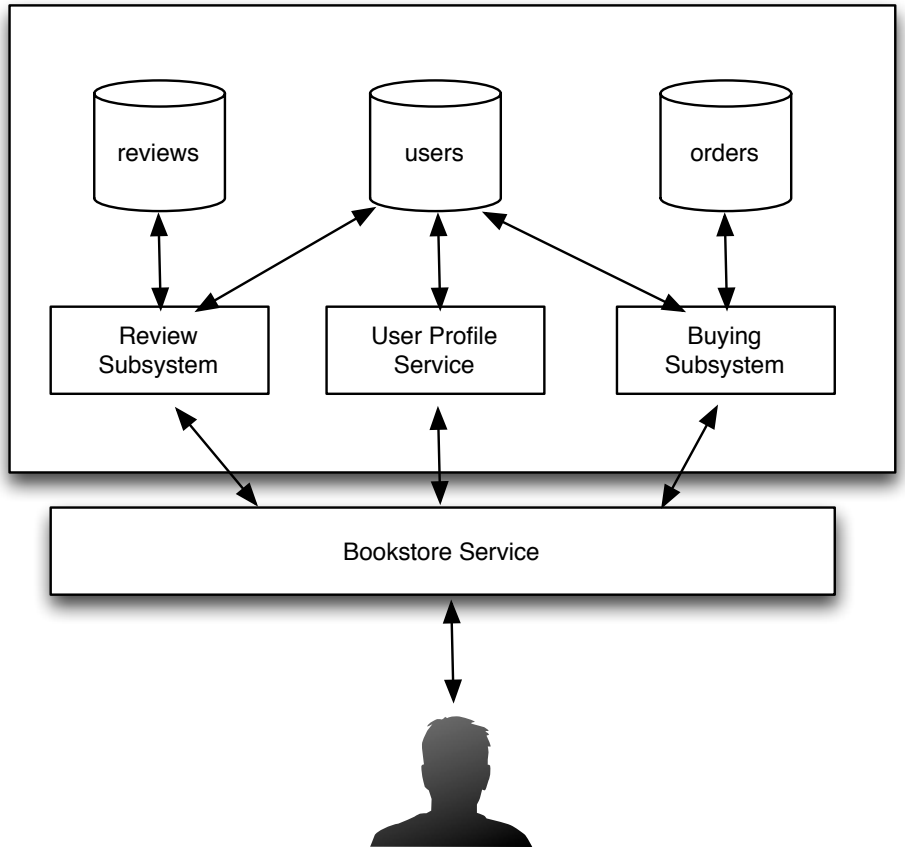


Figure 1.2: Silo version of a fictitious bookstore service, with all subsystems behind a single API.

had no APIs when it was launched on June 28, 2011 and had just one heavyweight API three months later: following the complete stream of everything a Google+ user sees.

To make these notions more concrete, suppose we wanted to create a bookstore service first as a silo and then as a SOA. Both will contain the same three subsystems: reviews, user profiles, and buying.

Figure 1.2 shows the silo version. The silo means subsystems can internally share access to data directly in different subsystems. For example, the reviews subsystem can get user profile info out of the users subsystem. However, all subsystems are inside a single external API ("the bookstore").

Figure 1.3 shows the SOA version of the bookstore service, where all subsystems are separate and independent. Even though all are inside the "boundary" of the bookstore's datacenter, which is shown as a dotted rectangle, the subsystems interact with each other as if they were in separate datacenters. For example, if the reviews subsystem wants information about a user, it can't just reach directly into the users database. Instead, it has to ask the users ***service***, via whatever API is provided for that purpose. A similar restriction is true for

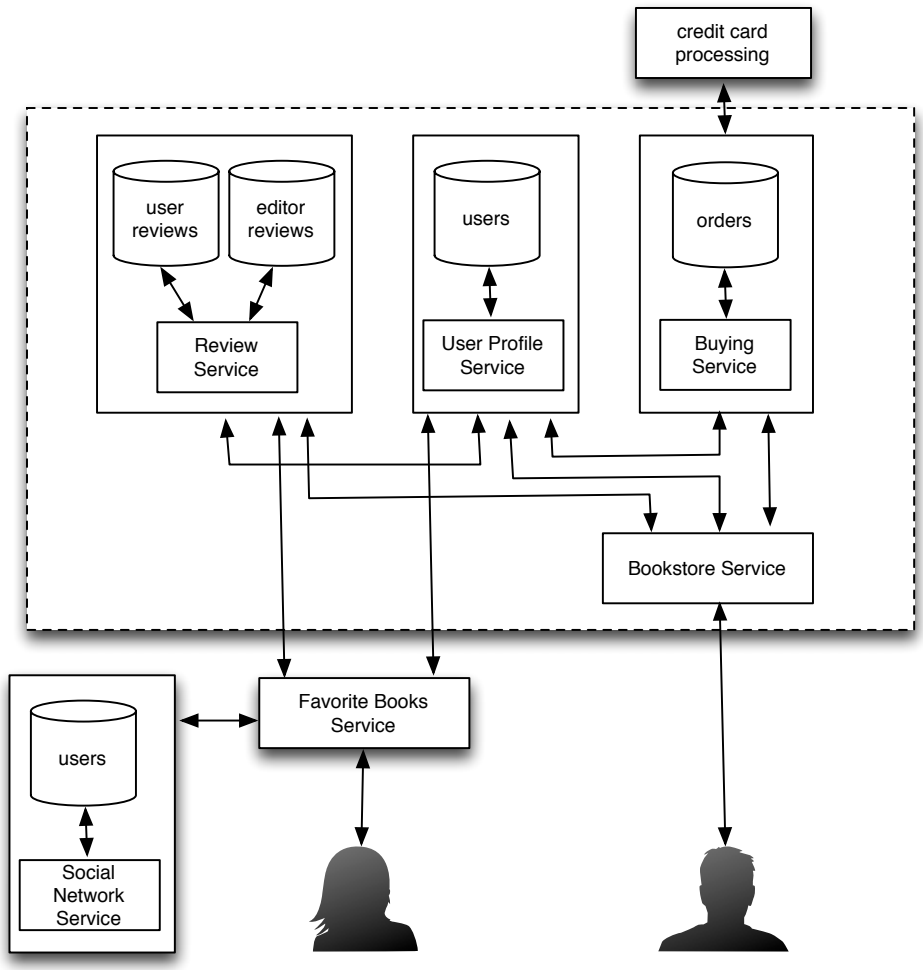


Figure 1.3: SOA version of a fictitious bookstore service, where all three subsystems are independent and available via APIs.

buying.

The "bookstore app" is then just one particular composition of these services. Consequently, others can recombine the services with others to create new apps. For example, a "my favorite books" app might combine the users service and reviews service with a social network, so you can see what your social-network friends think about the books you have reviewed (see Figure 1.3).

The critical distinction of SOA is that no service can name or access another service's data; it can only make requests for data through an external API. If the data it wants is not available through that API, then too bad. Note that SOA does not match the traditional layered model of software, in which each higher layer is built directly from the primitives of the immediately lower layer as in siloed software. SOA implies vertical slices through many layers, and these slices are connected together to form a service. While SOA usually means a bit more work compared to building a siloed service, the payback is tremendous reusability. Another upside of SOA is that the explicit APIs make testing easier.

There are two widely accepted downsides to SOA. First, each invocation of a service involves the higher cost of wading through the deeper software stack of a network interface, so there is a performance hit to SOA. Second, while a siloed system is very likely to be completely down on a failure, software engineers using SOA must deal with the sticky case of partial failures, so SOA makes dependability planning a bit more challenging.

Summary: Although the term was nearly lost in a sea of confusion, ***Service Oriented Architecture*** (***SOA***) just means an approach to software development where all the subsystems are only available as external services, which means others can recombine them in different ways. Following the tools and guidelines in this book ensures that your SaaS apps will be a good fit to SOA.

Self-Check 1.3.1. *Another take on SOA is that it is just a common sense approach to improving programmer productivity. Which productivity mechanism does SOA best exemplify: Clarity via conciseness, Synthesis, Reuse, or Automation and Tools?*

◇ Reuse! The purpose of making internal APIs visible is so that programmers can stand on the shoulders of others. ■

Given the case for SaaS and the understanding that it relies on a Service Oriented Architecture, we are ready to see the underlying hardware that makes SaaS possible.

1.4 Cloud Computing

If computers of the kind I have advocated become the computers of the future, then computing may someday be organized as a public utility just as the telephone system is a public utility . . . The computer utility could become the basis of a new and important industry.

—John McCarthy, at MIT centennial celebration in 1961

John McCarthy (1927–2011) received the Turing Award in 1971 and is the inventor of Lisp. As a pioneer of timesharing large computers, as early as 1961 he envisioned an "ecosystem" foreshadowing today's Software as a Service in which large computers provide continuous service to large numbers of users with a utility-pricing-like model. Clusters of commodity hardware and the spread of fast networking have helped make this vision a reality.

SaaS places three demands on our information technology (IT) infrastructure:

1. Communication, to allow any customer to interact with the service.

2. Scalability, in that the central facility running the service must deal with the fluctuations in demand during the day and during popular times of the year for that service as well as a way for new services to add users rapidly.

3. Dependability, in that both the service and the communication vehicle must be continuously available: every day, 24 hours a day ("24×7").

The gold standard set by the US public phone system is 99.999% availability ("five nines"), or about 5 minutes of downtime per year.

The Internet and broadband to the home easily resolve the communication demand of SaaS. Although some early web services were deployed on expensive large-scale computers—in part because such computers were more reliable and in part because it was easier to operate a few large computers—a contrarian approach soon overtook the industry. Collections of commodity small-scale computers connected by commodity Ethernet switches, which became known as ***clusters***, offered several advantages over the "big iron" hardware approach:

- Because of their reliance on Ethernet switches to interconnect, clusters are much more scalable than conventional servers. Early clusters offered 1000 computers, and today's datacenters contain 100,000 or more.

- Careful selection of the type of hardware to place in the datacenter and careful control of software state made it possible for a very small number of operators to successfully run thousands of servers. In particular, some datacenters rely on ***virtual machines*** to simplify operation. A virtual machine monitor is software that imitates a real computer so successfully that you can even run an operating system correctly on top of the virtual machine abstraction that it provides (Rosenblum and Garfinkel 2005). The goal is to imitate with low overhead, and one popular use is to simplify software distribution within a cluster.

- Two senior architects at Google show that the cost of the equivalent amount of processors, memory, and storage is much less for clusters than for "big iron," perhaps by a factor of 20 (Barroso and Hoelzle 2009).

- Although the cluster components are less reliable than conventional servers and storage systems, the cluster software infrastructure makes the whole system dependable via extensive use of redundancy. The low hardware cost makes the redundancy at the software level affordable. Modern service providers also use multiple datacenters that are distributed geographically so that a natural disaster cannot knock a service offline.

As Internet datacenters grew, some service providers realized that their per capita costs were substantially below what it cost others to run their own smaller datacenters, in large part due to economies of scale when purchasing and operating 100,000 computers at a time. They also benefit from higher utilization given that many companies could share these giant datacenters, which (Barroso and Hoelzle 2009) call ***Warehouse Scale Computers***, as smaller datacenters often run at only 10% to 20% utilization. Thus, these companies realized they could profit from making their datacenter hardware available on a pay-as-you-go basis.

The result is called ***public cloud services*** or ***utility computing***, which offers computing, storage, and communication at pennies per hour (see (Armbrust et al. 2010). Moreover, there is no additional cost for scale: Using 1000 computers for 1 hour costs no more than using 1 computer for 1000 hours. Leading examples of "infinitely scalable" pay-as-you-go

Rapid growth of FarmVille The prior record for number of users of a social networking game was 5 million. FarmVille had 1 million players within 4 days after it was announced, 10 million after 2 months, and 28 million daily players and 75 million monthly players after 9 months. Fortunately, FarmVille used the Elastic Compute Cloud (EC2) from Amazon Web Services, and kept up with its popularity by simply paying to use larger clusters.

computing are Amazon Web Services, Google App Engine, and Microsoft Azure. The public cloud means today that anyone with a credit card and a good idea can start a SaaS company that can grow to millions of customers without first having to build and operate a datacenter.

Today, we call this long held dream of computing as a utility ***Cloud Computing***. We believe that Cloud Computing and SaaS are transforming the computer industry, with the full impact of this revolution taking the rest of this decade to determine. Indeed, this revolution is one reason we decided to write this book, as we believe engineering SaaS for Cloud Computing is radically different from engineering shrink-wrap software for PCs and servers.

Summary

- The Internet supplies the communication for SaaS.
- ***Cloud Computing*** provides the scalable and dependable hardware computation and storage for SaaS.
- Cloud computing consists of ***clusters*** of commodity servers that are connected by local area network switches, with a software layer providing sufficient redundancy to make this cost-effective hardware dependable.
- These large clusters or ***Warehouse Scale Computers*** offer economies of scale.
- Taking advantage of economies of scale, some Cloud Computing providers offer this hardware infrastructure as low-cost ***utility computing*** that anyone can use on a pay-as-you-go basis, acquiring resources immediately as your customer demand grows and releasing them immediately when it drops.

Self-Check 1.4.1. *True or False: Internal datacenters could get the same cost savings as Warehouse Scale Computers if they embraced SOA and purchased the same type of hardware.*
◇ False. While imitating best practices of WSC could lower costs, the major cost advantage of WSCs comes from the economies of scale, which today means 100,000 servers, thereby dwarfing internal datacenters. ■

1.5 Beautiful vs. Legacy Code

Unlike hardware, software is expected to grow and evolve over time. Whereas hardware designs must be declared finished before they can be manufactured and shipped, initial software designs can easily be shipped and later upgraded over time. Basically, the cost of upgrade in the field is astronomical for hardware and affordable for software.

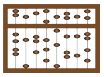
Hence, software can achieve a high-tech version of immortality, potentially getting better over time while generations of computer hardware decay into obsolescence. The causes of ***software evolution*** are not only fixing faults, but also adding new features that customers request, adjusting to changing business requirements, improving performance, and adapting to a changed environment. Software customers expect to get notices about and install improved versions of the software over the lifetime that they use it, perhaps even submitting bug reports to help developers fix their code. They may even have to pay an annual maintenance fee for this privilege!

Just as novelists fondly hope that their brainchild will be read long enough to be labeled a classic—which for books is 100 years!—software engineers should hope their creations would also be long lasting. Of course, software has the advantage over books of being able to be improved over time. In fact, a long software life often means that others maintain and enhance it, letting the creators of original code off the hook.

The Oldest Living Program might be MOCAS[13] ("Mechanization of Contract Administration Services"), which was originally purchased by the US Department of Defense in 1958 and was still in use as of 2005.

This brings us to a few terms we'll use throughout the book. The term ***legacy code*** refers to software that, despite its old age, continues to be used because it meets customers' needs. Sixty percent of software maintenance costs are for adding new functionality to legacy software, vs. only 17% for fixing bugs, so legacy software is successful software.

The term "legacy" has a negative connotation, however, in that it indicates that the code is difficult to evolve because of inelegance of its design or use of antiquated technology. To contrast to legacy code, we use the term ***beautiful code*** to indicate long-lasting code that is easy to evolve. The worst case is not legacy code, however, but ***unexpectedly short-lived code*** that is soon discarded because it doesn't meet customers' needs. We'll highlight examples that lead to beautiful code with the Mona Lisa icon. Similarly, we'll highlight text that deals with legacy code using an abacus icon, which is certainly a long-lasting but little changed calculating device.



Surprisingly, despite the widely accepted importance of enhancing existing software, this topic is traditionally ignored in college courses and textbooks. We feature such software in this book for two reasons. First, you can reduce the effort to build a program by finding existing code that you can reuse. One supplier is open source software. Second, it's advantageous to learn how to build code that makes it easier for successors to enhance, as that increases software's chances of a long life. In the following chapters, we show examples of both beautiful code and legacy code that we hope will inspire you to make your designs simpler to evolve.

Summary: Successful software can live decades and is expected to evolve and improve, unlike computer hardware that is finalized at time of manufacture and can be considered obsolete within just a few years. One goal of this book is to teach you how to increase the chances of producing beautiful code so that your software lives a long and useful life.

We next define software quality and see how to test for it to increase our chances of writing beautiful code.

1.6 Software Quality Assurance: Testing

And the users exclaimed with a laugh and a taunt:
"It's just what we asked for, but not what we want."

—Anonymous

We start this topic with a definition of quality. A standard definition of ***quality*** for any product is "fitness for use," which must provide business value for both the customer and for the manufacturer (Juran and Gryna 1998). For software, quality means both satisfying the customer's needs—easy to use, gets correct answers, does not crash, and so on—*and* to be easy for the developer to debug and enhance. ***Quality Assurance*** (***QA***) also comes from manufacturing, and refers to processes and standards that lead to manufacture of high-quality products and to the introduction of manufacturing processes that improve quality. Software

QA then means both ensuring that products under development have high quality and the creation of processes and standards in an organization that lead to high quality software. As we shall see, some conventional software processes even use a separate QA team that tests software quality (Section 8.9).

Determining software quality involves two terms that are commonly interchanged but have subtle distinctions (Boehm 1979):

- ***Verification***: Did you build the thing *right*? (Did you meet the specification?)
- ***Validation***: Did you build the right *thing*? (Is this what the customer wants? That is, is the specification correct?)

Software prototypes typically help with validation rather than verification, since customers often change their minds on what they want once they begin to see the product work.

The main approaches to verification and validation is ***testing***; we'll describe an alternative to testing, called ***formal methods***, in Chapter 8. The motivation for testing is that the earlier developers find mistakes, the cheaper it is to repair them. Given the substantial costs of repair, testing involve large communities of researchers and practitioners with many books on the topics. We highlight just some of the main issues here.

Infeasibility of exhaustive testing
Suppose it took just 1 nanosecond to test a program and it had just one 64-bit input that we wanted to test exhaustively. (Obviously, most programs take longer to run and have more inputs.) Just this simple case would take 2^{64} nanoseconds, or 500 years!

Given the vast number of different combinations of inputs, testing cannot be exhaustive. One way to reduce the space is to perform different tests at different phases of software development. Starting bottom up, ***unit testing*** makes sure that a single procedure or method does what was expected. The next level up is ***module testing***, which tests across individual units. For example, unit testing works within a single class whereas module testing works across classes. Above this level is ***integration testing***, which ensures that the interfaces between the units have consistent assumptions and communicate correctly. This level does not test the functionality of the units. At the top level is ***system testing*** or ***acceptance testing***, which tests to see if the integrated program meets its specifications.

Another perspective, less common today, distinguishes ***black-box tests***, whose design is based solely on the software's external specifications, from ***white-box tests*** (also called ***glass-box tests***), whose design reflects knowledge about the software's implementation that is not implied by external specifications. For example, the external specification of a hash table might just state that when we store a key/value pair and later read that key, we should get back the stored value. A black-box test would specify a random set of key/value pairs to test this behavior, whereas a white-box test might exploit knowledge about the hash function to construct worst-case test data that results in many hash collisions.

Given the multiple approaches and many levels of testing, from unit tests to acceptance tests, testing needs an indication of the fraction of the possible paths that have been tested. If we represent the paths as a graph, the term ***test coverage*** means what fraction of the paths is covered by the test. (Section 8.8 goes into more depth on test coverage.) Note that even 100% test coverage is no guarantee of design reliability, however, since it says nothing about the *quality* of the tests that were run.

Given the long lifetime of software, another concern is whether later changes in design will cause failures in tests that it previously passed. To prevent such a backwards step, software engineers use ***regression testing*** to automatically rerun old tests to ensure that the current version still works at least as well as it used to. A related term is ***continuous integration*** or ***CI*** testing, which means the entire program is tested every time new code is checked in, versus having a separate test phase after completing development. Section 8.8

discusses testing your tests by tweaking source code (***mutation testing***) and testing your interfaces by throwing random data at your application to see what breaks (***fuzz testing***).

Summary: Testing reduces the risks of errors in designs.

- In its many forms, testing helps ***verify*** that software meets the specification and ***validates*** that the design does what the customer wants.
- Attacking the infeasibility of exhaustive testing, we divide in order to conquer by focusing on ***unit testing***, ***module testing***, ***integration testing***, and full ***system testing*** or ***acceptance testing***. Each higher-level test delegates more detailed testing to lower levels.
- ***Black-box*** vs. ***white-box*** or ***glass-box testing*** refers to whether tests rely on the external specifications or the implementation of a module.
- By mapping designs to graphs and recording which nodes and arcs are traversed, ***test coverage*** indicates what has and has not been tested.
- ***Regression testing*** reapplies old tests to reduce the chance of new revisions breaking designs that have worked in the past.

Self-Check 1.6.1. *While all of the following help with verification, which form of testing is most likely to help with validation: Unit, Module, Integration, or Acceptance?*
◇ Validation is concerned with doing what the customer really wants versus whether code met the specification, so acceptance testing is most likely to point out the difference between doing the thing right and doing the right thing. ■

After this review of quality assurance, let's see how to make developers productive.

1.7 Productivity: Conciseness, Synthesis, Reuse, and Tools

> *Most software today is very much like an Egyptian pyramid with millions of bricks piled on top of each other, with no structural integrity, but just done by brute force and thousands of slaves.*
>
> —Alan Kay, *ACM Queue*, 2005

Moore's Law means that hardware resources double every 18 months. These faster computers with much larger memories could run much larger programs. To build bigger applications that could take advantage of the more powerful computers, software engineers needed to improve their productivity.

Engineers developed four fundamental mechanisms to improve their productivity:

1. Clarity via conciseness
2. Synthesis
3. Reuse

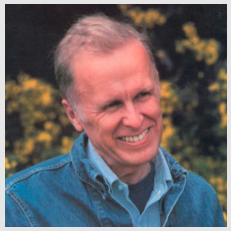
4. Automation via Tools

One of the driving assumptions of improving productivity of programmers is that if programs are easier to understand, then they will have fewer bugs and to be easier to evolve. A closely related corollary is that if the program is smaller, it's generally easier to understand. We capture this notion with our motto of "clarity via conciseness."

Programming languages do this two ways. The first is simply offering a syntax that lets programmers express ideas naturally and in fewer characters. For example, below are two ways to express a simple assertion:

```
assert_greater_than_or_equal_to(a, 7)
a.should be >= 7
```

John Backus (1924–2007) received the 1977 Turing Award in part for "profound, influential, and lasting contributions to the design of practical high-level programming systems, notably through his work on Fortran," which was the first widely used high-level language.



Unquestionably, the second version (which happens to be legal Ruby) is shorter and easier to read and understand, and will likely be easier to maintain. It's easy to imagine momentary confusion about the order of arguments in the first version in addition to the higher cognitive load of reading twice as many characters (see Chapter 3).

The second way to improve clarity is to raise the level of abstraction. That initially meant the invention of higher-level programming languages such as Fortran and COBOL. This step raised the engineering of software from assembly language for a particular computer to higher-level languages that could target multiple computers simply by changing the compiler.

As computer hardware performance continued to increase, more programmers were willing to delegate tasks to the compiler and runtime system that they formerly performed themselves. For example, Java and similar languages took over memory management from the earlier C and C++ languages. Scripting languages like Python and Ruby have raised the level of abstraction even higher. Examples are ***reflection***, which allows programs to observe themselves, and ***metaprogramming***, which allows programs to modify their own structure and behavior at runtime. To highlight examples that improve productivity via conciseness, we will use this "Concise" icon.

The second productivity mechanism is synthesis; that is, the implementation is generated rather than created manually. Logic synthesis for hardware engineers meant that they could describe hardware as Boolean functions and receive highly optimized transistors that implemented those functions. The classic software synthesis example is ***Bit blit***. This graphics primitive combines two bitmaps under control of a mask. The straightforward approach would include a conditional statement in the innermost loop to chose the type of mask, but it was slow. The solution was to write a program that could synthesize the appropriate special-purpose code *without* the conditional statement in the loop. We'll highlight examples that improve productivity by generating code with this "CodeGen" gears icon.

The third productivity mechanism is to reuse portions from past designs rather than write everything from scratch. As it is easier to make small changes in software than in hardware, software is even more likely than hardware to reuse a component that is almost but not quite a correct fit. We highlight examples that improve productivity via reuse with this "Reuse" recycling icon.

Procedures and functions were invented in the earliest days of software so that different parts of the program could reuse the same code with different parameter values. Standardized libraries for input/output and for mathematical functions soon followed, so that programmers could reuse code developed by others. Hardware engineers also had the equivalent of procedures and design libraries.

Procedures in libraries let you reuse implementations of individual tasks. But more commonly programmers want to reuse and manage ***collections*** of tasks. The next step in software reuse was therefore ***object-oriented programming***, where you could reuse the same tasks with different objects via the use of inheritance in languages like C++ and Java.

While inheritance supported reuse of implementations, another opportunity for reuse is a general strategy for doing something even if the implementation varies. ***Design patterns***, inspired by work in civil architecture (Alexander et al. 1977), arose to address this need. Language support for reuse of design patterns includes ***dynamic typing***, which facilitates composition of abstractions, and ***mix-ins***, which offers ways to collect functionality from multiple methods without some of the pathologies of multiple inheritance found in some object oriented programming. Python and Ruby are examples of languages with features that help with reuse of design patterns.

Note that reuse does *not* mean copying and pasting code so that you have very similar code in many places. The problem with cutting and pasting code is that you may not change all the copies when fixing a bug or adding a feature. Here is a software engineering guideline that guards against repetition:

> *Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.*
>
> —Andy Hunt and Dave Thomas, 1999

This guideline has been captured in the motto and acronym: ***Don't Repeat Yourself (DRY)***. We'll use a towel as the "DRY" icon to show examples of DRY in the following chapters.

A core value of computer engineers is finding ways to replace tedious manual tasks with tools to save time, improve accuracy, or both. Obvious Computer Aided Design (CAD) tools for software development are compilers and interpreters that raise the level of abstraction and generate code as mentioned above, but there are also more subtle productivity tools like Makefiles and version control systems (see Section 10.4) that automate tedious tasks. We highlight tool examples with the hammer icon.

The tradeoff is always the time it takes to learn a new tool versus the time saved in applying it. Other concerns are the dependability of the tool, the quality of the user experience, and how to decide which one to use if there are many choices. Nevertheless, one of the software engineering tenets of faith is that a new tool can make our lives better.

Learning new tools
Proverbs 14:4 in the King James Bible discusses improving productivity by taking the time to learn and use tools: *Where there are no oxen, the manger is clean; but abundant crops come by the strength of oxen.*

Your authors embrace the value of automation and tools. That is why we show you several tools in this book to make you more productive. The good news is that any tool we show you will have been vetted to ensure its dependability and that time to learn will be paid back many times over in reduced development time and in the improved quality of the final result. For example, Chapter 7 shows how ***Cucumber*** automates turning user stories into integration tests and it also demonstrates how ***Pivotal Tracker*** automatically measures ***Velocity***, which is a measure of the rate of adding features to an application. Chapter 8 introduces ***RSpec*** that automates the unit testing process. The bad news is that you'll need to learn several new tools. However, we think the ability to quickly learn and apply new tools is a requirement for success in engineering software, so it's a good skill to cultivate.

Thus, our fourth productivity enhancer is automation via tools. We highlight examples that use automation with the robot icon, although they are often also associated with tools.

Summary: Moore's Law inspired software engineers to improve their productivity by:

- Coveting conciseness, in using compact syntax and by raising the level of design by using higher-level languages. Recent advances include ***reflection*** that allows programs to observe themselves and ***metaprogramming*** that allows programs to modify their own structure and behavior at runtime.
- Synthesizing implementations.
- Reusing designs by following the principle of ***Don't Repeat Yourself (DRY)*** and by relying upon innovations that help reuse, such as procedures, libraries, object-oriented programming, and design patterns.
- Using (and inventing) CAD tools to automate tedious tasks.

Self-Check 1.7.1. *Which mechanism is the weakest argument for productivity benefits of compilers for high-level programming languages: Clarity via conciseness, Synthesis, Reuse, or Automation and Tools?*

◇ Compilers make high-level programming languages practical, enabling programmers to improve productivity via writing the more concise code in a HLL. Compilers do synthesize lower-level code based on the HLL input. Compilers are definitely tools. While you can argue that HLL makes reuse easier, reuse is the weakest of the four for explaining the benefits of compilers. ■

Given the need of SaaS for rapid evolution, we need to find a software development process where change is the norm rather than the exception. Let us start with traditional software development processes, which turn out to be a poor fit to SaaS.

1.8 Software Development Processes: Plan and Document

> *If builders built buildings the way programmers wrote programs, then the first woodpecker that came along would destroy civilization.*
>
> —Gerald Weinberg, *Weinberg's Second Law*

The general unpredictability of software development, along with the software disasters like those mentioned above, led to the study of how high-quality software could be developed on a predictable schedule and budget. Drawing the analogy to other engineering fields, the term ***software engineering*** was coined (Naur and Randell 1969). The goal was to discover methods to build software that are as predictable in quality, cost, and time as those used to build bridges in civil engineering.

One thrust of software engineering was to bring an engineering discipline to what was often unplanned software development. Before starting to code, come up with a plan for the project, including extensive, detailed documentation of all phases of that plan. Progress is then measured against the plan. Changes to the project must be reflected in the documentation and possibly to the plan.

An early version of this "Plan-and-Document" software development process was developed in 1970 (Royce 1970). It follows this sequence of phases:

1. Requirements analysis and specification
2. Architectural design
3. Implementation and Integration
4. Verification
5. Operation and Maintenance

Given that the earlier you find an error the cheaper it is to fix, the philosophy of this process is to complete a phase before going on to the next one, thereby removing as many errors as early as possible. Getting the early phases right could also prevent unnecessary work downstream. As this process could take years, the extensive documentation helps to ensure that important information is not lost if a person leaves the project and that new people can get up to speed quickly when they join the project.

Windows 95 was heralded by a US$300 million outdoor party[14] for which Microsoft hired comedian Jay Leno, lit up New York's Empire State Building using the Microsoft Windows logo colors, and licensed "Start Me Up" by the Rolling Stones as the celebration's theme song.

Because it flows from the top down to completion, this process is called the ***Waterfall*** software development process or Waterfall software development ***lifecycle***. Understandably, given the complexity of each stage in the Waterfall lifecycle, product releases are major events toward which engineers worked feverishly and which are accompanied by much fanfare.

In the Waterfall lifecycle, the long life of software is acknowledged by a maintenance phase that repairs errors as they are discovered. New versions of software developed in the Waterfall model go through the same several phases, and take typically between 6 and 18 months.

The Waterfall model can work well with well-specified tasks like NASA space flights, but it runs into trouble when customers change their minds about what they want. A Turing Award winner captures this observation:

> *Plan to throw one [implementation] away; you will, anyhow.*
>
> —Fred Brooks, Jr.

That is, it's easier for customers to understand what they want once they see a prototype and for engineers to understand how to build it better once they've done it the first time.

This observation led to a software development lifecycle developed in the 1980s that combines prototypes with the Waterfall model (Boehm 1986). The idea is to iterate through a sequence of four phases, with each iteration resulting in a prototype that is a refinement of the previous version. Figure 1.4 illustrates this model of development across the four phases, which gives this lifecycle its name: the ***Spiral model***. The phases are

1. Determine objectives and constraints of this iteration
2. Evaluate alternatives and identify and resolve risks
3. Develop and verify the prototype for this iteration
4. Plan the next iteration

Rather than document all the requirements at the beginning, as in the Waterfall model, the requirement documents are developed across the iteration as they are needed and evolve with the project. Iterations involve the customer before the product is completed, which reduces chances of misunderstandings. However, as originally envisioned, these iterations were 6 to

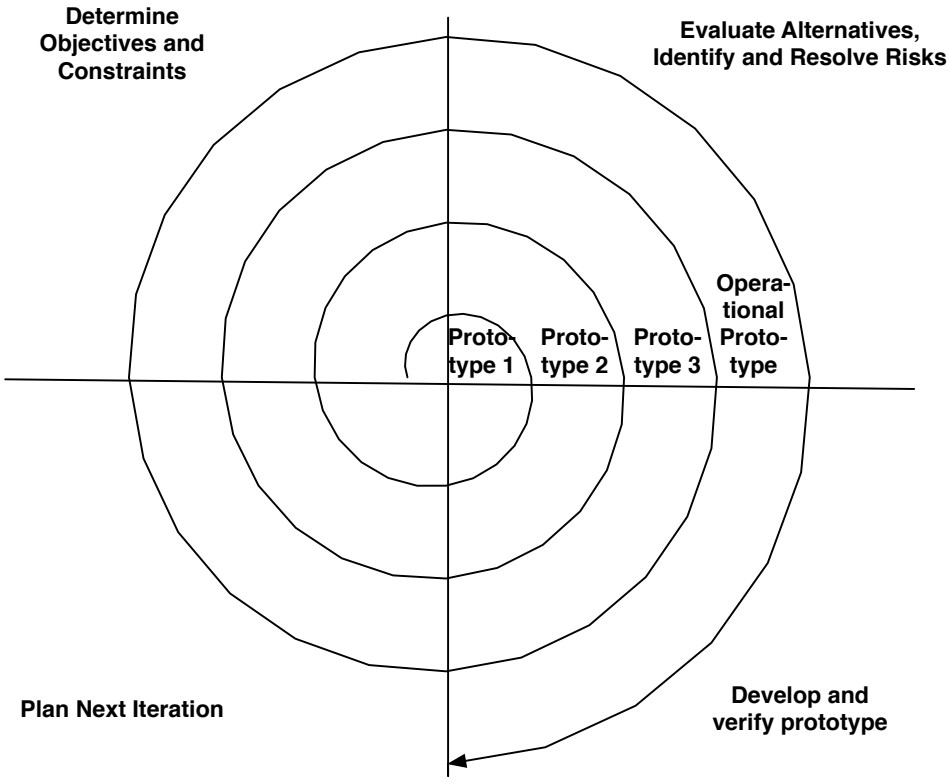


Figure 1.4: The Spiral lifecycle combines Waterfall with prototyping. It starts at the center, with each iteration around the spiral going through the four phases and resulting in a revised prototype until the product is ready for release.

24 months long, so there is plenty of time for customers to change their minds during an iteration! Thus, Spiral still relies on planning and extensive documentation, but the plan is expected to evolve on each iteration.

Given the importance of software development, many variations of Plan-and-Document methodologies were proposed beyond these two. A recent one is called the ***Rational unified process*** (***RUP***) (Kruchten 2003), which combines features of both Waterfall and Spiral lifecycles as well standards for diagrams and documentation. We'll use RUP as a representative of the latest thinking in Plan-and-Document lifecycles. Unlike Waterfall and Spiral, it is more closely allied to business issues than to technical issues.

Big Design Up Front, abbreviated ***BDUF***, is a name some use for software processes like Waterfall, Spiral, and RUP that depend on extensive planning and documentation. They are also known variously as ***heavyweight***, ***plan-driven***, ***disciplined***, or ***structured*** processes.

Like Waterfall and Spiral, RUP has phases:

1. Inception: makes the business case for the software and scopes the project to set the schedule and budget, which is used to judge progress and justify expenditures, and initial assessment of risks to schedule and budget.
2. Elaboration: works with stakeholders to identify use cases, designs a software architecture, sets the development plan, and builds an initial prototype.
3. Construction: codes and tests the product, resulting in the first external release.
4. Transition: moves the product from development to production in the real environment, including customer acceptance testing and user training.

Unlike Waterfall, each phase involves iteration. For example, a project might have one inception phase iteration, two elaboration phase iterations, four construction phase iterations, and two transition phase iterations. Like Spiral, a project could also iterate across all four phases repeatedly.

In addition to the dynamically changing phases of the project, RUP identifies six "engineering disciplines" (also known as workflows) that people working on the project should collectively cover:

1. Business Modeling
2. Requirements
3. Analysis and Design
4. Implementation
5. Test
6. Deployment

These disciplines are more static than the phases, in that they nominally exist over the whole lifetime of the project. However, some disciplines get used more in earlier phases (like business modeling), some periodically (like test), and some more towards the end (deployment). Figure 1.5 shows the relationship of the phases and the disciplines, with the area indicating the amount of effort in each discipline over time.

The goal of all these Plan-and-Document lifecycles is to improve the predictability of the software development process via extensive documentation, which must be changed whenever the goals change. Here is how textbook authors put it (Lethbridge and Laganiere 2002; Braude 2001):

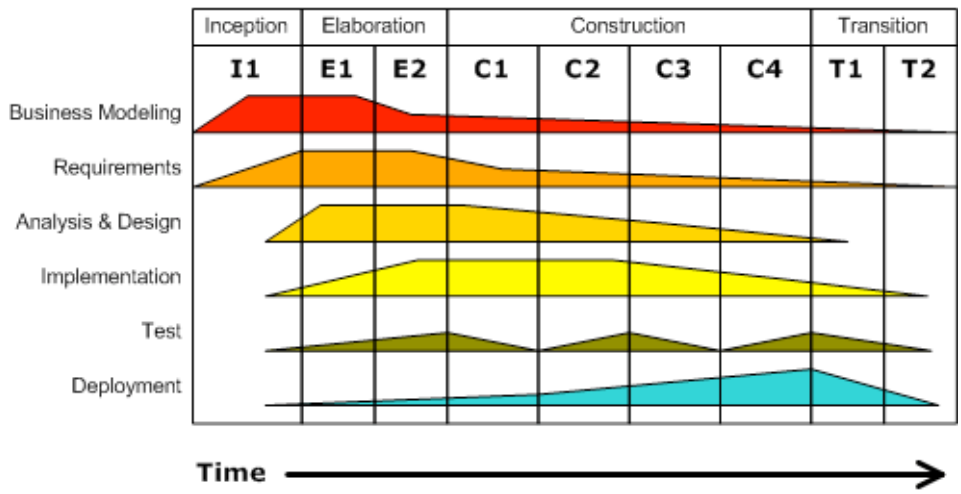


Figure 1.5: The Rational Unified Process lifecycle allows the project to have multiple iterations in each phase and identifies the skills needed by the project team, which vary in effort over time. RUP also has three "supporting disciplines" not shown in this figure: Configuration and Change Management, Project Management, and Environment. (Image from Wikipedia Commons by Dutchgilder.)

Documentation should be written at all stages of development, and includes requirements, designs, user manuals, instructions for testers and project plans.

—Timothy Lethbridge and Robert Laganiere, 2002

Documentation is the lifeblood of software engineering.

—Eric Braude, 2001

This process is even embraced with an official standard of documentation: IEEE/ANSI standard 830/1993.

An unfortunate downside to teaching this disciplined approach is that students may find software development tedious (Nawrocki et al. 2002; Estler et al. 2012). Given the importance of predictable software development, this is hardly a strong enough reason not to teach it; the good news is that there are alternatives that work just as well for some kinds of projects that are a better fit to the classroom, as we describe in the next section.

Summary: The basic *activities* of software engineering are the same in all the software development process or ***lifecycles***, but their interaction over time relative to product releases differs among the models. The Waterfall lifecycle is characterized by much of the design being done in advance of coding, completing each phase before going on to the next one. The Spiral lifecycle iterates through all the development phases to produce prototypes, but like Waterfall the customers may only get involved every 6 to 24 months. The more recent Rational Unified Process lifecycle includes phases, iterations, and prototypes, while identifying the people skills needed for the project. All rely on careful planning and thorough documentation, and all measure progress against a plan.

■ ***Elaboration: SEI Capability Maturity Model (CMM)***

The Software Engineering Institute at Carnegie Mellon University proposed the ***Capability Maturity Model*** (***CMM***) (Paulk et al. 1995) to evaluate organizations' software-development processes based on Plan-and-Document methodologies. The idea is that by modeling the software development process, an organization can improve them. SEI studies observed five levels of software practice:

1. Initial or Chaotic—undocumented/ad hoc/unstable software development.
2. Repeatable—not following rigorous discipline, but some processes repeatable with consistent results.
3. Defined—Defined and documented standard processes that improve over time.
4. Managed—Management can control software development using process metrics, adapting the process to different projects successfully.
5. Optimizing—Deliberate process optimization improvements as part of management process.

As bigger is better, CMM implicitly encourages an organization to move up the CMM levels. While not proposed as a software development methodology, many consider it one. For example, (Nawrocki et al. 2002) compares CMM Level 2 to the Agile software methodology (see next section).

Self-Check 1.8.1. *What are a major similarity and a major difference between processes like Spiral and RUP versus Waterfall?*

◇ All rely on planning and documentation, but Spiral and RUP use iteration and prototypes to improve them over time versus a single long path to the product. ■

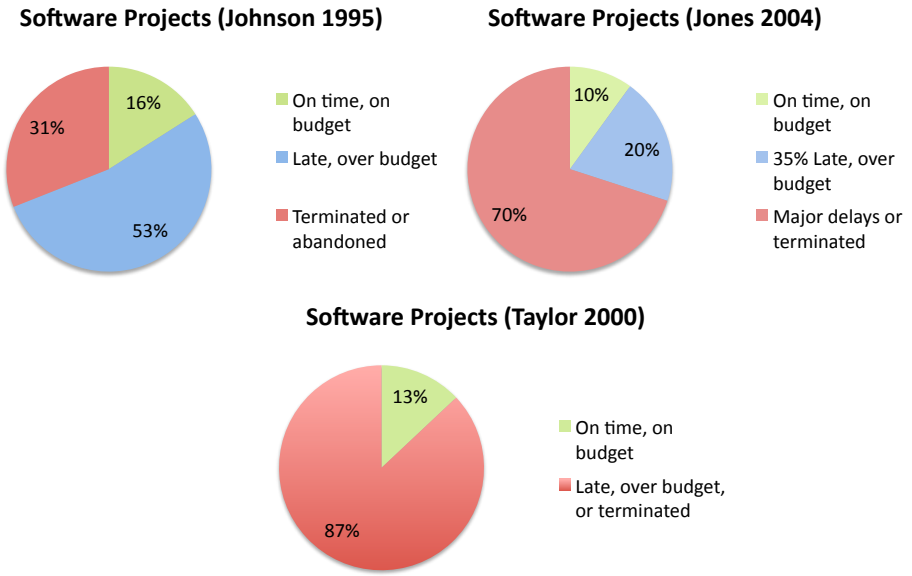


Figure 1.6: The first study of software projects found that 53% of projects exceeding their budgets by a factor of 2.9 and overshot their schedule by a factor of 3.2 and another 31% of software projects were cancelled before completion (Johnson 1995). The an estimated annual cost in the United States for such software projects was $100B. The second survey of 250 large projects, each with the equivalent of more than a million lines of C code, found similarly disappointing results (Jones 2004). The final survey of members of the British Computer Society found that only 130 of 1027 projects met their schedule and budget. Half of all projects were maintenance or data conversion projects and half new development project, but the successful projects divided into 127 of the former and just 3 of the latter (Taylor 2000).

Self-Check 1.8.2. *What are the differences between the phases of these processes?*
◇ Waterfall phases separate planning (requirements and architectural design) from implementation. Testing the product before release is next, followed by a separate operations phase. The Spiral phases are aimed at an iteration: set the goals for an iteration; explore alternatives; develop and verify the prototype for this iteration; and plan the next iteration. RUP phases are tied closer to business objectives: inception makes business case and sets schedule and budget; elaboration works with customers to build an initial prototype; construction builds and test the first version; and transition deploys the product. ■

1.9 Software Development Processes: The Agile Manifesto

If a problem has no solution, it may not be a problem, but a fact—not to be solved, but to be coped with over time.

—Shimon Peres

While plan-and-development processes brought discipline to software development, there were still complaints. One article included a "Software Wall of Shame" with 31 highly-visible software projects that collectively were responsible for losses of $17B, with the majority of these projects abandoned (Charettte 2005). Figure 1.6 summarizes three surveys of software projects. With just 10% to 16% on time and on budget, more projects were cancelled or

abandoned than met their mark. A closer look at the 13% success of third study is even more sobering, as less than 1% of new development projects met their schedules and budgets.

Perhaps these problematic projects did not try to follow the Plan-and-Document methodologies, but then again, perhaps they did. Indeed, the FBI Virtual Case File considered as a software disaster (Section 1.1) started with an 800-page system requirements specification (see Section 7.10) that was developed over six months in conjunction with FBI stakeholders (Goldstein 2005).

Agile is also known variously as a ***lightweight*** or ***undisciplined*** process.

Perhaps the "Reformation moment" for software engineering was the ***Agile Manifesto*** in February 2001. A group of software developers met to develop a lighter-weight software lifecycle. Here is exactly what the ***Agile Alliance*** nailed to the door of the "Church of Plan and Document":

"*We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:*

- **Individuals and interactions** *over processes and tools*
- **Working software** *over comprehensive documentation*
- **Customer collaboration** *over contract negotiation*
- **Responding to change** *over following a plan*

That is, while there is value in the items on the right, we value the items on the left more."

Variants of Agile There are many variants of Agile software development (Fowler 2005). The one we use in this book is ***Extreme Programming***, which is abbreviated ***XP***, and credited to Kent Beck.

This alternative development model is based on embracing change as a fact of life: developers should continuously refine a working but incomplete prototype until the customer is happy with result, with the customer offering feedback on each iteration. Agile emphasizes ***test-driven development (TDD)*** to reduce mistakes by writing the tests *before* writing the code, ***user stories*** to reach agreement and validate customer requirements, and ***velocity*** to measure project progress. We'll cover these topics in detail in later chapters.

Regarding software lifetimes, the Agile software lifecycle is so quick that new versions are available every week or two—with some even releasing every day–so they are not even special events as in the Plan-and-Document models. The assumption is one of basically continuous improvement over its lifetime.

We mentioned in the prior section that newcomers can find Plan-and-Document processes tedious, but this is not the case for Agile. This perspective is captured by a software engineering instructor's early review of Agile:

> *Remember when programming was fun? Is this how you got interested in computers in the first place and later in computer science? Is this why many of our majors enter the discipline—because they like to program computers? Well, there may be promising and respectable software development methodologies that are perfectly suited to these kinds of folks. . . . [Agile] is fun and effective, because not only do we not bog down the process in mountains of documentation, but also because developers work face-to-face with clients throughout the development process and produce working software early on.*
>
> —Renee McCauley, "Agile Development Methods Poised to Upset Status Quo," *SIGCSE Bulletin*, 2001

By de-emphasizing planning, documentation, and contractually binding specifications, the Agile Manifesto ran counter to conventional wisdom of the software engineering intelligentsia, so it was not universally welcomed with open arms (Cormick 2001):

	Question: A no answer suggests Agile; a yes suggests Plan and Document
1	Is specification required?
2	Are customers unavailable?
3	Is the system to be built large?
4	Is the system to be built complex (e.g., real time)?
5	Will it have a long product lifetime?
6	Are you using poor software tools?
7	Is the project team geographically distributed?
8	Is team part of a documentation-oriented culture?
9	Does the team have poor programming skills?
10	Is the system to be built subject to regulation?

Figure 1.7: Ten questions to help decide whether to use an Agile lifecycle (the answer is no) or a Plan-and-Document lifecycle (the answer is yes) (Sommerville 2010). We find it striking that when asking these questions for projects done by student teams in a class, virtually all answers point to Agile. As this book attests, open source software tools are excellent, thus available to students (question 6). Our survey of industry (see Preface) found that graduating students do indeed have good programming skills (question 9). The other eight answers are clearly no for student projects.

> *[The Agile Manifesto] is yet another attempt to undermine the discipline of software engineering... In the software engineering profession, there are engineers and there are hackers... It seems to me that this is nothing more than an attempt to legitimize hacker behavior... The software engineering profession will change for the better only when customers refuse to pay for software that doesn't do what they contracted for... Changing the culture from one that encourages the hacker mentality to one that is based on predictable software engineering practices will only help transform software engineering into a respected engineering discipline.*
>
> —Steven Ratkin, "Manifesto Elicits Cynicism," *IEEE Computer*, 2001

One pair of critics even published the case against Agile as a 432-page book! (Stephens and Rosenberg 2003)

The software engineering research community went on to compare Plan-and-Document lifecycles to the Agile lifecycle in the field and found—to the surprise of some cynics—that Agile could indeed work well, depending on the circumstances. Figure 1.7 shows 10 questions from a popular software engineering textbook (Sommerville 2010) whose answers suggest when to use Agile and when to use Plan-and-Document methods.

An example of the acceptance of Agile is a recent study comparing it to Plan-and-Document for geographically distributed programming teams, including outsourced projects (Estler et al. 2012). Note that question 7 in Figure 1.7 suggests using Plan-and-Document for geographically distributed teams, so it would seem that Agile would be at a disadvantage. The study found that the majority of the 66 projects in their survey used Agile, which speaks to its rising popularity even with distributed teams. Second, while the Agile teams were smaller than the Plan-and-Document teams (related to question 3), some Agile teams had more than 60 people. Third, project management difficulties were more frequent in Plan-and-Document projects, perhaps in part because Agile projects had fewer communication problems. The authors' bottom line was that projects using Agile had outcomes as good as those of Plan-and-Document projects for geographically distributed development teams.

Indeed, the authors of the 1995 survey of software projects above partially credit the increasing popularity of Agile for the improvement of the number of on-time and on-budget projects. They rose from 16% to 32% in 2009, the reduction of the time and cost overruns for late projects fell from about 3.0 to 1.7, and their percentage of projects fell from 53% to

44% (Johnson 2009).

Conventional wisdom among the software engineering intelligentsia today is to use a Plan-and-Document lifecycle when requirements are accurately known and not subject to radical change—since they scale to larger teams and hence larger projects—and to use an Agile lifecycle when requirements change frequently or when creating a formal specification with stakeholders is difficult or unrealistic (Estler et al. 2012).

Note that frequent upgrades of SaaS—due to only having a single copy of the software—perfectly align with the Agile software lifecycle. Hence, Amazon, eBay, Facebook, Google, and other SaaS providers all rely on the Agile lifecycle, and traditional software companies like Microsoft are increasingly using Agile in their product development.

Thus, we concentrate on Agile in the six software development chapters in Part II of the book, but each chapter also gives the perspective of the Plan-and-Document methodologies on topics like requirements, testing, project management, and maintenance. This contrast allows readers to decide for themselves when each methodology is appropriate. Part I introduces SaaS and SaaS programming environments, including Ruby, Rails, and Javascript.

Summary: In contrast to the Plan-and-Document lifecycles, the Agile lifecycle works with customers to continuously add features to working prototypes until the customer is satisfied, allowing customers to change what they want as the project develops. Documentation is primarily through user stories and test cases, and it does not measure progress against a predefined plan. Progress is gauged instead by recording ***velocity***, which essentially is the rate that a project completes features. The Agile process is an excellent match to the fast-changing nature of SaaS applications.

Self-Check 1.9.1. *True or False: A big difference between Waterfall and Agile development is that Agile does not use requirements.*

◇ False: While Agile does not develop extensive requirements documents as does Waterfall, the interactions with customers lead to the creation of requirements as user stories, as we shall see in Chapter 7. ■

Self-Check 1.9.2. *True or False: A big difference between Spiral and Agile development is building prototypes and interacting with customers during the process.*

◇ False: Both build working but incomplete prototypes that the customer helps evaluate. The difference is that customers are involved every two weeks in Agile versus up to two years in with Spiral. ■

■ ***Elaboration: Testing: Plan-and-Document vs. Agile lifecycles***

For the Waterfall development process, testing happens after each phase is complete and in a final verification phase that includes acceptance tests. For Spiral it happens on each iteration, which can last one or two years. Assurance for Agile comes from test-driven development, in that the tests are written *before* the code when coding from scratch. When enhancing existing code, test-driven design means writing the tests before writing the enhancements. The amount of testing depends on whether you are enhancing beautiful code or legacy code, with the latter needing a lot more.

■ ***Elaboration: Productivity: Plan-and-Document vs. Agile lifecycles***

Productivity is measured in the people-hours to implement a new function. The difference is the cycles are much longer in Waterfall and Spiral vs. Agile—on the order of 6 to 24 months vs. 1/2 a month—so much more work is done between releases that the customer sees, and hence the chances are greater for Waterfall and Spiral that more work will ultimately be rejected by the customer.

1.10 Guided Tour of the Book

I hear and I forget. I see and I remember. I do and I understand.

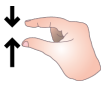
—Confucius

With this introduction behind us, we can now explain what follows and what paths you might want to take. To do and understand, as Confucius advises, begin by reading Appendix A. It explains how to obtain and use the "bookware," which is our name for the software associated with the book.

The rest of the book is divided into two parts. Part I explains Software as a Service, and Part II explains modern software development, with an emphasis on Agile.

Chapter 2 starts Part I with an explanation of the architecture of a SaaS application, using an altitude analogy of going from the 100,000-foot view to the 500-foot view. During the descent you'll learn the definition of many acronyms that you may have already heard—APIs, CSS, IP, REST, TCP, URLs, URIs, and XML—as well as some widely used buzzwords: cookies, markup languages, port numbers, and three-tier architectures. It also demonstrates the importance of design patterns, particularly Model-View-Controller that is at the heart of Rails.

Rather than just tell you how to build long lasting software and watch you forget, we believe you must do to understand. It is much easier to try good guidelines if the tools encourage it, and we believe today the best SaaS tools support in the Rails framework, which is written in Ruby. Thus, Chapter 3 introduces Ruby. The Ruby introduction is short because it assumes you already know another object-oriented programming language well, in this case Java. As mentioned above, we believe successful software engineers will need to routinely learn new languages and tools over their careers, so learning Ruby and Rails is good practice.



Chapter 4 next introduces the basics of Rails. We cover more advanced features of Rails in Chapter 5. We split the material into two chapters for readers who want to get started writing an app as soon as they can, which just requires Chapter 4. While the material in Chapter 5 is more challenging to learn and understand, your application can be DRYer and more concise if you use concepts like partials, validations, lifecycle callbacks, filters, associations, and foreign keys. Readers already familiar with Ruby and Rails should skip these chapters.

Building on familiarity with Ruby and Rails by this point in the book, Chapter 6 introduces the programming language JavaScript, its productive framework jQuery, and the testing tool Jasmine.

Given this background, the next six chapters of Part II illustrate important software engineering principles using Rails tools to build and deploy a SaaS app. Figure 1.8 shows one iteration of the Agile lifecycle, which we use as a framework on which to hang the next chapters of the book.

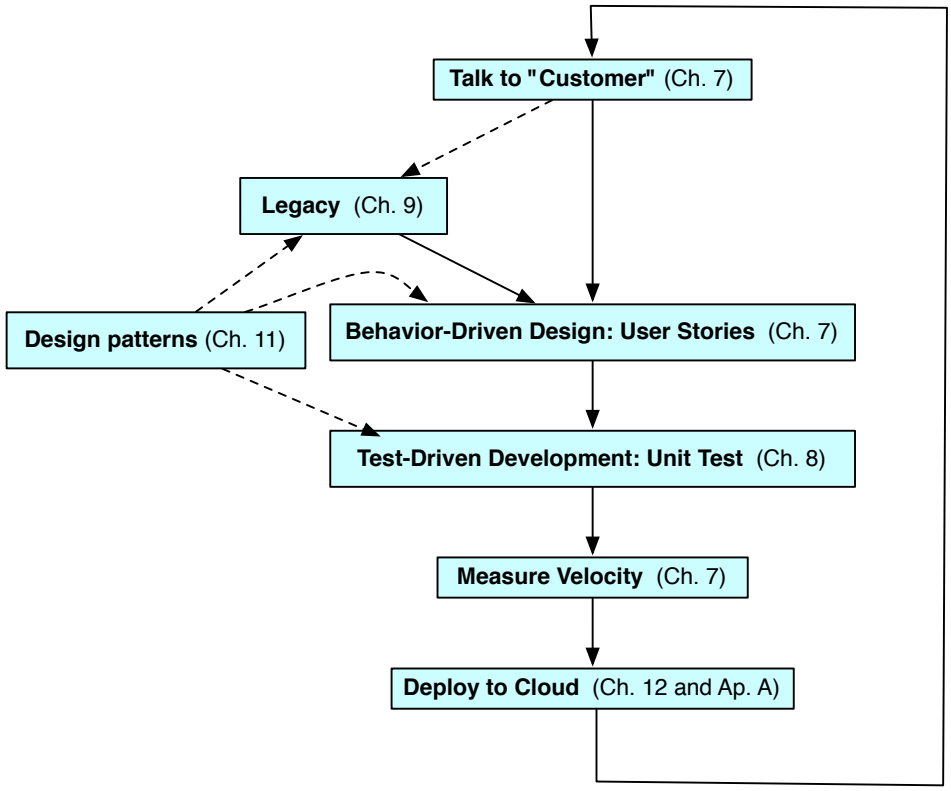
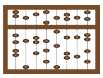


Figure 1.8: An iteration of the Agile software lifecycle and its relationship to the chapters in this book. The dashed arrows indicate a more tangential relationship between the steps of an iteration, while the solid arrows indicate the typical flow. Note that the Agile process applies equally well to existing legacy applications and new applications, although the customer may play a smaller role with legacy apps.

Chapter 7 discusses how to talk to the customer. ***Behavior-Driven Design (BDD)*** advocates writing tests that customers without a programming background can read, called ***user stories***, and Chapter 7 shows how to write user stories so that they can be turned into integration tests as well as acceptance tests. It introduces the ***Cucumber*** tool to help automate this task. This testing tool can be used with any language and framework, not just Rails. As SaaS apps are often user facing, the chapter also covers how to prototype a useful user interface using "Lo-Fi" prototyping. It also explains the term ***Velocity*** and how to use it to measure progress in the rate that you deliver features, and introduces the SaaS-based tool ***Pivotal Tracker*** to simplify such measurements.

Chapter 8 covers ***Test-Driven Development (TDD)***. The chapter demonstrates how to write good, testable code and introduces the ***RSpec*** testing tool for writing unit tests, the ***Autotest*** tool for automating test running, and the ***SimpleCov*** tool to measure test coverage.



Chapter 9 describes how to deal with existing code, including how to enhance legacy code. Helpfully, it shows how to use BDD and TDD to both understand and refactor code and how to use the Cucumber and RSpec tools to make this task easier.

Chapter 10 gives advice on how to organize and work as part of an effective team versus doing it all by yourself. It also describes how the version control system ***Git*** and the corresponding services ***GitHub*** and ***ProjectLocker*** can let team members work on different features without interfering with each other or causing chaos in the release process.

To help you practice Don't Repeat Yourself, Chapter 11 introduces design patterns, which are proven structural solutions to common problems in designing how classes work together, and shows how to exploit Ruby's language features to adopt and reuse the patterns. The chapter also offers guidelines on how to write good classes. It introduces just enough UML (Unified Modeling Language) notation to help you notate design patterns and to help you make diagrams that show how the classes should work.

Note that Chapter 11 is about software architecture whereas prior chapters in Part II are about the Agile development process. We believe in a college course setting that this order will let you start an Agile iteration sooner, and we think the more iterations you do, the better you will understand the Agile lifecycle. However, as Figure 1.8 suggests, knowing design patterns will be useful when writing or refactoring code that is fundamental of the BDD/TDD process.

Chapter 12 offers practical advice on how to first deploy and then improve performance and scalability in the cloud, and briefly introduces some reliability and security techniques that are uniquely relevant to deploying SaaS.

We conclude with an Afterword that reflects on the material in the book and projects what might be next.

1.11 How *NOT* to Read this Book

Don't skip the screencasts. The temptation is to skip sidebars, elaborations, and screencasts to just skim the text until you find what you want to answer your question.

While elaborations are typically for experienced readers who want to know more about what is going on behind the curtain, and sidebars are just short asides that we think you'll enjoy, screencasts are *critical* to learning this material. While we wouldn't say you could skip the text and just watch the screencasts, we would say that they are some of the most important parts of the book. They allow us to express a lot of concepts, show how they interact, and demonstrate how you can do the same tasks yourself. What would take many pages and be

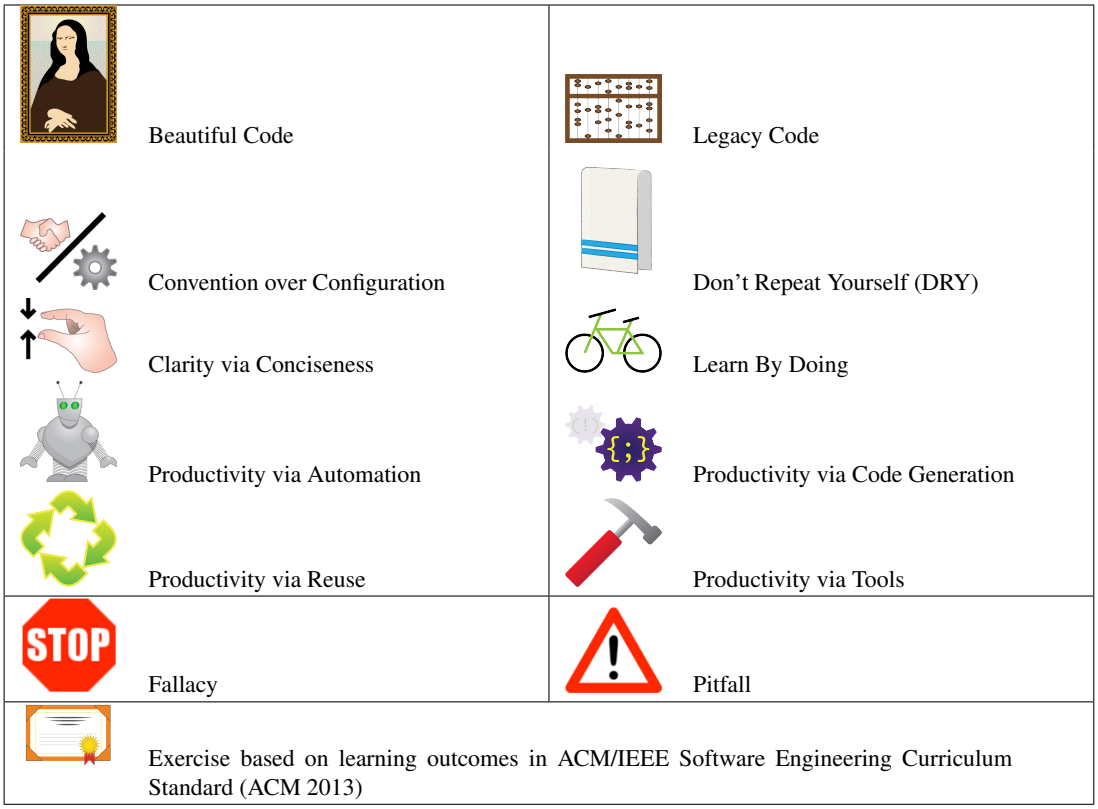


Figure 1.9: Summary of icons used in the book.

difficult to describe can come alive in a two to five minute video. Screencasts allow us to follow the advice of Confucius: "I see and I remember." So please watch them!

Learn by doing. Have your computer open with the Ruby interpreter ready so that you can try the examples in the screencasts and the text. We even make it easy to copy-and-paste the code using the service Pastebin[15], (If you're reading the ebook, the link accompanying each code example will take you to that code example on Pastebin.) This practice follows the advice of Confucius: "I do and I understand." Specific opportunities to learn by doing are highlighted by a bicycle icon.

There are topics that you will need to study to learn, especially in our buzzword-intensive ecosystem of Agile + Ruby + Rails + SaaS + Cloud Computing. Indeed, Figure 13.2 in the Afterword lists nearly 120 new terms introduced in just the first three chapters. To help you identify important terms, text formatted ***like this*** refers to terms with corresponding Wikipedia entries. (In the ebook, such terms are links to Wikipedia itself.) We also use icons to remind you of the common themes throughout the book, which Figure 1.9 summarizes as a single handy place to look them up.

Depending on your background, we suspect you'll need to read some chapters more than once before you get the hang of it. To help you focus on the key points, each chapter starts with a 1-page ***Concepts*** summary, which lists the big concepts of each chapter, and each chapter ends with ***Fallacies and Pitfalls***, which explains common misconceptions or prob-

lems that are easy to experience if you're not vigilant. Each section concludes with a ***summary*** of the key concepts in that section and ***self-check questions*** with answers. ***Projects*** at the end of each chapter are more open-ended than the self-check questions. To give readers a perspective about who came up with these big ideas that they are learning and that information technology relies upon, we use sidebars to introduce 20 Turing Award winners. (As there is no Nobel Prize in IT, our highest honor is known as the "Nobel Prize of Computing".)

We deliberately chose to keep the book concise, since different readers will want additional detail in different areas. Links are provided to the Ruby and Rails online documentation for built-in classes or methods, to definitions of important concepts you may be unfamiliar with, and to the Web in general for further reading related to the material. If you're using the Interactive Edition or the Kindle edition, the links should be live if you're connected to the Internet; in the print version, the link URIs appear at the end of each chapter.

1.12 Fallacies and Pitfalls

> *Lord, give us the wisdom to utter words that are gentle and tender, for tomorrow we may have to eat them.*
>
> —Sen. Morris Udall

We include this section near the end of chapters to explain the ideas once again from another perspective and to give readers a chance to learn from the mistakes of others. ***Fallacies*** are statements that seem plausible (or are actually widely held views) based on the ideas in the chapter, but they are not true. ***Pitfalls***, on the other hand, are common dangers associated with the topics in the chapter that are difficult to avoid even when you are warned.

Fallacy: **The Agile lifecycle is best for software development.**

Agile is a nice match to some types of software, particularly SaaS, which is why we use it in this book. However, Agile is *not* best for all software. Agile is inappropriate for safety-critical apps, for example.

Our experience is that once you learn the classic steps of software development and have a positive experience in using them via Agile, you will use these important software engineering principles in other projects no matter which methodology is used.

Nor will Agile be the last software lifecycle you will ever see. We believe that new development methodologies develop and become popular in response to new opportunities, so expect to learn new methodologies and frameworks in the future.

Pitfall: **Ignoring the cost of software design.**

Since there is no cost to manufacture software, the temptation is to believe there is almost no cost to changing it so that it can be remanufactured the way the customer wants. However, this perspective ignores the cost of design and test, which can be a substantial part of the overall costs for software projects. Zero manufacturing costs is also one rationalization used to justify pirating copies of software and other electronic data, since pirates apparently believe no one should pay for the cost of development, just for manufacturing.

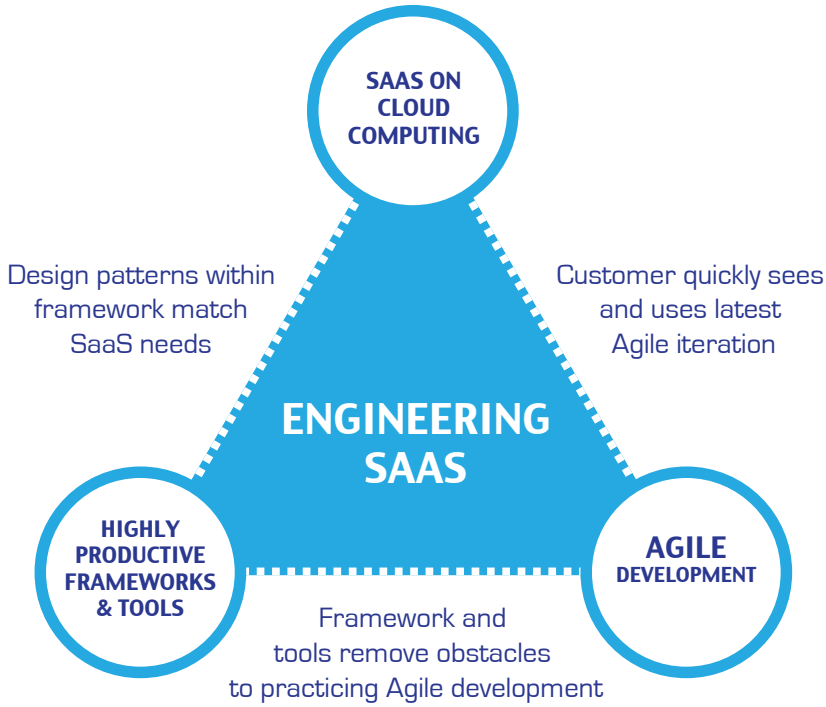


Figure 1.10: The Virtuous Triangle of Engineering SaaS is formed from the three software engineering crown jewels of (1) SaaS on Cloud Computing, (2) Agile Development, and (3) Highly Productive Framework and Tools.

1.13 Concluding Remarks: Engineering Software is More Than Programming

> *But if Extreme Programming is just a new selection of old practices, what's so extreme about it? Kent's answer is that it takes obvious, common sense principles and practices to extreme levels. For example:*
> *— If short iterations are good, make them as short as possible–hours or minutes or seconds rather than days or weeks or years.*
> *— If simplicity is good, always do the simplest thing that could possibly work.*
> *— If testing is good, test all the time. Write the test code before you write the code to test.*
> *— If code reviews are good, review code continuously, by programming in pairs, two programmers to a computer, taking turns looking over each other's shoulders.*
>
> —Michael Swaine, interview with Kent Beck, (Swaine 2001)

You could develop SaaS apps using a Plan-and-Document methodology, use Agile to develop shrink-wrap application, or write them in any framework. While there is no inherent dependency among SaaS, Agile, and highly productive frameworks like Rails, Figure 1.10 suggests there is a synergistic relationship among them. Agile development means continuous progress while working closely with the customer, and SaaS on Cloud Computing enables the customer to use the latest version immediately, thereby closing the feedback loop (see Chapter 7). SaaS on Cloud Computing matches the Model–View–Controller design pat-

tern, which Highly-Productive SaaS Frameworks expose (see Chapter 2). Highly Productive Frameworks and Tools designed to support Agile development remove obstacles to practicing Agile (see Chapters 8 and 10). We believe these three "crown jewels" form a "virtuous triangle" that leads to the engineering of beautiful Software as a Service, and they form the foundation of this book.

This virtuous triangle also helps explain the innovative nature of the Rails community, where new important tools are frequently developed that further improve productivity, simply because it's so easy to do. We fully expect that future editions of this book will include tools not yet invented that are so helpful that we can't imagine how we got our work done without them!

As teachers, since many students find the Plan-and-Document methods tedious, we are pleased that the answers to the 10 questions in Figure 1.7 strongly recommend using Agile for student team projects. Nevertheless, we believe it is worthwhile for readers to be familiar with the Plan-and-Document methodology, as there are tasks for which it is clearly a better match and some companies follow it for all tasks, so we include sections near the end all the chapters in Part II that offer the Plan-and-Document perspective.

As researchers, we are convinced that software of the future will increasingly be built and rely on services in the Cloud, and thus Agile methodology will continue to increase in popularity given the synergy between them. Hence, we are at a happy point in technology where the future of software development is more fun both to learn and to teach. Highly productive frameworks like Rails let you can understand this valuable technology by *doing* in a remarkably short time. The main reason we wrote this book is to help more people become aware of and take advantage of this extraordinary opportunity.

We believe if you learn the contents of this book and use the "bookware" that comes with it, you can build your own (simplified) version of a popular software service like FarmVille or Twitter while learning and following sound software engineering practices. While being able to imitate currently successful services and deploy them in the cloud in a few months is impressive, we are even more excited to see what *you* will invent given this new skill set. We look forward to your beautiful code becoming long-lasting and to becoming some of its passionate fans!

1.14 To Learn More

Computer science curricula 2013, Ironman Draft (version 1.0). Technical report, The ACM IEEE-Computer Society Joint Task Force, February 2013. URL `http://ai.stanford.edu/users/sahami/CS2013/`.

C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language: Towns, Buildings, Construction (Cess Center for Environmental)*. Oxford University Press, 1977. ISBN 0195019199.

M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Communications of the ACM (CACM)*, 53(4):50–58, Apr. 2010.

L. A. Barroso and U. Hoelzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines (Synthesis Lectures on Computer Architecture)*. Morgan and

Claypool Publishers, 2009. ISBN 159829556X. URL `http://www.morganclaypool.com/doi/pdf/10.2200/S00193ED1V01Y200905CAC006`.

B. W. Boehm. Software engineering: R & D trends and defense needs. In P. Wegner, editor, *Research Directions in Software Technology*, Cambridge, MA, 1979. MIT Press.

B. W. Boehm. A spiral model of software development and enhancement. In *ACM SIGSOFT Software Engineering Notes*, 1986.

E. Braude. *Software Engineering: An Object-Oriented Perspective*. John Wiley and Sons, 2001. ISBN 0471692085.

R. Charettte. Why software fails. *IEEE Spectrum*, 42(9):42–49, September 2005.

M. Cormick. Programming extremism. *Communications of the ACM*, 44(6):109–110, June 2001.

H.-C. Estler, M. Nordio, C. A. Furia, B. Meyer, and J. Schneider. Agile vs. structured distributed software development: A case study. In *Proceedings of the 7th International Conference on Global Software Engineering (ICGSE'12))*, pages 11–20, 2012.

M. Fowler. The New Methodology. *martinfowler.com*, 2005. URL `http://www.martinfowler.com/articles/newMethodology.html`.

H. Goldstein. Who killed the virtual case file? *IEEE Spectrum*, 42(9), September 2005.

J. Johnson. The CHAOS report. Technical report, The Standish Group, Boston, Massachusetts, 1995. URL `http://blog.standishgroup.com/`.

J. Johnson. The CHAOS report. Technical report, The Standish Group, Boston, Massachusetts, 2009. URL `http://blog.standishgroup.com/`.

C. Jones. Software project management practices: Failure versus success. *CrossTalk: The Journal of Defense Software Engineering*, pages 5–9, Oct. 2004. URL `http://cross5talk2.squarespace.com/storage/issue-archives/2004/200410/200410-Jones.pdf`.

J. M. Juran and F. M. Gryna. *Juran's quality control handbook*. New York: McGraw-Hill, 1998.

P. Kruchten. *The Rational Unified Process: An Introduction, Third Edition*. Addison-Wesley Professional, 2003. ISBN 0321197704.

T. Lethbridge and R. Laganiere. *Object-Oriented Software Engineering: Practical Software Development using UML and Java*. McGraw-Hill, 2002. ISBN 0072834951.

P. Naur and B. Randell. *Software engineering*. Scientific Affairs Div., NATO, 1969.

J. R. Nawrocki, B. Walter, and A. Wojciechowski. Comparison of CMM level 2 and extreme programming. In *7th European Conference on Software Quality*, Helsinki, Finland, 2002.

M. Paulk, C. Weber, B. Curtis, and M. B. Chrissis. *The Capability Maturity Model: Guidelines for Improving the Software Process*. Addison-Wesley, 1995. ISBN 0201546647.

M. Rosenblum and T. Garfinkel. Virtual machine monitors: Current technology and future trends. *Computer*, 38(5):39–47, 2005.

W. W. Royce. Managing the development of large software systems: concepts and techniques. In *Proceedings of WESCON*, pages 1–9, Los Angeles, California, August 1970.

I. Sommerville. *Software Engineering, Ninth Edition*. Addison-Wesley, 2010. ISBN 0137035152.

M. Stephens and D. Rosenberg. *Extreme Programming Refactored: The Case Against XP*. Apress, 2003.

M. Swaine. Back to the future: Was Bill Gates a good programmer? What does Prolog have to do with the semantic web? And what did Kent Beck have for lunch? *Dr. Dobb's The World of Software Development*, 2001. URL `http://www.drdobbs.com/back-to-the-future/184404733`.

A. Taylor. IT projects sink or swim. *BCS Review*, Jan. 2000. URL `http://archive.bcs.org/bulletin/jan00/article1.htm`.

Notes

[9] `http://en.wikipedia.org/wiki/Turing_Award`
[10] `http://www.youtube.com/watch?v=kYUrqdUyEpI`
[11] `https://www.edx.org/`
[12] `https://plus.google.com/112678702228711889851/posts/eVeouesvaVX`
[13] `http://developers.slashdot.org/story/08/05/11/1759213/`
[14] `http://www.youtube.com/watch?v=DeBi2ZxUZiM`
[15] `http://www.pastebin.com/u/saasbook`

1.15 Suggested Projects

Project 1.1. *(Discussion) In your opinion, how would you rank the software disasters in the first section from most terrible to the least? How did you rank them?*

Project 1.2. *(Discussion) Identify the principal issues associated with software evolution and explain their impact on the software life cycle. Note: We use this margin icon to identify all projects that from come from the ACM/IEEE 2013 Computer Science Curriculum for Software Engineering standard (ACM 2013).*

Project 1.3. *(Discussion) Discuss the challenges of evolving systems in a changing environment.*

Project 1.4. *(Discussion) Explain the concept of a software life cycle and provide an example, illustrating its phases including the deliverables that are produced.*

Project 1.5. *(Discussion) Referring to Figure 1.7, compare the process models from this chapter with respect to their value for development of particular classes of software systems: information management, embedded, process control, communications, and web applications.*

Project 1.6. *(Discussion) The closest hardware failure to the software disasters mentioned in the first section is probably the Intel Floating Point Divide bug*[1]*. Where would you put this hardware problem in the ranked list of software examples from the exercise above?*

Project 1.7. *(Discussion) Measured in lines of code, what is the largest program in the world? For purposes of this exercise, assume it can be a suite of software that is shipped as a single product.*

Project 1.8. *(Discussion) Which programming language has the most active programmers?*

Project 1.9. *(Discussion) In which programming language is the most number of lines of code written annually? Which has the most lines of active code cumulatively?*

Project 1.10. *(Discussion) Make a list of, in your opinion, the Top 10 most important applications. Which would best be developed and maintained using the four lifecycles from this chapter? List your reasons for each choice.*

Project 1.11. *(Discussion) Given the list of Top 10 applications from the exercise above, how important are each of the four productivity techniques listed in the chapter?*

Project 1.12. *Distinguish between program validation and program verification.*

Project 1.13. *(Discussion) Given the list of Top 10 applications from the exercise above, what aspects might be difficult to test and need to rely on formal methods? Would some testing techniques be more important for some applications than others? State why.*

Project 1.14. *(Discussion) What are the Top 5 reasons that SaaS and Cloud Computing will grow in popularity and the Top 5 obstacles to its growth?*

Project 1.15. *(Discussion) Discuss the advantages and disadvantages of software reuse.*

Project 1.16. *(Discussion) Describe and distinguish among the different types and levels of testing (unit, integration, module, system, and acceptance).*

Project 1.17. *Describe the difference between principles of the waterfall model and Plan-and-Document models using iterations.*

Project 1.18. *Describe the different practices that are key components of Agile and various Plan-and-Document process models.*

Project 1.19. *Differentiate among the phases of software development of Plan-and-Document models.*

YOUR COMPLIMENTARY CHAPTER