Chapter 1

Introduction

Decade after decade, software systems have seen orders-of-magnitude increases in their size and complexity. This is remarkable — and more than a little scary for those of us who build software. In contrast, imagine how hard basketball would be if it scaled up the same way, with 5 people on the floor one decade, then 50, then 500. Because of this growth, today's software systems are arguably the largest and most complex things ever built.

Software developers are always battling the ever-stronger foes of complexity and scale, but even though their opponent grows in strength, developers have staved off defeat and even reveled in victory. How have they done this?

One answer is that the increases in software size and complexity have been matched by advances in software engineering. Assembly language programming gave way to higher-level languages and structured programming. Procedures have, in many domains, given way to objects. And software reuse, which used to mean just subroutines, is now also done with extensive libraries and frameworks.

It is no coincidence that the battle between developers and software complexity always seems to be at a stalemate. Since developers cannot grow bigger brains, they have instead improved their weapons. An improved weapon gives developers two options: to more easily conquer yesterday's problems, or to combat tomorrow's. We are no smarter than developers of the previous generation, but our improved weapons allow us to build software of greater size and complexity.

Software developers wield some tangible weapons, such as Integrated Development Environments (IDEs) and programming languages, but intangible weapons arguably make a bigger impact. Returning to our basketball metaphor, consider a coach

and a rookie (a novice sports player) watching the same game. The coach sees more than the rookie — not because the coach's eyes are more acute, but because he has an intangible weapon. He has built up a set of mental abstractions that allow him to convert his perceptions of raw phenomena, such as a ball being passed, into a condensed and integrated understanding of what is happening, such as the success of an offensive strategy. The coach watches the same game that the rookie does, but he understands it better. Alan Kay has observed that your "point of view is worth 80 IQ points" (Kay, 1989).

Software is similar in that there are lots of low-level details. If developers have built up a set of mental abstractions, they can convert those details into a condensed understanding: where before they saw just code, perhaps they now see a thread-safe locking policy or an event-driven system.

1.1 Partitioning, knowledge, and abstractions

To be successful at combating the scale and complexity of software in the next decade, developers will need improved weapons. Those weapons can be categorized, perhaps with a bit of shoehorning, into three categories: partitioning, knowledge, and abstraction. Developers *partition* a problem so that its parts are smaller and more tractable, they apply *knowledge* of similar problems, and they use *abstractions* to help them reason. Partitioning, knowledge, and abstraction are effective because they enable our fixed-sized minds to comprehend an ever-growing problem.

- **Partitioning**. Partitioning is effective as a strategy to combat complexity and scale when two conditions are true: first, the divided parts must be sufficiently small that a person can now solve them; second, it must be possible to reason about how the parts assemble[1] into a whole. Parts that are encapsulated are easier to reason about, because you need to track fewer details when composing the parts into a solution. You can forget, at least temporarily, about the details inside the other parts. This allows the developer to more easily reason about how the parts will interact with each other.

- **Knowledge**. Software developers use knowledge of prior problems to help them solve current ones. This knowledge can be implicit know-how or explicitly written down. It can be specific, as in which components work well with others, or general, as in techniques for optimizing a database table layout. It comes in many forms, including books, lectures, pattern descriptions, source code, design documents, or sketches on a whiteboard.

[1] Mary Shaw has remarked that when dividing and conquering, the dividing is the easy part.

- **Abstraction**. Abstraction can effectively combat complexity and scale because it shrinks problems, and smaller problems are easier to reason about. If you are driving from New York to Los Angeles, you can simplify the navigation problem by considering only highways. By hiding details (excluding the option of driving across fields or parking lots), you have shrunken the number of options to consider, making the problem easier to reason about.

You should not expect any *silver bullets*, as Fred Brooks called them, that will suddenly eliminate the difficulties of software development (Brooks, 1995). Instead, you should look for weapons that help you partition systems better, provide knowledge, and enable abstraction to reveal the essence of the problem.

Software architecture is one such weapon and it can help you address the complexity and scale of software systems. It helps you *partition* software, it provides *knowledge* that helps you design better software, and it provides *abstractions* that help you reason about software. It is a tool in the hands of a skilled developer. It helps software developers to routinely build systems that previously required virtuosos (Shaw and Garlan, 1996), but it does not eliminate the need for skilled software developers. Instead of removing the need for ingenuity, it allows developers to apply their ingenuity to build bigger and more complex systems.

1.2 Three examples of software architecture

That is what software architecture *does* for you, but what *is* it? Roughly speaking, architecture is the macroscopic design of a software system. Chapter 2 of this book discusses a more careful definition, but perhaps it is best to understand software architecture using some concrete examples first.

It can be hard to "see the forest for the trees", which in this case means finding the architecture amidst the design details. But by comparing multiple similar systems with different architectures you should be able to notice what is different and therefore identify their architectures. What follows is a description of three systems with the same functionality, yet different architectures, based on the experiences at Rackspace.

Rackspace is a real company that manages hosted email servers. Customers call up for help when they experience problems. To help a customer, Rackspace must search the log files that record what has happened during the customer's email processing. Because the volume of emails they handle kept increasing, Rackspace built three generations of systems to handle the customer queries (Hoff, 2008b; Hood, 2008).

Version 1: Local log files. The first version of the program was simple. There were already dozens of email servers generating log files. Rackspace wrote a script that

would use ssh to connect to each machine and execute a grep[2] query on the mail log file. Engineers could control the search results by adjusting the grep query.

This version initially worked well, but over time the number of searches increased and the overhead of running those searches on the email servers became noticeable. Also, it required an engineer, rather than a support tech, to perform the search.

Version 2: Central database. The second version addressed the drawbacks of the first one by moving the log data off of the email servers and by making it searchable by support techs. Every few minutes, each email server would send its recent log data to a central machine where it was loaded into a relational database. Support techs had access to the database data via a web-based interface.

Rackspace was now handling hundreds of email servers, so the volume of log data had increased correspondingly. Rackspace's challenge became how to get the log data into the database as quickly and efficiently as possible. The company settled on bulk record insertion into merge tables, which enabled loading of the data in two or three minutes. Only three days worth of logs were kept so that the database size would not hinder performance.

Over time, this system also encountered problems. The database server was a single machine and, because of the constant loading of data and the query volume, it was pushed to its limit with heavy CPU and disk loads. Wildcard searches were prohibited because of the extra load they put on the server. As the amount of log data grew, searches became slower. The server experienced seemingly random failures that became increasingly frequent. Any log data that was dropped was gone forever because it was not backed up. These problems led to a loss of confidence in the system.

Version 3: Indexing cluster. The third version addressed the drawbacks of the second by saving log data into a distributed file system and by parallelizing the indexing of log data. Instead of running on a single powerful machine, it used ten commodity machines. Log data from the email servers streamed into the Hadoop Distributed File System, which kept three copies of everything on different disks. In 2008, when Rackspace wrote a report on its experiences, it had over six terabytes of data spanning thirty disk drives, which represented six months of search indexes.

Indexing was performed using Hadoop, which divides the input data, indexes (or "maps") in jobs, then combines (or "reduces") the partial results into a complete index. Jobs ran every ten minutes and took about five minutes to complete, so index results were about fifteen minutes stale. Rackspace was able to index over 140 gigabytes of log data per day and had executed over 150,000 jobs since starting the system.

[2]Grep is a command-line utility that performs regular expression queries.

As in the second system, support techs had access via a web interface that was much like a web search-engine interface. Query results were provided within seconds. When engineers thought up new questions about the data, they could write a new kind of job and have their answer within a few hours.

1.3 Reflections

The first thing to notice from looking at these three systems is that they all have roughly the same functionality (querying email logs to diagnose problems) yet they have different architectures. *Their architecture was a separate choice from their functionality.* This means that when you build a system, you can choose an architecture that best suits your needs, then build the functionality on that architectural skeleton. What else can these systems reveal about software architecture?

Quality attributes. Despite having the same functionality, the three systems differ in their modifiability, scalability, and latency. For example, in the first and second version, ad hoc queries could be created in a matter of seconds, either by changing the grep expression used for the search or by changing the SQL query. The third system requires a new program to be written and scheduled before query results can be obtained. All three support creating new queries, but they differ in how easy it is to do (modifiability).

Notice also that there was no free lunch: promoting one quality inhibited another. The third system was much more scalable than the other two, but its scalability came at the price of reduced ability to make ad hoc queries and a longer wait before results were available. The data in the first system was queryable online (and the second could perhaps be made nearly so), but the third system had to collect data then run a batch process to index the results, which means that query results were a bit stale.

If you are ever in a situation in which you can get great scalability, latency, modifiability, etc., then you should count yourself lucky, because such *quality attributes* usually trade off against each other. Maximizing one quality attribute means settling for less of the others. For example, choosing a more modifiable design may entail worse latency.

Conceptual model. Even without being a software architecture expert, you could read about these three systems and reason about their design from first principles. What advantage would come from being a software architecture expert (i.e., being a coach instead of a rookie)? Both coaches and rookies have an innate ability to reason, but the coach has a head start because of the *conceptual model* he carries in his head that helps him make sense of what he sees.

As an architecture expert, you would be primed to notice the partitioning differences in each system. You would distinguish between chunks of code (modules),

runtime chunks (components), and hardware chunks (nodes, or environmental elements). You would notice and already know the names of the architectural patterns each system employs. You would know which pattern is suited for achieving which quality attributes, so you would have predicted that the client-server system would have lower latency than the map-reduce system. You would sort out and relate domain facts, design choices, and implementation details — and notice when someone else has jumbled them together. Being an expert in software architecture helps you to use your innate reasoning abilities more effectively.

Abstractions and constraints. In software, bigger things are usually built out of smaller things. You can always reason about the smaller things in a system (like individual lines of code) but usually you will find it more efficient to reason about the larger things (like clients and servers). For example, the third system in the Rackspace example scheduled jobs and stored data in a distributed file system. If you needed to solve a problem about how jobs flow through that system, then it would be most efficient to reason about the problem at that level. You could begin your reasoning process by considering the small bits, like objects and procedure calls, but that would be inefficient and likely to swamp you with details.

Furthermore, a "job" is an abstraction that obeys more constraints than an arbitrary "chunk of code" does. The developers imposed those constraints to make reasoning about the system easier. For example, if they constrain jobs to have no side effects, they can run the same job twice, in parallel, just in case one becomes bogged down. It is hard to reason about an arbitrary chunk of code specifically because you cannot tell what it *does not* do. Developers voluntarily impose constraints on the system in order to amplify their reasoning abilities.

1.4 Perspective shift

In 1968 Edsger Dijkstra wrote a now famous letter titled "GOTO Considered Harmful" advocating the use of structured programming. His argument is roughly as follows: Developers build programs containing static statements that execute to produce output. Developers, being human, have a hard time envisioning how the static statements of a program will execute at runtime. GOTO statements complicate the reasoning about runtime execution, so it is best to avoid GOTO statements and embrace structured programming.

Looking back at this debate today, it is hard to imagine disagreeing strongly, but at the time the resistance was substantial. Developers were accustomed to working within the old set of abstractions. They focused on the constraints of the new abstractions rather than the benefits and objected based on corner cases that were hard to express using structured programming. Each similar increase in abstraction is opposed by some who are familiar with the old abstractions. During my programming

career, I have seen developers resist abstract data types and object-oriented programming, only to later embrace them.

New architecture abstractions rarely replace old abstractions, but instead coexist with them. Using abstractions such as components and connectors does not mean that objects, methods, and data structures disappear. Similarly, forest fire fighters switch between thinking about individual trees or entire forests depending on what part of their job they are doing at the moment.

Effectively applying software architecture ideas requires a conscious and explicit shift to embrace its abstractions, such as components and connectors, rather than only using the abstractions found in mainstream programming language (often just classes or objects). If you do not consciously choose the architecture of your systems, then it may end up being what Brian Foote and Joseph Yoder call a *big ball of mud* (Foote and Yoder, 2000), which they estimate is the most common software architecture. It is easy to understand this architecture: Imagine what a system with 10 classes would look like, then scale it up to 100, 1000, ..., without any new abstractions or partitions, and let the objects in the system communicate with each other as is expedient.

1.5 Architects architecting architectures

I have sometimes seen software developers design systems with beautiful architectures, then voice their resistance to software architecture. This resistance may stem from resistance to bureaucracy-intensive up-front design processes, to pompous architects, or to having been forced to waste time making diagrams instead of systems. Fortunately, none of these problems need to be a part of creating software architecture.

Job titles, development processes, and engineering artifacts are separable, so it is important to avoid conflating the *job title* "architect", the *process* of architecting a system, and the *engineering artifact* that is the software architecture.

- **The job role: architect.** One possible job title (or role) in an organization is that of a software architect. Some architects sit in corner offices and make pronouncements that are disconnected from the engineering reality, while other architects are intimately involved in the ongoing construction of the software. Either way, the title and the office are not intrinsic to the work of designing or building software. All software developers, not just architects, should understand their software's architecture.

- **The process: architecting.** There is no software at the beginning of a project, but by the end of the project there is a running system. In between, the team performs activities (i.e., they follow a process) to construct the system. Some teams design up-front and other teams design as they build. The process that

the team follows is separable from the design that emerges. A team could follow any number of different processes and produce, for example, a 3-tier system. Or put another way, it is nearly impossible to tell what architecting process a team followed by looking only at the finished software.

- **The engineering artifact: the architecture.** If you look at an automobile, you can tell what type of car it is, perhaps an all-electric car, a hybrid car, or an internal combustion car. That characteristic of the car is distinct from the process followed to design it, and distinct from the job titles used by its designers. The car's design is an engineering artifact. Different choices about process and job titles could still result in their creating, for example, a hybrid car. Software is similar. If you look at a finished software system, you can distinguish various designs; for example: collaborating peer-to-peer nodes in a voice-over-IP network, multiple tiers in information technology (IT) systems, or parallelized map-reduce compute nodes in internet systems. Every software system has an architecture just as every car has a design. Some software is cobbled together without a regular process, yet its architecture is still visible.

This book discusses process in Chapter 2 and Chapter 3. The rest of the book treats architecture as an engineering artifact: something to be analyzed, understood, and designed so that you can build better systems.

1.6 Risk-driven software architecture

Different developers have had success with different processes. Some succeeded with agile processes that have little or no up-front work. Others succeeded with detailed up-front design work. Which should you choose? Ideally, you would have a guiding principle that would help you choose appropriately.

This book suggests using *risk* to decide how much architecture work to do. One way to understand how risk can guide you to good choices about architecture it is to consider the story of my father installing a mailbox.

My father has two degrees in mechanical engineering, but when he put up a mailbox he did it like anyone else would: he dug a hole, put in the post, and filled the hole with some cement. Just because he could calculate moments, stresses, and strains does not mean he must or should. In other situations it would be foolish for him to skip these analyses. How did he know when to use them?

Software architecture is a relatively new technology, and it includes many techniques for modeling and analyzing systems. Yet each of these techniques takes time that could otherwise be spent building the system. This book introduces the *risk-driven model* for software architecture, which guides you to do just enough archi-

tecture by selecting appropriate architecture techniques and knowing when you can stop.

Your effort should be commensurate with your risk of failure. Perhaps your system has demanding scalability requirements because you run a popular web service. It would be best to ensure that your system will handle the expected number of users[3] before you invest too much energy in its design (or the site goes live). If modifiability is less of a concern for your system (or usability, etc.), you would spend little time worrying about that risk.

Each project faces different risks, so there is no single correct way to do software architecture: you must evaluate the risks on each project. Sometimes the answer is to do no architecture work, because some projects are so highly precedented that there is almost no risk as long as you reuse a proven architecture. However, when you work in novel domains or push an existing system into uncharted territory you will want to be more careful.

The idea of consistently working to reduce engineering risks echoes Barry Boehm's *spiral model* of software development (Boehm, 1988). The spiral model is a full software development process that guides projects to work on the highest risk items first. Projects face both management and engineering risks, so managers must prioritize both management risks (like the risk of customer rejection) and engineering risks (like the risk that the system is insecure or inefficient).

Compared to the spiral model, the risk-driven model helps you answer narrower questions: How much architecture work should you do, and what kind of architecture techniques should you use? Because the risk-driven model only applies to design work, it means that it can be applied to agile processes, waterfall processes, spiral processes, etc. Regardless of process, you must design the software — the difference is when design happens and which techniques are applied.

1.7 Architecture for agile developers

Agile software development is a reaction to heavyweight development processes and it emphasizes efficiently building products that customers want (Beck et al., 2001). It is increasingly popular, with one study showing 69% of companies trying it on at least some of their projects (Ambler, 2008).

In their desire to cut out unnecessary steps in software development, some agile developers believe they should avoid software architecture techniques. This reluctance is not universal, as many important voices in agile community support some planned design work, including Martin Fowler, Robert Martin, Scott Ambler, and Granville Miller (Fowler, 2004; Martin, 2009; Ambler, 2002; Miller, 2006). Refactor-

[3] Recall that before the popular Facebook and MySpace social networking sites there was Friendster, but it could not handle the rush of users and became too slow to use.

ing a poor architecture choice can be prohibitively expensive on large systems. This book can help agile developers to use software architecture in a way that is consistent with agile principles for two primary reasons:

- **Just enough architecture.** The risk-driven model of architecture guides developers to do just enough architecture then resume coding. If you only foresee risks on your project that can be handled by refactoring then you would not do any architecture design. But if you are an agilist who is also concerned that refactoring may not be enough to get the security or scalability you need, now you have a way to mitigate those risks — even in your Nth iteration — then get back to coding.
- **Conceptual model.** Agile's primary contribution is to software development process, not design abstractions, and offers a limited number of techniques (such as refactoring and spikes) to produce good designs. The contents of this book augment agile processes by providing a conceptual model to reason about a system's architecture and design, a set of software design and modeling techniques, and expert software architecture knowledge.

This is not a book specifically about agile software architecture, but you will find that its risk-driven approach is well suited to agile projects. In particular, Section 3.11 provides a sketch of how to integrate risks into an iterative, feature-centric process.

1.8 About this book

This book focuses on software architecture as it relates to the *construction of software*, and describes the techniques used to ensure that software satisfies its engineering demands. This book is largely process agnostic because the engineering techniques themselves are largely process agnostic. You will not find advice on management activities like the political responsibilities of architects, when to hold specific kinds of meetings, or how to gather requirements from stakeholders.

This book is divided into two parts. This first part introduces software architecture and the risk-driven approach. The second part helps you build up a mental conceptual model of software architecture and describes in detail the abstractions like components and connectors. What follows is short summaries of each part.

Part I: Risk-driven software architecture

A definition of software architecture is difficult to pin down precisely, but several things about it are quite clear. Software developers, like engineers in other specialties, use abstraction and models to solve large and complex problems. Software architecture acts as the skeleton of a system, influences quality attributes, is orthogonal

to functionality, and uses constraints to influence a system's properties. Architecture is most important when the solution space is small, the failure risks are high, or you face difficult quality attribute demands. You can choose from architecture-indifferent design, architecture-focused design, or even architecture hoisting.

Risks can be used to guide you regarding which design and architecture techniques you should use and regarding how much design and architecture you should do. At its core, the risk-driven model is simple: (1) identify and prioritize risks, (2) select and apply a set of techniques, and (3) evaluate risk reduction.

To reveal the risk-driven model in practice, Chapter 4 provides an example of applying the risk-driven model to a Home Media Player system. The developers on that system have the challenges of team communication, integration of COTS components, and ensuring metadata consistency.

The first part of the book concludes with advice on using models and software architecture, including: use models to solve problems, add constraints judiciously, focus on risks, and distribute architecture skills throughout your team.

Part II: Architecture modeling

The second part of the book helps you build a mental conceptual model of software architecture. It starts with the canonical model structure: the domain model, the design model, and the code model. The domain model corresponds to things in the real world, the design model is the design of the software you are building, and the code model corresponds to your source code. You can build additional models that show selected details, called views, and these views can be grouped into viewtypes.

Building encapsulation boundaries is a crucial skill in software architecture. Users of a component or module can often ignore how it works internally, freeing their minds to solve other hard problems. And the builders of an encapsulated component or module have the freedom to change its implementation without perturbing its users. Builders will only have that freedom if the encapsulation is effective, so this book will teach you techniques for ensuring that it is effective.

A great number of architectural abstractions and modeling techniques have been built up over the years. This book consolidates software architecture techniques found in a variety of other sources, integrating techniques emphasizing quality attributes as well as techniques emphasizing functionality. It also discusses pragmatic ways to build effective models and debug them.

The second part of the book concludes with advice on how to use the models effectively. Any book that covered the advantages but not the pitfalls of a technology should not be trusted, so it also covers problems that you are likely to encounter. By the end of the second part, you should have built up a rich conceptual model of abstractions and relationships that will help you see software systems the way a coach sees a game.