

---

## Chapter 1

# Introduction

---

When a coach and a rookie watch the same game, the coach sees more than the rookie – not because the coach’s eyes are more acute, but because 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 understanding of what is happening, such as the success of an offensive strategy. He sees the same game that the rookie does, but he understands it better. A developer who understands software architecture sees not only lines of code but also larger abstractions like components, styles, constraints, and quality attributes.

What developers see and understand is different than the software development process they follow. A small project team may all work in the same room and occasionally sketch on whiteboards, while a large project may be distributed and need to publish documents that describe their designs. But both teams need to understand the core software architecture abstractions and be able to make design decisions that reconcile opposing design forces. Both teams will understand that any kind of engineering, including software engineering, requires understanding the risks of failure and how to apply appropriate techniques to mitigate those risks.

Software architecture does not require lots of up-front designing. Software architecture is a way of understanding and structuring the system to achieve engineering objectives. That architectural understanding is applied when the software is designed and built, whether the project management is iterative, waterfall, or something else. Developers following the tightest iterative process still design a solution then build code. Developers with architectural understanding ensure that what they build at each step is consistent with their previous architectural choices, such as constraints that enable better security or architectural styles that enable concurrent processing.

If your project is small and simple then almost any management process or engineering technique can yield success. As projects get larger and more complex, the risks of failure grow and success requires skilled developers following sensible processes and employing good engineering techniques. Smaller projects do fine with lower-level abstractions, such as objects and methods, but larger and more complex projects will benefit from more ab-

stract concepts such as components, connectors, and ports. This book describes how to efficiently use software architecture concepts and techniques to address risks faced by your project.

## 1.1 Foes: complexity and scale

Modern software systems are arguably the largest and most complex things ever built. There are no obvious limits to our desires for ever-grander software, so software developers must fight the foes of complexity and scale. This battle always seems to be a stalemate, because with our ever more powerful weapons we attack ever more difficult problems.

We can learn how this battle has been fought by looking at the evolution of the abstractions that were invented as weapons. The earliest abstractions included state machines, flowcharts, and message sequence diagrams, and were invented in the earliest days of computing. Petri nets, abstract data types, specification languages like VDM and Z, objects, and module interconnection languages followed. More recently, the Unified Modeling Language (UML) and architecture description languages raised the abstractions further.

The trend of rising abstractions is clear, and each abstraction was invented to attack the problems of the day. It is hard to imagine, for example, module interconnection languages being invented before there were enough modules to be worth the effort. So this historical record of rising abstractions mirrors a trend of rising complexity and scale faced by software developers.

We can also learn from the evolution of software engineering publications. The 1950's saw the first description of the subroutine. In the 1960's, Donald Knuth published *The Art of Computer Programming*, followed closely by the NATO software conference where the term *software engineering* was coined and software development was declared to be in a crisis. In the 1970's, David Parnas described how to allocate responsibilities to modules based on encapsulation. Many inventions were heralded as *silver bullets* that would solve the software crisis, but Fred Brooks concluded that no silver bullets would be found based on his experience managing the OS/360 project at IBM. The 1990's introduced many books on software architecture and design patterns. Since 2000, the patterns have grown larger and cover patterns of enterprise architecture. Again we see a trend towards coping with increasing complexity and scale.

## 1.2 Weapons: partitioning, knowledge, and abstractions

Our primary strategy for combating complexity and scale is to use intelligent people – people who are good at working on large, complex problems. But even the most intelligent person with the best mind has limits, and modern software systems are devilishly large and complex, so getting good people is a good start, but not enough.

Software developers overcome the difficulties of complexity and scale with three primary strategies. They *partition* the problem so that the parts are smaller and more tractable, they use *knowledge* of similar problems, and they use *abstractions* to help them reason. These three strategies help people work on large problems by allowing a part of the problem to fit in their heads.

## Partitioning

It is difficult to find software of even moderate size that lacks partitions. Partitioning can be seen in distinct chunks that run on separate machines, distinct compilation units, or distinct runtime components. Software varies considerably in how fine-grained the partitioning is, and the style of decomposition. Code libraries and frameworks are kinds of partitions that enable reuse of code. In object-oriented programming, each object is a partition.

Partitioning works 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 go together into a whole. Mary Shaw, a prominent software architecture researcher, has remarked that when dividing and conquering, the dividing is the easy part. Parts that are encapsulated make the reasoning easier, since fewer details need to be tracked when composing them into a solution. The developer can forget, at least temporarily, about the details inside each of those 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 with the current one. This knowledge can be explicitly written down or implicit as know-how. It can be specific, as in which components work well with others, or general, as in techniques for optimizing a database table layout. It can take many forms, including books, lectures, pattern descriptions, source code, design documents, or sketches on a whiteboard.

It would be great to know exactly how a software developer uses knowledge to solve problems, because if we did, then we could write a program to do the software developer's job. For now we have to simply admire what software developers can achieve. It is clear that knowledge does help them solve problems, and here is my intuition as to why.

When problems are small enough, a developer can just solve them without much thought, almost automatically or unconsciously, such as swapping the value of two variables. The more experience the developer has, the less thought the solution requires, and the more automatic it becomes. A problem that was tricky to solve the first time becomes progressively easier each time it is solved.

One can think of the developer's solution as "spanning the gap" between what exists now and what is needed to solve the problem. The developer spans the gap by designing parts that fill the gap. As the developer becomes more knowledgeable, the size of the part that he reasons about automatically becomes larger. His ability to solve the overall problem is a combination of what he can design automatically (or unconsciously) along with his explicit designing of the solution. So as the automatically generated parts get larger, there is less of a cognitive burden on his design skills, and he can solve larger problems.

## Abstractions

Abstraction is effective in combating complexity and scale because abstraction shrinks the problem and because smaller problems are easier to reason about. You can simplify the problem of driving from New York to Los Angeles by considering only highways. By excluding the option of driving across fields or parking lots, the number of options to consider has been reduced, making the problem easier to reason about. Similarly constraining your

driving to only paved roads, or only highways, further simplifies the problem and your reasoning.

Ideally, the details you elide by abstracting are irrelevant to solving the problem. Some roads have signposts made of wood, others concrete, and others metal – this choice can be safely abstracted when we consider the shortest path. However, the shortest path from New York to Los Angeles may include a road smaller than a highway. The use of abstraction may include a tradeoff: The problem is too difficult to reason about without abstraction, but reasoning with the abstraction is imperfect.

### 1.3 Partial solution: software architecture

If complexity and scale are the foes facing software developers, and their weapons are partitioning, knowledge, and abstractions, where does software architecture fit? Software architecture contributes to all three, but its primary contribution is a set of carefully defined new abstractions, including *components*, *connectors* and *ports*. These abstractions enable larger and more complex problems to fit in the heads of software developers. Models containing these new abstractions can be analyzed, either by the software developers or by automated analysis tools.

Software architecture also contributes knowledge. Patterns of software architecture reoccur, and are called *architectural styles*. An architectural style is better suited to some problems than others, so by identifying and naming styles, such as client-server or pipe-and-filter, a body of knowledge concerning their benefits and pitfalls has accumulated. The hallmark of engineering is the design of a system using principled decisions, and having a catalog of architectural styles enables software developers to choose from known alternative designs, weighing known pros and cons.

Software architecture aids in partitioning of problems, as software developers can use the new abstractions, including components and connectors, when partitioning their systems. When a component is described in an *architectural description language* (ADL), its relationships with other components and connectors is also expressed, and therefore reveals how well that component is encapsulated.

You should not expect software architecture to be either a silver bullet or a complete solution to any single problem you are facing. It is an improvement over current software development practice and is a tool in the hands of a skilled developer. It helps software developers 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.

### 1.4 What is software architecture?

Much ink has been spilled arguing over an appropriate definition for *software architecture*. Instead of offering yet another one, this book uses the definition (Bass, Clements and Kazman, 2003) from the Software Engineering Institute (SEI) :

The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.

This definition focuses our attention on topics that are important: structure, elements, externally visible properties, and relationships. This book will additionally discuss ideas that are implicit in this definition, such as allocation of responsibility, design decisions, and styles. These topics fall under “visible properties of those elements and the relationships among them” in the definition.

The definition is careful to distinguish software architecture from software design, which looks more closely inside of the elements. This book will deliberately blur the distinction between architecture and design because in practice you rarely see software architecture taking place as an activity separate from software design. The close relationship between architecture and design is discussed in more depth in Section 4.11.

For many people a vague definition of architecture is sufficient. If you find yourself in a heated argument over the definition so that you can compartmentalize roles in your company, then you already have bigger problems to work out. On the other hand, if you build software and are simply looking for techniques that help you build it better and reduce the risks of failure, then you are likely tolerant of some fuzziness around where architecture stops and design begins.

One can view software architecture as a part of a *story at many levels*. The invention of subroutines allowed developers to tell a story of a master and servant routines. The master task could be understood at an abstract level without reading each of the subroutines. The invention of modules, structured programming, and object-oriented programming enabled the story to be told with increasingly large codebases. The story from the subroutine level was still there, but it was augmented with a story about what each module did. The concepts in software architecture allow us to tell a story about larger chunks – for example, that this is a 3-tier system with one tier behind a security firewall.

## 1.5 Required perspective shift

In 1968 Edsger Dijkstra wrote a now famous letter advocating the use of structured programming titled “GOTO Considered Harmful”. His argument is roughly as follows. Programmers build programs containing static statements that execute to produce output. Programmers, being human, have a hard time envisioning how the static statements of 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. Programmers were accustomed to working within the old set of abstractions. They focused on the constraints of the new abstractions rather than the benefits. Each similar increase in abstraction is opposed by some who are familiar with the old abstractions. During my programming career, I have seen programmers resist abstract data types and object-oriented programming, only to later embrace them.

New abstractions do not replace old ones, but instead coexist with them. Designing your system so that communication between components occurs only via ports and uses explicit connectors does not mean that objects, methods, and data structures disappear. Similarly, forest fire fighters switch between forest and tree abstractions depending on what part of their job they are doing at the moment.

Solving harder problems can be done either by refining the existing weapons, or by inventing new ones. While we can improve by sharpening our old tools, we expect only

modest improvements. Jumps in improvement come from improvements in abstractions. We are constantly inventing new housing materials to provide us with better buildings, but improved bricks and carpentry did not enable the leap to build skyscrapers. Similarly, a more effective state machine will not enable us to build the distributed, heterogeneous, reliable systems of the next decade.

Using software architecture requires a conscious and explicit shift to creating components and connectors, rather than just using the existing abstractions (often just objects). Every computer program has a software architecture, but if it has not been consciously chosen then it is probably what Brian Foote and Joseph Yoder call a *big ball of mud* (Foote and Yoder, 2000), and they estimate that it is the most common software architecture. It is easy to understand this architecture: Imagine what a system with 10 objects 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.

It can be harder to build systems with a deliberate architecture. By analogy, imagine that you have developed a system that uses a *stack* abstract data type, but then later want to remove an item from the middle of the stack. Since stacks prohibit this, either the stack was not the right abstraction to use, or you will need to re-work the program to maintain the integrity of the stack abstraction. The user of the software system may not care if the abstraction is violated, but software developers do, as violations pile up and evolving the system becomes harder. What the system *does not do*, including that it does not violate the invariants in its abstractions, can be as important as what it does.

I have seen software developers who design systems in accordance with the abstractions of software architecture, yet voice their resistance to software architecture. I believe this contradiction may stem from resistance to high bureaucracy processes, or having seen time wasted making diagrams instead of systems, both of which can be conflated with software architecture. Through this book I hope to disentangle these ideas, and to promote the efficient use of software architecture abstractions.

## 1.6 Risk

My father has a degree in mechanical engineering, but when he put up a mailbox he did it like anyone else would: he dug a hole, put in some cement, and put the post into the cement. Just because he could calculate moments, stresses, and strains does not mean he must or should. In other situations it would be foolish to skip these analyses. How do we know when to use them? Our general principle is that our effort should be commensurate with our risk of failure.

With new technologies, it is hard to know when to use them and when to stop. Software architecture is a relatively new technology, and it includes many techniques for modeling and analyzing systems. Each of these activities takes time that could otherwise be spent building the system. This book will provide some guidance and help software developers choose appropriate techniques.

Each software project faces risks of failure. Will it run fast enough? Will it interoperate with existing systems? Can it be maintained? Will it scale to many users? While every project has risks, risks differ from project to project. Not all software architecture techniques are good at reducing a particular risk. A great way to waste time is to engage in activities that reduce the wrong risks, ones you are not worried about.

Assuming I can produce an accurate list of risks for my project, I can work systematically to reduce their likelihood until they drop off my list of worries. For example, I might fear how well my web system will scale, but as I model the expected load, choose the infrastructure, and prototype the system I gain confidence in its ability to scale up, and consequently I can remove scalability from my list of fears. Other risks might never make it onto my list of fears, such as interoperability if my project is a standalone system.

The idea of working systematically to reduce engineering risks echoes Barry Boehm's spiral model of software development (Boehm, 1988). The spiral model instructs engineers to build the system incrementally, starting from the highest risk items to the lowest risk. Projects face many risks besides engineering risks, however, which means that managers must decide how the risk of customer rejection should be prioritized versus the risk that the system is insecure or inefficient.

While the kinds of risks will vary from project to project, we can make some generalizations by grouping projects based on their type. We can divide projects into Systems projects (e.g., operating systems, device drivers), Information Technology (IT) projects (e.g., banking applications, inventory control), and Web projects (e.g., online retail, web forums). Software developers on Systems projects tend to fear that their system will not be fast enough, reliable enough, small enough, or secure enough. Software developers on IT projects tend to fear their system will not be sufficiently modifiable, interoperable, or comprehensive. Software developers on Web projects tend to fear that their developers will not be productive, their website will be insecure, and that too many users will overload their website. Not all projects will follow these patterns exactly, and will have risks in addition to these.

Software developers working on one type of system often do not understand why developers on other types of systems choose different activities and prioritize risks differently. Looking at the canonical risks on each system type helps us to understand why this happens. A Systems software developer fears a program that is too slow, or one that uses too much memory, and so avoids automatic garbage collection. An IT software developer fears maintenance costs, and so embraces automatic garbage collection. Few choices have no tradeoffs, so a Systems software developer who fears reliability more than slowness may choose garbage collection to eliminate a category of bugs by trading off some speed.

## 1.7 Concerns about software architecture

Many software developers fear software architecture and design because it can lead to *analysis paralysis*. This fear is well motivated, because adding work to a project can slow it down, especially if that work does not have clear bounds. But they are wrong to fear that software architecture must be wall-sized UML models of the whole system, or binders of documents that are never read. When someone suggests starting some work, they should be able to tell you when to stop and how much is enough. This book shows how to do a minimum amount of architecture so that it does not become analysis paralysis.

Your enthusiasm for software architecture depends on your temperament, and software developers have different temperaments. Some prefer writing code to building models, and others prefer the reverse. It is important to know yourself so that you do not advocate one simply because you enjoy doing it rather than because it is what is best for the project. So ask yourself which one you prefer and keep an open mind about where the ideal balance

lies.

Software architecture has been criticized for having poor impact. This will occur when models are built because your process says so, rather than originating from a recognized engineering risk. We do not yet have good techniques for every risk, but we do have many good techniques.

Anything that is not code may become outdated or otherwise inconsistent with the system's source code. Chapter 5 describes techniques for handling this problem, including modeling only a the most important aspects and those least likely to change rapidly.

Some advocates of agile development shun architecture and design work because they feel the time is better spent coding. They worry that software architecture implies Big Design Up Front. Their aversion comes from having seen too many projects lose sight of the ultimate goal: efficiently producing software loved by the customer. After reading this book, many agile developers on small projects will likely do exactly what they do today, but they will better understand why their choices are appropriate and will have some shared design vocabulary with their co-workers. And when they work on larger projects or have particularly tricky design challenges then they will know the techniques to overcome them. While this is not a book specifically about agile architecture, it encourages agile and spiral software development processes and one of its goals is to show how software architecture techniques are compatible with agile development.

## 1.8 About this book

Software architecture techniques are now mature enough to reduce the burdens of complexity and scale in software development, and should be widely adopted. Not every project needs every model or analysis, so software developers should prioritize project risks and apply the techniques that are best at mitigating those risks.

This book, as much as is possible, focuses on software architecture as it relates to the *construction of software*, and describes the techniques used to ensure software satisfies its engineering demands. This book is largely process agnostic because the engineering techniques themselves are largely process agnostic. Many books on software design / architecture describe management activities like the political responsibilities of architects, or when to hold specific kinds of meetings. These activities are valuable, and are analogous to the bridge-building activities of coordinating with the community to ensure that the project receives funding. A different set of activities ensure that the bridge satisfies its engineering demands – whether or not it withstands its load. This book focuses on engineering techniques used to ensure the software withstands its load. Another important area not covered by this book is requirements. Gathering and analyzing requirements is hard and error-prone. Worse, stakeholders will change their minds while the software is under construction. Yet every book has boundaries and cannot cover every topic.

The primary contribution of this book is to consolidate software architecture techniques found in a variety of other sources, integrating both techniques emphasizing on quality attributes as well as ones emphasizing functionality. It also shows how architecture is sufficiently lightweight to work within an agile or iterative process. The key insight is that developers can use risk to mediate how much architecture and design is needed, and thereby avoid both Big Design Up Front and analysis paralysis.

Readers of the book can expect three things. First, they will have a mental framework

of software architecture and how it can be used to combat complexity and scale. Software architecture abstractions will be in their heads so that when they look at systems it will be easier to see how and why the pieces fit together. Second, they will know a set of interconnected software design and modeling techniques. They will be able to choose appropriate techniques that address risks faced on their projects. And third, they will have expert software architecture knowledge, including a catalog of architecture styles, a set of standard relationships between models, and a set of component decomposition strategies.

This book is divided into two parts. This first part introduces and walks through the basic ideas of software architecture and the second part contains reference material. The remaining chapters in this first part prepare you to apply software architecture by answering common questions: Why should I use software architecture, and how much should I use? What architecture techniques should I use to mitigate my risks? Are there standard risks for my type of system? What are the basics of building models? How do I recognize a good model? How do I make an architecture model? What are the pitfalls of architecture modeling?