6

Abstraction Works Only in a Perfect World

"There is no abstract art. You must always start with something. Afterward you can remove all traces of reality."
—Pablo Picasso

Chapter Contents

In the first part of the book, we saw how IT systems have grown increasingly larger and more complex over time. This growing complexity is challenging the capability of businesses to innovate as more of the IT budget is channeled into regulatory compliance, replatforming, and maintenance of the status quo. As this book has shown, changing these systems is not primarily a technical difficulty, but one of coordinating and disambiguating human communication. In overcoming such difficulties, we have introduced the concept of an Elephant Eater and the Brownfield development approach.

This second part of the book explains the technical and practical aspects of Brownfield for someone who might want to implement such an approach. This chapter examines the necessary technical context, requirements, and characteristics of the Elephant Eater. The chapter then goes on to analyze existing IT elephant-eating approaches and highlights the problems these approaches present with their extensive use of decomposition and abstraction.

Considerations for an Elephant Eater

The following sections outline considerations for the Elephant Eater. The problems with large scale developments are many, and the first half of the book illustrated some of the problems that such developments pose. The high failure rate for such projects is the reason why the creation of an Elephant Eater was necessary. Like any problem, the starting point for a solution is the understanding of the requirements, so if an Elephant Eater is going to be created, it needs to cater to the considerations in this section.

Lack of Transparency

On very large-scale developments, the problem being solved usually is unclear. At a high level, the design and development task might seem to be understood—for example, "build a family home," "design a hospital," or "implement a customer relationship management system." However, such terms are insufficient to describe what is actually required.

For any complex problem, some degree of analysis and investigation is essential to properly frame the detailed requirements of the solution and understand its context. In conventional building architectures, the site survey is a fundamental part of the requirements-gathering process.

A thorough analysis of a complex site takes a great deal of time and effort. Even using traditional Greenfield methods, the analysis effort is often as large as the build effort. Despite this effort, however, IT architects and business analysts rarely do as thorough a job of surveying a site as building architects do. As discussed in previous chapters, a thorough analysis that encompasses functional and nonfunctional requirements and multiple constraints requires vast documentation. As such, the real requirements in any situation are always less than transparent.

Unfortunately, in IT, relatively little time is spent on the equivalent of a site survey.

Multiple Conflicting Goals

Another problem is conflicting requirements. In any complex situation, a single optimal solution is rarely a given for such a problem. The problem itself might even be poorly described.

In the example of the house building discussion in Chapter 1, "Eating Elephants Is Difficult," the mother-in-law and the landowner could have very different perspectives on what is desirable. Will their combined requirements be entirely coherent and compatible? Whose job will it be to resolve these conflicts?

We have seen the same problem on multiple $100 million programs. Any big program owned by more than one powerful stakeholder is likely to fail because of confusing and conflicted directions. As we saw in Chapter 1, life is much easier when one powerful person is consistently in charge. Of course, assigning a single stakeholder is not easy, but failing to identify this stakeholder at the start of the project only ignores the problem.

Spotting requirements that are clearly expressed but in conflict is reasonably easy, and it is usually possible to resolve these through careful negotiation. No one would seriously demand two mutually incompatible set of requirements, right?

Let's return to the analogy of home building as an example. When designing a house, increasing the size of the windows will increase the feeling of light and space within the building and improve the view. But bigger windows will contribute to energy loss. Improved insulation in the walls or ceilings might compensate for this, but this could result in increased building costs or a reduced living area. Alternatively, the architect could request special triple glazing. That would make the windows more thermally efficient but could make the glass less translucent. As more concerns arise, the interplays between them become more complex. As a result, the final solution becomes a trade-off between different aspects or characteristics of the solution. Possibly, the requirements are actually mutually incompatible—but this can be known only in the context of a solution.

These conflicting requirements also come up repeatedly when designing large computer systems. We hear comments similar to these: "We need the system to be hugely scaleable to cope with any unexpected demand. It must be available 24 hours a day, 7 days a week—even during upgrades or maintenance—but must be cheaper to build, run, and maintain than the last system." Obviously, such requirements are always in conflict.

Dynamic Aspects

The difficulty in coping with these requirements is compounded by the fact that they don't stand still. As you begin to interfere and interact with the problem, you change it. For example, talking to a user about what the system currently does could change that user's perception about what it needs to do. Interacting with the system during acceptance testing might overturn those initial perceptions again. Introducing the supposed solution into the environment could cause additional difficulties.

In IT systems, these side effects often result from a lack of understanding that installing a new IT system changes its surroundings. Subsequent changes also might need to be made to existing business procedures and best practices that are not directly part of the solution. These changes might alter people's jobs, their interaction with customers, or the skills they require.

In addition to these impacts, the time involved in such projects usually means that the business environment in which the solution is to be placed has evolved. Some of the original requirements might need to change or might no longer be applicable

Therefore, one of the key requirements for any Elephant Eater is tight and dynamic linkage between the business and IT.

Systems Integration and Engineering Techniques

But the problem we're talking about isn't new, is it? People have been trying to deliver complex systems for more than 40 years. There must already be some pretty reasonable Elephant Eaters out there.

Now that we have a good understanding of the problem, it's a good idea to take a closer look at some of the solutions that are already out there and see why, given the meager 30 percent success rate noted in the Preface, we need a new Elephant Eater.

Generally, these big problems need to be approached via formal techniques. These techniques work from two different directions. They either work their way down from the top, gaining increasing levels of detail, or they start from the bottom, examining needs in detail and working their way upward, building toward a complete solution.

Walk the Easy Path or…

If you're infinitely lucky, the bottom-up approach might work. Considering a very simple example, you could select a package that seems close to what you need. You could then walk through the business processes you want to execute. As you go, you can write down all the changes you need to make to the package, and, *presto!* After you've made the changes, you've got a solution! You've designed the whole system from the ground up because the package dictates your choices for how you do pretty much everything else.

If you don't allow the package to dictate your choices, chances are, you will find yourself in a very sticky mess: Each major change you make will require extra development, testing, and long-term maintenance costs. If you've chosen the bottom-up approach, you must stick to it religiously and accept the changes it will impose on the process and the business.

Ultimately, a package with a good fit, whether imposed or a lucky choice, is the very best in bottom-up solutions. Start halfway up the hill—the package already approximates what you want. Then modify the solution iteratively with the end user and find a happy endpoint near the top of the hill.

However, chances are, for a really complex project, using the bottom-up approach with a single package will not work. You must break down the problem into smaller pieces and then integrate them to create a single solution. You can divide up the problem in two fundamental ways.

…Break the Boulders and Make Them Smooth

You can decompose the problem into smaller, more easily managed Views through two methods: splitting and abstraction. Splitting simply divides complex big chunks into smaller, more manageable pieces. Abstraction removes detail from each larger chunk to form more manageable and understandable pieces. These two techniques, splitting and abstraction, allow almost any gargantuan problem to be subdivided into smaller, better contained problems. Think of it as slicing the problem into little squares.

Abstraction gives you horizontal cuts, while View splitting gives you vertical ones. Everything becomes a manageable "chunk." This is the basis for most systems integration and engineering methods. Many of these methods are proprietary, but some, such as The Open Group Architecture Framework (TOGAF) from the Open Foundation, are freely available. Each approach tries to create a continuum of knowledge, from high-level representations to more detailed. These paths vary but can be characterized as moving in some way from logical to physical, general to specific, or taxonomy to specification.

When good methods or tools are used, there is traceability from the high level to the low level. This helps a reader understand why something has been designed the way it has.

Such movement is unsurprisingly characterized as a progression, starting from the high-level principles and overall vision of what needs to be achieved, and moving down through the perspectives of business, process, roles, and models of information. Figure 6.1 highlights the basic stages of the TOGAF method.

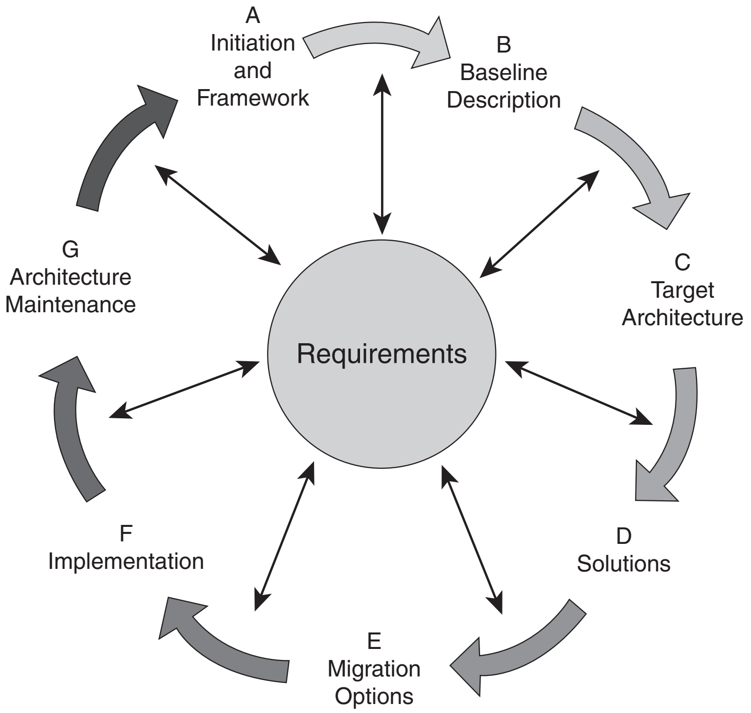

Figure 6.1 Even with its cyclic diagram, TOGAF is part of the progressive school of architecture.

Some approaches go even further. They segment each level of abstraction into a number of separate perspectives. Of these “frameworks,” the enterprise architecture framework produced by John Zachman of IBM in the 1980s is probably the most famous. Called the Zachman Framework, it considers the additional dimension of Data, Function, Network, People, Time, and Motivation. Figure 6.2 illustrates how the Zachman Framework segments the architecture into these perspectives.

	What? Data	How? Function	Where? Network	Who? People	When? Time	Why? Motivation	
Planner							Scope
Owner							Enterprise Models
Designer							System Models
Builder							Technology Models
Sub-contractor							Detailed Represent-ations
Enterprise							Actual Systems

Figure 6.2 The Zachman Framework of Enterprise Architecture segments the architecture into a variety of perspectives.

These approaches enable you to decompose the full width and breadth of the problem (including the existing constraints) into separate Views so that a suitably skilled guru can independently govern and maintain them.

At the very top of this top-down approach is a simple sheet of paper that purports to show or describe the scope of the whole problem for that particular perspective. A single sheet of paper might even purport to summarize the 10,000-foot view for *all* the perspectives.

Below that top sheet are many more sheets that describe each element on the sheet above. This technique is so well recognized that it's applied to almost everything in complex problems, whether we're talking about the shape of the system, the business processes that it executes, or the description of the plan that will build it.

In this hierarchy of paper, the top tier is labeled Level 0; the next tier down, Level 1; and so on. At each layer, the number of sheets of paper increases, but each of these sheets is a manageable View. The problem has been successfully decomposed. In the example in Figure 6.3, our single-page business context that describes the boundaries of the problem we're solving is gradually decomposed into 60,000 pages of code, deployment information, and operational instructions that describe the whole solution. At each step of the way, the intermediate representations all correspond to a View.

After the problem has been decomposed into single sheets, or Views, rules must be written and applied to specify how they work together.

Surely that solves our problem. The elephant has been eaten. Complexity is reduced, so each area becomes manageable. Each person is dealing with only a bit of the problem.

This is, of course, precisely what the world's largest systems integrators do. They define their Views in terms of work products or deliverables. They come from different perspectives and at different levels of abstraction. The systems integrators have design and development methods that describe who should do what to which View (or work product) and in what order.

So if the problem is essentially solved, why does it go wrong so often?

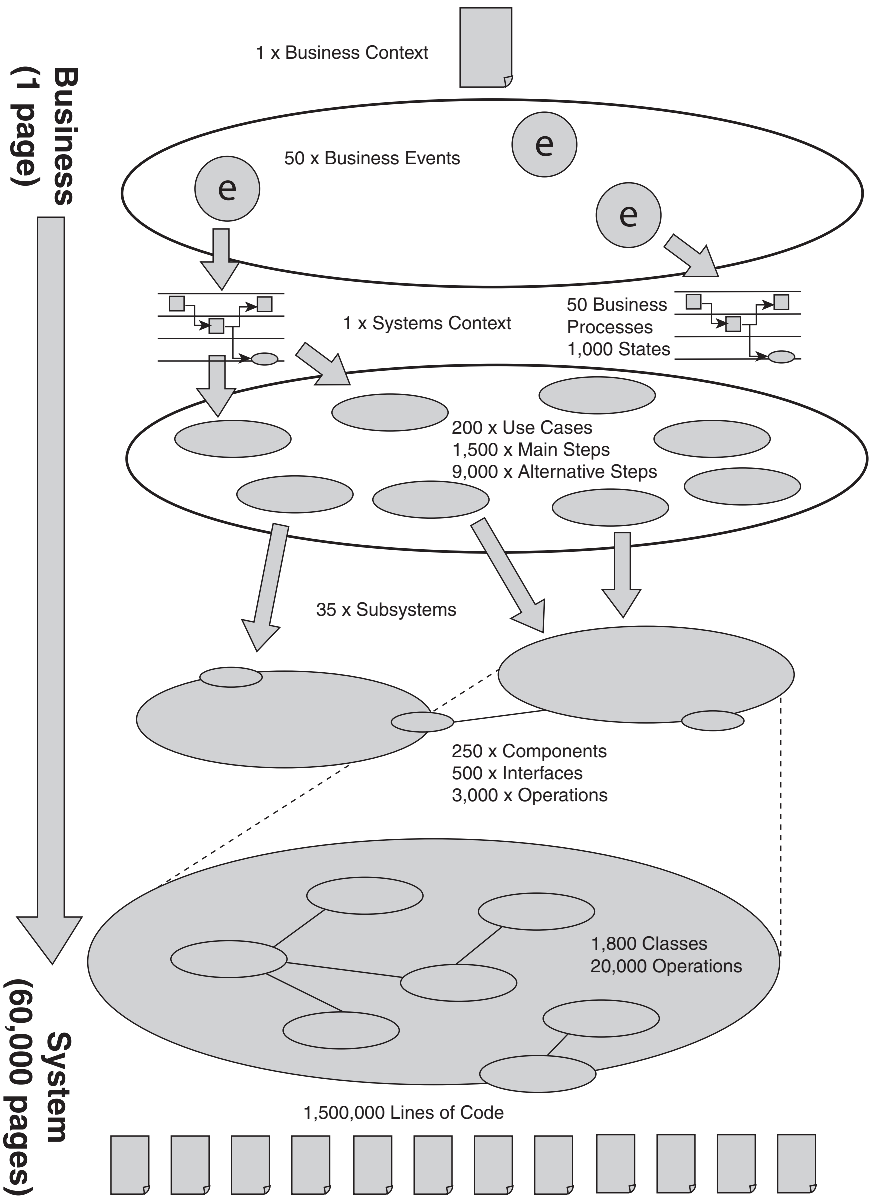

Figure 6.3 Decomposition of a complex problem space

Abstraction Is the Heart of Architecture

In all these cases, we move from the general to the specific, with the next layer of detail expanding upon the previous level of abstraction. This movement from general to specific gives architecture its power to simplify, communicate, and make ghastly complexity more aesthetically pleasing.

Abstraction is the heart of architecture. This powerful and persuasive concept has been at the center of most of the advances in complex systems architecting for the last 15 years. It underpins the history of software engineering—objects, components, and even IT services have their roots in abstraction. Because abstraction is one of our most powerful tools, we should consider its capabilities and limitations.

As systems have become more complex, additional layers of abstraction have been inserted into the software to keep everything understandable and maintainable. Year by year, programmers have gotten further away from the bits, registers, and native machine code, through the introduction of languages, layered software architectures, object-oriented languages, visual programming, modeling, packages, and even models of models (metamodeling).

Today, programs can be routinely written, tested, and deployed without manually writing a single line of code or even understanding the basics of how a computer works. A cornucopia of techniques and technologies can insulate today's programmers from the specifics and complexities of their surrounding environments. Writing a program is so simple that we can even get a computer to do it. We get used to the idea of being insulated from the complexity of the real world.

Mirror, Mirror on the Wall, Which Is the Fairest Software of All?

Software engineering approaches the complexity and unpredictability of the real world by abstracting the detail to something more convenient and incrementally improving the abstraction over time.

Working out the levels of abstraction that solve the problem (and will continue to solve the problem) is the key concern of the software architect. IBM's chief scientist Grady Booch and other leaders of the software industry are convinced that the best software should be capable of dealing with great complexity but also should be inherently simple and aesthetically pleasing.[1]

Thus, over time, we should expect that increasing levels of abstraction will enable our software to deal with more aspects of the real world. This is most obviously noticeable in games and virtual worlds, where the sophistication of the representation of the virtual reality has increased as individual elements of the problem are abstracted. Figure 6.4 shows how games architectures have matured over the last 20 years.

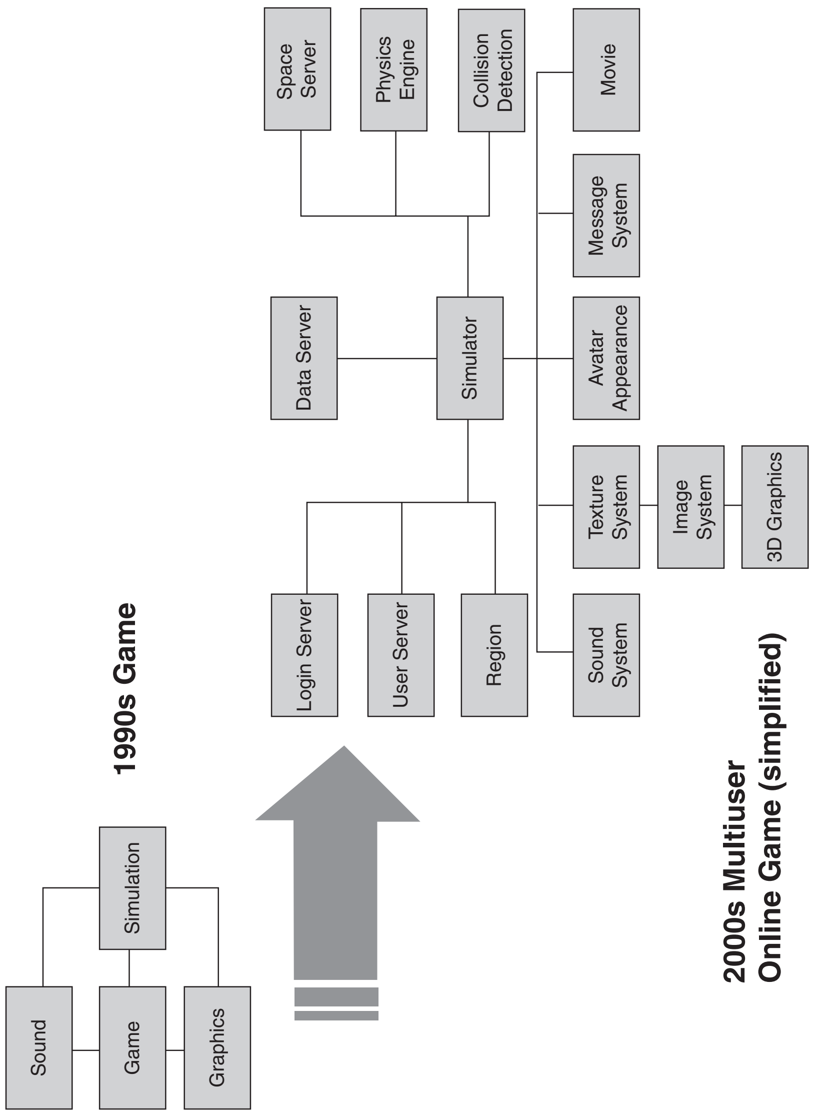

Figure 6.4 Games architectures have matured immensely over the last 20 years.

The current sophisticated, shared online games of the early twenty-first century exhibit greater descriptive power compared to the basic 2D games of the 1970s. Hiding the complexity of the physics engine from the graphical rendering system, and hiding both of these from the user server and the system that stores the in-world objects, enables increasing levels of sophisticated behavior.

Abstraction has its drawbacks, however. Each level of abstraction deliberately hides a certain amount of complexity. That's fine if you start with a complete description of the problem and work your way upward, but you must remember that this isn't the way today's systems integration and architecting methods work.

These methods start from the general and the abstract, and gradually refine the level of detail from there. Eventually, they drill down to reality. This sounds good. Superficially, it sounds almost like a scientific technique. For example, physicists conduct experiments in the real world, which has a lot of complexity, imperfection, and "noise" complicating their experiments. However, those experiments are designed to define or confirm useful and accurate abstractions of reality in the form of mathematical theories that will enable them to make successful predictions. Of course, the key difference between software engineering and physics is that the physicists are iteratively creating abstractions for something that already exists and refining the abstraction as more facts emerge. The architects, on the other hand, are abstracting first and then creating the detail to slot in behind the abstraction. Figure 6.5 should make the comparison clearer.

The IT approach should strike you as fundamentally wrong. If you need some convincing, instead of focusing on the rather abstract worlds of physics or IT, let's first take a look at something more down to earth: plumbing.

Plumbing the Depths

The IT and plumbing industries have much in common. Participants in both spend a great deal of time sucking their teeth, saying, "Well, I wouldn't have done it like that," or, "That'll cost a few dollars to put right." As in many other professions, they make sure that they shroud themselves in indecipherable private languages, acronyms, and anecdotes.

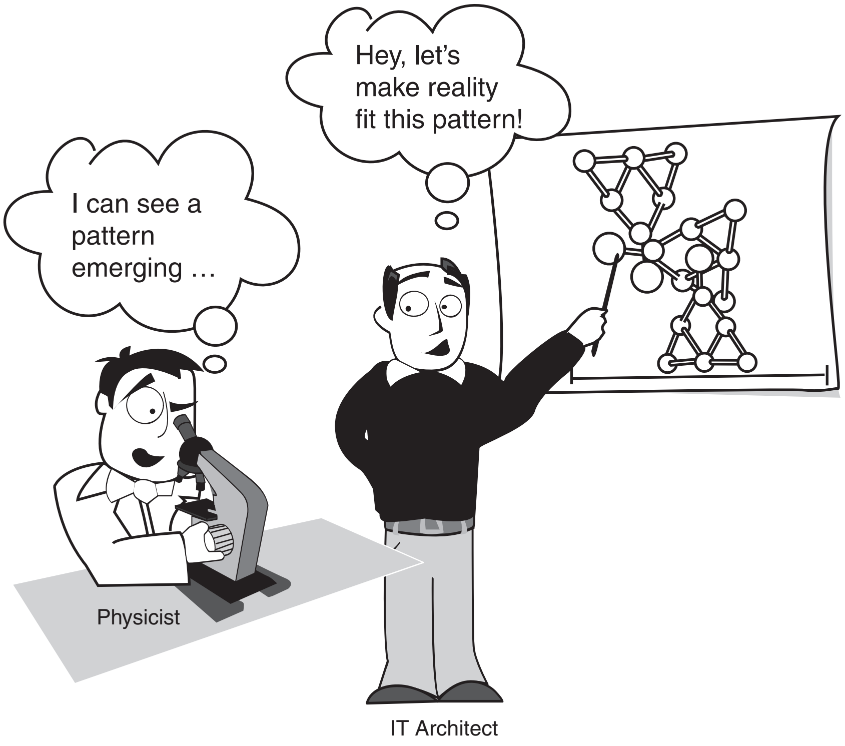
I can see a pattern emerging …
Hey, let's make reality fit this pattern!
Physicist
IT Architect

Figure 6.5 Who's right? Physicists or IT architects?

Imagine for a moment a heating engineer who has been asked to install a radiator in a new extension. He has looked at the plans and knows how he's going to get access to the pipes. From the specifications he's read, he knows what fixtures he needs. After doing some pretty easy calculations based on room size, window area, and wall type, he even got hold of the right size radiator to fit on the wall that will deliver the right amount of heat for the room. It's an hour's work, at most.

The job is done and he leaves a happy man. A few days later, the homeowner is complaining that the room is still cold. Only when the plumber arrives back on-site and investigates the boiler does he find out that the output of the boiler is now insufficient for the needs of the house. He recommends that the homeowner order a new 33-kilowatt boiler and arranges to come back in a week.

A week later, he's back to begin fitting the new boiler. Right at the start of the task, it becomes obvious that the old boiler was oil-fired and the new one is gas. This is slightly inconvenient because the property is not connected to the gas main, even though it runs past the property.

Another few weeks pass while the homeowner arranges for the house to be connected to the gas supply. On the plumber's third visit, everything is going swimmingly. Then he notices that there are no free breaker slots on the electricity circuit board to attach the new boiler. A week later, he replaces the circuit board. The boiler is installed, but another problem arises: Although the heat output of the boiler is sufficient, a more powerful pump is required to distribute the heat throughout the house.

And that's when the problems really start.

Don't Abstract Until You See the Whole Elephant

Judging from the architect's top-level view, the solution seemed pretty obvious. Only when the job was almost done was it obvious that it hadn't worked. Those other aspects of the problem—the supply, the pump, and the circuit board—were invisible from the Level 0 perspective the plumber received, so he ignored them in doing his analysis.

After all, nothing was fundamentally wrong with the plumber's solution; he just didn't have a good specification of the problem. The process of abstracting the problem to the single architectural drawing of the new room meant that he had no visibility of the real problem, which was somewhat bigger and more complex. He simply couldn't see the hidden requirements—the environmental constraints—from his top-level, incorrectly abstracted view of the problem.

Unfortunately, abstractions, per se, always lack details of the underlying complexity. The radiator was a good theoretical solution to the problem, but it was being treated as a simple abstract component that, when connected to the central heating system, would issue the right amount of heat. Behind that simple abstraction lays the real hidden complexity of the boiler, gas main, and circuit board that leaked through and derailed this abstracted solution.[2] Will such complexity always leak up through the pipe and derail simple abstract solutions?

Well, imagine for a moment that the abstraction was absolute and that it was impossible to trace backward from the radiator to the source of the heat. Consider, for example, that the heat to each radiator was supplied from one of a huge number of central utilities via shared pipes. If the complexity of that arrangement was completely hidden, you would not know who to complain to if the radiator didn't work. Of course, on the positive side, the utility company supplying your heat wouldn't be able to bill you for adding a new radiator!

Is this such an absurd example? Consider today's IT infrastructures, with layers of software, each supposedly easier to maintain by hiding the complexities below. Who do you call when there is a problem? Is it in the application? The middleware? Maybe it is a problem with the database?

If you become completely insulated from the underlying complexity—or if you simply don't understand it, then it becomes very difficult to know what is happening when something goes wrong. Such an approach also encourages naïve rather than robust implementations. Abstractions that fully hide complexity ultimately cause problems because it is impossible to know what is going wrong.

Poorly formed abstractions can also create a lack of flexibility in any complex software architecture. If the wrong elements are chosen to be exposed to the layers above, people will have to find ways around the architecture, compromising its integrity. Establishing the right abstractions is more of an art than a science, but starting from a point of generalization is not a good place to start—it is possibly the worst.

Successful Abstraction Does Not Come from a Lack of Knowledge

In summary, abstraction is probably the single most powerful tool for the architect. It works well when used with care and when there is a deep understanding of the problem.

However, today's methods work from the general to the specific, so they essentially encourage and impose a lack of knowledge. Not surprisingly, therefore, the initial abstractions and decompositions that are made at the start of a big systems integration or development project often turn out to be wrong. Today's methods tend to ignore complexity while purporting to hide it.

The Ripple Effect

Poor abstractions lead to underestimations and misunderstandings galore. Everything looks so simple from 10,000 feet. On large projects, a saying goes that "All expensive mistakes are made on the first day." From our experience, it's an observation that is very, very true.

Working with a lack of information makes abstraction easy but inaccurate.

All projects are most optimistic right at the start. These early stages lack detailed information; as a result, assumptions are made and the big abstractions are decided.

Assumptions are not dangerous in themselves—as long as they are tracked. Unfortunately, all too often they are made but not tracked, and their impact is not understood. In some ways, they are treated as "risks that will never happen." Assumptions must always be tracked and reviewed, and their potential impact, if they're untrue, must be understood. Chances are, some of them will turn out to be false assumptions—and, chances are, those will be the ones with expensive consequences.

We need to move away from this optimistic, pretty-diagram school of architecture, in which making the right decisions is an art form of second guessing based on years of accumulated instinct and heuristics.[3] We need a more scientific approach with fewer assumptions and oversimplifications. A colleague, Bob Lojek, memorably said, "Once you understand the full problem, there is no problem."

Fundamentally, we need to put more effort into understanding the problem than prematurely defining the solution. As senior architects for IBM, we are often asked to intervene in client projects when things have gone awry. For example:

An Agile development method was being used to deliver a leading-edge, web-based, customer self-service solution for a world-leading credit card processor. The team had all the relevant skills, and the lead architect was a software engineering guru who knew the modern technology platform they were using and had delivered many projects in the past.

Given the new nature of the technology, the team had conformed strictly to the best-practice patterns for development and had created a technical prototype to ensure that the technology did what they wanted it to do. The design they had created was hugely elegant and was exactly in line with the customer requirement.

A problem arose, though. The project had run like a dream for 6 months, but it stalled in the final 3 months of the development. The reporting system for the project recorded correctly that 80 percent of the code had been written and was working, but the progress meter had stopped there and was not moving forward. IBM was asked to take a look and see what the problem was.

As usual, the answer was relatively straightforward. The levels of abstraction, or layering, of the system had been done according to theoretical best practice, but it was overly sophisticated for the job that needed to be done. The architecture failed the Occam's Razor test: The lead architect had induced unnecessary complexity, and his key architectural decisions around abstraction (and, to some extent, decomposition) of the problem had been made in isolation of the actual customer problem.

Second, and more important, the architect had ignored the inherent complexity of the solution. Although the user requirements were relatively straightforward and the Level 0 architecture perspectives were easy to understand, he had largely ignored the constraints imposed by the other systems that surrounded the self-service solution.

Yes, the design successfully performed a beautiful and elegant abstraction of the core concepts it needed to deal with—it's just that it didn't look anything like the systems to which it needed to be linked. As a result, the core activity for the previous 3 months had been a frantic attempt to map the new solution onto the limitations of the transactions and data models of the old. The mind-bending complexity of trying to pull together two mutually incompatible views of these new and old systems had paralyzed the delivery team. They didn't want to think the unthinkable. They had defined an elegant and best-practice solution to the wrong problem. In doing so, they had ignored hundreds of constraints that needed to be imposed on the new system.

When the project restarted with a core understanding of these constraints, it became straightforward to define the right levels of abstraction and separation of concerns. This provided an elegant and simple solution with flexibility in all the right places—without complicating the solution's relationship with its neighbors.

—R.H.

As a final horror story, consider a major customer case system for an important government agency:

> We were asked to intervene after the project (in the hands of another systems integrator) had made little progress after 2 years of investment.
>
> At this point, the customer had chosen a package to provide its overarching customer care solution. After significant analysis, this package had been accepted as a superb fit to the business and user requirements. Pretty much everything that was needed to replace the hugely complex legacy systems would come out of a box.
>
> However, it was thought that replacing a complete legacy system would be too risky. As a result, the decision was made to use half of the package for the end-user element of the strategic solution; the legacy systems the package was meant to replace would serve as its temporary back end (providing some of the complex logic and many of the interfaces that were necessary for an end-to-end solution).
>
> The decision was to eat half the elephant. On paper, from 10,000 feet, it looked straightforward. The high-level analysis had not pointed out any glitches, and the layering of the architecture and the separation of concerns appeared clean and simple.
>
> As the project progressed, however, it became apparent that the legacy system imposed a very different set of constraints on the package. Although they were highly similar from an end user and data perspective, the internal models of the new and old systems turned out to be hugely different—and these differences numbered in the thousands instead of the hundreds.
>
> Ultimately, the three-way conflict between the user requirements (which were based on the promise of a full new system), the new package, and the legacy system meant that something had to give. The requirements were deemed to be strategic and the legacy system was immovable, so the package had to change. This decision broke the first rule of bottom-up implementations mentioned earlier.

> Although the system was delivered on time and budget, and although it works to this day for thousands of users and millions of customers, the implementation was hugely complicated by the backflow of constraints from the legacy systems. As a result, it then proved uneconomic to move the system to subsequent major versions of the package. The desired strategic solution became a dead end.
>
> —K.J. and R.H.

In each of these cases, a better and more detailed understanding of the overall problem was needed than standard top-down approaches could provide. Such an understanding would have prevented the problems these projects encountered.

Each of these three problems stems from a basic and incorrect assumption by stakeholders that they could build a Greenfield implementation. At the credit card processor, this assumption held firm until they tried to integrate it with the existing infrastructure. The government department failed to realize that its original requirements were based on a survey of a completely different site (the one in which the legacy system was cleared away), resulting in large-scale customization of the original package that was supposedly a perfect fit.

Fundamentally, today's large-scale IT projects need to work around the constraints of their existing environment. Today's IT architects should regard themselves as Brownfield redevelopers first, and exciting and visionary architects second.

Companies that try to upgrade their hardware or software to the latest levels experience the same ripple effect of contamination from the existing environment. Despite the abstraction and layering of modern software and the imposed rigor of enterprise architectures, making changes to the low levels of systems still has a major impact on today's enterprises.

As we mentioned before, no abstraction is perfect and, to some extent, it will leak around the edges. This means there is no such thing as a nondisruptive change to any nontrivial environment. As a supposedly independent layer in the environment changes—perhaps a database, middleware, or operating system version—a ripple of change permeates around the environment.

As only certain combinations of products are supported, the change can cascade like a chain of dominoes. Ultimately, these ripples can hit applications, resulting in retesting, application changes, or even reintegration.

Thus, to provide good and true architectures, we need to accept that we need a better understanding of the problem to engineer the right abstractions. Additionally, we need all the aspects of the problem definition (business, application, and infrastructure) to be interlinked so that we can understand when and where the ripple effect of discovered constraints or changes will impact the solution we are defining.

Do We Need a Grand Unified Tool?

The problem definition is too big for one tool or person to maintain, so there appears to be a dilemma. The full complexity of the problem needs to be embraced, and an understanding is required of everything that's around, including the existing IT and business environments. But all that information needs to be pulled together so that the Views aren't discrete or disconnected.

Many people have argued for tool unification as a means to achieve this, to maintain all these connected Views in a single tool and, thus, enable a single documented version of the truth to be established and maintained. But that is missing a vital point about Views.

As explained in Chapter 2, "The Confusion of Tongues," Views need to be maintained by people in their own way, in their own language. Imposing a single tool will never work. Simply too many preferred perspectives, roles, and prejudices exist within our industry to believe that everyone is going to sit down one day and record and maintain their Views in one specific tool.[4] If such combinations of Views into single multipurpose tools were possible, desirable, and usable, then it is arguable that Microsoft® Office user interfaces Word®, PowerPoint®, and Excel® would have merged long ago.

Moreover, these integrated approaches that have been at the heart of traditional tooling are usually pretty poor at dealing with ambiguity or differences of opinion. On large projects with many people working on the same information, it is not unusual to have formal repositories that enable people to check out information, make changes to it, and then check it back in. Such systems prevent two people from updating the same information at the same time, which would result in confusion and conflicts. The upshot of this approach, however, is that the information that is checked into the repository is the information that everyone else is then forced to use. The implications of your changes are not always apparent to you—or perhaps immediately to your colleagues, either. Maintaining a single source of truth when hundreds of people are changing individual overlapping elements is less than straightforward. A change made by one individual can have serious consequences for many other areas of the project, and no mechanism exists for highlighting or resolving ambiguity—whoever checks the information into the repository last wins!

In summary, grand unified tools are to software engineering what grand unified theories are to modern physics—tricky to understand, multidimensional, and elusive, often involving bits of string. No one has created a single tool to maintain the full complexity of a complex IT project. Likewise, no one will do so unless the tool enables people to maintain Views in their own way, in their own language, and to identify and deal with ambiguity cooperatively.

The Connoisseur's Guide to Eating Elephants

This chapter set out to define the kinds of things the Elephant Eater must do, the kinds of problems it needs to deal with, and the kinds of environments with which it must cope. We've covered a lot of ground, so it's worth recapping the key requirements that we have established—a connoisseur's guide to eating elephants.

The Elephant Eater machine must recognize that the environment imposes many more constraints beyond functional and nonfunctional requirements. We rarely work on Greenfield sites anymore; the elephant-eating machine must be at home on the most complex Brownfield sites—the kind of Brownfield sites that have had a lot of IT complexity built up layer on layer over many years.

The Elephant Eater must also address the lack of transparency that is inherent within our most complex projects. This will enable us to x-ray our elephant to see the heart of the problem. To achieve this transparent understanding, the Elephant Eater must acknowledge the fundamental human limitation of the View and enable us to break down the problem into smaller chunks.

However, we suspect that a one-size-fits-all approach to maintaining Views is doomed to failure. A high-level business process View will always look very different than a detailed data definition. Therefore, an elephant-eating machine that relies on a single tool for all users is pretty impractical.

In addition, we now know that, despite the best efforts of architects to keep them insulated and isolated via abstractions and enterprise architectures, many of these Views are interlinked. Therefore, the only way to understand the problem properly is to make the interconnections between Views explicit and try to make them unambiguous. We should also note, however, that establishing a consolidated picture of all these Views needs to be a process of cooperation and communication—one View cannot overwrite another one, and ambiguity must be dealt with within its processing. We also know that the View should cover the entire solution (business, application, and infrastructure).

By using the formal View and VITA approach introduced in Part I, "Introducing Brownfield," it should be possible to see how the Elephant Eater proposed can address these requirements. The following facets are an intrinsic part of Brownfield development.

Our Brownfield abstractions—and, therefore, architectures—will be a good fit for the problem: Those decisions will be made based on detailed information fed in via a site survey instead of vague generalization. This adopts an engineer's approach to the solution instead of the artisan's heuristics and intuition.

We will be able to preempt the ripple effect, often understanding which requirements are in conflict or at least knowing the horrors hiding behind the constraints. Therefore, the requirements can be cost-effectively refined instead of the abstractions of the solution or its code. Resolving these problems early will have significant economic benefit.

The solution will become easier to create due to a deeper understanding of the problem. A precise and unambiguous specification will enable the use of delivery accelerators such as these:

- Global delivery and centers of excellence
- Code generation via Model Driven Development and Pattern Driven Engineering because the precise specification can be used to parameterize the generation processes
- Iterative delivery as possible strategies for appropriate business and IT segmentation of the problem become clearer

Therefore, the Brownfield approach conceptually solves many of the problems presented in this chapter and previous chapters, avoiding the early, unreliable, and imprecise abstractions and decompositions of existing approaches. In the remaining chapters, we examine how Brownfield evolved and how it can be deployed on large-scale IT projects.

Endnotes

1 Booch, Grady. "The BCS/IET Manchester Turing Lecture." Manchester, 2007. http://intranet.cs.man.ac.uk/Events_subweb/special/turing07/.

2 Splolsky, Joel. "Joel on Software." www.joelonsoftware.com/articles/LeakyAbstractions.html.

3 Maier, Mark W. and Eberhardt Rechtin. *The Art of Systems Architecting*. CRC Press, Boca Raton, Florida, 2000.

4 For example, IBM's Rational Tool Set.