book reviews

SOFTWARE ARCHITECT BOOTCAMP, 2ND ED.

RAPHAEL C. MALVEAU AND THOMAS J. MOWBRAY

Upper Saddle River, NJ: Prentice Hall, 2003. Pp. 400. ISBN 0-13-027407-0.

Reviewed by Harry DeLano
hdelano@adelphia.net

OVERALL IMPRESSIONS

Reviewing books isn't as much fun as writing software, but it can be pretty satisfying. I was a bit skeptical when I began reading this book, mostly because of a personal aversion to things military and bureaucratic. The bootcamp analogy kind of bothered me because, as I see it, the primary purpose of military training is to turn a human being into a mindless, obedient drone. My attitude softened as I realized that there is a lot of common sense here that would help one to understand the culture if one happens to get involved in a large software project. Actually, this book is more like Officer Candidate School than bootcamp and seems designed to help someone's career develop from "just" programming into leading a team of developers to successfully deliver a system.

If that's what you're interested in, this book might be for you. It reviews the techniques and tools used to help design a system at very high levels of abstraction, taking into account a wide range of points of view. It also covers many of the software development environments currently available. The reader is also offered tips on how to mold oneself professionally, including advice on how to improve communication skills and a discussion of various aspects of the psychology of software development.

In spite of my background in engineering application development, system administration, and systems programming on various platforms, as well as participation as a systems analyst on large software development projects, I found the perspective taken in this book was pretty much new territory. The overview to the use of components here is enlightening.

OVERVIEW OF BOOK

Bootcamp is about the need for software architects and the role they should play in large commercial software development projects. "The era of limitless demand for IT talent is over." This is partly due to the methods described here to build these systems. Techniques and technology in this field have evolved to allow more regimented and controlled development. (This, by the way, has made outsourcing easier.) What are these techniques and technologies? A set of formal models has evolved for specifying (1) what the problem is, (2) who cares, and (3) how that system will be built. The new technologies include the use of components (as opposed to just objects) in building large, distributed software systems.

The book is organized by chapters corresponding to some aspect of military training (e.g., "Jump School," "Military Intelligence"), with a final chapter containing advice on how to design your career. Again, the military analogy used to explain how one might be a software architect fits well with the authors' rigid and hierarchical view of how teams might best work.

THE DETAILS

What follows is a collection of what seemed to be the salient points from each of the first five chapters of the book.

CHAPTER ONE: INTRODUCTION

This chapter includes some interesting (unverified) facts worth mentioning: "Corporate America spends more than $275 billion each year on approximately 200,000 application software development projects"; application development success rates were only 16% in 1994 vs. 26% by 1998; the cost of failed projects went from $80 billion in 1995 to $75 billion in 1998; cost overruns dropped from “$59 billion in 1005 [sic, probably 1995] to . . . $22 billion in 1998.”

Chapter Two: Military History

This is an overview of the field of software architecture. It seemed a little incoherent to me in that it introduces five “schools of thought” on the subject and then goes on to describe only a couple of them thoroughly, interspersed with garbled references to others. For example, they throw in a mention of “Enterprise Architecture” without providing previous noticeable context to help understand how it fits into the discussion.

An attempt is made to justify the need for architects and to describe how one operates, including what tools they use to build system specifications. In designing a system, the architect needs to consider multiple viewpoints to achieve simplicity, maintain strict consistency in terminology to achieve system understanding, and use the notion of “complete models,” describing multiple phases of development while taking care not to get too detailed. There are techniques available to allow the consideration of several points of view of the project. One of the approaches to defining a model (the Zachman Framework) provides for 30(!) viewpoints.

Information systems have evolved from static and local to dynamic and global, so that we now have distributed multi-organizational systems with heterogeneous hardware and software configurations. The architect needs to be able to separate concerns about business application functionality from concerns about distributed-system complexity. Requirements change frequently and account for the majority of system software costs of the life cycle, so there’s a need to “future proof” the architecture.

Once careful consideration has been given to what problem the system will solve, there needs to be a way of describing the solution in fairly general terms. We used to write systems in which the code was all in one place and data was in another, but both were in the same neighborhood. Then the object-oriented approach came along and software is now written so that the code and the data it is associated with are encapsulated into elements (objects). These objects work together through message-passing mechanisms (fairly local and fairly primitive (sockets, RMI, etc.). Lately, it has been found that these object-oriented elements need to work with each other in very heterogeneous, widely distributed environments. There are many common approaches used in the various communication environments (“idioms”) that could be abstracted and used in specifying how the software project might solve a problem.

The drift of this chapter seems to be that there is a need for a good set of architectural tools to describe how to have a successful software development project. One reason is that there is a need for a system to be resistant to requirements and context changes. In the past, writing specifications and using object-oriented development techniques were sufficient. However, that was before the need for globally distributed enterprise requirements. Now the problem needs to be described from various stakeholders’ points of view. These techniques allow one to propose solutions that are abstract enough that the underlying technology can be ignored above a certain level in the hierarchy of system stakeholders (investor, architect, development manager, developer). Development can then proceed by taking advantage of component-based tools to bring the higher-level vision to fruition. Development and implementation technologies can be left as details.

The authors make the point that 70% of the code in a typical application is infrastructure. Component technology supposedly means that reinventing the wheel is no longer necessary. That was the promise of OO programming, but now the “idioms” we see over and over again that are used to have objects play together are defined as abstract elements of models and have been made part of the infrastructures that are now available for component-based development.

The authors review various approaches to this need and seem to settle on the OSI’s X.900 Reference Model for Open Distributed Processing as about the best currently available. There are other choices (e.g., IBM’s 4+1 View Model), but most are variations on the theme of RM ODP.

These tools force one to take several points of view in describing a solution to the problem: enterprise (what the system’s purchaser needs it to do); information (what data will flow and how); engineering (familiarity with the guts of the infrastructure, similar to an OS engineer); computation (partition of the system into components that can interoperate in a distributed fashion and definition of the boundaries between components, and use of CORBA IDL [see below]); technology (component interoperability concerns).

RM ODP is described over the course of about 200 pages in a set of four documents that are concise but “relatively inscrutable.” This includes conformance assessment criteria that can be used to decide whether or not the project is going well.

The notion of “design patterns” is introduced, which has been used to codify and document a lot of software knowledge. The authors state that patterns represent a rejection of originality as a technical goal (so leave your imagination at home). A very formal mechanism exists for documenting patterns, and these are collected in catalogs you can buy and which architects should study to be pattern literate. Design patterns are derived as follows: a single design occurrence is an event; two occurrences are a coincidence; three constitute a pattern.

The formality referred to as "anti-patterns" are patterns, enhanced with annotations, known to have failed in their attempt to solve a problem. Included is a description of how a new version might be derived from that original ill-used pattern to better solve the original problem (a sort of tale of woe with a happy ending). There is a class of patterns called idioms; these are programming-language specific (think cookbooks).

CHAPTER THREE: BASIC TRAINING

Here the authors go over the tools available to a software architect for doing the actual development. It reviews a history of software environments, starting with procedural technology, in which program code exists separate from the data it deals with. This is OK, but if data representation is modified, there can be a large impact on the program. OO technology (pieces of data and program elements to access and manipulate that data all together – OK but weak for distributed processing since language-specific encapsulation is insufficient to support software reuse and distributed systems); objects communicating with each other via messages, devoid of software-architecture approach; specification objects representing modules; rapid iterative prototyping, with ruthless disregard for architectural principles (bad).

They describe the evolution of distributed technology, from file servers through database servers and transaction processing monitors through distributed objects to *N*-tier componentware. Object-oriented middleware is an outgrowth of procedural predecessors, including RPC, socket-based Open Network Computing, and Distributed Computing Environment. Later, Microsoft's Distributed Common Object Model attempted to add another layer of abstraction, but, according to the authors, it still exposed too much of its underlying distribution mechanism.

CORBA (Common Object Request Broker Architecture) attempts to provide a standard interface to services used by applications, no matter what platform they run on. Using CORBA Interface Definition Language is a way to standardize on APIs throughout a system. Vendors usually provide hundreds of APIs, and developers pick and choose from that list as they see fit, causing there to be many more used in the project than really need be. Providing a layer between the application and the OS services needed by the application that is tailored to the needs of the organization simplifies the use of those OS services, making them more manageable and maintainable.

In component technology, specification objects represent constraints rather than programming objects. It emphasizes larger-grained software interfaces and modules. Component infrastructures include MS .Net and Sun Java Enterprise Java Beans, including CORBA. Software architecture for componentware allows parallel, independent development of the system or its parts (good for outsourcing). It tries to standardize means of component interaction so that custom interfaces between individual components are minimized. Distributed components can communicate using standard interfaces by way of CORBA and its IDL, which is centered around the Object Request Broker; CORBA Services provide a way to implement CORBA on particular OS platforms, including Netscape Communicator(!); XML allows for universal data interchange.

A comparison of J2EE and .Net shows that .Net is easier to use but J2EE is more robust and might be preferred by experienced developers, since it allows greater programmer flexibility (hey, I thought that was bad!).

CHAPTER FOUR: SOFTWARE ARCHITECTURE: GOING TO WAR

Here the authors go over some ways that large software projects happen and then lay out steps for an architectural approach. Most large software is very fragile (sucks) except for "telecommunication systems, video games, mainframe operating systems, or rigorously inspected systems (e.g., CMM Level 5)." (Hmm, how about the Linux kernel?) They point out that traditional system assumptions are local and assume that the system will be stable, but in a distributed system one needs to assume that things will be global and unstable. This is like comparing Newtonian mechanics to quantum mechanics. Also, distributed systems typically involve more than one organization to deal with. They recommend the following approach: proactive thinking (actively anticipate problems); use design patterns and anti-patterns to avoid redesigning the wheel; use the Universal Modeling Language to lay out and describe the design.

Traditionally, large systems are pulled off by "heroic programmers" coming in to rescue a flagging project. They make it work under extreme time pressure but typically leave a fragile and undocumented system. Architects must avoid this by initially laying out the project very carefully (using the latest tools like CORBA IDL and UML), making sure that appropriate development tools are used (componentware), and staying on top of the development process to make sure that no heroes are needed.

The architecture-centered development process should include the following steps: system envisioning, requirements analysis, mock-up prototype, architecture planning, architecture prototype, project planning, parallel development (of components), system transition (quality assurance), operations and maintenance (rolling it out), and system migration (moving the organization over to the new system). What's new about all of this? The process has been very much more formalized as the componentware approach has evolved. For example, the architecture-planning step involves documenting these architectures using OSI's ODP: enterprise (how the business works), logical information architecture (what objects are needed to represent the business), computational interface (what will flow between these objects and how), distributed engineering (allocation of responsibilities), and technical selection (component mechanisms).

Chapter Five: Software Architecture: Drill School

Here we are introduced to the idea of using "design levels" to lay out a model. Design levels have been applied to hardware design for decades and are used to simplify by separating concerns. Software may be looked at as having various levels of granularity (objects and classes [the finest, defined by programming language], configurations of objects [the next level up], micro-architecture, frameworks, applications, systems [the coarsest]).

The last five chapters of the book cover how best to provide leadership in a software development project. Topics include how to be a good leader, project management basics, the architect's role vis-à-vis the project manager, various roles in the software design process, teamwork communication skills, using UML, architectural mining, and the psychology of software development. Some highlights follow.

The architect acts as an assistant to the project manager.

Many experienced programmers will not be able make the shift from the procedural to the object paradigm.

Component architectures can be looked at as having four layers: foundation (classes to manage basic object services), domain (business entities), application (specialized domain classes for particular views), and user interface (tailored application classes for various types of user).

There's a "process for creating processes" (sub-projects?). The software architect must be an expert in teamwork and make sure that the team works well together. Also, he or she has responsibility for explaining incremental development to upper management, justifying required shifts in architecture.

It's often necessary to force the "lone wolf" developer into constructive interaction.

There are lot of issues with regard to communications, including running good brainstorming sessions (and being able to decide when they're needed), keeping good records of meetings, making sure consistent terminology and modeling notation is used, and listening well.

Use Universal Modeling Language to document the meta-model of applications and systems. "Design a thing considering its next larger context – a chair in a room."

Architectural mining is extracting from preexisting designs information on how those designs could be applied to the problem at hand.

Polya's paradox: It's often easier to solve a general problem than a specific one (which may have an overwhelming amount of detail).

The traditional approach of "analyze requirements, design, code, test" is being replaced by a more iterative approach wherein this cycle is done repeatedly over the life of the project for a particular piece of the system. This provides for more feedback opportunities.

Psychological techniques include being able to propagandize without seeming to do so. Taking advice is difficult by nature. Architects need to avoid situations where positive feedback causes developers to get carried away in the next phase by trying to top themselves. This usually leads to a system that's too complex.

Signs of egomania in software architects (sampling): Forgetting that the job is about communicating, not winning arguments; use of the royal "we"; referring to developers as "grunts"; believing these signs are about someone else.

"A typical adult gets angry about ten times every day." Not me.

Career advice is offered, including how architecture isn't taught much in schools yet; so, for now, you'll have to develop your own curriculum.

CONCLUSION

As I said earlier, I was skeptical when I first got this book, but I'm now glad I had a chance to spend time with it. It covers a fast-developing field and provides lots of detail that might be handy to delve into if ever one finds oneself working on a large enterprise software project. In fact, it might convince you that you want to be a software architect. Further information can be found at the Worldwide Institute of Software Architects (*http://www.wwisa.org*).