原版风暴·软件工程系列

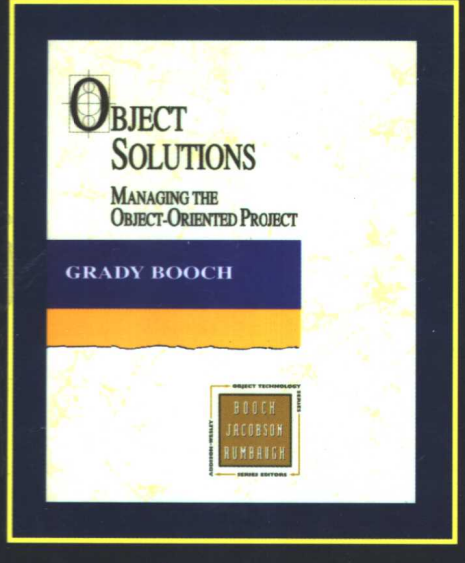
PEARSON Addison Wesley

Object Solutions

Managing the Object-Oriented Project

对象解决方案——管理面向对象项目（影印版）

[美] Grady Booch 著



Rational 首席科学家 Grady Booch 经典作品 ■

软件开发和管理人员必读经典 ■

原汁原味，零距离领悟大师思想精髓 ■

Pearson Education

中国电力出版社
www.infopower.com.cn

原版风暴·软件工程系列

Object Solutions

Managing the Object-Oriented Project

对象解决方案——管理面向对象项目

（影印版）

[美]Grady Booch 著

中国电力出版社

Object Solutions: Managing the Object-Oriented Project (ISBN 0-8053-0594-7)
Grady Booch

北京市版权局著作合同登记号：图字：01-2003-2436

图书在版编目（CIP）数据

对象解决方案——管理面向对象项目 /（美）布什（Booch, G.）著. —影印本. —北京：中国电力出版社，2003
（原版风暴·软件工程系列）
ISBN 7-5083-1509-X
Ⅰ.对... Ⅱ.布... Ⅲ.面向对象语言-软件开发-项目管理-英文 Ⅳ.TP311.52
中国版本图书馆 CIP 数据核字（2003）第 027885 号

责任编辑：闫宏
丛 书 名：原版风暴·软件工程系列
书　　名：对象解决方案——管理面向对象项目（影印版）
编　　著：（美）Grady Booch
出 版 者：中国电力出版社
地址：北京市三里河路6号　邮政编码：100044
电话：（010）88515918　传真：（010）88423191
印　　刷：北京地矿印刷厂
发 行 者：新华书店总店北京发行所
开　　本：787×1092 1/16　**印　　张：**21.5
书　　号：ISBN 7-5083-1509-X
版　　次：2003年7月北京第一版
印　　次：2003年7月第一次印刷
定　　价：42.00 元

Preface

Early adopters of object-oriented technology took it on faith that object orientation was A Good Thing, offering hope for improving some ugly aspect of software development. Some of these primordial efforts truly flourished, some failed, but overall, a number of such projects quietly began to experience the anticipated benefits of objects: better time to market, improved quality, greater resilience to change, and increased levels of reuse. Of course, any new technology is fun to play with for a short while. Indeed, there is a part of our industry that thrives on embracing the latest fad in software development. However, the real business case for any mature technology is that it delivers measurable and sustainable benefits for real projects.

Object-oriented technology has demonstrated its value in a multitude of applications around the world. I have seen object-oriented languages and methods used successfully in such diverse problem domains as securities trading, medical electronics, enterprise-wide information management, air traffic control, semiconductor manufacturing, interactive video gaming, telecommunications network management, and astronomical research. Indeed, I can honestly say that in every industrialized country and in every conceivable application area, I have come across some use of object-oriented technology. Object-oriented stuff is indisputably a part of the mainstream of computing.

There exists an ample and growing body of experience from projects that have applied object-oriented technology. This experience – both good and bad – is useful in guiding new projects. One important conclusion that I draw from all such projects is that object-orientation can have a very positive impact upon software development, but that a project requires much more than just an object-oriented veneer to be successful. Programmers must not abandon sound development principles all in the name of objects. Similarly, managers must understand the subtle impact that objects have upon traditional practices.

SCOPE

In almost every project I have come across, be it a modest two- or three-person effort, to undertakings of epic proportions wherein geopolitical issues dominate, a common set of questions always appears: How do I transition my organization to object-oriented practices? What artifacts should I manage to retain

control? How should I organize my staff? How do I measure the quality of the software being produced? How can I reconcile the creative needs of my individual programmers with management's needs for stability and predictability? Can object-orientation help me help my customers better articulate what they really want? These are all reasonable questions, and their answers strike at the heart of what is different and special about object-oriented technology.

This book serves to answer these and many other related questions, by offering pragmatic advice on the recommended practices and rules of thumb used by successful projects.

This is not a theoretical book, nor is its purpose to explain all the dark corners of object-oriented analysis, design, and programming. My previous work, *Object-Oriented Analysis and Design with Applications*, serves those purposes: it examines the theoretical underpinnings of all things object-oriented, and offers a comprehensive reference to a unified method of object-oriented analysis and design.

Object Solutions provides a direct and balanced treatment on all the important issues of managing object-oriented projects. I have been engaged in hundreds of projects; this book draws upon that broad experience. My intent is to explain what has worked, what has not, and how to distinguish between the two.

AUDIENCE

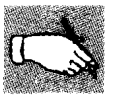
My intended audience includes project managers and senior programmers who want to apply object-oriented technology successfully to their projects, while avoiding the common mistakes that can befall the unwary. Professional programmers will find this book useful as well, giving them insight into the larger issues of turning cool looking object-oriented code into real products; this book will also help to explain why their managers do what they do. Students on their way to becoming professional programmers will come to understand why software development is often not very tidy in the real world, and how industrial-strength projects cope with this disorder.

ORGANIZATION

I have organized this book according to the various functional aspects of managing an object-oriented project. As such, it can either be read from cover to cover or selectively by topic. To make this material more accessible, my general style is to present an issue, discuss its implications, and then offer some recom-

mended practices and rules of thumb. To distinguish these elements in the text, I use the following typographic conventions:

> This is an issue, usually stated in the form of a question followed by its answer, regarding some functional area of project management.



This is a recommended practice, which represents a generally acceptable way of addressing a given issue.

P #

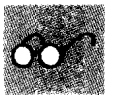
This is a rule of thumb, which represents some quantifiable measure about a particular practice.

R #

I've numbered these practices and rules sequentially, so that specific ones can be referred to easily.

To reinforce certain lessons, I offer examples drawn from a variety of production object-oriented projects, whose details have been changed to protect the guilty. I highlight these examples in the following manner:

This is an example, drawn from some production object-oriented project.



ACKNOWLEDGMENTS

As a compendium of object-oriented wisdom, *Object Solutions* owes an enormous debt to the many professional managers and programmers whose contributions have advanced the state of the practice in object-oriented technology.

The following individuals deserve a special mention for reviewing my work in progress, and providing me with many useful comments and suggestions: Gregory Adams, Glen Andert, Andrew Baer, Dave Bernstein, Mike Dalpee, Rob Daly, Mike Devlin, Richard Dué, Jim Gillespie, Jim Hamilton, Larry Hartweg, Philippe Kruchten, Brian Lyons, Joe Marasco, Sue Mickel, Frank Pappas, Jim Purtilo, Rich Reitman, Walker Royce, Dave Tropeano, Mike Weeks, and Dr. William Wright.

A special thanks goes to my wife, Jan, for keeping me sane during the development of yet another book, and who always gently shows me that there is a rich life beyond all things object-oriented.

Contents

Chapter 1

First Principles

Chapter 1

First Principles

Nature uses only the longest threads to weave her pattern, so each small piece of the fabric reveals the organization of the entire tapestry.

RICHARD FEYNMAN

First, some good news.

From the perspective of the software developer, we live in very interesting times. Consider all the software behind many of the activities that we take for granted in an industrialized society: making a phone call, buying shares in a mutual fund, driving a car, watching a movie, having a medical examination. Sophisticated software is already pervasive, and the fact that it continues to weave itself deeply into the fabric of society creates an insatiable demand for creative architects, abstractionists, and implementers.

Now, some bad news. It is not radical to predict that future software will be evolutionarily more complex. Indeed, two dominant forces drive this trend: the increased connectivity of distributed, high-performance computing systems and greater user expectations for better visualization of and access to information. The first force—increased connectivity—is made possible by the emergence of increasingly high-bandwidth conduits of information and is made practical by economies of scale. The second force—greater user expectations—is largely a consequence of the Nintendo generation that is socially aware of the creative possibilities of automation. Under the influences of both forces, it is reasonable for a consumer to expect that a movie ordered over cable television can be billed directly to his or her bank account. It is reasonable for a scientist to expect on-line access to information in distant laboratories. It is reasonable for an architect to expect the ability to walk through a virtual blueprint created by remote collaborators. It is reasonable for a player to interact with a game whose images are virtually indistinguishable from reality. It is reasonable for a retail business to expect there to be no seams in its mission-critical systems, connecting the event of a customer buying an item to the activities of the company's buyers (who react to rapidly changing consumer tastes) as well as to the activities of the company's marketing organization (which must target new offerings to increasingly specialized groups of consumers). The places where we find seams in such systems, those times when users ask "why can't I do *x*," hint at the fact that we have not yet mastered the complexity of a particular domain.

Even if we ignore the substantial amount of resources already being spent on software maintenance, the world's current supply of software developers would easily be consumed just by the activities of writing software that derive from the natural consequences of these two simple forces. If we add to this equation the largely unproductive tasks of coping with the microprocessor wars, the operating system wars, the programming language wars, and even the methodology wars, we find that scant resources are left to spend on discovering and inventing the next class of so-called "killer" applications. Ultimately, every computer user suffers.

On the positive side, however, software development today is far less constrained by hardware. Compared to a decade ago, many applications operate in a world of abundant MIPS, excess memory, and cheap connectivity. Of

course, there are two sides to this blessing. On the one hand, this means that our hardware must no longer dramatically shape the software architectures that we craft. On the other hand, this embarrassment of riches tends to encourage an appetite for software than will never be satiated.

Thus, we are faced with a simple but fundamental truth:

> Our ability to imagine complex applications will always exceed our ability to develop them.

Actually, this is the most positive of situations: the demands of our imagination drive us continually to improve the ways in which we craft our software.

WHEN BAD THINGS HAPPEN TO GOOD PROJECTS

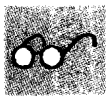
Most software projects start out for all the right reasons: to fill a market need, to provide certain much-needed functionality to a group of end users, to explore some wild theory. For many, writing software is a necessary and unavoidable part of their business, in other words, a secondary concern. A bank is in the business of managing money, not managing software, although software is an essential means to that end. A retail company achieves a competitive advantage in manufacturing and distribution through its software, although its primary business might be providing consumers with the latest fashions in clothes. For other people, writing software is their life, their passion, their joy, something they do while others focus on more mundane problems. In either case, developing software is something that consumes time and significant intellectual energy.

> Why do some software projects fail? Most often, it is because of:
> - A failure to properly manage the risks
> - Building the wrong thing
> - Being blindsided by technology

Unfortunately, as the work of a team unfolds, more than a few projects lose their way. Many projects fail because of a lack of adult supervision.* Increas-

* I don't mean to sound condescending. It is not that such projects fail because of poor management; rather, they fail because of *no* management.

ingly unrealistic schedules and plans are drawn up, cumulatively forming a succession of lies, with no one having the nerve to stand up and acknowledge reality. Petty empires form. Every problem is viewed as "a simple matter of programming," rather than as a reflection of a more systemic problem in the system's architecture or the development process itself. The project's direction and activities are set by the most obnoxious people in the group, because it is easier for management to let this group have its way than it is to make hard decisions when problems arise. Unmanaged projects such as these eventually enter into a "free fall" with no one taking responsibility and everyone waiting for the impact. Usually, the most merciful thing to do in these circumstances is to kill the project before it ruins everything in its path.



A company landed a major contract to supply a common suite of development tools which would ultimately be used by a variety of other contractors on a mission-critical space project. The company did a number of things right: it selected a seasoned architect, trained its people well, selected some good tools, and even instrumented their project so that management could tune the development process over time. However, once the project got going, upper management essentially stepped back and let the technologists on the project run amok. Free from the constraints of any clear goals or firm schedules for incremental releases, the programmers pursued what seemed to them some really cool, albeit irrelevant, implementation issues. While the project's managers spent their time playing political games, the programmers kept rolling forward their schedules, aided by the fact that there was no internal pressure to deliver anything real. In the meantime, end users kept demanding certain high priority deliverables, which were promptly brushed aside as things that would eventually be completed once the team had finished building all the necessary infrastructure. The testing team raised concerns about performance, warning that the framework being crafted would eventually collapse of its own sheer weight, once exposed to the demands of real users. After burning several tens of million dollars, the project was canceled, with hardly any deliverable software to show for all its effort.

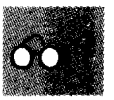
There is a simple lesson to be learned from this project's demise:

Management must actively attack a project's risks, otherwise they will actively attack you.*

P 1

* Gilb, p. 73.

Quite often, projects lose their way because they go adrift in completely uncharted territory. There is no shared vision of the problem being solved. The team is clueless as to the final destination, and so it thrashes about, throwing its scarce energies on what appear to be the most important technical tasks, which often turn out to be of secondary concern once the end users finally see the product. No one takes the time to validate what is being built with any end users or domain experts. Occasionally, so-called analysts capture the essence of the system's real requirements, but for a number of political and social reasons, that essence is never communicated to the people who must design and implement the system. A false sense of understanding pervades the project, and everyone is surprised when users reject the delivered software that was so lovingly crafted in a complete vacuum.



A company was selected to develop a large interstate traffic control system. Early sizing estimates suggested the need for several hundred developers (in itself an early warning sign). Two new buildings were erected, one to house the project's analysts, and the other to house the project's designers and implementers. Not surprisingly, these artificial physical boundaries introduced significant amounts of noise in the communication of the user's requirements down to the level of the project's programmers. Memo wars raged, with analysts lobbing reports containing their view of the problem over the walls to the poor, isolated designers, who would from time to time fire back with reports of their own. The project's programmers were rarely exposed to any real end users; they were too busy coding, and besides, as the project's culture dictated it was thought that most of its programmers would not know how to deal with users anyway. As time unfolded and the project's real requirements became clear, there was considerable delay in communicating these changing needs from the users through the analysts to the designers, resulting in significant schedule slips as well as much broken glass in the customer/vendor relationship.

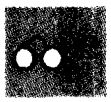
The experience from this project, and too many others like it, prompts the following recommended practice:

Involve real users throughout the software development process; their presence is a constant reminder why and for whom the software is being crafted.

P 2

Occasionally, projects fail because they are blindsided by the very technology being employed to build the software itself. Tools break at the most inop-

portune moments, lacking the capacity to handle the project's exponentially increasing complexity. From time to time, the project's tools prove to be just plain erroneous, requiring programmers to perform unnatural acts to get around these limitations. Third-party software suppliers sometimes do not deliver what they originally promised; often some expected functionality is lacking, or performance is less than expected. In the worst of all possible scenarios, the supplier simply goes out of business, leaving the project totally exposed. Technology backlash happens most often when forces in the marketplace, beyond a project's local control, change the technology rules out from under it: a hardware platform vendor stops making a product, operating system interfaces and features are changed by their supplier faster than the project can meaningfully keep up, end user's tastes change and their expectations rise because of some other really neat program one of them recently saw mentioned in the latest trade magazine (even though that product later proved to be vaporware). Although the latest language/tool/method selected by a project might promise real benefits, extracting those benefits is usually much harder than it first appears. When blindsided by technology, there usually are not enough programming hours in the day to recover, without reducing the functionality of the system the project had promised to deliver. Ultimately, this is all very embarrassing for a software development organization: as professionals, the last thing to expect is for your own technology to turn on you.



A securities trading company made a significant investment in object-oriented stuff, buying the latest brand workstations and programming tools for all its developers. Shortly thereafter, the workstation vendor decided that it really was a software company after all, and so it stopped making any more of its hardware.

Too often, problems with the underlying technology take the blame for a project's failure when the real culprit is really non-technical, namely, the lack of active management that should have anticipated and planned contingencies for the technical risk in the first place. Still, we do not live in a perfect world, and thus:

P 3

Where possible, do not bind your project to any single-source technology, but if you must (such as when that technology offers some compelling advantage even in the face of its risk), build firewalls into your architecture and process so that your project will not unravel even if the technology does.

ESTABLISHING A PROJECT'S FOCUS

Despite these three failure modes, many software projects that start out for all the right reasons really do achieve at least some modest amount of success. However, even the most successful projects seem to take longer, involve more intellectual effort, and require more crisis management than we really believe they ever should. Unfortunately, as Parnas suggests, we can never have a completely rational development process because:*

- A system's users typically do not know exactly what they want and are unable to articulate all that they do know.
- Even if we could state all of a system's requirements, there are many details about a system that we can only discover once we are well into its implementation.
- Even if we knew all of these details, there are fundamental limits to the amount of complexity that humans can master.
- Even if we could master all this complexity, there are external forces, far beyond a project's control, that lead to changes in requirements, some of which may invalidate earlier decisions.
- Systems built by humans are always subject to human error.
- As we embark on each new project, we bring with us the intellectual baggage of ideas from earlier designs as well as the economic baggage of existing software, both of which shape our decisions independent of a system's real requirements.

Parnas goes on to observe that "For all of these reasons, the picture of the software designer deriving his design in a rational, error-free way from a statement of requirements is quite unrealistic." Fortunately, as Parnas observes, and as I'll discuss further in the next chapter, it is possible, and indeed desirable, to fake it. In one way or another, establishing the semblance of a rational design process is exactly what every successful project has to do.

Every successful project also entails the making of a plethora of technical decisions. Some of these decisions have sweeping implications for the system under construction, such as the decision to use a certain client/server topology, the decision to use a particular windowing framework, or the decision to use a relational database. These I call *strategic decisions*, because each denotes a fundamental architectural pattern. Other decisions are much more local in nature, such as the decision to use a particular programming idiom for iteration, the decisions that shape the interface of an individual class, or the decision to use a specific vendor's relational database. These I call *tactical decisions*. Together,

* Parnas, p. 251.