Foreword

Be Careful What You Wish For. You May Get It.

A common theme in folk tales is the story of a man who is given three wishes. After spending the first two wishes on a golden castle and a beautiful princess, and discovering the operations and maintenance implications of each, he is happy to spend the third wish getting back to where he came from.

This kind of story happens a lot in software development. Much of it is due to an overfixation in hurrying to get a set of requirements templates completely filled out by the Software Requirements Review deadline. The deadline is usually set very early by the upper management, based on such self-fulfilling logic as, "We need to hurry up and start coding, because we will have a lot of debugging to do."

This pushes the requirements analysts into a hurried activity to minimize such dysfunctional progress metrics as "percentage of incomplete requirements" so that the requirements can be signed off and the "real software work" can begin. As a result, the project gets locked into requirements for hasty and later-regretted wishes.

Suzanne and James Robertson have seen a lot of software projects fail in this way. They have seen it happen with teams misapplying the Robertsons' own requirements templates (named Volere after the Italian word for "to wish or want"), and have written this book in response. The book rightly emphasizes that the main purpose of the requirements activity is not template-filling. It is rather a *mutual learning activity* in which specialists from different areas and cultures try to fit *what people in a user culture need* to *what people in a development culture can produce*, subject to some *real-world constraints that people in an ownership and management culture need to live within*.

These real-world constraints lead to another important conclusion about software requirements.

It is not a requirement if you cannot afford it.

A good example of this happened on a project I was involved in at TRW. A hasty requirements activity had locked the project into a commitment to a 1-second response time to user queries. This was based largely on marketing enthusiasm and some small-scale COTS product demos.

Belatedly, the designers found that the COTS products could only achieve a 2.5-second response time when scaled up to the customer's full workload. The best custom solution they could find carried a price tag of $100 million, as compared to the customer's real-world constraint of a $30 million budget.

Belatedly, the customer commissioned some prototypes. These determined that a 4-second response time was adequate for 90% of the queries, and that the COTS product could be used to provide an acceptable solution within the $30 million budget constraint. Fortunately, this was discovered before delivery, but only after 18 months and a lot of money spent on a design that had to be thrown away.

This experience leads to some further important conclusions about software requirements (and requirements for a system in general).

Don't lock yourself into a set of requirements until you're sure you have a design that will satisfy them.

This is covered very well by the Robertsons in Chapter 11 on the need for strong overlap between requirements, design, building, and integration activities; and in Chapters 5 and 6 emphasizing the importance of early prototypes and simulations to validate requirements before committing to them. Also in Chapter 9, they emphasize the importance of "signing on" to requirements and solution commitments rather than "signing off" on a set of requirements templates.

Watch out for nonfunctional requirements (NFRs).

The best design solution is often a discontinuous function of the NFR level, and you don't want to get caught in the wrong side of the discontinuity, as did our project in the 1-second response time, $100 million cost situation. NFRs are emphasized by the Robertsons in Chapters 6 and 11, and covered more in their previous book, *Mastering the Requirements Process*. Again, they require a lot of caution because:

A tiny change in NFRs can cause a huge change in the cost.

Just think. All it takes is changing one character from a "4" to a "1" in a 1000-page requirements specification to turn a $30 million project into a $100 million project.

In Chapter 8, the Robertsons advocate using function points to estimate the cost of the requirements activity. Note the restricted target of this estimate, as they are aware of the flawed notion advanced by overenthusiastic function point advocates that you can estimate costs from the requirements without thinking about the design. As we saw in the example above, one character change in a 1000-page requirements specification changed cost by a factor of over three. The big thing that changed was the design, which is the preferred basis for estimating costs.

Thus, when following the Robertsons' excellent guidance in Chapter 2 on developing a return on investment (ROI) analysis for your system and its requirements, it is important to recognize that:

- An ROI analysis is based on both benefit and cost estimates.
- Benefits are best estimated from the requirements, which tell you *what* the system will do for the users, and *how well*.
- Costs are best estimated from the design, which tells you *how* the system will be built (including the cost implications of using very high-level languages and using COTS, open source, or product-line components, none of which are included in the estimation formulas in Chapter 8).

That said, you'll find this book a treasure trove of experience-based guidelines and illustrative examples on how to get the requirements right on your project. These include guidelines and examples on treating the requirements as an investment activity in Chapter 2; getting the right people involved and understanding their cultures in Chapter 3; techniques for stimulating mutual learning and a shared vision among stakeholders in Chapter 4; the use of prototypes and simulations in Chapters 5 and 6; dealing with legacy systems in Chapter 7; and managing systems requirements, systems of systems requirements, and requirements processes in Chapters 9, 10, and 11. Each chapter concludes with a nicely balanced set of "What do I do right now?" and "What's the least that I can get away with?" checklists.

As a bottom line, the book does a wonderful job of lifting its readers from a focus on templates and objects to a focus on people's needs, capabilities, and ability to work together to achieve a shared vision of the requirements (and the design) for a system that will satisfy all their needs and constraints. I hope you have the opportunity to use its practices on your next project.

Barry Boehm