Contents

1

Why I Wrote This Book

As an undergraduate in physics, I'd been entranced by the idea that the world could be captured by simple equations like $F = ma$. When I became a programmer, and later a computer science researcher, I gravitated towards the field of formal methods, because it promised to do something similar for software: to express its very essence in a succinct logic.

A Passion for Design

My main research contribution in the 30 years since my PhD has been Alloy,[3] a language for describing software designs and analyzing them automatically. It's been an exciting and satisfying journey for me, but I came to realize over time that the essence of software doesn't lie in any logic or analysis. What really fascinated me wasn't the question that consumed most formal methods researchers—namely how to check that a program's behavior conforms exactly to its specification—but rather the question of *design*.[4]

I mean "design" here in the same sense that the word is used in other design disciplines: the shaping of some artifact to meet a human need. Design, as the architect Christopher Alexander put it, is about creating a *form* to fit a *context*. For software, that means determining what the behavior of the software should be: what controls it will offer, and what responses it will provide in return. These questions have no right or wrong answers, only better or worse ones.[5]

I wanted to know why some software products seem so natural and elegant, react predictably once you master the basics, and let you combine their features in powerful ways. And to pinpoint why other products just seem wrong: cluttered with needless complexity, and behaving in unexpected and inconsistent ways. Surely, I thought, there must be some essential principles, some theory of software design, that could explain all of this. It would not only explain why some software products are good and some are bad, but it would help you fix the problems and avoid them in the first place.

Design in Computer Science and Other Fields

I started to look around. Within my own subfield (formal methods, software engineering and programming languages), such a theory exists for what you might call "internal design"—namely the design of the structure of the code. Programmers have a rich language of design, and well-established criteria for what distinguishes good designs from bad ones. But no such language or criteria exist for software design in the user-facing sense, namely design that determines how software is experienced as a form in context.[6]

Internal code design is very important and influences primarily what software engineers call "maintainability," which means how easy (or hard) the code is to change over time as needs evolve. It also influences performance and reliability. But the key decisions that determine whether a software application or system is useful and fulfills its users' needs lie elsewhere, in the kind of software design in which the functionality and the patterns of interaction with the user are shaped.

These big questions were at one time more central in computer science. In the field of software engineering, they came up in workshops on software design, specification and requirements; in the field of human-computer interaction, they permeated early work on graphical user interfaces and computational models of user behavior.[7]

But as time passed, they became less fashionable, and they faded away. Research in software engineering narrowed, and eliminating defects—whether by testing or more sophisticated means such as program verification—became synonymous with software quality.[8] But you can't get there from here: if your software has the wrong design, there's no amount of defect elimination that will fix it, short of going back to the very start and fixing the design itself.[9]

Research in human-computer interaction (HCI) shifted to novel interaction technologies, to tools and frameworks, to niche domains, and to other disciplines (such as ethnography and sociology). Both software engineering and HCI embraced empiricism enthusiastically, largely in the misguided hope that this would bring respectability. Instead, the demand for concrete measures of success seems to have led researchers towards less ambitious projects that admit easier evaluation, and has stymied progress on bigger and more important questions.[10]

Puzzlingly, even as interest in design seems to have waned, talk of "design" is everywhere. This is not in fact a contradiction. The talk, almost exclusively, is about the *process* of design, whether in the context of "design thinking" (a compelling packaging of iterative design processes), or of "agile" software development. These processes are undoubtedly valuable (so long as they are applied judiciously and not as panaceas), but they are for the most part content-free. I mean that not to disparage but to describe. Design thinking, for example, might tell you to develop your solution hand in hand with your understanding of the problem, or to engage in alternating phases of brainstorming ("divergence") and reduction ("convergence"). But no design thinking book that I have read talks in depth about any particular designs and how the process sheds light on them. The very domain-independence of design thinking may be the key to its widespread appeal and applicability—but also the reason it has little to say about deeper challenges of design in a particular domain such as software.[11]

Clarity & Simplicity in Design

When I began the Alloy project, with the goal of creating a design language that was amenable to automatic analysis, I was critical of existing modeling and specification languages whose lack of tool support rendered them "write-only." This snide dismissal was not entirely unwarranted. After all, why would you go to the trouble of constructing an elaborate design model if you couldn't then do anything with it? I argued, in particular, that the designer's effort should be rewarded immediately with "push-button automation" that would instantly give you feedback in the form of surprising scenarios that would challenge you to think more deeply about your design.[12]

I don't think I was wrong, and Alloy's automation did indeed change the experience of design modeling. But I had underestimated the value of writing down a design. In fact, it was a not very well guarded secret amongst formal methods researchers (who were eager to demonstrate the efficacy of their tools by finding flaws in existing designs) that a high proportion of the flaws were detected *before* the tools were even run! Just transcribing the design into logic was enough to reveal serious problems. The software engineering researcher Michael Jackson credits not the logic per se but the very difficulty of using it, and once mischievously suggested that the quality of software systems might be improved if designers were simply required to record their designs in Latin.

Clarity is good not only for finding design flaws after the fact. It is also the key to good design in the first place. In teaching programming and software engineering over the last thirty years, I've become increasingly convinced that the determinant of success when you're developing software isn't whether you use the latest programming languages and tools, or the management process you follow (agile or otherwise), or even how you structure the code. It's simply whether you know what you are trying to do. If your goals are clear, and your design is clear—and it's clear how your design meets the goals—your code will tend to be clear too. And if something isn't working, it will be clear how to fix it.[13]

It is this clarity that distinguishes great software from the rest. When the Apple Macintosh came out in 1984, people could see immediately how to use folders to organize their files; the complexities of previous operating systems (such as Unix, which made even the command to move files between folders complicated) seemed to have evaporated.

But what exactly is this clarity, and how is it achieved? As early as the 1960s, the central role of "conceptual models" has been recognized. The challenge was not merely to *convey* the software's conceptual model to the user so that her internal version ("mental model") was aligned with the programmers', but to treat it as a subject of design in its own right. With the right conceptual model, the software would be easy to understand and thus easy to use. This was a great idea, but nobody seems to have pursued it, and so until now "concepts" have remained a vague, if inspiring, notion.[14]

How This Project Came About

Convinced that conceptual models were indeed the essence of software, I started about eight years ago trying to figure out what they might be. I wanted to give them concrete expression, so that I could point to some software's conceptual model, compare it to others (and to the mental models of users), and have an explicit focus for design discussions.

That didn't seem so hard. After all, a plausible first cut at a conceptual model might be just a description of the software's behavior, made suitably abstract to remove incidental and "non-conceptual" aspects (such as the details of the physical user interface). What proved much harder was finding appropriate

structure in the model. I had an inkling that a conceptual model should be made up of concepts, but I didn't know what a concept was.

In a social media app such as Facebook, for example, it seemed to me that there should be a concept associated with liking things. This concept surely wasn't a function or action (such as the behavior bound to the button you click to like a post); there are too many of those, and they only tell part of the story. It also surely wasn't an object or entity (such as the "like" itself that your action produced), since at the very least the concept seemed to be about the *relationship* between things and their likes. It also seemed essential to me that the concept of liking was not associated with any particular kind of thing: you could like posts, comments, pages, and so on. The concept, in programming lingo, is "generic" or "polymorphic."

This Book: Opening a Conversation

This book is the result of my explorations to date. Driven by dozens of design issues in widely used applications, I've evolved a new approach to software design, refining and testing it along the way. A happy aspect of this project has been that every app failure or frustration had a silver lining: a chance to extend my repertoire of examples. It has also given me greater sympathy and respect for the designers when my analysis revealed the full complexity of the problem they faced.

Of course, the problem of software design is not solved. But as my friend Kirsten Olson wisely advised me: a book should aim to start a conversation, not to end one. In the course of giving many talks about this project, I've been thrilled to discover that it seems to resonate with audiences more than any of my previous ones. I suspect this is because software design is something we all want to talk about, but we have not known how to have that conversation.

So to you, my readers—fellow researchers, designers and users—I present this book as my opening gambit in what I hope to be a fruitful and enjoyable conversation.

Index of Applications

Index of Concepts

Index of Names

Index of Topics