SOFTWARE DEVELOPMENT PEARLS

Lessons from Fifty Years of Software Experience

KARL WIEGERS

Foreword by Steve McConnell

Praise for *Software Development Pearls*

"This is a collection of lessons that Karl has learned over his long and, I can say this honestly, distinguished career. It is a retrospective of all the good things (and some of the bad) he picked up along the way. However, this is not a recollection of 'It was like this in my day' aphorisms, but lessons that are relevant and will benefit anybody involved, even tangentially, in software development today. The book is surprising. It is not simply a list of pearls of wisdom—each lesson is carefully argued and explained. Each one carries an explanation of why it is important to you, and importantly, how you might bring the lesson to your reality."

—**James Robertson,** *Author of* Mastering the Requirements Process

"Wouldn't it be great to gain a lifetime's experience early in your career, when it's most useful, without having to pay for the inevitable errors of your own experience? Much of Karl Wiegers's half-century in software and management has been as a consultant, where he's often been called upon to rectify debacles of other people's making. In *Software Development Pearls,* Karl lays out the most common and egregious types of maladies that he's run into. It's valuable to know where the most expensive potholes are and which potholes people keep hitting time and time again.

"Not just a disaster correspondent, Karl is well versed in the best techniques of business analysis, software engineering, and project management. So from Karl's experience and knowledge you'll gain concise but important insights into how to recover from setbacks as well as how to avoid them in the first place.

"Forty-six years ago I was lucky enough to stumble onto Fred Brooks's classic *The Mythical Man-Month*, which gave me tremendous insights into my new career. Karl's book is in a similar vein, but broader in scope and more relevant for today's world. My own half-century of experience confirms that he's right on the money with the lessons that he's chosen for *Software Development Pearls*."

—**Meilir Page-Jones,** *Senior Business Analyst, Wayland Systems Inc.*

"Karl has created yet another wonderful book full of well-rounded advice for software developers. His wisdom will be relatable to all development professionals and students—young and old, new and experienced. Although I've been doing software development for many years, this book brought timely reminders of things my team should do better. I cannot wait to have our new-to-the-job team members read this.

"*Software Development Pearls* is rooted in actual experiences from many years of real projects, with a dose of thorough research to back up the lessons. As with all of Karl's books, he keeps it light and engaging, chock-full of relatable stories and a few funny comments. You can read it from front to back or just dive into a particular section that's relevant to the areas you're looking to improve today. An enjoyable read plus practical advice—you can't go wrong!"

—**Joy Beatty,** *Vice President at Seilevel*

- **Abstraction.** Abstraction allows developers to write code that doesn't depend on specific implementation details, such as the platform's operating system or the user interface. Abstraction facilitates portability and reuse.
- **Defined and respected interfaces.** A well-defined module interface makes it easy for developers of other code modules to access that module's services. It also facilitates replacing a module when necessary, because the interface it presents to the rest of the system remains unchanged. The same principle applies to external interfaces the system presents to the outside world.

Design discussions sometimes treat design as a straightforward extension of requirements or bundle it in with implementation as "development," but it's better to regard design as a distinct exercise. Someone will design the software on every project, whether or not they treat design as a discrete activity and whether or not they record the designs in some form.

I worked on a project where coding directly from the requirements would have yielded a far more complex program than we devised by exploring design options first. It wasn't apparent from the requirements, but three of the system's eight computational modules used the same algorithm, three more shared a common algorithm, and the last two used a third algorithm. Eventually, we would have noticed that we were writing the same code multiple times, but we were happy to spot the repetition before implementation.

Instead of leaping straight from requirements to code, it's well worth the time to evaluate alternative design approaches and choose the most appropriate one. This chapter describes six valuable lessons I've learned from my software design experiences.

First Steps: Design

I suggest you spend a few minutes on the following activities before reading the design-related lessons in this chapter. As you read the lessons, contemplate to what extent each of them applies to your organization or project team.

1. List design practices that your organization is especially good at. Is information about those practices documented to remind team members about them and make it easy to apply them?
2. Identify any problems—points of pain—that you can attribute to shortcomings in how project teams deal with architectural, detailed, database, user experience, or other design activities.

3. State the impacts that each problem has on your ability to complete projects successfully. How do the problems impede achieving business success for both the development organization and its customers? Design shortcomings can lead to brittle systems that are not easily modified or improved, subpar performance, duplicated code, inconsistencies within a product or across related products, and usability problems.
4. For each problem from Step 2, identify the root causes that trigger the problem or make it worse. Problems, impacts, and root causes can blur together, so try to tease them apart and see their connections. You might find multiple root causes that contribute to the same problem, as well as several problems that arise from a single root cause.
5. As you read this chapter, list any practices that would be useful to your team.

Lesson 17 Design demands iteration.

In his classic book *The Mythical Man-Month,* Frederick P. Brooks, Jr. (1995) advises, "Plan to throw one away; you will, anyhow." Brooks is referring to the idea that, on large projects, it's advisable to create a pilot or preproduction system to figure out how best to build the complete system. That's an expensive prospect, particularly if the system includes hardware components. However, a pilot system is valuable if you have technical feasibility questions or if a suitable design strategy isn't clear initially. A pilot system also reveals the unknown unknowns, factors you hadn't yet realized were significant.

While you're unlikely to build and then discard a preliminary version of most products, you do need to iterate on potential designs before the team gets very far into construction. Creating the simplest possible design sounds attractive, and it does accelerate solution delivery. Rapid delivery might meet a customer's short-term perception of value, but it may not be the best long-term strategy as the product grows over time.

There's always more than one design solution for a software problem and seldom a single best solution (Glass 2003). The first design approach you conceive won't be the best option. Norman Kerth, a highly experienced designer of software, furniture, and other items, explained it to me nicely:

> You haven't done your design job if you haven't thought of at least three solutions, discarded all of them because they weren't good enough, and then combined the best parts of all of them into a superior fourth solution. Sometimes, after considering three options, you realize that you don't really understand the problem. After reflection, you might discover a simple solution when you generalize the problem.

Software design isn't a linear, orderly, systematic, or predictable process. Top designers often focus first on the hard parts where a solution might not be obvious or perhaps even feasible (Glass 2003). Several methods facilitate iteration as a designer moves from an initial concept to an effective solution. One method is to create and refine graphical models—diagrams—of the proposed designs. This technique is addressed in Lesson 18, "It's cheaper to iterate at higher levels of abstraction." Prototyping is another valuable technique for iterating on both technical and UX designs.

The Power of Prototypes

A prototype is a partial, preliminary, or possible solution. You build a piece of the system as an experiment, testing the hypothesis that you understand how to design the system well. If the experiment fails, you redesign it and try again. A prototype is valuable for assessing and reducing risk, particularly if you're employing a novel architectural or design pattern that you want to validate before committing to it.

If you intend a prototype to grow into the product, you must build it with production-level quality from the beginning.

Before you construct a prototype, determine whether you intend to discard it and then develop the real thing, or grow the preliminary solution into the product. A key point is that if you intend a prototype to grow into the product, you must build it with production-level quality from the beginning. That takes more effort than building something temporary that you'll discard after it has served its purpose. The more work you put into a prototype, the more reluctant you become to change it significantly or throw it away, which impedes the iteration mindset. Your prototyping approach should encourage cyclical refinement and even starting over if necessary.

Agile teams sometimes create stories called *spikes* to research technical approaches, resolve uncertainty, and reduce risk before committing to a specific solution (Leffingwell 2011). Unlike other user stories, a spike's prime deliverable is not working code, but rather knowledge. Spikes could involve technical prototypes, UI prototypes, or both, depending on the information sought. A spike should have a clear goal, just like a scientific experiment. The developer has a hypothesis to test. The spike should be designed to provide evidence for or against the hypothesis, test and confirm the validity of some approach, or allow the team to make an informed technical decision quickly.

Proofs of Concept

Proof-of-concept prototypes, also called *vertical prototypes,* are valuable for validating a proposed architecture. I once worked on a project that envisioned an unconventional client–server approach. The architecture made sense in our computing environment, but we wanted to make sure we weren't painting ourselves into a technical corner. We built a proof-of-concept prototype with a vertical slice of functionality from the UI through the communication layers and the computational engine. It worked, so we felt confident this design was workable.

Experimenting on a proof-of-concept prototype is a way to iterate at a relatively low cost, although you do need to build some executable software. Such prototypes are valuable for assessing the proposed design's technical aspects: architecture, algorithms, database structure, system interfaces, and communications. You can evaluate architectures against their needed properties—such as performance, security, safety, and reliability—and then refine them progressively.

Mock-ups

User interface designs always require iteration. Even if you're following established UI conventions, you should perform at least informal usability testing to choose appropriate controls and layouts to meet your ease-of-learning, ease-of-use, and accessibility goals. For instance, A/B testing is an approach in which you present users with two UI alternatives for a given operation so that they can choose which one makes the most sense to them. The people who conduct an A/B test can observe user behaviors with the different approaches to determine which option is more intuitive or leads to more successful outcomes. It's simpler, faster, and cheaper to conduct such experiments while you're still exploring the design than to react to post-delivery customer complaints or lower-than-expected click-through rates on a web page.

As with requirements, UX designs benefit from the progressive refinement of detail through prototyping. You can create *mock-ups,* also called *horizontal* *prototypes* because they consist of just a thin layer of user interface with no functional substance below it. Mock-ups range from basic screen sketches to executable interfaces that look authentic but don't do real work (Coleman and Goodwin 2017). Even simple paper prototypes are valuable and are quick to create and modify. You can use a word processing document or even index cards to lay out the data elements in boxes representing potential screens, see how the elements relate to each other, and note which elements are user input and which are displayed results. Watch out for these traps with user interface prototyping.

- Spending too much time perfecting the UI's cosmetics ("How about a darker red for this text?") before you've mastered the screen flow and functional layouts. Get the broad strokes right first.
- Customers or managers thinking the software must be nearly done because the UI looks good, even if there's nothing behind it but simulated functions. A less polished prototype shows that it isn't yet finished.
- Coaching prototype evaluators as they attempt to perform a task that isn't obvious to them. You can't judge usability if you're helping the users learn and use the prototype.

If you don't invest in repeatedly exploring both user experience and technical designs before committing to them, you risk delivering products that customers don't like. Thoughtlessly designed products annoy customers, waste their time, erode their goodwill toward your product and company, and generate bad reviews (Wiegers 2021). A few more iteration cycles will get you much closer to useful and enjoyable designs.

Lesson 18 It's cheaper to iterate at higher levels of abstraction.

One way to revise a design is to build the entire product several times, improving it with each cycle. That's not practical. Another way is to implement just small portions of the solution, including the hard parts or the parts you don't understand yet, to determine what design approaches will work best. That's the idea behind prototyping, as we saw in the preceding lesson.

Yet a third strategy is to build an operational portion of the system so that users can work with it and provide feedback that improves the subsequent extensions.

This incremental approach is the thrust of agile software development. It's a good way to solicit user input on something tangible so that you can adjust the work to meet customer needs better. You might discover that your initial design was satisfactory for that first chunk of the product, but it won't support the product's growth through continued development. Or you could find that the team didn't make well-thought-out design decisions in the rush to deliver working software, so they must revisit those decisions later. (See Lesson 50, "Today's 'gotta get it out right away' development project is tomorrow's maintenance nightmare.") Shortcomings in system architecture and database designs often are costly and time consuming to rectify. Therefore, building a hasty implementation in the first few iterations without carefully exploring the technical underpinnings can come back to painfully bite the team.

The common factor for all three of these design strategies is building working software to evaluate your design ideas. Incrementally improving designs in this fashion is relatively slow and expensive. You could find yourself reworking what you've built several times to reach a suitable design.

An alternative approach is to iterate at a higher abstraction level than executable software. As Figure 3.3 illustrates, it's more expensive to iterate on artifacts at low abstraction levels than high. That's because you have to invest more work in creating the artifacts you're assessing and revising. Design modeling provides a cost-effective iteration alternative.

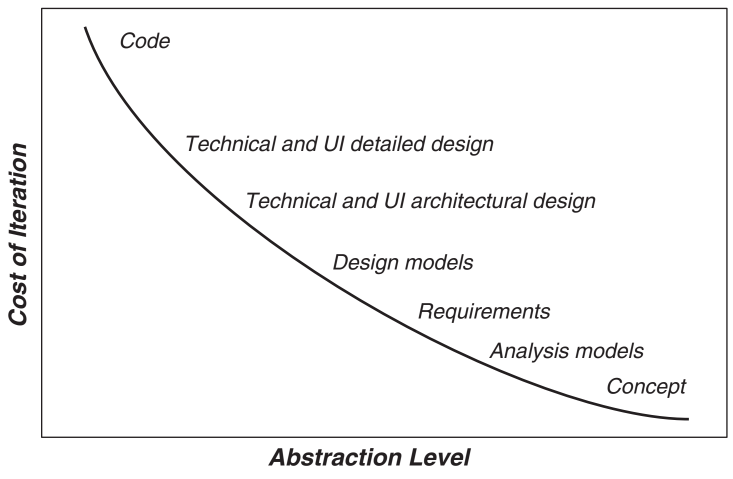

Figure 3.3 *The cost of iteration is lower at higher levels of abstraction.*