developer.*
A Web Magazine for Software Developers

Software Design and Programmers

by Daniel Read

> *The architect without the stonemason is not designing cathedrals, but castles in the air. --Gerald Weinberg*

How important are software design skills to a programmer? Programmers, in the traditional, and perhaps most widespread, view of the software development process, are not themselves seen as designers but rather as people who implement the designs of other people. The job of the programmer, after all, is to write code. Code is viewed as a "construction" activity, and everyone knows you have to complete the design before beginning construction. The real design work is performed by specialized software designers. Designers create the designs and hand them off to programmers, who turn them into code according to the designer's specifications. In this view, then, the programmer only needs enough design skills to understand the designs given to him. The programmer's main job is to master the tools of her trade.

This view, of course, only tells one story, since there is great variety among software development projects. Let's consider a spectrum of software development "realities." At one end of the spectrum we have the situation described above. This hand-off based scenario occurs especially on larger, more complex projects, and especially within organizations that have a longstanding traditional software engineering culture. Specialization of function is a key component on these kinds of projects. Analysts specialize in gathering and analyzing requirements, which are handed off to designers who specialize in producing design specifications, which are handed off to programmers who specialize in producing code.

At the opposite end of the spectrum, best represented by the example of Extreme Programming (XP), there are no designers, just programmers, the programmers are responsible for the design of the system. In this situation, there is no room for specialization. According to Pete McBreen, in his excellent analysis of the Extreme Programming methodology and phenomenon, *Questioning Extreme Programming*, "The choice that XP makes is to keep as many as possible of the design-related activities concentrated in one role--the programmer." [McBreen, 2003, p. 97] This reality is also well represented in a less formal sense by the millions of one or two person software development shops in which the same people do just about everything--requirements, design, construction, testing, deployment, documentation, training, and support.

Many other realities fall somewhere in between the two poles a) of pure, traditional, segmented software engineering, where highly detailed "complete designs" are handed off to programmers, and b) Extreme Programming and micro-size development teams, where programmers are the stars of the show. In the "middle realities" between these poles there are designers, lead programmers, or "architects" who create a design (in isolation or in

collaboration with some or all of the programmers), but the design itself is (intentionally or unintentionally) not a complete design. Furthermore, the documentation of the design will have wide disparities in formality and format from one reality to another. In these situations, either explicitly or implicitly, the programmers have responsibility over some portion of the design, but not all of it. The programmer's job is to fill in the blanks in the design as she writes the code.

There is one thing that all of the points along this spectrum have in common: even in the "programmers just write the code" software engineering view, *all programmers are also software designers*. That bears repeating: all programmers are also software designers. Unfortunately, this fact is not often enough recognized or acknowledged, which leads to misconceptions about the nature of software development, the role of the programmer, and the skills that programmers need to have. (Programmers, when was the last time you were tested on, or even asked about, your design skills in a job interview?)

In an article for *IEEE Software* magazine called "Software Engineering Is Not Enough," James A. Whittaker and Steve Atkin do an excellent job of skewering the idea that code construction is a rote activity. The picture they paint is a vivid one, so I will quote more than a little from the article:

> Imagine that you know nothing about software development. So, to learn about it, you pick up a book with "Software Engineering," or something similar, in the title. Certainly, you might expect that software engineering texts would be about engineering software. Can you imagine drawing the conclusion that writing code is simple--that code is just a translation of a design into a language that the computer can understand? Well, this conclusion might not seem so far-fetched when it has support from an authority:
>
> > *The only design decisions made at the coding level address the small implementation details that enable the procedural design to be coded.* [Pressman, 1997, p. 346]
>
> Really? How many times does the design of a nontrivial system translate into a programming language without some trouble? The reason we call them *designs* in the first place is that they are *not* programs. The nature of designs is that they abstract many details that must eventually be coded. [Whittaker, 2002, p.108]

The scary part is that the software engineering texts that Whittaker and Atkin so skillfully deride are the standard texts used in college software development courses. Whittaker and Atkin continue with this criticism two pages later:

> Finally, you decide that you simply read the wrong section of the software engineering book, so you try to find the sections that cover coding. A glance at the table of contents, however, shows few other places to look. For example, *Software Engineering: A Practitioners Approach*, McGraw-Hill's

> best-selling software engineering text, does not have a single program listing. Neither does it have a design that is translated into a program. Instead, the book is replete with project management, cost estimation, and design concepts. *Software Engineering: Theory and Practice*, Prentice Hall's bestseller, does dedicate 22 pages to coding. However, this is only slightly more than four percent of the book's 543 pages. [Whittaker, 2002, p. 110]

(I recommend seeking out this article as the passages I have quoted are only a launching point for a terrific discussion of specific issues to consider before, during, and after code construction.)

Given a world where "coding is trivial" seems to be the prevailing viewpoint, it is no wonder that many working software professionals sought a new way of thinking about the relationship between and nature of design and construction. One approach that has arisen as an alternative to the software engineering approach is the craft-based approach, which de-emphasizes complex processes, specialization, and hand-offs.[1] Extreme Programming is an example of a craft-centric methodology. There are many others as well.

Extreme Programming, and related techniques such as refactoring and "test first design," arose from the work Smalltalk developers Kent Beck and Ward Cunningham did together. The ideas Beck and Cunningham were working with were part of a burgeoning object oriented movement, in which the Smalltalk language and community played a critical role. According to Pete McBreen in *Questioning Extreme Programming*, "The idea that the source code is the design was widespread in the Smalltalk community of the 1980s." [McBreen, 2003, p. 100]

Extreme Programming has at its core the idea that the code *is* the design and that the best way to simultaneously achieve the best design and the highest quality code is to keep the design and coding activities tightly coupled, so much so that the they are performed by the same people--programmers. Refactoring, a key XP concept, codifies a set of methods for incrementally altering, in a controlled manner, the design embodied in code, further leveraging the programmers role as designer. Two other key XP concepts, "test first design" and automated unit testing, are based on the idea that, not only is the code the design, but the design is not complete unless it can be verified through testing. It is, of course, the programmer's job to verify the design through unit testing.

It is not much of a stretch to conclude that one of the reasons Extreme Programming (and the Agile Methodology movement in general) have become so popular, especially with people who love to write code, is that they recognize (explicitly or implicitly) that programmers have a critical role to play in software design--even when they are not given the responsibility to create or alter the "design." Academics and practitioners who champion the traditional software engineering point of view often lament that the results of their research and publications do not trickle down fast enough to practicing software developers.

Perhaps this is because, as Whittaker and Atkin point out, too much of the software engineering literature neglects the role of the programmer. Turning back to the specific

example of Extreme Programming, McBreen is right on the money when he writes, "XP does directly challenge some of the sacred cows of the software engineering community. By elevating the status of the programmer, it is turning nearly 30 years of software engineering orthodoxy upside-down." [McBreen, 2003, p. 159]

I don't want to give the impression that the Smalltalk community and XP are the only places where ideas about the close relationship between designing and programming have flourished. As one would expect with an association of practitioners, academics, authors, publications, universities, and conferences, such as the software development profession enjoys, a healthy cross-pollination of ideas takes place. For example, in 1992, C++ guru Jack W. Reeves published an influential article called "What is Software Design?" in the magazine *C++ Journal*. McBreen identifies this article as having been influential on the Extreme Programming community, and one would expect, Kent Beck.[2] Reeves offers some key insights relative to the discussion at hand:

- "C++ has become popular because it makes it easier to design software and program at the same time." [Reeves, 1992]
- "After reviewing the software development life cycle as I understood it, I concluded that the only software documentation that actually seems to satisfy the criteria of an engineering design is the source code listings." [Reeves, 1992] (Actually, Gerald Weinberg made this exact point many years earlier in an essay entitled "A Lesson from the University," published in the book *Understanding the Professional Programmer*.)
- "The overwhelming problem with software development is that everything is part of the design process. Coding is design, testing and debugging are part of design, and what we typically call software design is still part of the design." [Reeves, 1992]
- "Testing is not just concerned with getting the current design correct, it is part of the process of refining the design." [Reeves, 1992]

Another example: in describing his book *What Every Programmer Should Know About Object Oriented Design*, Meiler Page-Jones said:

> Some programmers don't think they're doing design when they program, but whenever you write code, you're always doing design, either explicitly or implicitly. One of the aims of this book is to make programmers explicitly aware of the design patterns that they're creating in their code. [Page-Jones, ?]

***

What about the role of the programmer on a maintenance project, and the resulting importance of programmer-owned design? Most of the time, I think, when developers talk about software development in the abstract and when authors write about development techniques, there is an implicit assumption that everyone is talking about the development of *new* software. However, many (most?) programmers are not working on the from-scratch development of brand new systems (even though that's probably everyone's

preference). Instead, they are working on the maintenance of existing production systems, either in a bug fixing mode or in an extension mode, adding new features or otherwise evolving the software in new directions.[3]

It is even more important, then, that we recognize that the design skills of the programmer are of even greater import when a project is in maintenance mode. This is because, from my observations, many companies reduce the staff of a development team when a system goes into maintenance mode. Analysts and designers move on to the next assignment, while programmers are left to work on the "completed" system. In fact, when a project shifts to maintenance mode, less expensive (and probably less experienced and less skilled) programmers are often brought in to replace the original programmers. When new features need developing, or when an existing part of the system needs redesigning or performance tuning, it is often the maintenance programmers who do the design *and* implementation. (Incidentally, the resulting increase in learning opportunities can be one of the definite advantages of working as a maintenance developer--especially for a less experienced developer.)

In his landmark 1988 book *Rethinking Systems Analysis and Design*, Gerald Weinberg writes:

> Successful designers have learned to avoid the temptation to design everything in one big lump. Instead, they may build one small part at a time, analyze the actual behavior, and then repeat the process for the next part. In this way, design becomes not only evolution-like, but actually evolutionary.
>
> This process of incremental design takes place at two levels--in the mind of the designer, and in reality. When it takes place in reality, it is design as maintenance, which is the principle mode of design today. The vast majority of design decisions actually put into effect today are created by maintenance programmers, not designers.
>
> We[4] do not mean to imply that this situation is a good thing--much of the 'design' being done in maintenance can be equally well viewed as systematic deterioration. [Weinberg, 1988, p.103]

I could go on for ten more pages in directions suggested by this passage, but I will resist and be as brief as possible.[5] First, though, an aside: notice the connection to Extreme Programming, which had not been invented when these words were written. Extreme Programming fully embraces Weinberg's idea of "design as maintenance" and joins it together with Reeves's assertion that "it is cheaper and simpler to just build the design and test it than to do anything else." [Reeves, 1992] The result is a process that explicitly leverages that which Weinberg (rightly) labels as risky and dangerous. What makes this work for Extreme Programming is that it uses techniques such as test first design, automated testing, short iterations, pair programming, and refactoring to simultaneously mitigate the risk and transform that risk into an advantage.

But back to the topic at hand: if Weinberg is right that "the vast majority of design decisions actually put into effect today are created by maintenance programmers, not designers," then design skills for programmers on maintenance projects (which I believe is most of us, for at least some percentage of our time) are that much more important. The ultimate care, humility, and integrity are required when extending a production software system.

The maintenance programmer/designer must constantly balance competing concerns: the need to preserve consistency within the existing system, both in the code and in the external interface; the desire to use the best and most correct solution; the desire to use the latest techniques; the desire to create the best experience for the user; the absolute requirement to not break or otherwise destabilize the production system; the critical charge to protect the production data; the desire or need to improve the design or code that already exists; and the need to craft a solution that can be deployed into the existing system with the least disruption and downtime possible. If these competing concerns are not managed with care, then Weinberg's "systematic deterioration" (as well as "systemic" deterioration) is inevitable. Is it not ironic, then, that the job of the maintenance programmer is most often viewed as being beneath the best and brightest developers?

Given everything I've said up to this point, I might be giving the impression that I am lobbying against the software engineering approach--with its dependency on specialization, documentation, and hand-offs--in favor of an exclusively programmer-centric, craft-based approach. This is not my position at all. I agree wholeheartedly with Alan Cooper, who said in a 2002 interview,

> These two movements in the software world--engineering and craft--appear to be moving in opposite directions. But I think they're both right. The problem is that you can't focus craft methods on engineering problems and vice versa. So the places they break down is misapplying RUP to something that needs three craftsmen working on it, or trying to use individual craft methods to do big projects. Where is it carved in stone that we have to use the same method for all projects? [Thé, 2002]

My point in emphasizing the role of the programmer-as-designer is not to say that specialist designers are a bad idea. Nor am I suggesting the programmers working on a project where someone else does the designing should revolt and ignore the designs they are given. My point is, other than those few projects that explicitly embrace the programmer-as-designer idea (such as XP projects), there is a danger in not recognizing the inherent design role of programmers.

The danger exists on both sides of the hand-off. Whittaker and Atkin wrote, "The nature of designs is that they abstract many details that must eventually be coded." [Whittaker, 2002, p.108] Designers then must be sensitive to the kinds of things they are abstracting in their designs (that is, what blanks they are leaving for programmers to fill), and programmers must realize that a certain amount of design is part of their job and to do their best to a) hone their design skills so that their designs will be the best possible, and b) to design consistently within the framework of the designs they are given.

On a recent project I was tasked with designing a set of PL/SQL stored procedures that would create and move a large amount of critical application data. I had three advantages: the database was designed largely by me, I had spent a lot of time in the company for which we were building this software, and I had intimate knowledge of the structure these procedures needed to have and the logic they needed to follow. I had two disadvantages: I had a lot of procedures to design under an extremely tight deadline, and the two developers who were assigned to write these procedures from my designs were brand new to the project and the domain. They had very little knowledge of the database or even what this complex system was all about. Also, having never worked with them before, I had no knowledge of what kind of coders they were. To top it off, the QA team that would be putting these procedures through their paces did not have the kind of knowledge or documentation they would need to test them.

Given this set of circumstances, I wrote *very* detailed designs. I spelled out every single detail, using elaborate pseudocode--even writing many of the SQL statements the programmers would need--and adding extra expository explanation wherever I could. When I delivered the designs to the developers, it was as close as it gets to those software engineering books Whittaker and Atkin were making fun of. I could not leave anything to chance in terms of the programmers' misinterpreting the intent or logic of any part of the design. The programmers even joked about how it wasn't too much fun writing the code when everything was spelled out for them that way. But we got those procedures done within the deadline, and they were quite solid.[6]

Even in this example, though, the programmers did have to interpret certain things about my designs, and they had to come back to me to get clarifications and fix mistakes I had made. Furthermore, there really was a fair amount of design decision-making for them to do. They had to turn my pseudocode into real PL/SQL code, they had to design an error handling scheme, and they had to design internal data structures.

In another situation on a different project (this one a maintenance project), I needed to design a feature that a programmer on my team was going to build. I had done the analysis with the business people, and I had a vision about how to make the new feature work. This time, however, I had a programmer who had been on the project for awhile, and he had good knowledge of the domain and the existing system. Furthermore, I knew him to be an excellent coder with great design skills and instincts. In this situation, I knew that I did not have to write a highly detailed design. I wrote up a short requirements/design document that described the feature in general terms, explained where it fit in with the existing application, spelled out a few key business rules and constraints, and described a rough user interface concept, with some specifics where necessary. I laid out some rough class and database design ideas, but explicitly spelled out in the design that the developer was free to design it the way he thought best.

The developer and I had a short meeting, and I handed the design document over to him, requesting that he keep it up to date as he finished the design and built the user interface and code. Also, I pointed him to a business person that he could go to if he needed some

requirements clarifications and asked him to keep me in the loop. Here we have one of those “middle realities” we talked about earlier.

As a final example, I worked on another project where I was not the designer. I observed that the designers were not being sensitive to the kinds of concerns described in the previous two examples. The designs they created and handed to the developers were incomplete, as all designs are, but not in the appropriate ways. The developers had not been in the requirements and design meetings, they were not domain experts, and the project did not have good analysis documentation. The designers left critical business rules out of the documents, and left many crucial points open to interpretation with vague language.

As you can imagine, the developers were very frustrated in this situation. They had to scramble around asking various people questions to try to fill in the blanks. They had to go back to the designers to ask for clarifications, and the designers would get frustrated too. When the developers did make an interpretation and design things as they thought best, the designers would come back and say “No, no, that’s not the way it’s supposed to be,” which of course caused even more frustration on both sides.

Now, one could argue that these designers were simply creating poor designs, but I think there was more to it than that. It was not that the designs were bad, but rather that they were incomplete at a level that was inappropriate to the situation. In my opinion, these designers were not sensitive to the fact that the developers would have responsibility for some portion of the design, and more importantly, they were not sensitive to the portions of the design for which the developers were *equipped* to have responsibility. The designs were not tailored to the process, the situation, or the audience, and the designers erroneously viewed their own designs as “complete.”

***

All of this discussion about the relationship between design and construction is intended to underscore three points: first, designers who create specifications that will be handed off to other people must be aware of the inherent incompleteness of any design. Consideration must be given to the audience for the design, as well as situational concerns such as schedule and risk.

Second, it is critical that programmers have an intimate knowledge of the principles and techniques of software design; study is required. Programmers design when they code. The only difference for a programmer from one situation to the next is a matter of degree: at one time you might be working on a project that keeps as much design decision making upstream as possible, and at another time, you might find yourself on a team where most or all of the design decisions occur during construction. In either case, your knowledge of design has a direct effect on the quality of your work and the quality of the resulting product. Mastering your language, tools, and platform is not enough.

Third, it is critical for managers and leaders in charge of development projects to recognize the inevitable role of design in the construction phase and to ensure that the overall process creates opportunities to leverage that fact as an advantage. To ignore the fact that your programmers are also designers, and that the designs handed to them can never be complete, is to invite the most expensive kinds of errors, quality problems, and schedule overruns.

Special thanks to fellow developers Trevor Conn, Robert MacGrogan, and Andy Tegethoff for reviewing this essay and providing feedback essential to its improvement.

Notes

(1) For a good discussion of the craft-based philosophy in the software development universe, I recommend Pete McBreen's book *Software Craftsmanship* (Addison-Wesley, 2002). For a great discussion of craft in general, I recommend you track down a book called *The Nature of Art and Workmanship*, by David Pye (Cambium, 1968, 1998).

(2) I would be remiss if I did not acknowledge McBreen's *Questioning Extreme Programming* [McBreen, 2003] for pointing me to Reeves's article.

(3) In an article for *IEEE Software*, Chilean researchers Jose Pablo Zagal, Raul Santelices Ahues, and Miguel Nussbaum Voehl assert an even broader definition for software maintenance:

> What happens when we shift our perspective and embed the implementation stage inside of the maintenance step? Software development becomes an initial base design followed by maintenance...This focus ensures that the software to be designed is maintainable, because once the base design is finished, maintenance starts, even if there is nothing implemented yet....In other words, maintenance begins where development begins. [Zagal, 2002]

Given this definition of maintenance, *all* programmers are maintenance programmers! The article presents a compelling argument, and includes a case study that illustrates the idea.

(4) At the beginning of *Rethinking Systems Analysis and Design*, Weinberg credits Don Gause as a co-author on some of the book's chapters.

(5) For more on the dangers of "systematic deterioration," I encourage you to check out another essay of mine, "Consistency, Correctness, and Craftsmanship."

(6) Ironically, the project manager gave me grief over the fact that my designs were "too detailed." I tried to explain my position that we would never have made our schedule and quality goals if the designs had not been so detailed, but we never reached a point where we saw eye-to-eye on the issue.

Please note also that the source of the Gerald Weinberg quote beneath the title of this essay is [Weinberg, 1988, p. 101].

References

[McBreen, 2003] McBreen, Pete. *Questioning Extreme Programming*. Boston: Addion-Wesley, 2003.

[Page-Jones, ?] Page-Jones, Meilir. Quoted in Dorset House Quarterly. Quote located on the web at `http://www.dorsethouse.com/books/weps.html`. Original publication date not available.

[Pressman, 1997] Pressman, Roger S. *Software Engineering: A Practitioners Approach*. New York: McGraw-Hill, 1997.

[Reeves, 1992] Reeves, Jack W. "What Is Software Design?" 1992 Located on the web at `http://www.bleading-edge.com/Publications/C++Journal/Cpjour2.htm`.

[Thé, 2002] Thé, Lee. "Why We Don't Build Software for Users," an interview with Alan Cooper for Fawcette Technical Publications. Located on the web at `http://www.fawcette.com/vsm/2002_12/online/the/default_pf.asp`.

[Weinberg, 1988] Weinberg, Gerald M. *Rethinking Systems Analysis and Design*. New York: Dorset House, 1988.

[Whittaker, 2002] Whittaker, James A. and Steve Atkin. "Software Engineering Is Not Enough." *IEEE Software*; July/August 2002.

[Zagal, 2002] Zagal, Jose Pablo, with Raul Santeclices Ahues and Miguel Nussbaum Voehl. "Maintenance-Oriented Design and Development: A Case Study." *IEEE Software*; July/August 2002.

###

Daniel Read is editor and publisher of the **developer.*** web magazine. He lives in Atlanta, GA, where he works as a software architect. He is currently at work on a book about software development crafted for a business audience.

