



Studying Professional Software Design

Alex Baker, Visitrend • André van der Hoek, University of California, Irvine
Harold Ossher, IBM Research • Marian Petre, The Open University

ALL SOFTWARE IS designed. No software is put to use without someone thinking about what it should do and how it should do it. Such thought might be explicit, deliberate, and collaborative, such as when a group of programmers meet around a table. Alternatively, it might be more implicit and personal, such as when a lone programmer ponders a particular piece of the code. It might even happen subconsciously—a sudden thought or insight during an architect's morning shower.

Throughout the life cycle, decisions are made that shape the software to be

produced, and hence the experiences that users will have with it. These decisions must be made with care—making them is a *design* activity.

Given design's crucial role, one would expect our field to have a “culture of design,” as more mature design disciplines do, with a rich portfolio of well-understood design approaches and methods, and an appreciation for both the commonalities and variations in practice. When faced with a design problem, a software engineer should be able to draw from a rich array of techniques that help match familiar

problems to effective solutions and help explore unfamiliar problems systematically in a way that reveals key considerations and alternatives.

Unfortunately, this simply isn't true at this time. Although we have quite a few high-level design methodologies, we don't have a sufficient understanding of what effective professional software designers do when they design, nor much well-founded guidance about how to match method to context. How software developers (whether addressing requirements, architecture, implementation, maintenance, or any other



aspect of development) truly engage in design is a relative mystery. We have reflective pieces written by experienced developers,^{1–3} but relatively little systematic empirical investigation that reveals the mechanisms of designerly thinking⁴ about software, especially early in the design process. What representations do software designers use? What strategies do they employ? How do they work collaboratively and communicate to address a problem effectively? How do they accommodate context in their practice and thinking?

This special issue emerged from “Studying Professional Software Design,” a 2010 workshop at the University of California, Irvine, that studied software designers in action to fuel the discussion and determine what software designers actually do when faced with a design problem. Three pairs of professional designers were asked to design a traffic simulator for civil engineering students (see www.ics.uci.edu/design-workshop). Each pair took the design prompt, worked together at a whiteboard for two hours, and discussed several aspects of both the problem and its potential solutions. The sessions were videotaped, and all workshop participants analyzed one or more of the tapes, resulting in many different perspectives, including cognition, representation, discourse, collaboration, requirements, problem analysis, interaction design, assumptions, rationale, coordination, tools, and design theory. We present some of these perspectives here; several others appear in a special issue of *Design Studies* (www.sciencedirect.com/science/journal/0142694X/31/6).

A Dialogue

Our goal with this special issue of *IEEE Software* is to encourage dialogue between practitioners and researchers. All too often, researchers propose design methods disconnected from practice (and, indeed, teach those in their university classes). Practitioners, on

the other hand, tend to be too busy to look for new approaches—so focused on their own practices that they aren’t concerned with others or simply not tuned in to help steer academia toward innovations that could actually help them in their daily work. Neither practitioners nor researchers are helped by this lack of connection.

The discipline can benefit from systemic investigations by outsiders looking in, a necessity to reveal what we ourselves do not see. A range of empirical studies have emerged (for example, from the Empirical Studies of Programmers [ESP] and the Psychology of Programming Interest Group [PPIG] communities) that have focused on programming and on software development more broadly. We have seen these contributions build new knowledge, challenge previous assumptions, and lead to new kinds of tools that slowly make their way into practice, thereby providing evidence that research grounded in actual practices and building on observed problems and opportunities can have real impact.

But similar empirical studies of software design—particularly of early, formative design—are less common, and we haven’t yet achieved a cohesive body of work nor a coordinated community of researchers or a regular dialogue between researchers and practitioners. Our hope is that the articles in this special issue—and the workshop from which they emerged—can provide the seeds for such an ongoing dialogue.

The workshop itself provided evidence of the power of dialogue between researchers and practitioners. Several designers who were videotaped participated in the workshop alongside the researchers. In a wonderfully cooperative environment, a dialogue emerged in which the researchers reflected on their observations and analyses with the software designers who provided the basis for them. Sometimes, findings were contradicted, sometimes

wholeheartedly affirmed, and sometimes clarifying interpretations resulted in new insight. Similarly, the practitioners were able to reflect on their own practices and strategies. Once they came to terms with being the subject of so much scrutiny by the researchers, they returned their own challenges and questions. They reported that they found the discussions enlightening, delivering insights that were useful to take back into practice.

This kind of dialogue needs to take place much more frequently and more broadly in order for our community’s understanding and practices of design to advance.

Early Evidence

Some early work provides interesting observations that show the potential impact of the study of professional software designers at work, especially through the kind of dialogues we envision. A sample (our introductory annotated bibliography on the *IEEE Software* website contains a larger set of readings; <http://doi.ieeecomputersociety.org/10.1109/MS.2011.155>):

- Raymonde Guindon and colleagues were among the first to study in great detail how software designers work through a design problem, showing a relative absence of the exploration of alternatives.⁵
- Bill Curtis and colleagues showed how the linear notion of design being a phase in the life cycle was a fallacy.⁶
- Mauro Cherubini and colleagues highlighted just how prevalent and transient whiteboard software design is, serving a crucial role in the development process.⁷
- Uri Dekel and Jim Herbsleb demonstrated that designers engaged in collaborative design used formal notations much less than expected, although the notations they did use resembled existing notations.⁸

- Robin Jeffries and colleagues studied differences between experts and novices, observing that experts tend to work initially to understand the problem at hand and have better insight into how to decompose a problem into subproblems, for example, by choosing the appropriate subproblem to work on next.⁹

depth-first solution development, and that expert behavior is informed by longer-term considerations of cost-effectiveness.¹⁴

- Carmen Zannier and colleagues proposed an empirically based model of design decision-making in which the nature of the design problem determines the structure

and important collaborative discussions can occur unpredictably. We're all familiar with design meetings in which a group of developers are brought together to discuss a certain aspect of the architecture or impromptu meetings in which a developer gets stuck working on some code, gathers one or two other developers, and retreats to a conference room to work through the issue. But design takes place less overtly, too. The stuck developer might just have a quick IM conversation, possibly supported by some screen-sharing software, during which the issue is resolved and certain critical design decisions are made. Or design might take place during a team lunch, when some developers spontaneously discuss a feature or issue.

Design isn't "pure": it involves intuition, engineering, drawing upon domain knowledge, explorations of multiple lines of thought, and mistakes.

- Marian Petre documented several strategies that designers use, often subconsciously, as part of their design repertoire and experience (for instance, by using provisionality in designs to leave room for future options).¹⁰
- Alex Baker and André van der Hoek showed how designers work in design cycles, progressing in their design work by juxtaposing different design topics for relatively short periods of time.¹¹
- Sabine Sonnentag found that high-performing professional software designers structured their design process using local planning and attention to feedback, whereas lower performers were more engaged in analyzing requirements and more distracted by irrelevant observations.¹²
- Willemien Visser argued, with empirical support, that the organization of actual design activities, even by experts involved in routine tasks, is opportunistic—in part, because opportunistic design provides cognitive economy.¹³
- Linden Ball and Thomas Ormerod, on the other hand, argued that opportunistic design behavior is actually a mix of breadth- and

of the designer's decision-making processes. The more certain and familiar the design problem, the less a designer considers options.¹⁵

More is needed. Actually studying what software designers do and how they express themselves while they design is necessary if we're to build appropriate support tools, document effective design techniques for the current generation of software designers, and educate the next generation effectively.

Challenges

Significant challenges lie in conducting this kind of work. First, data collection is difficult. Unlike the large body of work that mines preexisting software repositories to study development practices and patterns, no equivalent data source exists for design processes. Many design activities result in transient artifacts, such as paper notes and sketches, whiteboard drawings, and even conversations. These transient artifacts disappear, often quickly, and the researcher is left with personal recollections and, perhaps, more formal design documents produced after the fact. Although these can provide important insights, they tell only part of the story.

Second, design is socially embedded,

Third, design isn't "pure": it involves intuition, engineering, drawing upon domain knowledge, explorations of multiple lines of thought, and mistakes. Furthermore, it's influenced by a large number of human factors that confound what's already a muddled picture of how design truly progresses. Consequently, the study of software design is inherently an interdisciplinary study.

Essentially, "studying software design" must mean studying design over time, over many authentic contexts, and from a variety of perspectives. This entails abandoning any notion of a single definitive study and committing instead to accumulating a body of studies that aggregate in meaningful ways to give a rich overview and cross-cutting insights. It also requires accommodating the balance between detailed study and breadth; attending to the trade-offs between focused studies that give attention to context and studies that might generalize beyond context; taking account of the impact of the problem domain; and so on. This is another reason why the dialogue between researchers and practitioners is crucial: we must access meaningful data that represents effective practice.

Agenda

With a few exceptions, earlier works have been at a relatively high level of abstraction, not diving too deeply into design as a human activity. The studies tend to have a single focus and don't attempt to crosslink different perspectives on early software design. Our agenda, then, calls for studies that examine the following:

- *Commonalities and variations in design approaches across the life cycle.* Early design might or might not share techniques, characteristics, and attitudes with maintenance design; similarly, aspects of the design of a set of requirements might or might not reflect the design of a set of test cases. Moreover, the question of how design decisions made in one part of the process influence the design decisions made in other parts is highly pertinent.
- *The various modes of working.* Design is sometimes solitary and other times highly collaborative. It's sometimes advanced through introspective thought and reflection and at other times through the creation of diagrams, documents, and other artifacts. Design might be wildly creative and free-form, or it might involve the careful analysis of trade-offs. How can each form of design be leveraged? How do they support and interleave? Can we determine what sort of design is appropriate when?
- *Different roles and expertise.* Substantial work on novice-expert differences in programming has resulted in insights about both the nature of expertise and what sorts of expert strategies might be articulated and transferred to others. We need similar novice-expert comparisons for the design reasoning and practice that takes place elsewhere in the process. Can we learn how experts approach a design task and

navigate a design problem—and from that, extract patterns in their behavior that can be described and taught? Are there common mistakes, oversights, or biases that can be recognized, detected, and thus avoided?

Cutting across such studies is the issue of software development context. Early studies in the psychology of the programming community focused on programming-in-the-small, examining how programmers program and make decisions that affect the structure of the software they're developing. Today, this activity still takes place, but the nature of software has changed rapidly toward complex, often distributed and rapidly evolving systems. How does this influence the nature of design, both in-the-small, where program-level decisions must take into account the massive software infrastructure upon which they build, and in-the-large, where decisions about the infrastructure must live up to years of highly varied use?

In This Issue

For this special issue, we selected five articles representing a range of perspectives on professional software design and how designers work. The first, "Toward Unweaving Streams of Thought

have been taken and distilled. Their tool, design practice streams, provides an inventive method of accessing videotaped design meetings by allowing designers to choose a region of the whiteboard or to alternatively select a few keywords from the transcript, upon which the tool retrieves the segments where pertinent design aspects were discussed in order to reconsider design decisions in the context in which they were made.

"Strategies for Early-Stage Collaborative Design," by Ania Dilmaghani and Jim Dibble, recognizes that much of design takes place in a collaborative manner. Based on their collective decades of experience in interaction design, they prescribe 10 strategies for managing effective design meetings. These strategies, ranging from "agree on an agenda and goals for each session" and "work from a shared understanding of user requirements" to "sketch the problem domain" and "mine disagreements," are appropriate in any meeting. However, as the self-evaluation of their own performance in addressing the workshop design prompt shows, without explicitly recognizing and working with the strategies, it's easy (yet problematic!) to forget one or two.

Mary Shaw's article on "The Role

The nature of software has changed rapidly toward complex, often distributed and rapidly evolving systems.

for Reflection in Professional Software Design," by Kumiyo Nakakoji, Yasuhiro Yamamoto, Nobuto Matsubara, and Yoshinari Shirai, seeks to address the challenge that design meetings are fleeting, with no opportunity to return to what was said or decided other than through the memory of individuals present or through the notes that might

of Design Spaces" rekindles the topic of design spaces, presenting an explicit analysis of the alternatives for each of the main design decisions to be made in the prompt that was used in the design workshop. She then highlights how each of the three teams chose quite different points in the design space and compares the three designs to that of a



ALEX BAKER is a senior software engineer at Visitrend in Boston, Massachusetts. His professional focus is on information visualization and user interface design, and his research interests include empirical study of software design processes and software engineering education. Baker has a PhD in computer science from the University of California, Irvine. Contact him at abaker@visitrend.com.



ANDRÉ VAN DER HOEK serves as chair of the Department of Informatics at the University of California, Irvine. He heads the Software Design and Collaboration Laboratory, which focuses on understanding and advancing the roles of design, collaboration, and education in software development. Van der Hoek has a PhD in computer science from the University of Colorado at Boulder. Contact him at andrew@ics.uci.edu.



HAROLD OSSHER is a researcher at the IBM Thomas J. Watson Research Center, currently in the Services Innovation Laboratory working on tool support for solution engineering in the smarter commerce domain. His research interests include modularity and separation of concerns, software development tools and environments, and flexible modeling. Ossher has a PhD in computer science from Stanford University. Contact him at ossher@us.ibm.com.



MARIAN PETRE is a professor of computing at the Open University and a Royal Society Wolfson Research Merit Award holder, in recognition of her research on expertise in software design. Petre has a PhD in computer science from University College London. Contact her at m.petre@open.ac.uk.

Dibble. However, Rooksby and Ikeya carefully analyze the transcripts and videos and add some lessons of their own, the most important, perhaps, being that a sense of humor plays a key role. Design, after all, remains a human and often social activity, and thus, the cooperation and openness among those in a design meeting will shape the give and take of the design dialogue that ultimately determines a design's effectiveness.

The concept of “design” has developed for decades, spanning a variety of approaches, product types, and fields. Software is often compared to a broad range of other fields (architecture, engineering, movie creation), and we have lessons to learn from the study of design in other disciplines: similarities emerge in how people navigate design problems that could well inspire how we should proceed in software design. We suggest a more enthusiastic embrace of a design-oriented perspective in software research, starting with the rejection of notions that design can only exist in a phase, in code, or in a system's interface. What if we considered requirements engineering from a design-oriented perspective? Or the creation of a suite of test cases, or the development of incremental changes during maintenance? Design is a powerful way of considering creative endeavors with a long history of research and practice. We believe that applying this perspective to software development stands to improve significantly how it is researched and practiced.


We also argue that effective study of software design requires dialogue between practitioners and researchers. To produce insights that are relevant to practice, researchers need to relate to practice and be informed by it. We need to understand the contexts in which software design is conducted, in order to address both technical and social

commercial simulator. The implication is that the exploration of design spaces, and indeed their articulation up front when faced with a design problem, is extremely important in order to make proper design decisions.

The fourth article, “Design Strategy and Software Design Effectiveness” by Antony Tang and Hans van Vliet, examines design from a two-dimensional space of breadth- or depth-first design versus problem- or solution-oriented design. They articulate four distinct design strategies (scoping/questioning,

scoping/solving, no scoping/questions, no scoping/solving) that they observed the designer pairs engage in at different times during the recorded design meetings. They suggest that software designers match their design strategy to the requirements of the situation.

Finally, John Rooksby and Nozomi Ikeya present a detailed analysis of one designer pair in “Collaboration in Formative Design: Working Together at a Whiteboard.” The article echoes, from a researcher's perspective, the reflective observations made by Dilmaghani and

aspects. Existing reports reflect a reluctance by practitioners to adopt tools that are at odds with their professional ethos and practice. Grounding research in dialogue between practitioners and researchers should inspire tools that suit existing cultures in industry—or that are profound enough to warrant the transitional cost of adopting new ways of thinking. 

References

1. B. Boehm, *Software Engineering: Barry Boehm's Lifetime Contributions to Software Development, Management, and Research*, Wiley/IEEE CS, 2007.
2. F.P. Brooks Jr., *The Design of Design: Essays from a Computer Scientist*, Addison-Wesley/Pearson Education, 2010.
3. T. Winograd, *Bringing Design to Software*, ACM Press, 1996.
4. N. Cross, *Designerly Ways of Knowing*, Springer, 2007.
5. R. Guindon, H. Krasner, and B. Curtis, *Breakdowns and Processes during the Early Activities of Software Design by Professionals*, Empirical Studies of Programmers: Second Workshop, Ablex Publishing, 1987.
6. B. Curtis et al., "A Field Study of the Software Design Process for Large Systems," *Comm. ACM*, vol. 31, no. 11, 1988, pp. 1268–1287.
7. M. Cherubini et al., "Let's Go to the Whiteboard: How and Why Software Developers Use Drawings," *Proc. CHI 2007*, ACM Press, 2007, pp. 557–566.
8. U. Dekel and J.D. Herbsleb, "Notation and Representation in Collaborative Object-Oriented Design: An Observational Study," *SIG-PLAN Notices*, vol. 42, no. 10, 2007, pp. 261–280.
9. R. Jeffries et al., "The Processes Involved in Designing Software," *Cognitive Skills and Their Acquisition*, J.R. Anderson, ed., Erlbaum, 1981, pp. 225–283.
10. M. Petre, "Insights from Expert Software Design Practice," *Proc. European Software Eng. Conf./Foundations of Software Eng. (ESEC/FSE)*, ACM, 2009, pp. 233–242.
11. A. Baker and A. van der Hoek, "Ideas, Subjects, and Cycles as Lenses for Understanding the Software Design Process," *Design Studies*, vol. 31, no. 6, 2010, pp. 590–613.
12. S. Sonnetag, "Expertise in Professional Software Design," *J. Applied Psychology*, vol. 83, no. 5, 1998, pp. 703–715.
13. W. Visser, "Designers' Activities Examined at Three Levels: Organization, Strategies and Problem-Solving Processes," *Knowledge-Based Systems*, vol. 5, no. 1, 1992, pp. 92–104.
14. L.J. Ball and T.C. Ormerod, "Structured and Opportunistic Processing in Design: A Critical Discussion," *Int'l J. Human-Computer Studies*, vol. 43, 1995, pp. 131–151.
15. C. Zannier, M. Chiasson, and F. Maurer, "A Model of Design Decision Making Based on Empirical Results of Interviews with Software Designers," *Information and Software Technology*, vol. 49, no. 6, 2007, pp. 637–653.



Selected CS articles and columns are also available for free at <http://ComputingNow.computer.org>.

IEEE SOFTWARE CALL FOR PAPERS Special Issue on Technical Debt

SUBMISSION DEADLINE: 1 APRIL 2012 • PUBLICATION: NOVEMBER/DECEMBER 2012

The ability to deliver increasingly complex software-reliant systems demands better ways to manage the long-term effects of short-term expedient decisions. The idea of technical debt is that developers sometimes accept compromises in a system in one dimension (for example, modularity and code quality) to meet an urgent demand in another dimension (such as a deadline). Such compromises incur a "debt." Time spent dealing with the compromised code is considered "interest" that has to be paid, and the cost of building in the originally planned quality is the "principal" that should be repaid at some point for the long-term health of the project.

IEEE Software seeks submissions for a special issue on technical debt in software development. Possible topics include

- Definitions, models, or theories behind the concept of technical debt
- Case studies and lessons learned on technical debt in large-scale software development
- Practical guidelines, strategies, and frameworks for evaluating and paying back technical debt
- How to integrate technical debt management with soft-

ware development practices (for example, Scrum, architecture analysis, design/code review and documentation, test-driven development, evolution, and maintenance)

- Approaches, applications, and tools for visualizing, analyzing, and managing technical debt
- Types, taxonomy, symptoms, and root causes of technical debt

QUESTIONS?

For more information about the special issue, contact the guest editors:

- Philippe Kruchten, University of British Columbia, Canada; pbk@ece.ubc.ca
- Robert L. Nord, Carnegie Mellon University, Software Engineering Institute; rn@sei.cmu.edu
- Ipek Ozkaya, Carnegie Mellon University, Software Engineering Institute; ozkaya@sei.cmu.edu

For full call for papers: www.computer.org/software/cfp6

For full author guidelines: www.computer.org/software/author.htm

For submission details: software@computer.org