

**Position Paper:
Science of Design for Software Intensive Systems**

Gregor Kiczales
Department of Computer Science
University of British Columbia
gregor@cs.ubc.ca

Personal Background

The focus of my research is to enable programmers to write code that looks like the design it implements.

This goal implies a willingness to expand our current notions of what “code” means, and most of my work has been in programming languages (object-oriented, reflection, aspect-oriented programming). I have also done a small amount of work in software design principles and guidelines (open implementation).

I certainly do not believe that the entire design can be captured in the executable artifact. But I do believe that capturing as much design as possible in the code is a good goal. Most code is read more than it is written. Most design documents are never created, out of date, or lost. So the more the code says about what and why it is, the more valuable it will be over the long haul.

Perspective on Science of/for Design

I take “the design” of a software system to have a broad meaning. I take it to mean at least the code; formal and informal documents describing the code; the decisions and analyses that have gone into the code and documents; the technical and social contracts embodied in the above; and the work practice that produces the above.

I see the practice of software design as socially embedded technical work. It is technical in that we build a formal system and there are numerous objective technical issues. How does an algorithm scale? Is a mechanism type-safe? Can linear logic capture a certain domain model? Does a controller meet its real-time deadlines?

But design involves many issues that are a synthesis of the technical and the social (or at least aesthetic). Is this design elegant? How much flexibility can we afford? Will changing this interface a certain way help Joe with the problem he has?

I see a distinction between science of design and science for design. By the latter, I mean scientifically grounded techniques that can be used to good advantage in design work. By the former I mean something more encompassing, that accounts, in a scientific way, for the broad scope of design outlined above.

Science for Design

One approach to improving software design is to think about what is wrong—how current design practice fails, how science fails current design practice etc. I have instead tried to think about what is right. What recent developments have arguably improved the practice of software design? What can we learn from them? Can we develop a scientific basis that would support such results?

Design patterns are popular and well accepted among practitioners. A key claim is that design patterns improve communication between designers by providing a shared higher-level vocabulary. There is both anecdotal and scientific support for these claims.

Extreme programming (XP) has recently become popular. XP practices like pair programming, test-driven design, refactor mercilessly, and move people around appear to lead to better software and happier developers. (It is too soon to be sure how XP will hold up over time though.)

Refactoring tools are becoming popular. There is anecdotal evidence these help improve software quality. Because Eclipse makes it easy to rename packages, classes and methods, developers now fix “old confusing names” in a system. They make the actual code align more with their model of the design, and that makes the code easier to understand and work with.

Why do these techniques improve the practice of software design and development? Because they fully embrace the social nature of technical work. Design patterns are a semi-formal means of improving shared understanding of a design. Proponents of design patterns claim that their semi-formal nature is critical to their success. A “traditional scientific” result might not have produced this kind of semi-formal result. Refactoring tools help improve names, which help people work with code. But “traditional formal semantics” accounts of code would never suggest the dramatic value of such tools.

But design patterns, XP and refactoring tools are not just “soft”. They offer hard technical artifacts (code fragments, tools etc.) together with a soft theory, or paradigm about how those integrate with work practice. Critically, that theory is compatible with actual practice and actual practitioners.

As we work to understand what a science for software design must be, we should be careful to take into account the social work practice in which the science will be used. The science has to fit that practice in order to be truly useful.

(It is also worth noting that patterns, XP and refactoring are relatively simple ideas. This general point about what makes an idea suitable for mass adoption is probably worth keeping in mind.)

Science of Design

Is it possible to develop a field of study that has the critical properties of science, and that accounts for the technical and work practice aspects of design? I think so. Several examples suggest what this might be like.

Architecture (of buildings and places) produces designs that have much in common with what we produce. The designs have hard-core technical properties. But it also explicitly attends to issues like the technical and social contracts the design honors. Work to produce the design of new Java APIs feels a lot like work to produce the design of the new buildings and public spaces at ground zero.

Some work in cognitive science is a clear parallel for us. In a science of design, we must be concerned with representing design intent, with the relationship between such representations and executable code, and with the different kinds of perspectives different observers might have on the design intent, the representations and the executable code. These kinds of issues, and the way they span the technical/social boundary has direct parallels in cognitive science.

I am not advocating that the workshop go headlong into what some might call social science. We are unlikely to be well qualified to take the discussion there. But it is important that we embrace the context and the nature of software design practice. Science for software design must work in that context. Science of software design must account for that context.