

Software-Intensive Design: A Position Paper

Matthias Felleisen
Northeastern University

October 1, 2003

The nature of design

To design software means to create a piece of software for some context, subject to some constraints. The border between the context and the software is an interface; it specifies the extensional properties of the product. The constraints describe the intensional aspects of the software; for example, interactive software must respond within reasonable limits to a request, and embedded software should require commensurate amounts of electrical power.

The context of a software product and its constraints inevitably change. The very nature of software invites consumers to extend, reduce, or modify their demands in ways that no architect, aeronautical, chemical, electrical, or mechanical engineer could imagine. People also expect to deploy one and the same software product in several different contexts with minor adaptations. They correctly recognize that—at some level—the task remains the same and that more or less the same product should serve adequately.

Design at all levels

Design takes place, or ought to take place, at all levels of software production. Programmers should design their programs, not just write code. Teams design module and package interfaces. Software architects should design the components and connectors of the chosen software organization.

Research on design

Given the above, research on the nature of software design must face the following questions:

1. How do we specify and study the interfaces between contexts and software?
2. How do we create examples from the specifications of the interfaces that illustrate how the product should behave? How do we formulate these examples as tests?
3. How do we use the interface specifications to pick the “natural” organization for the pieces of a software product? (Ex: Why/when is a for-loop the natural function body? Why/when is a pipes-and-filters organization the natural software architecture?)
4. How do we use the tests? What do they tell people about the reliability of the product?

In short, the goal is to organize the design process around the idea of matching products to interfaces. After all, if the organization of a product (its design blueprint) matches the interface and if changes to the context are expressed as changes to the interfaces, the designer has some hope of reusing existing products to come up with a modified version for the new interfaces.

Constraints enter the picture when a designer must choose from several competing, systematically created products or blueprints:

1. How do we specify constraints? How do we translate them into stress tests?
2. How we use them to compare and evaluate competing designs?

Naturally, the product that satisfies constraints well (optimal is probably too much to ask for) should be the one that people choose.

Programming languages and design

Every software designer programs. Coders use programming languages in the traditional sense. Software architects use (often graphical) notations that specify the components of a large product and its connections. No matter at what level people program, the purpose of the program is to communicate ideas to someone else (including an older version of the designer).

Over the years, programming language researchers have learned that claims about code that aren't validated will eventually be false. It is therefore necessary that every programming notation includes a language of claims and means for validating them.

If we wish to develop a discipline of software design, we must learn to scale the programming language research to the level of architectural design and organization.

Teaching design

In the end, a discipline of software design can only succeed if it answers these questions and educates students about all levels of software design. First, students must learn that plain programming ought to be more than writing down something that works (for this moment, on some haphazardly chosen examples); they must learn that the organization of a function, method, class, package, module must match its interface and why this matters. Second, when students understand how proper design helps at the level of conventional coding, teachers of design can explain why and how systematic design will work at the scale of huge systems.

Teaching design means that teachers are able to explain why every step in the design process inevitably follows from what was done so far. A student must see that the steps are rational and that the end-product therefore has desirable qualities. The kind of design ideas that dominated the programming curriculum for the past 30 years are unfortunately broken from this perspective; for example, stepwise refinement was inadequate because teachers couldn't explain how to refine at each level or when to stop the refinement with uniform criteria. To get to such a point, we need a much more systematic exploration of software design than what we have done over the last three decades.

Background

Over the past eight years, I have explored the idea of systematic program design. My overall goal is to craft a curriculum for the first three courses on programming that teach programming at all levels as a design-oriented activity. At this point, the work has produced a text book for the introductory programming course (*How to Design Programs*, MIT Press, 2001). The second volume, *How to Design Class Hierarchies*, MIT Press) is in progress. The last volume (*How to Design Systems*) is in the planning stage.