Connecting Design to Code

Thomas D. LaToza
George Mason University
11 Oct 2022

Abstract

How can developers use what they know about design to more effectively understand code? This article explores the ways in which the brain helps developers understand code and how this process can be facilitated through tools and processes which help understand code in terms of its design.

Introduction

Code, design, and architecture are often conceived of as being entirely disjoint models of a system. Indeed, much of the value of thinking in terms of design and of architecture is to be able to think at a level of abstraction above the code, making decisions about how the system will satisfy its requirements and achieve quality attributes without worrying about all the details that code brings.

But for code-level tasks, day to day, it's crucial to *connect* design to code, to easily and quickly move from thinking about a line of code to how it's related to the design or architecture. Reasoning about how to correctly change code requires understanding *rules* about code imposed by the design. This is made easier by well-written code which exhibits conceptual integrity, allowing a small number of design rules to accurately explain it. Poorly written code often exhibits code decay, where code breaks its design rules.

So how does design made consistent with code help in reasoning about code? And how can this be facilitated by processes and tools?

Seeing "facts" about code

```
class HelloMessage extends React.Component {
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}

root.render(<HelloMessage name="Taylor" />);
```

Figure 1. An example of React code.

As a developer reads the code in Figure 1, they might read each word, token by token, noting a function declared in a class and a method call after the class declaration. However, if they're familiar with the React framework, they're likely to "see" (or, perceive) this code differently: it's building a simple component, with a string being passed into a render function. Their understanding immediately helps them perceive facts that would otherwise be unclear [1]: the name attribute value being passed into the HelloMessage is stored as a property on the element, which is then read off during the render process. And calling root.render() eventually results in the render method in HelloMessage being invoked.

Beyond noting facts about code, they might also begin to consider design decisions: given the little functionality in the render function, why not use a more concise syntax for declaring a new React component? Is this because the original developer is expecting to change this code, to add more functionality? Or, is this really the only functionality they expect to have, and did they just not know about other syntaxes for declaring a Component? Decisions tie together facts, explaining why one fact was chosen (or not chosen) in order to achieve another and imposing design rules that must be reflected in code.

Comprehending code is a process of reverse engineering, going from a series of tokens, defined by a programming language, to a mental model with facts and design rules. Facts may be in several forms: data flow or control flow information (e.g., the name attribute declared on HelloMessage is stored into a property which is later read in the render method), the domain model, and the design and architecture.

Lacking a good mental model of code, code becomes really hard and laborious to work with, where even a simple action like calling a method requires extended investigation into documentation and tutorials. But with this mental model, programming can become much easier, where consulting StackOverflow or documentation is necessary, but it's clear exactly what to look for to fill in code-level details they've forgotten.

Programming knowledge enables the developer to infer facts from code, facilitating translating back and forth between source code and a mental model of design, architecture, and rules. Leaving behind the world of source code, the developer can think at a higher level of abstraction. But at some point it's necessary to return to the source code, to seek and learn more facts. To do so, relevant parts of the source code must be located, and then comprehended in terms of the design.

IDEs can provide important support for this process. Autocomplete and tooltips offer quick reference to comments describing the use of methods and the meaning and use of parameters. More sophisticated tools can do even more, explaining, for example, how concepts in the problem domain translate into code, and vice versa.

But today's tools only go so far in translating the world of code into the world of design. The more a developer knows about the design of their specific project, of the design of APIs their code interacts with, or of design patterns and architectural styles in general, the faster they can jump across these abstraction gaps, comprehending code by using the facts they already know to generate more facts.

Reverse engineering design decisions

So why is it so important to bridge the gap between code and design and understand the facts behind code? One reason is that, whenever making a change, it's important to understand the implications of the change. Is it consistent with the current design? Does it break the design?

The intent of design and architecture rules are to constrain the implementation. When a developer makes an architectural or design choice that data should be persisted in a specific way or that specific functionality should be defined in a tier with specific interactions with other components, they want future code to follow these design rules.

Developers that don't understand the design and architecture and rules they impose will inevitably break them, causing code decay. Over time, the worlds of code and design and architecture diverge further, making it ever harder to make changes in a way that's consistent and forcing every decision to be considered in detail at the code level rather than in terms of higher level abstractions that can be easier to reason about. Instead of applying a single idea from the design or architecture consistently across the codebase, developers must relearn every detail of the implementation and its behavior every time a new source code file is read. Rather than thinking about the behavior of code in the abstract, developers end up thinking constantly in terms of code.

Documentation makes reasoning about facts easier

So how can developers more easily create a mental model of code? Anything that facilitates bridging the gap between code and design will help. Knowing more about frameworks and libraries, and their typical idioms, design patterns, and architectural styles helps. If a developer can look at some React code and already know, here's how a state change will propagate, or here's how the lifecycle of a component works, they're already a step ahead.

But beyond this, it's important to know the design and architecture of a project. What are the important abstractions, and the rules about how they should be used? What are the key architectural design decisions, and what lower level implications do these have? What are the key elements and the rules about how these elements should interact? What are the conventions about the way that events or state are managed, in an object-oriented or functional style? For a specific feature, what were some of the key goals driving the way that it was implemented?

Ideally, projects should have documentation that explains all this. Onboarding a developer to a project might mean simply getting up to speed on the key design documents.

But documentation carries risks. Perhaps the document reflected the intended reality at the time it was written. But did the implementation ever fully realize the design, as initially envisioned? In what ways have new features been implemented which change the initial design? Have all changes been made in a way that's consistent with the design?

Over time, documentation has a tendency to become dated and obsolete. It's easy to give up on documentation, to say why even bother, when the code is truth.

But, "the code is truth, but not the whole truth" [2]. Over time, this can lead to more tedious changes, where a codebase loses its conceptual integrity, where any change has myriad unforeseen consequences, and doing things the "right way" increasingly looks like giving up and starting over.

The fundamental problem is that design, as embodied in the mental model of the code, and the code itself are disconnected. Lacking a way to check that one is consistent with the other, they may diverge, particularly as others have slightly different, or widely diverging, mental models of what the design ought to be.

Active Documentation keeps code and design in sync

What if there were a way to ensure that the developer's mental model of the code was in sync with the code itself? What if it wasn't necessary to choose between reasoning only at the code level or at the design level, but one could easily map from one to the other?

One way to think about a design document is as a set of rules, each embodying some statements that should be true of code, and explanations justifying and explaining these rules in terms of the other rules, requirements, and quality attributes they help achieve.

Rather than conceiving of design as inherently nebulous and ill-defined prose in a document, it's possible to instead conceive of it as a precise set of design rules that should be true about code [3].

Ensuring that design and code are consistent then just means taking the rules written in the design and checking them against the code.

A tool could then make a design document *active* [4]:

- When code, or the design, changes, check code against the design to identify violations of design rules

- Link the design, as specified through design rules, with examples in the code which follow these rules

A design can then be a tool, helping a developer see code clearly. Rather than separate code and document artifacts, an IDE might place them side by side, showing code and rules expressing the design (Figure 3).

As a developer makes a change, the design provides guardrails, actively ensuring awareness of the design rules which apply to the code being written at the moment. The tool flags divergences between the code and the design, highlighting in red the violated design rules. It might be that the developer just forgot, or perhaps never knew about that part of the design in the first place.

When seeking to understand code, the developer can go straight from the code to linked design rules, explaining code in terms of the design. In contrast to comments, which can only exist at one point in the source code, design rules can be attached to every point in the source code they impact. When the code they impact is scattered across dozens of methods and files, design rules reify all of these points, helping developers quickly navigate from the singular declaration of the design rule to all of its implementations in code.

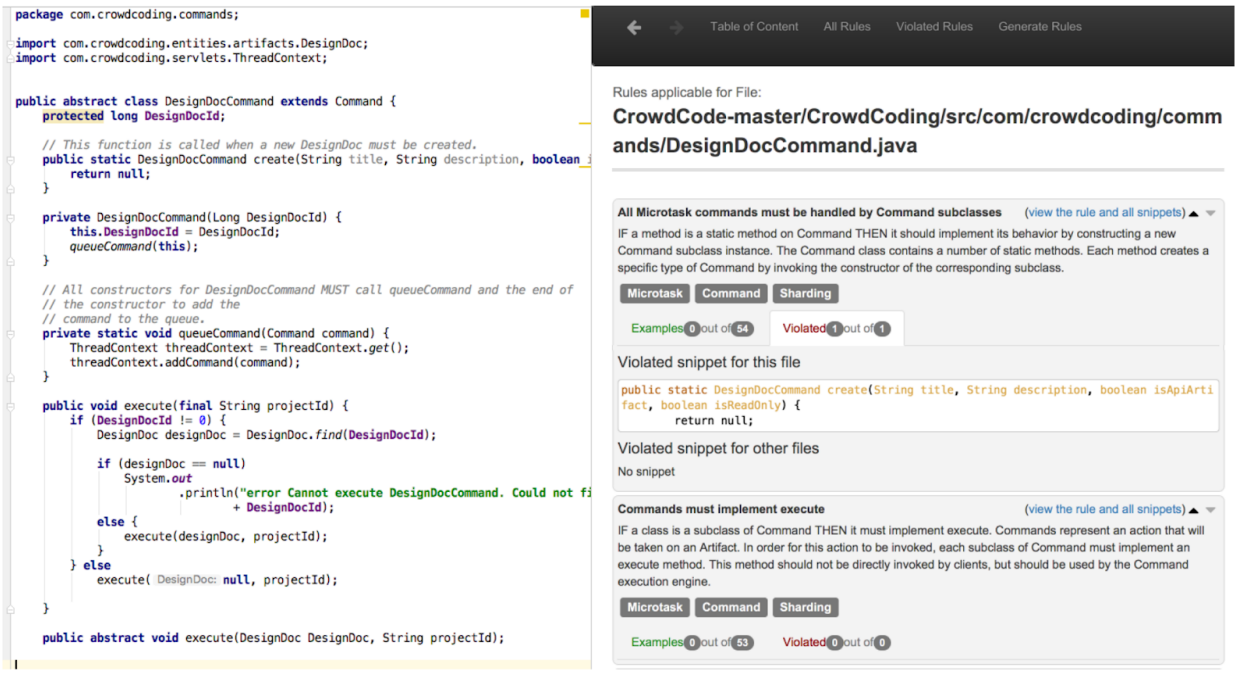

Figure 3. Active documentation connects code to the design, showing parts of the design relevant to the code, linking to code snippets illustrating the implementation of design decisions in code, and flagging divergences between the code and design as each are changed [4].

When finding a violated rule, a developer might decide that the design rule no longer makes sense, deleting it or replacing it with a new rule. They might then use the view of the design to find all of the instances of that rule in the code, helping plan a refactoring and offering a checklist as they change the code to follow the revised design.

Tools might go even further. When writing new code, a programming assistant, such as GitHub's CoPilot[1], might take the design into account, generating only code that is consistent with the design. Or it might explain suggestions, detailing how the way in which the code was written was necessary to follow the design.

By separating design from code, each may be simpler and therefore easier to understand in isolation. But that is not a practical option for developers because they must keep the design in mind with every edit they make to the code. In particular, thinking about the design is crucial to understanding code and ensuring that the code you write follows the design. Tools and processes that help connect the code to the design play an important role in avoiding design erosion and improving productivity.

About the Author

THOMAS D. LATOZA is an associate professor of computer science at George Mason University. His research focuses on investigating how developers interact with code and designing new ways to build software. Further information about him can be found at https://cs.gmu.edu/~tlatoza/. Contact him at tlatoza@gmu.edu.

References

[1] Thomas D. LaToza, David Garlan, James D. Herbsleb, and Brad A. Myers. 2007. Program comprehension as fact finding. European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC-FSE), 361–370. https://doi.org/10.1145/1287624.1287675

[2] G. Booch, "Architectural Organizational Patterns," in IEEE Software, vol. 25, no. 3, pp. 18-19, May-June 2008, doi: 10.1109/MS.2008.56.

[3] Sahar Mehrpour and Thomas D. LaToza. (2021). Programming tools for working with design decisions in code. Workshop on the Evaluation and Usability of Programming Languages and Tools (PLATEAU), 9 pages.

[1] https://github.com/features/copilot

[4] Sahar Mehrpour, Thomas D. LaToza, Rahul Kindi (2019). Active documentation: Helping developers follow design decisions. Symposium on Visual Languages and Human-Centric Computing (VL/HCC), 87-96.