Editor: **George Fairbanks**
gf@georgefairbanks.com

Two Kinds of Iteration

George Fairbanks

IN THE ALLEGORY of the cave, Plato argued that invisible concepts, such as geometry, could be more true than any figures that we imperfectly scratch in the sand. The triangles and squares that we can observe with our eyes are just shadows cast on the wall of a cave by the pure ideas that we cannot observe directly. This presents us with a choice: should we fixate on the shadows we can see, or use them to discover hidden truths?

Today, two and a half millennia after Plato wrote his allegory, software developers make that same choice. Some developers see source code as the truth. Others see source code as the shadow on the wall that provides clues about the truth, which is the problem and solution that cannot be observed directly. I doubt Plato would be surprised that we are still debating.

Digital Object Identifier 10.1109/MS.2021.3121737
Date of current version: 23 December 2021

Many developers consider themselves as pragmatic and therefore decide that seeking invisible truth is something left for philosophers and academics. I disagree. The Wright brothers were deeply pragmatic, yet in their quest to be the first to fly, they both built airplanes and developed theories about aviation. They built an airplane before others precisely because they pursued both.

Importantly, the Wright brothers used an iterative approach. Iterations forced them to build something instead of spending all their time on philosophy. Iteration is what makes it possible, and indeed pragmatic, for engineers to both get things working and seek the invisible truths that explain how to make things work better.

Many developers consider themselves as pragmatic and therefore decide that seeking invisible truth is something left for philosophers and academics.

Here's the rub: iteration means different things to different people. "Code is the truth" developers iterate on the code, adding features over time. I call this *code-focused iteration* (*CFI*). "Shadow on the wall" developers iterate on their understanding and on the code, making both better over time. I call this *design-focused iteration* (*DFI*).

Because the word "iteration" is ambiguous, developers can declare "we are iterating" and yet be doing quite different things. Small changes in day-to-day activities lead to different outcomes after just a few months. Developers doing CFI erode their designs, impair their readiness for the next requirement, and reduce their productivity. In contrast, developers doing DFI strengthen their design with each iteration, solve their problems better, and enjoy their work.

Kinds of Iteration

What do CFI and DFI look like in practice? Let's start with some familiar nonsoftware examples of iteration. When you get a new pair of eyeglasses, your optometrist uses iteration to adjust them to fit your head. The two of you alternate between wearing and adjusting the glasses until both of you are satisfied with how well they fit. The optometrist is using a hill-climbing algorithm: examining the situation and making a change for the better. In this kind of iteration, no one is seeking invisible truths. You and the optometrist attend solely to what is visible, using a technique to improve a machine (your eyeglasses) for the better. This is CFI but with eyeglasses instead of code.

0740-7459/22©2022IEEE

Car engines are another example. When a new generation of engine comes out, it typically has unforeseen problems. The automotive engineers identify and fix these problems iteratively and, over several years, as design flaws are fixed, that generation of engine becomes more reliable. This is CFI but with engines instead of code.

Those same automotive engineers, however, are also doing something else. Across generations of engines, they are building up their understanding of everything involved with building engines: the materials, the combustion, the wear on parts, the machines that create the engines, the environment the engines will be placed into, and so on. They are using their experience with the tangible to learn about the invisible. By building up their understanding of the invisible truths, their next generation of engines will be better than the previous. This is DFI applied to engines.

Let's return to software development. Imagine a system used to schedule university classes that already handles semesters, and let's say the developers receive a new requirement: trimesters. The developers make minor changes to the code to support the requirement. (You can imagine many similar changes that would not force any significant reflection on the nature of university classes or on the design of the software.) Developers can make those changes by attending solely to the code itself, applying a hill-climbing algorithm. This is CFI.

Consider a different requirement for this university software: that teachers can attend classes. Let's say the code has one data structure for teachers and another for students. If Prof. John Doe wants to take a class, the system would be tracking him twice, with his information duplicated in the two data structures. So, this requirement forces developers to reflect on what they understand about university classes. They iterate on their invisible understanding of how things work and revise their ideas. Perhaps they land on the idea of introducing two new concepts: people and roles. Where they previously thought of teachers and students, they now think of people who play the roles of teachers and students. They revise the code to match this new understanding. This is DFI.

> Small changes in day-to-day activities lead to different outcomes after just a few months.

Code Refines a Design

It's tempting to ignore distinctions between CFI and DFI, instead thinking only of developers making a series of edits to the code to improve it. After all, developers may interleave thinking and coding, and in fact this can accelerate their DFIs. But failure to distinguish CFI from DFI can doom a project. When Ward Cunningham coined the term *technical debt*, he described how iteration, done poorly, could bring "[e]ntire engineering organizations … to a stand-still."[1]

So, what is CFI missing? In a word, refinement. DFI improves both the design and the code so that, over time, the design becomes an increasingly good fit to the problem at hand. By contrast, CFI, by accumulating features, improves only the code.

Refinement is the relationship between design and code. Your design guides your code and limits some of your implementation choices. Anything present in the design must also be in the code but not vice versa. Consider the university class scheduling system in the earlier example. You have a lot of implementation choices. You could implement it in any programming language, using any variety of algorithms, and on any hardware platform. However, there are limits. The ideas from the design—people and roles, semesters and trimesters—may not be contradicted in the code.

As a developer, why should you voluntarily constrain yourself? How can shackles help you solve problems? A good design makes it easier to write good code. In the earlier example, the design change from teacher–students to people–roles isn't a shackle, it's a gift. Clear thoughts in the design can avoid any number of corner cases in the code. A good design allows you to make broad conclusions without reading through every line of code. For example, a map-reduce design insists that each map job be idempotent, so you can conclude that it's safe to reschedule jobs that are running slowly. The idea of idempotence is one of those invisible truths in a design that you cannot see directly in code.

Design can feel more true than source code, just as geometry can feel more true than any imperfect diagrams we might draw. Consider the vending machine problem that's often used in introductory programming courses along with a finite state machine design. Is that design not more true and real than any

Code-Focused Iteration	Design-Focused Iteration
1) Get the new requirement/feature. 2) Write the test case. 3) Edit the code minimally so the test passes. 4) Later on, the refactor removes the code duplication.	1) Get new requirement/feature. 2) Revise the design, if necessary. (Is the architecture OK? Is the domain model OK?) 3) Write the test case. 4) Revise the code to match the design.

FIGURE 1. The two kinds of iteration.

student's code that implements it? And if you had a new requirement, perhaps to handle a new coin, wouldn't you revise the state machine and then edit your code to match it?

Iteration With a Goal

Years ago, when waterfall processes were common, refinement was a fact of life. Developers were forced to confront the refinement relationship between design and code because design happened early in the project and code not until later. All developers were aware of how their design related to their code.

When I mention waterfall processes, some people misinterpret this as me advocating for upfront design. The goal is to have a refinement relationship between design and code, but that goal can be accomplished through upfront design or iterative design. As Desmond D'Souza and Alan Cameron Wills said: "Refinement is a relationship, not a sequence."[2] Plenty of articles have demonized upfront design, but the bigger problem is neglecting the goal of refinement.

Consider the two iterative processes shown in Figure 1. One will help you improve the code, while the other improves both the code and the design.[3] Teams using DFI are bringing the design with them on their journey. It is a constant companion. CFI and DFI are both iterative, but only DFI has the goal of nurturing the refinement between design and code.

CFI is vulnerable to problems that grow worse over time.[4] The first problem is the sedimentary buildup of old ideas. In the university example, you probably could have edited the code so that your teacher and student data structures survived. Obsolete ideas can accumulate in code like sediment, making it hard for other developers to understand the design and reason about it.

The second problem is loss of intellectual control. If you iterate only on the code, whatever design you have will deteriorate and provide less value. You lose your ability to reason through the system using abstraction

When you make a change to the code, is there often a corresponding change to the design?
In DFI, you coevolve the code and the design.

Over time, do your design abstractions fit the problem increasingly well?
In DFI, you evolve your design to avoid special cases and bent rules.

As time goes on, is it easier or harder to build the next feature?
In DFI, your abstractions are a foundation that speeds development, not a liability to be worked around.

Do you have design abstractions that are not directly expressible in code?
In DFI, developers think about and talk about design abstractions that their programming language cannot express (for example, idempotence).

If you had been given the requirements all at once instead of sequentially, would you have designed something like this?
In DFI, you course-correct your design in each iteration. Your design and code should look like you knew what you were doing all along, even though your understanding grew gradually.

Is the team gaining insight into the matters at hand?
In DFI, the team builds up a theory of the problem and solution.

In each iteration, do you build a revised running system?
In DFI, both design and code are updated in an iteration. If you iterate on the design alone, that's a phase in a waterfall process.

FIGURE 2. What kind of iteration are you using? How to recognize DFI.

and instead must trace the code line by line. When obsolete ideas accumulate and intellectual control is lost, projects become technical zombies without vitality.

Refactoring the Design, Not Just the Code

For decades, refactoring has been held up as the way to repair iteration's flaws. That's only partly right. Most projects use refactoring merely to textually rearrange code. How do developers make that mistake? If you look at books and websites on refactoring techniques, you'll see them describe mechanical activities, but those are shadows on the wall. Such refactoring is helpful, but it's akin to fixing the grammar and spelling in an essay with half-baked or obsolete ideas.

The truly valuable part of refactoring is invisible. The best description of how to use refactoring to evolve your design is in the Domain-Driven Design (DDD) book section on "Refactoring Toward Deeper Insight."[5] It suggests that the goal is to develop "deep models" and "supple designs," which happens during breakthroughs:

> *[C]ontinuous refactoring prepares the way for something less orderly. Each refinement of code and model gives developers a clearer view. This clarity creates the potential for a breakthrough of insights. A rush of change leads to a model that corresponds on a deeper level to the realities and priorities of the users. Versatility and explanatory power suddenly increase even as complexity evaporates.*

That is exactly what you hope to achieve by iterating. However, 17 years after that was written, most developers are still refactoring superficially, doing CFI. If I had to guess why, I would say it's because most developers haven't heard of the idea or think that the entire DDD package of ideas is a poor fit for their project and so neglect this critical technique.

Iterate Toward a Clean Design

Plato wrote the allegory of the cave to teach us that invisible ideas can be more important than the visible shadows on the wall. We read his words thousands of years later not because they are easy but because they are uncomfortable. It's far easier and comfortable to attend to what we can see directly than to heed someone ranting about hidden truths. In fact, the second half of the allegory discusses how people who have only ever seen shadows would react when told about the invisible figures casting those shadows. Plato's verdict was grim: they would kill the messenger.

When I look around our industry at what teams are doing, I see many good practices such as iteration, refactoring, testing, and automated deployments. Despite those similarities, some teams are succeeding and others are suffering. What distinguishes them is how well they nurture their design (see Figure 2). As teams abandon upfront design, I fear that many of them are doing CFI, accumulating technical debt through sedimentary layers of obsolete ideas, and building technical zombies.

For a long time, we believed that iteration and refactoring were sufficient to keep a design healthy, but we can no longer believe that after seeing so many tangled designs and zombie projects. Software development is an intensely cognitive activity that cannot be reduced to simple activities repeated mechanically. Good design, while invisible, is critical and must be a goal of refactoring. By recognizing the distinction between CFI and DFI developers can adjust their activities slightly to keep their design healthy.

ABOUT THE AUTHOR

GEORGE FAIRBANKS is a software engineer at Google. Contact him at gf@georgefairbanks.com.

References

1. W. Cunningham, "The WyCash portfolio management system," in *Proc. OOPSLA 92*, Vancouver, Canada, Oct. 5–10, 1992, pp. 29–30, doi: 10.1145/157709.157715.
2. D. F. D'Souza and A. C. Wills, *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Boston, MA, USA: Addison-Wesley, 1998.
3. M. Keeling, T. Halloran, and G. Fairbanks, "Garbage collect your technical debt," *IEEE Softw.*, vol. 38, no. 5, pp. 113–116, Sep./Oct. 2021, doi: 10.1109/MS.2021.3086578.
4. G. Fairbanks, "The rituals of iterations and tests," *IEEE Softw.*, vol. 37, no. 6, pp. 105–108, Nov./Dec. 2020, doi: 10.1109/MS.2020.3017445.
5. E. Evans, *Domain-Driven Design*. Reading, MA, USA: Addison-Wesley, 2004.