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

Intellectual Control

George Fairbanks

IN THE EARLY days of software engineering, Edsger Dijkstra warned us not to let the size and complexity of our programs cause us to lose "intellectual control" due to the limited nature of our minds. To my knowledge, he never defined precisely what intellectual control was. Our software today is staggeringly larger than the programs of the 1960s, so does that mean we have it under our intellectual control, or did we find ways to make progress without Dijkstra's high standards?

I see signs that we have some software that is under intellectual control and other software that is not. In this column, I'm going to discuss how we can recognize these two categories, what happens when engineers on a project have different attitudes about intellectual control, some advice on when we probably should insist on it, and some ideas about how we achieve it.

It's difficult to have a conversation about something as abstract as control over programs. To ease into it, we can use a metaphor from Rich Hickey's presentation called "Simple Made Easy" (github.com/matthiasn/talk-transcripts/blob/master/Hickey_Rich/SimpleMadeEasy.md). He asked us to imagine driving a car on a road with guardrails. In this metaphor, the guardrails are tests and driving the car is us writing programs. He then wondered if it's OK to successfully arrive at our destination after hitting the guardrails during the journey. His audience laughed because having a car under control means we are able to drive without hitting the rails.

The laughter had a nervous quality to it because we often write code that ends up breaking the tests, so perhaps we don't really have as much control as we thought. Are we monkeys at typewriters writing random programs hoping one will eventually pass the tests? No, we have some expectation that this code will do what we want. Failing tests are a signal that we didn't understand the program as well as we needed to.

If we're honest with ourselves, the way we write software today has a little bit of that monkey quality. We don't get everything straight in our minds before typing, but neither are we just throwing programs at the tests without thinking. In my experience, we use a proposed change to a program as a little hypothesis, "I think perhaps this is the way things work," and then run the tests to get feedback on that hypothesis. Over time, we build up a theory in our heads of how the program works. That theory gives us intellectual control.

Let me contrast intellectual control with another kind of control you may recognize. Imagine that you start looking at an existing codebase with tests, one where the original authors have departed, so you cannot ask them questions. You develop hypotheses about how the code works and gather evidence by seeing whether or not your code changes break the tests.

But this time, you never get that "Aha!" moment that results in a general understanding of the program. Your hypotheses never add up to a theory. You are still making forward progress because you can try several things until you find one that keeps the tests passing, but you keep hitting those guardrails and they are the primary things keeping you from failure.

My intent is to reveal these two approaches to building software, which I'll call *intellectual control* and *statistical control*, so that we can have a discussion about their natures and when we should use one or the other. To some extent, I've exaggerated the distinction between them so that we can see things more clearly. We rarely find ourselves fully invested in one approach or the other. More commonly, there are parts of the code that mostly make sense to you and other parts where you're mostly relying on the tests to keep you on track.

You can have enough control over the car that you don't hit the guardrails or you can lack control and hit the rails a few times along

Digital Object Identifier 10.1109/MS.2018.2874294
Date of publication: 8 January 2019

the journey. Both get you to your destination. In software development, sometimes we are able to obtain intellectual control because we have built up a theory of the program and, therefore, are generally able to write code without breaking the tests. Other times, we cannot find that general theory, and so we're relying on the tests to give us statistical confidence that it seems to be working.

But why call it *statistical control*? Another of Dijkstra's famous sayings is that tests can show the presence of bugs but not their absence. Tests are a sampling of the output space, so passing tests provide a statistical confidence that the program behaves as expected. Passing tests don't mean that the theory in my head is right, just that there's no evidence that it's wrong.

From this perspective, we can see that intellectual control comes from ideas in the heads of software developers but not the code or tests. The same is true of the car and the driver, where the driver may have control or not. How do we know if someone is in control of the code? If they are able to write new code and rarely break the tests, that's a good sign of control. If they break the tests dozens of times first, that's bad.

Recognizing Intellectual Control

Dijkstra's appeal to math and proofs has the benefit of being an objective standard that indicates control, but proofs have been hard to achieve. Short of that, what other signs can we look for that show developers have intellectual control over their code? Intellectual control comes from your mind, so we need to find external evidence of something that cannot be seen directly.

One sign is a specification. Before proving a program correct, you need to have thought about it sufficiently clearly that you can state what you think it should do. Just stating that abstractly is evidence that you have some amount of intellectual control. That clear statement is a necessary ingredient of a proof. Dijkstra would not be satisfied, but I see it as evidence of intellectual control. This specification probably isn't a big document full of "the system shall" statements.

Another bit of evidence is the user-defined types that the program manipulates. Few programs operate solely on built-in types like strings, integers, and lists. Instead, the programmers invent types that reveal their thinking about the problem and solution. A rich and expressive set of types that can be operated on simply indicates insight and control.

Simplicity is evidence of intellectual control. Blaise Pascal wrote, "I have made this longer than usual because I have not had time to make it shorter." It's possible to just get started on an activity and barrel through to the end, but that yields something verbose and convoluted rather than simple and comprehensible. Tony Hoare said it best in his Turing Award lecture,[1] "There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies."

There are several techniques commonly used to simplify software designs. If you see these, they are signs that the authors have intellectual control: separation of the problem and solution concerns, explicit data structure invariants, operations with clear contracts, minimization of the state space, attention to failure conditions, and a suitable architecture.

I've found that developers with control over the code have the ability to give impromptu "chalk talks" that explain the system. Much like a CAD tool can generate any crosscut of a building design, those developers can respond to questions that crosscut the software design and provide off-the-cuff explanations of how the system does, or does not, handle it. Explanations of the form "here's why and how this exists" are better evidence of control than just statements of what exists.

Mixing Statistical and Intellectual Control

Inevitably, code that we understand will coexist with code that we don't, and developers who are striving for intellectual control will sit next to developers who are content with the statistical control of passing tests. Over time, does one approach dominate?

Imagine you are on a team of programmers that relies mostly on the statistical control of passing tests. Are you able to insist on intellectual control of the code that you write? Probably not, either because your code is built on other code that you don't understand or because the team has collective code ownership that will degrade your control over time.

It might play out like this: one of your teammates is confronted with a tricky programming challenge. With tight deadlines, he codes up something that covers the cases seen in practice so far, but there is no explanation of why it does what it does. Then he turns his attention to the next feature. Since you hope to keep intellectual control over the system, you try to build up an understanding of that module. But it's not a job of recovering what's there, it's a job of inventing a theory that explains the operation of the code and possibly refactoring the code to better

reveal that theory. That's a tough job that's hard to justify when the standard being applied is passing tests and statistical control.

It's easy for statistical control to dominate a project. In my experience, once some of the team embraces statistical control via tests, that will become the dominant form of control in the system. There is an inherent asymmetry in the two approaches where test-controlled modules have no problems working with intellectually controlled modules, but not the reverse. Those who hold intellectual control dear still want test coverage, but those who only desire test coverage don't miss the intellectual control, or at least not enough to pay for it.

Faced with the volume of code today, some people see intellectual control as an unaffordable luxury. Extensive testing does seem to work pretty well for the kinds of software we build and the quality targets we are trying to hit. And programmers change jobs, taking their understanding of the software away with them. So perhaps the idea of intellectual control is elegant but quaint, suitable for bygone days of punch card programming but not for modern Internet-scale projects.

When to Insist on Intellectual Control

Let's say that you are like me in that intellectual control is appealing but you recognize that it can be expensive. Where should you invest in it? The biggest win is in shared code. I have programed in Java since it was released, but I was shocked and delighted to start using the Guava library (github.com/google/guava) about five years ago. This polished gem represents a high point of intellectual control because few areas in computer science have had as much attention as collection libraries and functional programming idioms. The more widely used the code is, the easier it is to justify the investment on pure economics. You might also choose intellectual control when other risks are high.

One of the ways you can find areas to bring under intellectual control is to listen to what your module dependency graph is telling you. Since core libraries don't depend on your application modules, you can get them under intellectual control without fear that they revert to statistical control. But the reverse is also true: using statistical control over modules at the bottom of the dependency graph may make it hard to get intellectual control anywhere.

I find it valuable to pay particular attention to the code at the boundaries of a system such as the code that accepts requests from the outside world or loads/stores data from persistent storage. Any other calculations or conclusions reached by the system depends on understanding what happens at those boundaries: garbage in, garbage out. So if you want intellectual control anywhere, make sure you have it at the boundary of your system and at the leaves of your dependency graph.

Benjamin Franklin advised that "A stitch in time saves nine." It's cheaper over time to keep intellectual control than it is to lose then recover it. Still, the investment in that stitch can be hard to argue for because nothing bad has happened yet and the tests are still passing.

It's worth mentioning that sophistication isn't free. If your team wants to have an efficient process that reliably delivers quality code in quick iterations, you will likely need to invest in supporting practices. I often think of how many countries have built infrastructure to deliver tap water but cannot keep that water healthy because there are other supporting practices they have yet to master. Similarly, it's easy for a team to write a script that pushes the code to production every day but far harder to have the supporting processes that ensure the push is of good quality. As you raise your standards for feature development velocity or code quality, you may find yourself investing in intellectual control as a support.

How to Achieve Control

So, how do we make that investment? Over time, the specific advice has differed, but there is surprising consistency across the decades.

In the 1960s and 1970s, Edsger Dijkstra and Tony Hoare advised mathematical rigor as a path to intellectual control. Today, we rarely prove that whole programs meet a full specification but static analysis tools are common, scouring our code and letting us know if a null reference might be slipping past our defenses or if our user-defined types don't fit together just right. And Bertrand Meyer found an economical way to reap much of the benefit of a proof with design by contract.

In the 1970s and 1990s, Fred Brooks reflected on his experiences guiding software development at IBM and pointed out the benefits of coherent designs from the mind of a single designer who was freed from other distractions. In the 1990s and 2000s, Paul Graham continued this theme, sharing his experiences from the Lisp community with bottom-up design and advising how to hold a program in your head to achieve control.

In the 2000s and 2010s, functional programming has become mainstream, bringing ideas such as

ABOUT THE AUTHOR

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

immutable data structures and pure functions into all kinds of languages, often enforced by the compiler. I see a connection between Brooks' drive for conceptual integrity and Rich Hickey's advice on how to find simplicity in design.

Across the decades, formalism has helped us think about our designs, and it's increasingly accessible. Where Dijkstra would have used a pencil and paper to formalize his thinking about a program, a programmer today following best practices might build abstractions bottom up with clear contracts and invariants, structure the overall system according to architectural patterns, and use the compiler and static analysis to ensure each of the user-defined types fit together as expected.

My intent with this column is to make it easier to talk about intellectual control and connect it with practices we already use on our projects. It's difficult to talk about the kind of control we have over our software and more difficult still to talk about different approaches like intellectual control and statistical control. I'm thankful for Rich Hickey's metaphor of driving a car without hitting the guardrails as it brings this topic to life.

Software is everywhere, and it's a common lament that it's big and buggy. As the software gets bigger and bigger, it's more and more tempting to settle for statistical control with tests. Nobody wants to write buggy software, but many don't know another way or how to avoid analysis paralysis. You can choose a few places in your code, build up your understanding to gain intellectual control, and share that understanding with your team. Soon you will be driving your car without fear of hitting the guardrails.

Reference

1. C. A. R. Hoare, "1980 Turing Award lecture," *Commun. ACM*, vol. 24, no. 2, pp. 75–83, Feb. 1981.

REQUIREMENTS

(continued from page 90)

also arisen in the form of conference participants who will now use RE Cares artifacts in their courses and as datasets for research. Based on the Banff experience, we believe that RE Cares is a concept that can, with additional improvement and with proper preparation, be turned into a *successful* fixture at future RE, and perhaps software engineering, conferences.

Our RE community saw and seized an opportunity to "do good" while attending a scientific conference. It was a success per our surveyed participants [they scored it a 1.4 of 5 as a useful undertaking (where 1 is strongly agree and 5 is strongly disagree); see also quotes from participants' feedback at https://wsrecares.wixsite.com/recares/quotes]. It can be improved; we have identified such opportunities. We will undertake this event again. Are you interested in assisting us or in trying an RE Cares event at your conference? Please contact us at hayes@cs.uky.edu and help spread the idea that the "software types" of the world can give a little something back.

Reference

1. G. Ruhe, M. Nayebi, and C. Ebert, "The vision: Requirements engineering in society," in *Proc. Int. Conf. Requirements Eng. (RE)*, 2017, pp. 478–479.