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

Better Code Reviews With Design by Contract

George Fairbanks

DESIGN BY CONTRACT (DBC) is a technique that improves the quality of your team's code. It yields code with both a logical and a procedural nature, where the contracts state declaratively what will happen, and the implementations procedurally cause the desired effect. The team can reason either logically, by using the contracts, or procedurally, by following the code line by line, but the former allows them to reason about far larger programs. It also creates conditions for deliberate practice so developers using DBC grow their design skills faster.

Teams that are looking for ways to improve their code should seriously consider DBC. It is a technique for designing software in which each method has a contract, much like a legal document, stating what the caller is responsible for and what the method body must do. It was introduced as a term and adapted to object-oriented design in the 1980s by Bertrand Meyer[1] and traces its roots to the late 1960s with the work of Robert Floyd, Tony Hoare, and Edsger Dijkstra on reasoning logically about procedural programs.

Digital Object Identifier 10.1109/MS.2019.2934192
Date of current version: 22 October 2019

DBC is not a magic elixir that guarantees great programs. It's more like standard sentence mechanics in an essay. When sentences are awkward, it's hard for a paragraph or essay to succeed. There are authors who have written great essays while breaking grammar and style rules, but they have done so after they've mastered them. DBC ensures that methods and functions are simple and easy to understand and are therefore great building blocks for a whole program.

Many teams do code reviews so that all code changes are reviewed by a teammate. Before being sent for review, however, the author has already tested the code and knows it works. So, code reviews check not whether it works but if it's well designed. Clean code usually has a contractual nature while spaghetti code does not, even if its author is not consciously following DBC. Who has not sighed in frustration when encountering something like "void process() {/* 1kLOC elided */}"?

DBC and code reviews are a great combination. Reviewers may themselves write clean code but struggle to guide others to do the same. They can say how they would write the code but not articulate why that is better. Such advice can devolve into a battle of opinions. The opportunity with DBC is that when both the author and reviewer agree to a goal of writing code with clear contracts, they can look out for the DBC practices being followed (or not) in the code being reviewed. A reviewer can point the author to the relevant DBC practice and, fingers crossed, help the author improve the design. This article includes a list of DBC practices that you can add to your code review style guide.

Hungry for Contracts

What are contracts exactly? Let's introduce this concept using an example that's familiar to everyone—buying a sandwich—and apply some practices to decide on a contract. A sandwich seller might tell us that buying a sandwich can be broken down into a series of steps: take payment, give change, collect the ingredients (bread, peanut butter, and jelly), spread the peanut butter on one piece of bread using a knife, spread jelly on the other piece using a knife, assemble the sandwich, and deliver the sandwich. Let's call those steps the implementation of the buySandwich method.

DBC asks us to add a contract to that implementation, for example (given in U.S. dollar): If I give you $5, you

> Clean code usually has a contractual nature while spaghetti code does not, even if its author is not consciously following DBC.

will give me a sandwich. That contract states the buyer's responsibility and what the seller will provide in return. We can make that a bit clearer by saying "buyer" and "seller" instead of "me" and "you." We can also state what you probably assumed, i.e., that the seller takes the money. Here's an improved version of that contract: If the buyer gives the seller $5, the seller will give the buyer a sandwich and keep the $5.

If you had read that contract as a code comment above the method buySandwich, you'd be able to understand that method fairly well without reading the code body. Notice that the contract isn't a translation of the procedure into natural language. The contract doesn't talk about how the sandwich is made (e.g., using a knife) or the sequence of its operations. Instead, it states what happens before and after the method.

You may have noticed that the contract talks about what happens using procedural language: The seller gives the buyer $5. We can change that to declarative language like this: The buyer has $5. Now that it's stated declaratively, we can use it as the precondition for the buySandwich method. The postcondition becomes: The seller has the buyer's $5, and the buyer has a sandwich. The contract and procedure are shown in Figure 1.

Perhaps you are thinking that this is just a mild improvement, as there could have been a comment on the buySandwich method saying something similar. Besides, everyone knows how buying a sandwich works. What is different is that callers know what they can depend on. Consider a few details of the implementation that do not appear in the contract:

- Making change: The method body says that the buyer will get change, but the contract doesn't guarantee that. You have probably seen similar best efforts, say, from a vending machine that might not have exact change or a public bus.
- Assembly with a knife
- Peanut butter before the jelly
- Sandwich type: I'm sure there are readers who were already questioning whether peanut butter and jelly is an acceptable sandwich at any price.

buySandwich
Precondition: The buyer has $5.
- Take payment.
- Give change.
- Collect the ingredients (bread, peanut butter, and jelly).
- Spread the peanut butter on one piece of bread using a knife.
- Spread jelly on the other piece using a knife.
- Assemble the sandwich.
- Deliver the sandwich.

Postcondition: The seller has $5, and the buyer has a sandwich.

FIGURE 1. An example of contract and implementation.

Without the contract, you could read the method name and implementation and then guess at the contract, but it's easy to infer the wrong things. You could convince yourself that any of these details are something a caller may depend on. If so, how will we ever fix bugs or change the implementation to run faster? Stating the contract removes the guesswork. As an implementer, writing a contract leads you to think about what the caller can rely on and separate that from how the method is implemented. As a caller, a contract tells you what's safe to depend on.

Logical Reasoning

Without the contract, you can reason through a method procedurally, animating the source code line by line in your head like a little machine, and draw conclusions about how it will behave. When methods have contracts, you can still use procedural reasoning if you want to, but you can also apply logical reasoning.

In this buySandwich example, you know that, before calling the method, you are rich and hungry, and afterward, you are poor and full. That's consistent with reasoning procedurally about the implementation, but it's different. It lets you employ formal logic, which is why contracts are used in automated program analysis, such as when your IDE warns you that a value in your program might be null.

Contracts also let humans reason informally with logic, and we do that all the time. Imagine an implementation of buySandwich that uses two helper methods: collectMoney and makeSandwich. Does that work? Your logical intuition says yes. But consider different helper methods blandly named *A* and *B*. Now your intuition is less sure. You actually don't have any more information about collectMoney than you do about A, but your mind inferred a logical contract such as "At the end of the method collectMoney, the seller has the $5." You couldn't have been reasoning procedurally through the implementation because there isn't one.

We all reason through programs procedurally, but there's a size limit, and it's not big. Can you keep 10,000 lines in your head and reason about it? Using procedural reasoning, that's shaky, but using logical reasoning it's pretty easy. Consider this: I bet you recall the postcondition for buySandwich, but do you recall every step in the implementation?

To me, the ability to scale our reasoning is the great benefit of DBC. When you structure your program with clear contracts on methods, you can always fall back to procedural reasoning, but you also unlock your ability to reason logically and can keep larger programs in your head.

How to Get Started

Every code review starts with the author thinking that the proposed change is a good idea, so we should be looking for ways to guide authors in advance, not just during the review. Authors can shoot at a known target as they write code by using a list of DBC practices and the overall guiding metaphor of a contract that is usable by callers.

I was unable to find a checklist suitable for use in code reviews, so I created the list shown in Figure 2, and I think it's consistent with DBC literature, such as in Mitchell and McKim.[2] You can think of the list as an extension of the team's coding style. Like any style guidance, you should discuss and tweak it for your project.

> Once developers start thinking about DBC, it changes how they write every method.

I find that once developers start thinking about DBC, it changes how they write every method. There are always choices about how to decompose a problem, and they will gravitate toward methods where the contract is easy to state. When introducing DBC to an existing team or codebase, it's better to do it gradually and skip contracts that you think callers can reliably guess. Overall, state contracts as terse comments that help callers. DBC is something that you can practice on your own but it's even better if the whole team adopts it.

Low-hanging fruit is the easiest to pick and coworkers are unlikely to protest, so I suggest starting with methods that represent predicates, such as isActive or hasAddress. Contracts for these methods can be stated as one-line comments of the form "Returns true iff … ," where "iff" is short for "if and only if." If the codebase already has methods like these, the change is just stating the contract, but if the methods don't exist, then you also benefit from making the code read better by reducing in-line logic.

Next, turn your attention to query methods, like getStatus or getAddress. Because a comment saying "Returns the status" is unhelpful noise, think about the corner cases and write a contract if you discover anything interesting, i.e., how it handles null, whether the method checks for invalid data, or whether the values are only a subset of the declared type (especially for primitives such as string or integer). Also consider whether the accessors should exist, as they could be coupling the

- Contracts state what must be true about the inputs.
- Contracts state what will be true about the outputs.
- Contracts use declarative, not procedural, language.
- Contracts omit implementation choices, including sequence.
- Contracts state when a subset of a type is used.
- Contracts state how nulls are handled.
- Contracts state what inputs or states trigger predictable failures.
- Contracts identify any side effects.
- Callers can understand the contract without reading the implementation.
- State ubiquitous conditions as invariants.
- Prefer simple contracts over complex ones.
- Omit contracts only when caller cannot infer the wrong contract.
- Align contracts with the contours of the problem.
- Separate predicates, queries, and commands.

FIGURE 2. The design-by-contract practices.

ABOUT THE AUTHOR

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

caller to the current implementation unnecessarily.

When you write contracts for commands (i.e., transactional methods that are impure), look for opportunities to split the method. Broad methods such as updateCustomer tend to have long contracts that cover all of the corner cases. It can be easier to write the contract for a narrower method that does less, and it's easier for clients to understand. A set of commands likely has an obligation to maintain invariants on the data structures it manipulates, so make sure those invariants are clearly stated.

Deliberate Practice

As they grow, developers must learn to detect vagueness, incompleteness, and clumsiness in their designs. Years on the job will eventually give them those skills, but mundane experience is less effective than deliberate practice. One nice thing about DBC is that it can turn routine programming into deliberate practice. To understand how, let's look at deliberate practice in another field.

William Zinsser, an English composition teacher, says that in his writing classes, he would not cross out unclear or unnecessary parts of students' sentences but would instead put square brackets around those parts. Rather than simply telling the students what he considered the right answer, his technique encouraged them to wrestle with it themselves. His experience was that, by the end of the semester, they had learned to write terse prose.[3]

By encouraging his students to wrestle with their work, he created the conditions for deliberate practice. I see the same thing happening with DBC. When I'm just grinding out code, I'm not deliberately practicing. But when I force myself to state the contracts for each method, I notice when an idea is fuzzy, when the contract is rambling, or when my code makes unstated assumptions about a data structure.

The act of stating contracts creates the conditions for deliberate practice. It makes the unstated visible, like the square brackets that direct attention in an essay. The contracts let me see my code from a different perspective, revealing design flaws, and nudging me toward clearer designs. If DBC is able to accelerate the careers of developers by helping them learn to detect vagueness, incompleteness, and clumsiness in their designs, it is worth trying for that reason alone.

DBC is a technique taught to computer science undergraduates at many universities, including the Massachusetts Institute of Technology in 6.031 and Carnegie Mellon University in 17-241. It encourages designs where you know what must be true when a method completes, rather than designs where a method does a bunch of stuff and you squint to infer what exactly it means.

If you are a technical lead or manager who wants to improve your system's code, you could just wait several years until the team has more experience. If you want to do something today, however, there are only a few techniques that are easy to teach and offer the benefits that DBC does. DBC helps if just one person on the team applies it, and it helps more with each additional person.

There are other ways to arrive at elegant designs, but DBC is a particularly good fit for code reviews because a reviewer can point the author to a practice that the code does not yet follow. What's more, DBC leads the team to think about the abstractions that the contracts refer to, so it's a gateway to other helpful techniques like precise modeling.

Having contracts on methods is like having an owner's manual in your car's glove box. The most loved owner's manuals are the ones that are never opened because the design is simple and obvious. Everyone wants software that is simple and probable, but wishing does not make it so. DBC provides an early warning about awkward designs, shows where complexity still lives, and often leads to methods that callers understand without reading the contract.

References

1. B. Meyer, *Object-Oriented Software Construction*. Reading, MA: Addison-Wesley, 1994.
2. R. Mitchell and J. McKim, *Design by Contract, by Example*. Reading, MA: Addison-Wesley, 2001.
3. W. Zinsser, *On Writing Well: The Classic Guide to Writing Nonfiction*. New York: Harper Perennial, 2016.