Software Design

James Wilcox and Kevin Zatloukal

July 2025

Writing large programs correctly is *significantly* harder than writing small ones correctly. Somewhere between 10 and 100 thousand lines of code, most programs go through a transition. Implementing individual methods correctly switches from being routine to being a struggle. Making changes to the code switches from being difficult to being impossible. This transition does not come with a clear signal, like a sonic boom, but as a programmer, you can feel it.

We will discuss some of the reasons for this phenomenon below, but ultimately, the only way to fully appreciate this is to write large programs yourself. That is not something we can do easily in a context of 10 week course. For now, I will ask the reader to trust me about this. However, any experienced programmer, who has written programs of this size, will tell the same story.

Fortunately, we do know how to get through this transition. To do so, it is not sufficient simply to write code that operates correctly. That code must have additional properties that fight against the difficulties that arise as the program gets large. As a shorthand, we say the code must be "high quality".

What is High Quality Code?

The most important properties of high quality code are the following:

- High quality code is **easy to test**.

  To ensure that the code operates correctly, we must test it carefully. As the program gets larger, it becomes difficult to do so, as some parts of the code are not easy to exercise from the user interface. They may only called under special circumstances or after a large part of the rest of the code has run first. In the latter case, debugging will be extremely painful since the bug could potentially be in any part of the code that executed.

  One standard solution to this is to write tests for each individual method. We call these *unit tests*. In addition to writing such tests, we also need to structure the code in such a manner that unit testing will be most effective. In particular, unit tests can only verify the output of a method; it cannot verify the state produced in the middle of a method. For that reason, it is important to avoid writing large methods. Instead, large methods should be split into smaller ones, each of which can be tested.[1]

  These issues arise not only for individual methods but also for larger units of the program. Testing each method individually does not guarantee that the code is correct: the program can have bugs that only appear when multiple methods are used together.

- High quality code is **easy to understand**.

  When debugging, in order to figure out exactly where the code is going awry, we need to understand what the code is supposed to be doing at every point. Debugging is difficult under the best of circumstances, so debugging code that is hard to understand can be impossible.

  In order to make our code understandable, we need to limit the amount of surrounding code ("context") that needs to be understood at the same time. Limiting the context needed to understand

[1] Even with small functions, it is not always sufficient just to verify that the outputs are correct: the correct output could be produced by accident when the code is not operating correctly. For this reason, we often practice *defensive programming*, which means adding extra checks into the code to verify that it is operating as intended.

each part of the code has always mattered for human programmers, but we now know that the same is true for AI-generated code. As the required context becomes larger, the quality of AI code generation degrades until it becomes unusable.

- High quality code is **easy to change**.

  Most programmers are not writing new programs from scratch but rather are adding new features to existing programs. As a result, most programming work involves changing existing code.

  Changing code is difficult because what happens in one part of the code affects other parts of it. If we change the operation of one method, that can break some of the methods that call it. If the first change is necessary, then we need to change the callers as well. However, changing those methods could break their callers and so on. In a large code base, these changes can cascade until tens of thousands of lines are affected. Making all of those changes correctly is difficult.

  In order to build large programs, the code needs to be structured in such a way that certain kinds of changes are possible. We cannot make all code changes easy, so this requires some foresight into which changes are the most likely to be necessary. We can then organize the code so that those particular kinds of changes are easy to make.

Note that **efficiency** is not in this list. Indeed, the most efficient code is usually not easy to test or understand or change, so it is usually not what we want in practice! Most of the time, we want the simplest, most understandable solution available with an *acceptable* running time.

The Need for Software Design

Source code does not naturally have these properties. In fact, the most straightforward changes to the code usually make all of these properties worse. The main reason for this is laziness on the part of individual programmers, but this is a problem even for diligent programmers. For example, when writing some code that calls another function that we are already familiar with, it is easy for our knowledge of that function to influence how we write the call to it. Thus, knowledge of one function creeps into the implementation of another, increasing the context required to understand the new code.

In short, the natural processes at work when programming are always pushing to make the individual parts of the code more interdependent. The natural result for any project, if allowed to develop naturally, is "spaghetti code", where each part of the program is intertwined with many others.

One common way that this manifests in Object Oriented programs is that one class contains all the core data structures and a significant fraction (10–25%) of all the code. That class cannot be split up any further because all the parts are intertwined. In order to work in the codebase, every programmer needs to understand that class, which requires significant effort due to its size and complexity. This sort of failure pattern is so well known that it has a name: it is called a "god class".

Spaghetti code has none of the properties we want. It is hard to write new code correctly due to the large context required. When a failure occurs in the code, it will be hard to debug: the intertwining nature of all the code means the bug could be almost anywhere in the code base. Debugging is always painful, but in spaghetti code, it becomes nearly impossible.

Making changes to the code also becomes significantly harder. As noted above, interdependencies cause changes to cascade into more and more methods that require fixes. In the worst case, this chain of changes can loop back on itself. That is, we may discover that a change to one of our methods causes a cascade of changes ending with a different, conflicting change to that method. If this happens, it is probably *impossible* to make the change! The code is tied in a knot, and the only way to fix it is to throw it out and rewrite it.

What is Good Software Design?

The primary way to avoid the problems mentioned above is through appropriate use of *abstraction*. An abstraction hides details of one part of the code from another. This creates less interdependency, making the code easier to understand, test, and change.

The simplest example of this is a *specification* for a method. The specification describes to the caller what the method does without explaining every detail of how that is accomplished. In order to use that method, the caller only needs to read the specification, which is usually much shorter than the code itself. Furthermore, the specification allows the implementation to be changed in the future. For example, if the initial implementation turns out to be too slow, we can replace it with more complex but faster implementation. As long as this new implementation satisfies the specification, this change will have no impact on the caller. In other words, changes no longer cascade from one method to another.

The most important type of abstraction is the Abstract Data Type (ADT), which hides not only code from the caller but also the data structures used. Done properly, this makes the code much simpler to understand, as data structures can be quite complicated, and it facilitates change of those data structures without breaking any other code.

In non-object-oriented languages, the primary unit of programming is not a class but rather a file or "module". Properly designed modules have simple specifications. The individual parts of the module (methods or functions) have high *cohesion*, meaning that they fit together well and could not be split up further. The individual modules have low *coupling*, meaning that each one can be understood without knowledge of many other modules (and even then, just their specifications, not their code).

> "Designing modules is the heart of software design."
> — M. Ernst

The Need for Software Design... In the Age of AI

The use of AI makes the tendency toward spaghetti code even stronger:

- AI tends to write long functions that are hard to test.[2]
- AI reads the code of functions it calls, not their specifications, immediately creating interdependency that makes the code hard to understand and change.
- AI often repeats itself, writing similar code over and over instead of creating a single function that could be more easily changed.
- AI does not clean up after itself, leaving unnecessary code around that makes changes harder.

Left unchecked, AI can write 20,000 lines of code a day and produce spaghetti code in record time!

Using AI to build large scale software, at present, requires constant supervision to ensure that it is writing code that is not only correct but also high-quality. In all likelihood, human supervisors will not only need to design the abstractions to be used but also constantly police the AI to ensure that it is respecting abstractions and not sneaking around them.

[2] This was noted by Steve Yegge, author of "Vibe Coding".