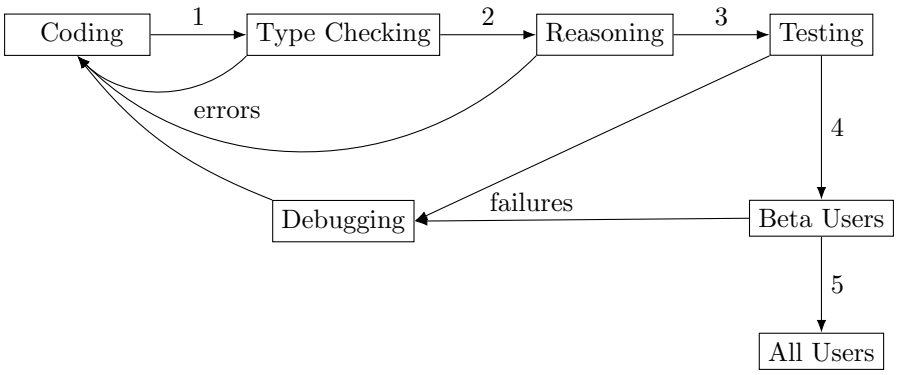
Software Implementation

James Wilcox and Kevin Zatloukal

July 2025

The process of creating software — be it a single method, a class, or a full program — starts with collecting requirements for what the software needs to do. Those requirement are codified in a *specification*. Next, we need to come up with an idea for how to achieve the desired behavior. A clear description of how to do so, along with a specification, is a *design*.

With those elements in hand, we turn to implementation. The time-tested approach to implementing software designs proceeds in the following steps:



- **Coding**: translates the design into source code in the appropriate programming language
- **Type Checking**: verifies that the code always produces values in the right set
  - checks all inputs (even invalid ones) but only checks simple properties (being in some set)
  - 100% reliable (as long as you have no type casts!)
- **Testing**: checks for correct outputs on *specifically chosen* inputs
  - checks for the exact answer but only on a handful inputs
  - tests can also be wrong! think carefully about each test case
- **Reasoning**: *thinking through* what the code does on *all allowed* inputs
  - only technique that checks that the code produces the correct outputs on all allowed inputs
  - usually done informally for most problems but formally for very hard or important problems
- **Debugging**: searching backward from a failure (e.g., wrong output) to find where the code is wrong
  - becomes more difficult as more code is involved
  - requires a full of understanding all of the code that executed

Once the code gets through testing, we let a small number of beta users try it. Only after it has worked reliably for them for some amount of time will we release it to all users.

Why These Steps?

To a newcomer, this may seem like a lot of steps and a lot of work. Why not just send the first version that compiles out to all users? The main reasons is due to the following, important fact: **users hate bugs**!

Encountering a single bug will typically result in a poor user report. If that bug causes the user to lose work, then they are unlikely to ever use the program again. If it calculates an incorrect answer where money is involved, expect a lawsuit.

Hence, one reason why new code is only released to a small number of users at first is to limit the damage if it contains bugs. Furthermore, beta users are typically warned about what they are getting into, and as a result, they are more understanding about failures. Regular users, however, are completely unforgiving! They do not give partial credit — only As and Fs!

Why *All* These Steps?

The reputation damage caused by bugs explains why we need beta users, but why do we need all the steps before that? There are two main reasons for that. One has to due with debugging, which we will discuss next. The other is that some bugs simply do not show up until you have a large number of users. A small number of beta users will not necessarily try every combination of features or every rare corner case that can occur. However, with millions or billions of users, someone eventually will.

Hence, if your goal is to avoid the reputation damage of sending bugs to regular users, then you need to find all the bugs lurking in every dark corner of the program before the code is sent to regular users. The hardest such bugs are unlikely to be found by beta users or by testing in general. The primary method of catching them, historically, has been *reasoning*, in particular, via the process of a *code review*, where a second programmer thinks through the code to make sure that no rare cases have been missed.

Debugging

Debugging is the search from a *failure*, something visible to the user, back to the *bug* that caused it. It is, in general, very difficult, often more-so than writing the code was initially. Indeed, Brian Kernighan once said "debugging is twice as hard as writing the code in the first place".

Debugging is generally harder the more code that needs to be searched through. The easiest case is a failure that occurs when a single method is being executed, as part of a "unit test". In that case, only that method and the methods it calls are part of the search space. The hardest case is when the failure is seen by a user. In that case, the bug could potentially be anywhere in the code.

Failures found by users are also harder because they rarely come with instructions for *reproducing* the failure. Debugging cannot be performed until we have the ability to reliably reproduce the failure. In some cases, finding a set of steps that reliably reproduce the failure is extremely difficult.

Once we have a way to reproduce the failure, the next step is to reduce the search space to a small enough portion of the code that we can step through it in the debugger. It would likely take weeks to step through one million lines of code to figure out where the bug occurs. Realistically, the search space must be reduced to a few hundred lines before stepping through them all becomes feasible.

The two principal techniques used to reduce the search space are the following:

- **Binary search**: If you can look at the state of the program in the debugger halfway through and confirm that the failure has already occurred (or has not), then you can cut the search space in half.
- **Scientific method**: If you can come up with an experiment, such as changing part of the code, that would prove the bug is in (or not in) part of the code, then you can perform that experiment and interpret the result to reduce the search space.

Both of these techniques require a complete **understanding** of how the program is *supposed* to work.

Even after the search space is down to a few hundred lines and we can step through the program in the debugger, we cannot identify the bug without understanding. When you are looking at the state of the program in the debugger, there isn't a red light that turns on to tell you that it is incorrect. You must know, from your own understanding, what the state *should* look like at that point so that you can determine whether that state is correct or not.

Binary search requires the same ability: to look at the state halfway through and identify whether that state is as it should be. The scientific method requires even more: the ability to imagine the behavior of the application in a hypothetical experiment (say, after a certain change is made to the code) and know how the resulting behavior would be different depending on where the bug in the code is located.

Debugging is hard!

In prior years of CSE 331, when writing the simplest full-stack applications, the average time to debug a single bug was over one hour, with many students encountering bugs that took several hours to find. This happened with programs that were typically less than 200 lines of new code. How can that be?

These assignments asked student to write applications consisting of two programs, a client and server. Debugging a failure observed by a user first required determining whether the bug was in the client or the server. That could be done by examining the requests and response messages sent between the two programs. To interpret them, the student had to understand whether the requests that were sent to the server were the ones that *should* have been sent in that circumstance. If one was wrong, then the bug was in the client. If the request was right, then the student would examine the response. To interpret it, the student had to know what response should have been sent. If the server sent the wrong response, then the bug was in the server. If it was correct, then the bug was in the client.

Even for assignments with only a client program, students spent an average 40 minutes per bug, again, with many students encountering individual bugs that took hours to find. How can that be? Well, the client application was a user interface. The user interacts with the application via events like clicking on buttons or typing in text. If the program crashes when the user clicks on a button, that does not necessarily mean that the bug was in the code that was executed after the click! Code that ran previously might have written an incorrect value somewhere in the state, but it was not until that value was read and used, much later, that a failure occurred.

Debugging is hard! Programmer productivity (and happiness and sanity) depend more on avoiding painful debugging than on anything else.

Hopefully, these examples give you some sense of why the software development process spends so much effort making sure the code is correct when it was written initially. These steps might seem like overkill, but amount of time saved by avoiding debugging more than makes up for it. Debugging is hard, harder than writing the code correctly in the first place.

Incorporating AI

At present, AI is being incorporated into this process in two different ways.

First, non-programmers are using AI to generate code and then testing it purely through its user interfaces (i.e., not via unit tests). This approach, often called "vibe coding", has the advantage that it does not require any programming knowledge. When the program does not work properly, they try prompting the AI to fix it. If that does not work, they try re-prompting the AI, having it replace some of the code, hoping the new version will not contain the same bug.[1]

Thus far, this technique has only been successful for very small programs, say, fewer than 1000 lines. It would be surprising for this approach to work for significantly larger programs. Even human programmers generate a few bugs every hundred lines of code, which means dozens of bugs in 1000 lines of code. Without the ability to test individual methods — which requires some knowledge of programming — it is extremely difficult for anyone, human or AI, to produce a large, error-free program. The inherently random nature of LLMs means there is always a nonzero probability of generating a version with bugs, and as the length of the generated code increases, the probability of it being bug-free tends to zero.[2]

The other way that AI is being used is under the supervision of a trained programmer, who can use the AI to generate the code and even to generate unit tests, provided that both are carefully reviewed. This method does not have the limitations of the approach described above and can be used to write large programs. We will explore this approach in the course.

[1] Sadly, even if it the old bug is gone, it could contain new bugs!

[2] Of course, the code is not written in a single shot, but rather, updated after being prompted each time. However, an update to the code can not only fix existing bugs but also introduce new bugs, so the odds of a series of updates also producing a long, bug-free program tend to zero.

It is important to note, however, that it remains the responsibility of the human programmer to careful review and understand every line of code and every test. As just noted, AI generated code frequently contains errors (just like human code), fixing those errors frequently involves debugging, and debugging requires understanding how the code is supposed to work. Skipping the work of understanding and carefully reviewing the code is likely to end up in extraordinarily painful debugging!

Debugging is already hard. Any approach that takes the hardest part of writing large, correct programs (debugging) and makes it even harder is not a good methodology. The fact that it requires less effort to do the work necessary to make sure the code is correct initially (testing and code reviews) than it does to debug that code is even more true with AI involved.

At present, careful attempts to measure improvements in developer productivity have found little if any benefit in having AI write the code. Features are not being shipped any faster. Meanwhile, code submissions are introducing more bugs and getting longer, making them harder to debug.

Surprisingly, the area where AI is having the most measurable improvement in productivity is not in writing code but rather in debugging, specifically the analyzing of stack traces, which is a laborious (and boring) process. It may be that these initial efforts have it backward: AI shouldn't be trying to help us write the code, it should be trying to help us debug.

Time will tell whether these trends continue, but at present, there is no reason *not* to think that careful testing and code reviews will remain essential to producing programs that work reliably when released to all users and to keeping the programmers that create them maximally productive.