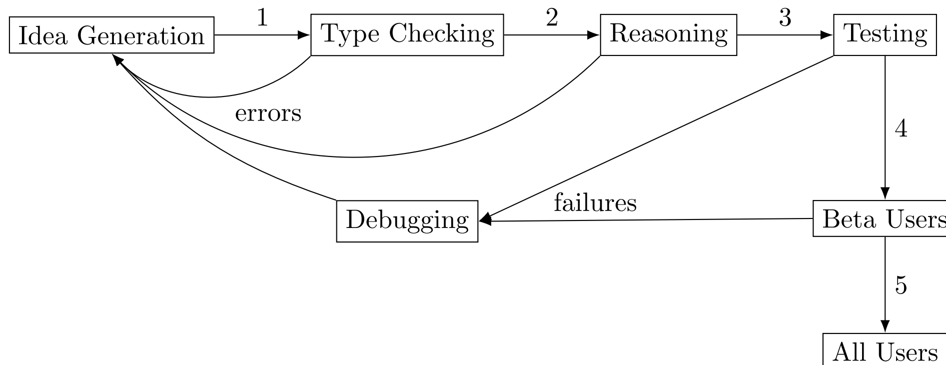
Core Concepts of Software Development

James Wilcox and Kevin Zatloukal

February 2024

High-Level Steps



- **Type Checking**: verifies that the code always produces values in a particular 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!)
- **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
- **Testing**: checks for correct outputs on *specifically chosen* inputs
  - checks for the exact answer but only on some inputs
  - can be wrong! think carefully about each test case
- **Debugging**: searching backward from a failure (e.g., wrong output) to find where the code is wrong
  - requires a full of understanding all of the code that executed
  - becomes more difficult as more code is involved

Once the code gets through testing, we let *some* users try it: alpha / beta users initially and ordinary users only later. Both the reputation damage and the cost to fix the bug increase the later it is found.

- Debugging a failure seen by a beta user is much harder than debugging a test failure. The search from the failure back to the bug will be longer and much more code needs to be understood.
- **Beta users** are usually understanding about failures, but ordinary users are completely unforgiving! Ordinary users do not give partial credit. They give only As and Fs.

With enough users, they will eventually try every possible type of case and discover every bug! The only way to avoid reputation damage is to find *all bugs* that cause failures before step 5 (see diagram).

Important Maxims

Here are some insights gleaned from the forty years of programming experience of your instructors as well as those of the general programming community:

1. **Programmers overestimate the importance of efficiency and underestimate the difficulty of achieving correctness.**

   The speed of the program is important, but only a small part of the code actually determines that. Most of the code you write is only executed on small inputs or only a small number of times. Those parts could usually be 100 times slower without users noticing any difference.

   What about the small part that does affect efficiency? You might think you can guess which code that will be, but the experience of almost all programmers says otherwise. Barbara Liskov once noted that "programmers are *notoriously* bad at guessing what will be slow".

   Time spent optimizing the rest of the code is wasted. Even worse, optimizations can introduce bugs, which will ruin the user experience. Don Knuth once quipped "premature optimization is the root of all evil". Most of the morality tales about programming start with someone foolishly trying to optimize some code (whose performance is probably immaterial) and end in tears.

   In summary, wait until you can measure the performance experienced by beta users before trying to improve efficiency. Before then, focus on keeping things simple and ensuring correctness. Usually, programmers underestimate the amount of time required to write the code correctly by *roughly 3x*.

2. **Reasoning is not optional on real problems: you either reason or debug and then reason**

   Reasoning can be informal rather than formal, but you have a professional responsibility to *think through* what the code will do on all inputs until you are convinced that it will produce the right answer in all cases. This is not only true in principle but also in practice.

   Any bugs you miss will eventually be found by users. The failure they see must first be debugged to find its cause. For real-world problems, fixing that one case without thinking all the cases through (i.e., without reasoning) usually just moves the bug somewhere else, leading to another round of debugging.

   Homework problems are often small enough that they can be solved "by accident" — i.e., without thinking them through — but that rarely happens on the kinds of problems people will pay you to solve. For those, you eventually have to think through all the cases to get them all right. As a result, the choice is not between reasoning and not reasoning; it is between just reasoning and debugging plus reasoning. It is usually less work to just do the reasoning the first time.

3. **Engineers are (only) paid to think and understand.**

   You should only expect to be paid for the reasoning (thinking) and debugging (understanding) parts. All the other steps can be automated and will eventually be available to anyone at ~ 0 cost.

   In particular, you should not expect to be paid to "just try things until the tests pass". That is (already) not a skill that someone will pay you for. In fact, you are unlikely to get through a coding interview if you work that way as interviews are done without a computer! This ensures that the candidate can solve problems by reasoning because no other tool is available. A typical coding problem has enough cases that you are unlikely to get them all correct without thinking through all the allowed inputs.

   Coding interviews are not usually about testing your knowledge of how to solve well-known problems. Interviewers want to see that you thinking through the problem. In fact, if you solve a problem too easily, without much evidence of careful thinking, they will often throw out the results of the interview, assuming that you have seen the problem before! Interviewers want to see you reasoning, so if you practice and get comfortable with reasoning, you can be confident going into coding interviews, even without any special preparation.