Michael Shintaku
UWid: shintaku
CSEid: shintaku

1)
One problem is translating English or other descriptive languages into a programming language. A simple example would be translating english to predicate logic. It is very easy to be imprecise when using general terms, as it requires the translator/coder to have the same knowledge, perspective and definitions of those terms as the one issuing the original statement. Without a strong specification and definitions, the translator will have to make decisions, which could create wildly unexpected output. Clear communication and feedback is vital for a good and correct translation.

I think this is a hard problem to solve, as a perfect solution would be for the client and translator to have the exact same perspective, or be the same person. To have the same perspective would require one to know all of the information and experiences that the other knows. The size of this knowledge can be very hard to work with, and harder to simply identify or retrieve. It would also be impractical for the client to perfectly specify every minutiae of their items, intent, and goals (if they are trying to simply convey their perspective). Also, it would be hard to determine how these might change with future events that have yet to happen.

I speculate that there isn't a clear-cut solution to this problem, because the uncertainty and problems that arise from it are minimal given continuous strong communication and feedback between the client and implementor. The work required for an automated tool could be hard as simulating the client's brain perfectly, and applying that with the implementing language. At that point, the client might as well be speaking in code instead of English.

2)
One issue is the effectiveness of testing. For exhaustive tests, it can be expensive to test against all possible inputs, and impossible given the many combinations of possible hardware/software states and variable states. So, there is always risk in that the tests don't cover some important domain or combination of inputs.

While automated tests may help increase test coverage, it can be impractical or cost ineffective for the test suite to determine what is correct for an arbitrary input. An example would be photo identification (can be overkill to acquire and train, vs human reasoning).

I speculate that there isn't a clear-cut solution to the most effective testing, because it can be impossible to have a program pass all possible tests, and it is hard to determine relevant domains and input combinations for testing. I think the best and cost-effective solution at the moment is to have a human programmer determine when to apply different testing approaches and tools

3)
Another problem is updating and/or upgrading legacy code. Incremental improvements is generally preferred versus a total overhaul of a codebase, but there must come a tipping point with long-running software, or it will fail to keep up as the maintainers dwindle in number and understanding.

There are some solutions, like building modern interfaces to work with the old legacy code, but they are short-sighted if the code is obsolete and the knowledgeable users are few to non-existent.

I speculate that there isn't a clear-cut solution to updating legacy code. It can be hard to determine when software is significantly outdated, and when/if it is cost-effective to upgrade the system to new technologies vs making improvements. In a perfect world, software would be kept on the most modern and powerful technologies, but this is simply impractical given businesses/people's resources and time.