Matthew Staehely
mstaehel

In CSE 403, my team and I developed an application for the Microsoft HoloLens. This required us to use substantial portions of Microsoft's HoloLens codebase, a.k.a. the HoloToolkit. The code we were provided by Microsoft's GitHub is a textbook example of most code not being documented. Those classes and functions which did have documentation were documented poorly, and many were simply not documented at all. Being that the Git is publicly posted, and much of the code is either based on open source work from MIT or provided (and maintained!) by Microsoft developers free of charge, we expected that the code would be very clear and easy to use. This was definitely not the case. While I can't say for certain *why* this hasn't been fixed yet, I suspect it ties in to what we discussed in class today: writing up clear and concise specs for the devs who will be using this software is a significant time investment. Since the HoloLens was originally intended to be developed on in-house, it does make some degree of sense that the code base would not have the sort of documentation that we'd expect from something released to the public. When it was opened up, what had previously been satisfactory was no longer so. I believe this hasn't been fixed for two reasons. The first is that, despite the poor documentation, people are still developing, muddling along as best they're able and constructing a knowledge base that's shared among users. The second reason is that it would be a substantial amount of time to fully document the entire HoloToolkit, and testing the device to ensure the specs are being met takes some time. Ensuring that voice recognition works properly does not seem to lend itself to unit testing, for example, and spatial recognition seems near-invulnerable to black-box testing.

The best solution I see to this is, unfortunately, to have a team comb through the code line-by-line and do just what I described above: develop the contract, and then test it for as many possible inputs as they can. Again, for something like voice recognition, that seems pretty implausible. While a good range of vocal inputs can be tested against, testing against all possible inputs isn't feasible.

Another difficulty we ran into while developing was that the number of warnings provided by the IDE we were using was outrageous- dozens at compile time, ballooning up to hundreds at runtime, if the application was run in debug mode. This was because a portion of the software we were using was deprecated, and had been for years, but a suitable replacement had never been provided by the engine we were using (Unity). The fix proposed by the engine's developers rendered the code unable to be compiled, or broke the graphics we were using. Because the number of warnings was so high, this caused the team to begin ignoring them. Later, flaws began to creep into the system and we spent a long time being overlooked because the software still compiled, and even ran under many circumstances. A long, stressful bug-hunt followed, and it wasn't until warnings were turned back on that we were pointed in the right direction. I don't think that this is a fault of the tool- it was warning us about a very real possibility, namely that the engine may not support our graphics someday. Fixing this could end up being a substantial amount of work, and would have forced us to rewrite a lot of what we were working on. In an ideal world, this is what we would have done. I know that the studio that developed the graphics has moved on to other projects, and because they were not provided by Unity themselves, Unity has no reason to fix the problem.

The solutions I see to this issue feel like 'kicking the can', to me. For instance, muting warnings entirely just didn't work. Adding the ability to tell the tool that "we were aware of these problems, and please stop warning us" would only make us stop being warned about those problems. In a non-classroom situation, once the project's team had cycled completely, then there may be nobody left working on the project who would even know that certain warnings were being suppressed or muted by the tool.

While working on our project, we used a lot of singletons. The singleton design pattern is great; it allows for easy instantiation and referencing of values or objects, and makes modification of that singleton relatively painless. However, as we were dealing with a multi-user system, each user needed their own instance of the object, and often times that object's state would need to be modified relative to the state of the other user's singleton. The up side of this design was that it made each user's state very easy for that user to track. The down side was that we were, in a way, violating the idea central to the singleton design pattern- each fully operational execution of the application would have multiple instances of several singletons, which themselves had different states. I think the main reason this problem exists is for the exact thinking we used while working on the project- despite not following the strict definition of a singleton, it made some of the debugging process much easier, as we could display each user's various states through different stages of application use. Fixing this would require creating a new design pattern that behaved in a similar fashion to a singleton- instantiate with specific set variables- but that would also have a baked in functionality that would change the state of various instances relative to the other instances. Though I think this would be a fun challenge, I'm not certain that every designer or developer would find it to be an effective use of their time. Also, I'm not convinced that this is an effective, general-purpose solution. It does solve *this problem*, but pulling back and looking at the overall design of our application seems like a more likely solution. If we had designed it to begin with such that each player wasn't represented by its own model, but rather the entire state of the game containing a set of all player objects (with the necessary fields/states), we would have had a much easier time. However, time once again played too large a constraint, and we did not do this. Looking back, this is almost certainly how I would solve the problem: either one player/user is nominated as the host, and contains all that information, or a centralized server all users connect to stores and maintains the model. Then we could more directly manipulate the players as needed, without having to use workarounds to ensure the users' states were correct.