Software Engineering Homework 7

1. The architecture was somewhat supportive of extensions. For one thing, the user interface was very well modularized. Every button, menu, or other graphical entity was self-contained, and they were all put together in the same place. This made it very easy to change the way the existing features were displayed, add or remove existing features, etc. without changing more than a few lines of code. However, once we delve into the classes that are behind each graphical component, we see that there is a whole lot of interplay between them. We cannot understand how one piece works without learning how several others work, and if we were to try to change one it would almost certainly require modifications in others to keep them from breaking.

Fortunately, our enhancement does not involve trying to modify the somewhat tightly coupled classes that already exist. We are adding a new class, which does implement several existing interfaces indirectly through inheritance. There are not any direct advantages to implementing these interfaces that we are aware of, but it can only help with keeping the project as cohesive and pliant as possible.

Our modification left the existing architecture almost completely intact, with only minor changes in the client builder class to include our addition. So while we did not do anything to destructure the existing code, we didn’t really improve it either.

1. The documentation associated with this project is extremely minimal. There is a website, which provides instruction on how to operate the program as an end user, but really nothing laying out how the code works. There is certainly no UML available anywhere, and even comments in the code are scare. Some interfaces have brief descriptions, but none of the classes do. The information that was provided seemed current, at least as up to date as the software was.

The end user manual was somewhat helpful; we were able to discover all of Rapla’s features with it, but it did not do much for us beyond that.

The documentation we provided was mostly user interface oriented, as our goal was to streamline the UI. Our UML diagrams mostly included classes that were displayed in the interface, or what steps would be taken to go from the interface to what we wanted. It was certainly tailored to the TA, as realistically it does not seem likely that anyone else will look at it in the future. If this were a more active project, we might have tried to make our diagrams more useful to other developers. But they were also made for ourselves to better understand the code we had in front of us, so they should provide some insight for other developers as well.

1. When we first looked over the suggested enhancements for the project we chose, they seemed almost trivial. We had forked a copy of each one of the suggested projects to test it out, and choose one that compiled, ran smoothly, was fully featured, and appeared to be bug free. It was not a complicated problem either, just a calendar. No physics or unfamiliar game concepts we would have to learn. How difficult could implementing a search feature possibly be? That would be a simple UI tweak. The program already had a method of filtering items, so we wouldn’t even need to write that part of it.

Then we began to explore the myriad of classes and packages that made up that simple calendar program, and it began to dawn on us just how daunting it was. Neither of us have previously had to work with code that somebody else had developed, at least not on this kind of scale. The largest software project we had formally been exposed to for school consisted of no more than a hundred or two lines of code in total. We had no idea how to even begin to understand the inner workings of our project, in fact we spent several days just looking through everything and getting no closer to knowing why it was put together the way it was. There was nothing we could point to and even have the slightest clue why it was included and what it was contributing to the calendar. We tried stepping through the program with eclipse’s debugger, but the launcher went into closed libraries before doing anything that made sense to us.

It was humbling, and we can see why in a situation like this where developers need to become accustomed to somebody else’s program, even minor modifications could easily take longer than one might expect. However, it also seems likely that once the developer *has* taken the time to learn what the important parts of a project are and how they go together, any subsequent modifications would be much easier. With this kind of a barrier to entry into any given software project, it seems there is value in keeping the same developers associated with the maintenance and expansion of a program as much as possible, or at least keeping some people who are familiar with it working on it so the amount of time wasted by the team just trying to understand everything is minimized. Some advice that someone at the beginning of the semester might find helpful is that, as we learned in class, the majority of the code is probably not directly involved in the particular application they are looking at. Much of it is just infrastructure, or if the code was written well it is generalized and potentially has the capability to do more than just what is displayed when it runs. There are likely only a few classes that pertain to whatever enhancement they might have in mind; it’s mostly a matter of finding them.

1. We tried a lot of things. Setting breakpoints was not at all helpful until we knew where exactly to put them. Some of the lab assignments turned out to be very good tools for better understanding and visualizing what was going on. The sequence diagrams in particular shed some light on what everything was doing when. Others made less sense though; the class diagrams didn’t do much for us before we could find any classes that were at all useful to us. The call hierarchy feature in eclipse was definitely a boon. We had found the FilterButton class early on but didn’t know where it was being used until we discovered the call hierarchy.

So the first challenge was just finding which classes were actually relevant to what we were interested in. Once we had, we still needed to determine how the relevant classes worked, and then how to mold them in the way we wanted. That also proved to be more difficult than we’d expected; we’d thought that once the classes we needed were isolated things would be easy. Unfortunately, there were almost no comments anywhere, and this too was a challenge. We mostly went about this part the same way we have been dissecting code since before this project-just by using the debugger, thoroughly examining the code, and throwing things against the wall until something stuck.

Someone starting out would probably appreciate knowing about the call hierarchy feature, as well as sequence diagrams. We tried to be as clear as possible with the documentation; describing how our enhancement worked as well as how the parts of the existing program it interacts with work. We hopefully gave enough detail for someone unfamiliar with the program to understand, but not so much that it is overwhelming.

1. Although we definitely learned a lot and are in a much better position now to make any given modification to Rapla than we were at the beginning of the year, it kind of felt like we just scratched the surface in understanding it. We isolated the classes and functions involved in creating the user interface. With what we learned it would be trivial to make cosmetic changes, and anything that just involved new or different buttons would be a very approachable task for us. But there is a lot more going on with Rapla that is very much beyond us. For one thing, we only ever ran the standalone version of the program. It appears that Rapla is intended to run on a network with several computers and users, and we would have absolutely no idea how any of that worked. Given that we only observed our enhancement working on the self-contained admin version of the program, we don’t even know if the changes we made would manifest for end users on an actual network. It could be that the launcher that is actually used when the program is implemented on a network builds the client GUI differently, and our modification was completely for naught. That seems unlikely, but we still wouldn’t even know how to approach it if our task was to change anything about how Rapla behaves on the network.

A student coming into the course would probably appreciate knowing that it is not at all necessary to understand the entire project to be able to work on it. They are all open source, so they were likely developed by multiple people who may not have had a complete understanding of every aspect of the code in the first place. Most of the projects we have worked on have been small enough that we could know exactly what was happening behind the scenes for every part of it-it usually wouldn’t even be unreasonable to step through them line by line from beginning to end. But these projects are very different. Provided the code is well modularized, it is feasible to have little to no understanding of anything outside of the part that is to be modified and still be successful.