504: Machine Learning meets Program Analysis

Assignment 1

Due: **Monday**, January 11 before class (at 10:29am)

In 1–2 pages, list several difficulties that you often encounter during software development — design, implementation, documentation, testing, maintenance, or otherwise. For each such difficulty, discuss the underlying cause. If you were to address this yourself, what data would you use? Can you imagine ways that tools could help?

Below are some examples (you should come up with your own). Your list need not be nearly this long, but each item should be more substantial — spend about half a page discussing each problem. Motivate why people should care about it, explain why it is not easy to solve or work around (for example, why current tools do not address the problem), speculate on why no one has solved it to date, and brainstorm some possible solutions.

To facilitate sharing them with the class, submit this assignment in a single PDF file. Please start the file name with your family name.

- In dynamically-typed languages, it is easy to apply illegal operations to data, and for such errors to be latent for long periods until some input exposes them. Even if easily reproducible, they might be exposed only after other long-running computations, making them difficult to detect. This sort of error is particularly frequent when accessing deeply-nested data structures: it is easy to forget one level of dereference.
- Interning values (replacing them by a canonical version) saves space, since only one version needs to be stored, and also saves time, since equality testing can be performed via pointer comparisons. Failure to properly intern can be extremely hard to track down, however.
- Reusing code requires an understanding of its behavior. Most code is not documented, however. This lack of specifications and documentation makes it difficult to use the code and difficult to know its assumptions or operational parameters (the values over which it has been tested and is known to work).
- Large components are often more worth reusing than small ones. However, they are also more likely to make assumptions (such as that they control execution of the program, that there is no need for thread safety, or that batch processing is acceptable) that may not be acceptable to an integrator who wishes to use these large components.
- Test suites are crucial to eliminating bugs and improving confidence in programs, but are difficult and tedious to create. For example, it may be difficult to create valid inputs to a program, because there are implicit constraints on the input. Given an arbitrary input to the program, it can be difficult to determine whether the program operated correctly.
- Large, comprehensive test suites tend to take a long time to execute. That discourages developers from using them as frequently as they ought to, or slows them down waiting for tests to complete when they could be moving on to other tasks.
- Some program analyses produce so much output that it is difficult or time-consuming to separate the useful information from spurious reports.
- "Maintenance" activities (modifying a program) tend to degrade its structure at both the macro (modules) and micro (specific functions) levels.
- It is hard to build new analyses for existing languages (such as Java); lots of grunge work is required, even though building a front end is no longer considered interesting research. A related problem is that real infrastructures are difficult to use, and it is difficult to integrate new tools with them.

Avoid discussing implementation annoyances unless you can identify an underlying principle. (Example: "Windows (or Unix) lacks this feature that Unix (or Windows) has," or "My favorite tool does not support *x*.") Avoid mentioning difficulties in performing tasks that don't matter. (Example: "I can't determine the cyclomatic complexity of my Tcl code.") Avoid problems that are extremely minor or that can be solved easily. (Example: "I often fail to balance delimiters before attempting to compile a file.")

While some people might be able to knock off an answer to this assignment quickly, it will reward careful thought about interesting problems and issues. Please introspect deeply and thoughtfully about program development and maintenance. Doing so will help you in this class, and beyond.

Be specific and concrete

A common problem that students suffer on this assignment is vagueness. Vagueness prevents the reader from understanding either the problem or the solution. For example, it is not helpful or informative to say, "programmers do not use testing tools" or "programmers have trouble with refactoring". This needs to be clarified in multiple ways.

First, what sort of testing tools are you talking about? Unit test frameworks? Mocking frameworks? Automated test generation tools? Test execution frameworks? The same goes for varieties of refactoring. It is worthwhile to do a bit of research to augment your own personal experience and knowledge. Also, you should give concrete examples of the sorts of tools you are talking about, since that will indicate to readers your context.

Second, you haven't said *why* programmers do not use the tools or have trouble with the task. If you are not sure of the reason, then think about it, or do a search to learn the reason. At the very least, lay out some possible reasons and discuss whether you believe those are the true reasons. Even this process will increase your understanding and may lead you to come up with new and better reasons.

If there is a problem that programmers have, then once you have described it, talk about possible solutions, whether manual or automated. Do not merely assert that there is no way to solve a given problem. This is usually false, and may indicate that you completed the assignment lazily, without doing background research. It might be the case that there is no automated way to solve the problem. If this is true, then think about how programmers solve it by hand. Whenever a task can be done manually, an approach to automation is to dissect and understand the manual process, and see whether some of the parts can be aided mechanically. Even if you don't solve the entire problem, it can help programmers in the overall process. Furthermore, an impossible problem is often just a distraction from your real work. It may be that a problem as posed is impossible, but that problem may not be what programmers truly care about. Maybe there is a way to finesse the problem or solve it in a different way. Think about these — even the ones that seem impossible at first may help you better understand the problem.

Finally, do not over-claim. If you don't know about a topic, then admit that, or look it up. If you make even a few false claims, then you will lose the reader's trust. You will also confuse yourself and may prevent yourself from trying an approach that could be productive.