CSE 490E1: Software Quality: Techniques and Tools

Assignment 1: Brainstorming about software development difficulties

Due: Thursday, January 5, 2017 before class (at 10:29am)
Submit to Canvas

Think about your sofware development experience. Oftentimes it is lots of fun, but at times it can be frustrating. Describe three concrete difficulties that you have encountered during software development — design, implementation, documentation, testing, maintenance, or otherwise. At least one should be about testing, and at least one should not. Below are some examples, but you will base yours on your own experience, you will discuss each item in more detail, and you will illustrate it with an anecdote.

For each difficulty, motivate why people should care about it, explain why it is not easy to solve or work around (discuss, why current tools and techniques do not address the problem), speculate on why no one has solved it to date, and brainstorm some possible solutions. Spend $\frac{1}{2}$ to $\frac{2}{3}$ page discussing each problem, and submit a **two-page PDF** file. For anything that you submit in this class, place your name and UW user ID on the first page (or first slide).

This assignment will reward careful thought about interesting problems and issues. Please introspect deeply and thoughtfully about software design, development, and maintenance. Doing so will help you in this class, and beyond.

- 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 code quality tools produce so many warnings that it is difficult or time-consuming to separate the useful information from spurious reports, prompting programmers to give up on these tools.
- "Maintenance" activities (modifying a program) tend to degrade its structure at both the macro (modules) and micro (specific functions) levels.
- Good performance can be achieved by storing computed values in a cache; the next time a value is needed, it can be looked up rather than recomputed. However, if values in the caches are side-effected

or if the computation depends on values that are not use as an index into the cache, the cache becomes corrupted, leading to extremely hard-to-debug errors.

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.")

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. Give concrete examples of the sorts of tools you are talking about, since that will indicate to readers your context.

Second, why don't programmers use the tools, or why do they have trouble with the task? If you are not sure of the reason, then think about it, or do some research 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. Search for solutions. If there is no automated way to solve the problem, 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. Beware of getting distracted by the fact that some task is impossible. It may be that a task as posed is impossible, but maybe there is a different way to solve the underlying problem that programmers care about, or maybe there is a special case that is tractable. Even approaches that seem impossible at first may help you better understand the problem.

It's also too vague, and not helpful, to mention a well-known engineering tradeoff, such as "modularity is useful, up until the point where the modules are unreasonably small" or "given a legacy program, is it better to improve it or to throw it away and reimplement it?" Give a concrete, actionable example.