Dani Lerner

114054883

Week 4 Written HW

5.1: How are software changes classified by their purpose? What is the most common purpose of the change?

Software changes are classified by the reason a change is necessary. There are four potential reasons we'd want to change software.

1. **Perfective changes** refer to the addition of features in the hopes that they improve the system and its corresponding value. This is the most common type of change.
2. **Adaptive changes**: When a program's environment changes, the programmers are forced to adapt the software and get it up and running in the new environment. This can refer to changes in the operating system, computer architecture, coding libraries, etc.
3. **Corrective changes**: When customers (or programmers) find bugs in the system, programmers must find and extract the bug from the system. These bugs are specific and display lapses in the code coverage. They might happen because of unexpected use cases or other "you don't know it will happen till it happens" moments.
4. **Protective changes**: Protective changes are proactive rather than reactive changes. These do not affect the system and are invisible to the user. They are generally changes in code architecture and enables more agility later on.

5.3: When is it permissible to do quick-fix changes?

It is permissible to perform quick-fix changes when the immediate value of the program is more important than the long-term value. Basically, you must assume that quick-fix changes will make development significantly more difficult, complex, and hard to follow in the future so it should generally be avoided. However, there are times where technical debt is worthwhile if the immediate performance is important. An example of this could be a sudden demo to stakeholders or customers. There may not be enough time to do a good job and immediately after, the programmers can perform a more legitimate change.

5.5: What is a product backlog?

Oftentimes a desired perfective change to the system is not a single change but a collection of many. A user defines their requested requirements in human language. This is called a user story. Programmers break this down further into a priority queue (that can be ranked by several factors) of requirements called the product backlog. These requirements are the set of actions needed to achieve the desired code from the existing code.

6.6: Describe a situation when a grep search fails. What would you do if this happened to you?

Grep searches generally fail when variable names and other programming implementations suffer from less-than-optimal naming conventions. There are 3 general cases of this.

1. No results in grep search.
2. Too many results.

3. Few results but results do not match targeted concept.

In each of these cases, reading the results and learning from them is key. It is often helpful to rephrase the search using different words (i.e., more or less specific depending on whether the problem corresponds to 1, 2, or 3). For example, let’s say there is some sort of payment software, and we are unable to find the word “payment” in a grep search. However, it’s possible that payment names are more specific and actually correspond to exact methods of payment. So, instead of repeatedly trying synonyms of payment, one could try words as in “credit,” “bank,” or “cash” instead. Furthermore, simple grep searches in a terminal are far from the best tools available for this kind of search. IDE’s such as VS Code have far better UI’s for the search and can provide better tooling for locking down a search. Additionally, we can find a state that we know is close to the problem and work backwards using other IDE tools such as “Find all references” for a given function, class, or variable. This can catch errors due to incorrect assumptions and accidental changes in global variables.