Lecture 16 - In Class Exercise

Goal: Understand how mutation testing works when applied directly to a program's source code.

1 Mutation Testing Over Source Code

Instructions: Work with your neighbors in groups of 2.

This exercise has two basic ideas. One idea is to simply understand the process of "killing" mutants, along with the unfortunate companion process of identifying "equivalent" mutants. The other idea is considering how certain mutation operators can be used to implement other coverage criteria. This second idea helps explain why mutation testing can be such a powerful coverage criterion.

Consider the program:

```
public final class GoodFastCheap {
    // boolean variables good, fast, and cheap other stuff omitted

    public boolean isSatisfactory() {
       if ((good && fast) || (good && cheap) || (fast && cheap)) {
          return true;
       }
       return false;
    }

    public boolean isSatisfactoryRefactored() {
       if (good && fast)  return true;
       if (good && cheap) return true;
       if (fast && cheap) return true;
       return false;
    }
}
```

Consider a mutation operator that replaces boolean variables with the constants "`true`" and "`false`".

1. How many mutants does this generate for:
   - `isSatisfactory()`
   - `isSatisfactoryRefactored()`
2. Some of these mutants are bound to be redundant. How many? Why?
3. Are any of these mutants "equivalent". What is going on here?
4. Consider a pair of corresponding mutants, one in each method. Will these mutants be killed by exactly the same tests? Is this a good thing?
5. For each mutant, find all the tests that kill that mutant.
6. Analyze weak vs. strong mutation for these mutants. Is there any difference? Does your answer depend on any assumptions?