SSW-555-6

Notes:
Midterm due October 23. It takes about 2 hours. One opportunity to do it.
Covers all the material including lectures and reading material.
We have a week off for Thanksgiving break.

Refactoring
Who, what, when, why, and how of refactoring

- If it ain't broke, don't fix it
    - This may not be the best strat for enhancing and maintaining software especially software with bad smells
- What is refactoring: changing the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior
    - Less complex, less risky, saving time
- **What is refactoring:** users should not notice that the code has been refactored. Refactoring should improve readability and reduce complexity. Refactoring is not about performance optimization, refactoring is critical for code that changes frequently.
- **Why refactor:** We start with good design and write good code to implement that design. But overtime the code changes to meet changes in the requirements. The design may not be updated to optimize those changes. Refactoring refreshes the design and the code to reflect the changes. Refactoring improves the design of software, easier to understand, helps you find bugs, helps you program faster, needed to pay off "technical debt"
- **Technical Debt:** additional development testing and maintenance effort
    - Caused by bad design, taking shortcuts, not implementing the right solution throughout the lifecycle. Quick hacks add technical debt; technical debt accumulates interest and makes changes even harder later on; refactoring helps to pay off technical debt as described earlier.
- **XP supports collective code ownership, anyone can refactor code**
- **Software development hats:** adding functionality (not changing existing code, adding code, adding tests) to the system or refactoring (not adding new functionality, not adding tests, small quick changes). Fowler suggests that developers should switch hats frequently but do not mix the modes.
- **Good test cases are critical:** research shows that new code is more likely to contain bugs than older; automated regression testing is critical
- **When to refactor:**
    - First time, just do it; second time, wince at duplication; third time refactor
    - When you add functionality; when you need to fix a bug; as you do a code review
- **Bad smells:** as you read code you realize that it needs restructuring; bad smells arent necessarily bugs, but instead is something that is not optimal and may suggest a bigger problem
    - Design flaws; opportunity for improvement
    - Experience guides you
    - Ie. duplicated code; unnecessary complexity

    - Bad smells in classes: large class; classes whose implementation depends on the implementation of another class; cyclomatic complexity/many different execution flows
    - Bad smells in methods: too many parameters; long method with too much code; bad method or variable names; returning too much data from the method
    - Duplicated code: extract method, pull up field, form template method, substitute algorithm
- **Refactoring: Extraction Method**
    - You have a code fragment that can be grouped together; turn the fragment into a method; replace the fragment with a call to the method
- **Bad smells: Long method**
    - Method is unreasonably long so you can use extraction method, replace temp with query, introduce parameter object
- **Soapbox Comments:** comments are not a replacement for clean, elegant code but judicious, concise comments are important and valuable for long term code maintenance
- Technical debt quadrant: reckless, prudent, deliberate, inadvertent
- **Refactoring workflows:** litter pickup, long term,
- **TDD refactoring workflow**: write tests, run the test, write some code, rerun the tests
- **Litter pickup:** reading code and discover some bad code
    - Less capable developer, didn't understand solution, replace bad code with an elegant solution
- **Comprehension refactoring:** reading code and gain insights into how the code works; refactor the code to help the next reader benefit
- **Preparatory refactoring:** need to add a new feature but it is easier to add the new feature after making changes to existing code.
- **Planned refactoring**: include time in the project schedule time for refactoring
- **Long term:** need a complex feature that cannot be added in a single sprint; new code to simplify change
- Eclipse support for refactoring
- Pycharm supports