CS 361
Software Engineering I

Improving Software Over Time

There is never enough time …

- The time to market has continually shortened in virtually every area of software
  - You could have taken a couple years in the 1980's
  - You could have taken a year in the 1990's
  - You could have taken a few months in the 2000's
  - Increasingly, entrepreneurs are beating each other to market by just weeks!!!!

There is never enough money…

- Established companies
  - Stockholders continually demand higher profits
  - Younger companies constantly try to disrupt your market ("change the rules of the game")
- Younger companies
  - There's no time to get substantial financial backing
  - You might lack deep market experience – is developing this product worth your own investment?

Key Strategy: Keep it Simple!

- You can improve the product later!
  1. Release a basic app
  2. Learn from users
  3. Improve the app
  4. Fund improvements with revenues

Talking about today…

- Adding features over time, and YAGNI
- Reorganizing code over time / refactoring
- Practical considerations of revenue & growth

Start simple and continually improve

- YAGNI ("You aren't gonna need it")
  - "A design which doesn't meet business needs is bad, no matter how pretty."
  - "If software is what you want to deliver then measure progress by how much you have working right now, not by how fancy the design is."

Rules of the Simplest Design

- The system (code and tests together) must communicate everything you want to communicate.
- The system must contain no duplicate code.
- The system should have the fewest possible classes.
- The system should have the fewest possible methods.

Identifying the simplest design may take effort

- Which is simpler, accessing a constructor directly or going through a Factory pattern?
  - Depends on the situation! If you get an idea like this while writing code, talk it over with your programming partner.
  - Agile says just use what's convenient – you can always fix up the code later.

Refactoring

- What it is: A program transformation that improves code's organization, not its function
- Examples:
  - Renaming variables or methods
  - Gathering duplicated code into a method
  - Splitting long methods into two methods
- When to do it: When code starts to "smell"

Bad Smell: Long Methods

- Sometimes you have a method that tries to do lots of different things.
  - Remember, code should have concerns!
  - This applies to methods, too.
  - Usually, >= 1 screen of code is too much.
- Most common way of refactoring:
  1. Split the method into smaller methods.
  2. Call each method.

Bad Smell: Duplicate Code

- Sometimes you have a few lines of code that appear in many different places
  - Often happens during copy-and-paste coding!
  - Usually, >= 3 duplicates are too many.
- Most common way of refactoring:
  1. Create a new method (and/or new class)
  2. Move the duplicated code into the new method
  3. Call the new method from each old place

Bad Smell: Large Classes

- Sometimes you have a class that tries to do too many things.
  - Usually, >= 7 member variables and/or >= 50 methods is too many.
- Most common way of refactoring:
  1. Pick the appropriate design pattern.
  2. Break the class into pieces, using the pattern.
  3. Fix up the code.

Bad Smell: Long Parameter Lists

- Sometimes your method has a parameter list as long as your arm.
  - How is somebody supposed to remember what parameters to pass into the method???
- Most common way of refactoring:
  1. Organize some/all of the parameters together into a hierarchy, using the composite pattern.
  2. Pass an instance of the composite, rather than a list of individual primitive values.
  3. Consider moving methods into

A Few More Refactorings

- Rename
- Delete unused method
- Move
- Introduce factory
- Change signature …

How to do refactoring right

- You absolutely have to have a working unit test suite.
  - No refactoring allowed until you're passing all your unit tests
- Then, when you get a whiff of a bad smell, talk with your pair programmer about it
- Try out the refactoring idea
- Run the unit test
- Iterate til you like the code & all unit tests pass

Refactoring early, refactor often

- If it takes you too long to refactor than you are not refactoring enough.
- Many tiny refactorings are easier than a single enormous refactorings.
- Each refactoring makes it easier to identify opportunities for further refactorings.

Be alert for opportunities

- Refactor when…
  - A new feature seems too difficult to code
  - You just created a new feature, and the code will be too hard for somebody else to understand
  - You "can't stand to look at your own code"
  - You can't stand to look at your teammate's code!!

REMEMBER: It's not *your* code, it's your *teams*.

Unit tests are your safety net

- When the test suite stops working (‘turns red’), it is unsafe to move forward

- “You might be driving, and then your copilot gets an idea to refactor, so you switch and he codes the changes then runs the test and voila! It works”