Refactoring

- During code review the code might pass all the tests and be understandable, but still not really be great and beautiful.
  - It is time to refactor the code!
- Refactoring is the act of rewriting code while preserving existing functionality.
  - Refactoring should make the code better in one or more aspects, such as
    - readability, understandability
    - maintainability
    - extendability
- How can we assure that the refactoring process preserves existing functionality?
  - Use the unit tests!
- What if there are no unit tests?
  - Not a uncommon situation..
  - Write unit tests (test data) using the existing code as the working base.
  - This is not easy as code not having unit tests is probably not easily testable, but it might be the best you have.
- Unit tests are a precondition for safe refactoring.
Refactoring

- The need for refactoring might arise as a result of changing requirements, i.e., you identify that you need to change the design to better accommodate the change.
  - Adding functionality and refactoring are separate activities and should not be done at the same time.
  - Refactor first, relying on the existing unit tests to preserve functionality.
  - Write new tests for the new/changed functionality.
  - Add/change functionality, using the newly written unit tests as guidance.
- Note, once again, how unit tests help us assure the quality and functionality of our code.
  - bad tests or bad test coverage will lead to unsafe refactoring - maintain your tests at all times; they are your automated guardians.
- Refactoring should be done regularly on code to keep it great and beautiful.
  - This is much like cleaning your home or tending your garden - it is a continuous activity.
  - Code and gardens grow slowly, but continuously so they don’t turn bad suddenly - you have to keep a constant watch.
Code Smells

- Code can be bad in many ways and it is good to have names for them when discussing, e.g, during a review.

- Martin Fowler and Kent Beck talks about *code smells* as a way of describing things that have gone bad. [Refactoring. Martin Fowler]

- The occurrence of code smells are also broken windows - fix them now or have them haunt you later.

- Code smells come in groups; some examples:
  - Unchecked growth
    - Long method
    - Long argument list
    - Large class
Code Smells

- Focus failure
  - Divergent change
    - a class has many reasons to change, i.e., different changes might affect one class
  - Shotgun surgery
    - a change that affects many classes
- A carefully crafted class expresses good abstraction and is focused on a number of closely related operations; this is the first half of *high cohesion, low coupling*.
- Data on the run
  - Data clumps
    - A group of data items that are passed around rather than being in a class.
  - Data class
    - A class containing only (public) instance variables that are manipulated externally.
Code Smells

- Code crowds
  - Speculative generality
    - Do we have code made for handling cases that we currently don’t need or use?
    - Are we too far ahead in the future?
  - Switch statements
    - High level code templates that indicates missing polymorphism.
    - Leads to high level DRY violations and shotgun surgery.
- Solutions:
  - Each smell can be fixed in different ways, depending on the goal at hand.
  - There are, in general, several different ways to fix a single smell.
  - Fowler has given names to different refactoring techniques that can be used to get rid of various smells.
  - Read the book to find out more!