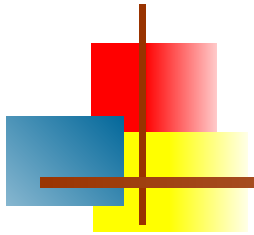
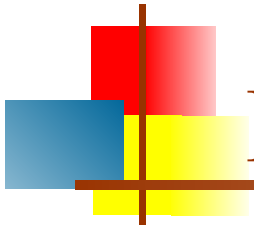
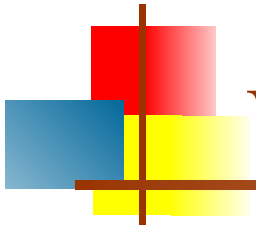


Refactoring



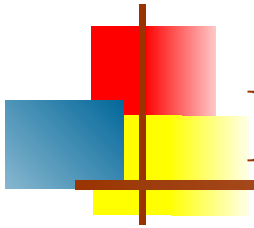
Refactoring

- Refactoring is:
  - restructuring (rearranging) code...
  - ...in a series of small, semantics-preserving transformations (i.e. the code keeps working)...
  - ...in order to make the code easier to maintain and modify
- Refactoring is *not* just any old restructuring
  - You need to keep the code working
  - You need small steps that preserve semantics
  - You need to have unit tests to prove the code works
- There are numerous well-known refactoring techniques
  - You should be at least somewhat familiar with these before inventing your own



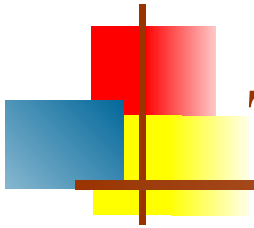
When to refactor

- You should refactor:
  - Any time that you see a better way to do things
    - "Better" means making the code easier to understand and to modify in the future
  - You can do so without breaking the code
    - Unit tests are essential for this
- You should *not* refactor:
  - Stable code (code that won't ever need to change)
  - Someone else's code
    - Unless you've inherited it (and now it's yours)



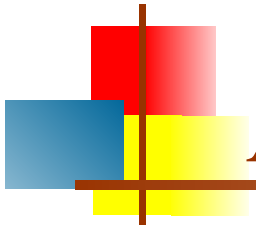
Design vs. coding

- "Design" is the process of determining, in detail, what the finished product will be and how it will be put together
- "Coding" is following the plan
- In traditional engineering (building bridges), design is perhaps 15% of the total effort
- In software engineering, design is 85-90% of the total effort
  - By comparison, coding is cheap



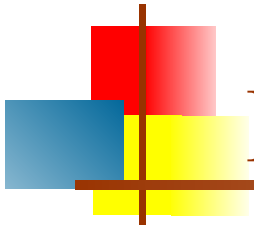
The refactoring environment

- Traditional software engineering is modeled after traditional engineering practices (= design first, then code)
- Assumptions:
  - The desired end product can be determined in advance
  - Workers of a given type (plumbers, electricians, etc.) are interchangeable
- "Agile" software engineering is based on different assumptions:
  - Requirements (and therefore design) change as users become acquainted with the software
  - Programmers are professionals with varying skills and knowledge
  - Programmers are in the best position for making design decisions
- Refactoring is fundamental to agile programming
  - Refactoring is sometimes necessary in a traditional process, when the design is found to be flawed



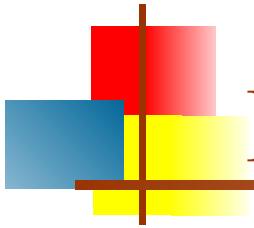
A personal view

- In my opinion,
  - Design, because it is a lot more creative than simple coding, is also a lot more fun
    - Admittedly, “more fun” is not necessarily “better”
    - ...but it does help you retain good programmers
  - Most small to medium-sized projects could benefit from an agile programming approach
    - We don’t yet know about large projects
  - Most programming methodologies attempt to turn everyone into a mediocre programmer
    - Sadly, this is probably an improvement in general
    - These methodologies work less well when you have some very good programmers



Back to refactoring

- When should you refactor?
  - *Any* time you find that you can improve the design of existing code
  - You detect a “bad smell” (an indication that something is wrong) in the code
- When *can* you refactor?
  - You should be in a supportive environment (agile programming team, or doing your own work)
  - You should have an adequate set of unit tests

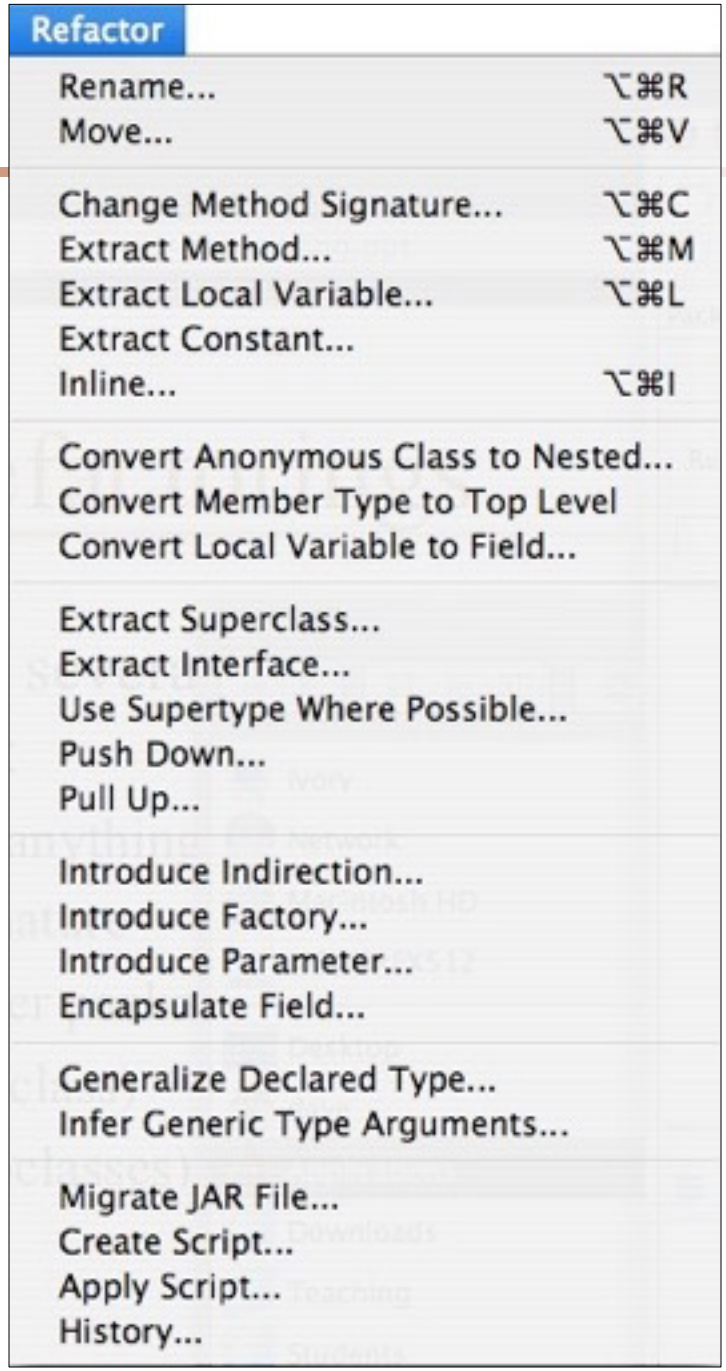


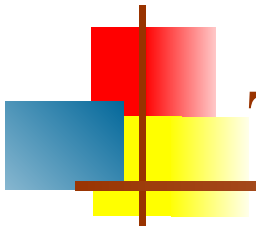
Bad Smell Examples

- You should refactor any time you detect a “bad smell” in the code
- Examples of bad smells include:
  - Duplicate Code
  - Long Methods
  - Large Classes
  - Long Parameter Lists
  - Multi location code changes
  - Feature Envy
  - Data Clumps
  - Primitive Obsession
- Good reading: https://sourcemaking.com/refactoring/smells

Eclipse

- The concept of refactoring applies to any programming language
- Natural languages, too!
- Eclipse (and some other IDEs) provide significant support for refactoring in Java





The End