Evolving Software

- Problem
  - The requirements of real software often change in ways that cannot be handled by the current design
  - Moreover, trying to anticipate changes in the initial implementation can be difficult and costly

- Solution
  - Redesign as requirements change
  - Refactor code to accommodate new design
Why not redesign/refactor?

- Conventional wisdom would discourage modifying a design
  - You might break something in the code
  - You have to update the documentation
  - Both expensive
- But, there are longer term concerns: sticking with an inappropriate design
  - Makes the code harder to change
  - Makes the code harder to understand and maintain
  - Very expensive in the long run

Refactoring Philosophy

- Make all changes small and methodical
  - Follow mechanical patterns (which could be automated in some cases) called **refactorings**, which are **semantics-preserving**
- Retest the system after each change
  - By rerunning all of your unit tests
  - If something breaks, it’s easy to know what you changed to cause the failure.
Principles of Refactoring

• In general, each refactoring aims to
  – decompose large objects into smaller ones
  – distribute responsibility

• Like design patterns
  – Adds composition and delegation (read: indirection)
  – In some sense, refactorings are ways of applying design patterns to existing code

Two Hats

• Refactoring Hat
  – You are updating the design of your code, but not changing what it does. You can thus rerun existing tests to make sure the change works.

• Bug-fixing/Feature-adding Hat
  – You are modifying the functionality of the code.

• May switch hats frequently
  – But know when you are using which hat, to be sure that you are reaching your end goal.
Some Refactorings

- **Rename Method**
  - Give a method a more meaningful name; update callers
- **Extract Method**
  - Pull out code in one method into a separate method
- **Replace Temp with Query**
  - Remove code that assigns a method call to a temporary, and replace references to that temporary with the call
- **Move Method**
  - Move method from one class to another

Some Refactorings

- **Extract Class**
  - Break a class that does many things into smaller classes
- **Inline Class**
  - A class isn’t doing very much, so inline its features into its users (reverse of Extract Class)
- **Replace Type Code with State/Strategy**
  - Basically replace state constants with typesafe Enum
- **Replace Conditional With Polymorphism**
  - Don’t switch on a type code, rather do the work in the type object, and use delegation (second half of State)
Refactoring with Tools

- Many refactorings can be performed automatically
- This reduces the possibility of making a silly mistake
- Eclipse provides support for refactoring in Java
  – http://www.eclipse.org

Spying Code to Refactor: Bad Smells

- What code needs to be refactored?
  – Bad code exhibits certain characteristics that can be addressed with refactoring; these are called “smells.”
- Different smells suggest different refactorings
Feature Envy

- A method seems more interested in a class other than the one it is actually in
  - e.g., invoking lots of get methods

- Can use **Move Method** and **Extract Method**
  - Did this by moving extracting getCharge() out of Customer.statement(), moving it into Rental

Long Method

- A method is too long. Long methods are harder to understand than lots of short ones.

- Can decompose with **Extract Method**
  - Pulled getCharge() out of statement()

- Also **Replace Temp with Query**
  - Replaced thisAmount with calls to getCharge()

- And others …
Switch Statements

- Usually not necessary in delegation-based OO programming

- **Replace Type Code with State/Strategy**
  - Define a class hierarchy, a subclass for each type code

- **Replace Conditional with Polymorphism**
  - Call method on state object to perform the check; switching is based on dynamic dispatch

- Did this for Movie pricing

Duplicated Code

- The same expression used in different places in the same class
  - Use Extract Method to pull it out into a method

- The same expression in two subclasses sharing the same superclass
  - Extract Method in each, then PullUp method into parent

- Duplicated code in two unrelated classes
  - Extract Class to create a fresh class, or make one class use the other
Long Parameter List

- Lots of parameters occlude understanding
- Replace parameters to method by calls in method to objects already known
  - Replace Parameter with Method
- Group parameters into a conceptually sound container object
  - Introduce Parameter Object

Divergent Change

- One class is commonly changed in different ways for different reasons
  - To add a new database I change these three methods
  - To add a new financial currency I change these four
- Identify everything that relates to a particular cause, express a new class for that variation
  - Extract Class
Shotgun Surgery

- Every time I make change X, I have to make lots of little changes to different classes
  - Opposite of Divergent change

- Goal: make common changes one-to-one with classes
  - Don’t know this until you’ve worked with the system for a while.

- Other smells too …

When should you refactor?

- Adding function
  - You wish to add a feature to your system and you realize it would work better with a modified design.

- Fixing a bug
  - Finding a bug can reveal flaws in the design that led to it. Thus, you change the design to prevent other bugs.

- During code review
  - This is a fruitful time to adjust design, because you are thoughtfully considering the current design’s value.
When to refactor: An analogy

- Unfinished refactoring is like going into debt.
- Debt is fine as long as you can meet the interest payments (extra maintenance costs).
- If there is too much debt, you will be overwhelmed.
  - [Ward Cunningham]

Obstacles to Refactoring

Complexity
  - Changing design is hard, understanding code is hard
Possibility to introduce errors
  - Mitigated by testing
    - Clean first Then add new functionality
Cultural Issues
  - Producing negative lines of code, what an idea!
    - “We pay you to add new features, not to improve the code!”
If it ain’t broke, don’t fix it
  - “We do not have a problem, this is our software!”
More information

• Textbook: Refactoring by M. Fowler
• Catalog of refatorings:
• Refactoring to patterns
  – http://industriallogic.com/xp/refactoring/