Refactoring

EECS 2311 - Software Development Project

Click to edit Master text styles
Second level
Third level
Fifth level

Wednesday, March 27, 2019

YORK UNIVERSITÉ UNIVERSITY

Software design

- Building software systems whose design is flexible, maintainable, and understandable requires experience
- You will discuss design guidelines in detail in EECS 3311, including design patterns, i.e. known solutions to common design problems
- In the meantime, we need to avoid bad design
- Known characteristics of bad design that should be avoided are referred to as “code smells”
- Let’s discuss some examples…

Duplicated code

- Same expression in two methods of the same class
  - Use **Extract Method** refactoring
- Same expression in two methods of sibling classes
  - Use **Extract Method** and **Pull Up Method**
  - If code is similar but not same, consider **Form Template Method**
- Duplicated code in unrelated classes
  - May need to **Extract Class** or otherwise eliminate one of the versions

Long Method

- The longer a method is, the more difficult it is to understand
- Be aggressive about decomposing methods
- Use *good naming*
- 90% of the time, just **Extract Method**
- What to extract? Look for comments explaining a piece of code

Large Class

- A class that tries to do too much often has too many instance variables
- Prime breeding ground for duplicated code
- **Extract Class**
- **Extract SubClass** for some of the
- **Extract Interface** variables

Long parameter list

- Hard to understand, requires frequent changes
- In OO systems, much fewer parameters are required
- Shorten parameter lists with
  - **Replace Parameter with Method**
  - **Preserve Whole Object**
  - **Introduce Parameter Object**

Divergent Change

- A class is commonly changed in different ways for different reasons
- “I will have to change these three methods every time I get a new database; I have to change these four methods every time there is a new financial instrument”
- **Extract Class** to alleviate this problem

Shotgun Surgery

- Every time you make a kind of change, you have to make a lot of little changes
- Easy to miss an important change
- **Move Method** and **Move Field** to put all changes into a single class
- You might even use **Inline Class**

Feature Envy

- A method seems more interested in a class other than the one it is in
  - Invokes many getter methods from another class
- **Move Method** to where it wants to be
- Strategy and Visitor design patterns result in code that has feature envy
  - Acceptable since this way we fight divergent change
- Often there are tradeoffs in fighting code smells

Data Clumps

- Bunches of data that hang around together ought to be made into their own object (**Extract Class**)
- You can then slim parameter lists down with **Introduce Parameter Object** or **Preserve Whole Object**

Switch statements

- Switch statements are often duplicated
- If you add a new clause, you need to find all related switch statements
- Polymorphism can solve this problem
- If switching on type code
  - **Extract Method**
  - **Move Method**
  - **Replace Type Code with Subclasses**
  - **Replace Conditional with Polymorphism**

Parallel Inheritance Hierarchies

- Special case of shotgun surgery
- Every time you make a subclass of one class, you also have to make a subclass of another
- Eliminate duplication by having instances of one hierarchy refer to instances of the other

Lazy class

- If a class is not doing enough to justify maintaining it, it should be removed
- Refactoring often results in lazy classes that can be removed with
  - **Collapse Hierarchy**
  - **Inline Class**

Speculative Generality

- Machinery added for future use that never gets implemented
- Makes system much harder to understand
- Often identified because test cases are the only users of a method of a class
- Remove unnecessary machinery with
  - **Inline Class / Collapse Hierarchy**
  - **Remove Parameter / Rename Method**

Temporary Field

- Fields that are not used (or used only in certain circumstances)
- Very difficult to determine their usefulness
- Maybe they are only used as global variables to avoid passing them around as parameters
- **Extract Class** for temporary fields

Refused Bequest

- Subclasses do not want or need methods or data of their parents
- **Push Down Method** and **Push Down Field** to move unwanted methods to siblings
- If the subclass does not want to support the interface of the superclass, **Replace Inheritance with Delegation**

Comments

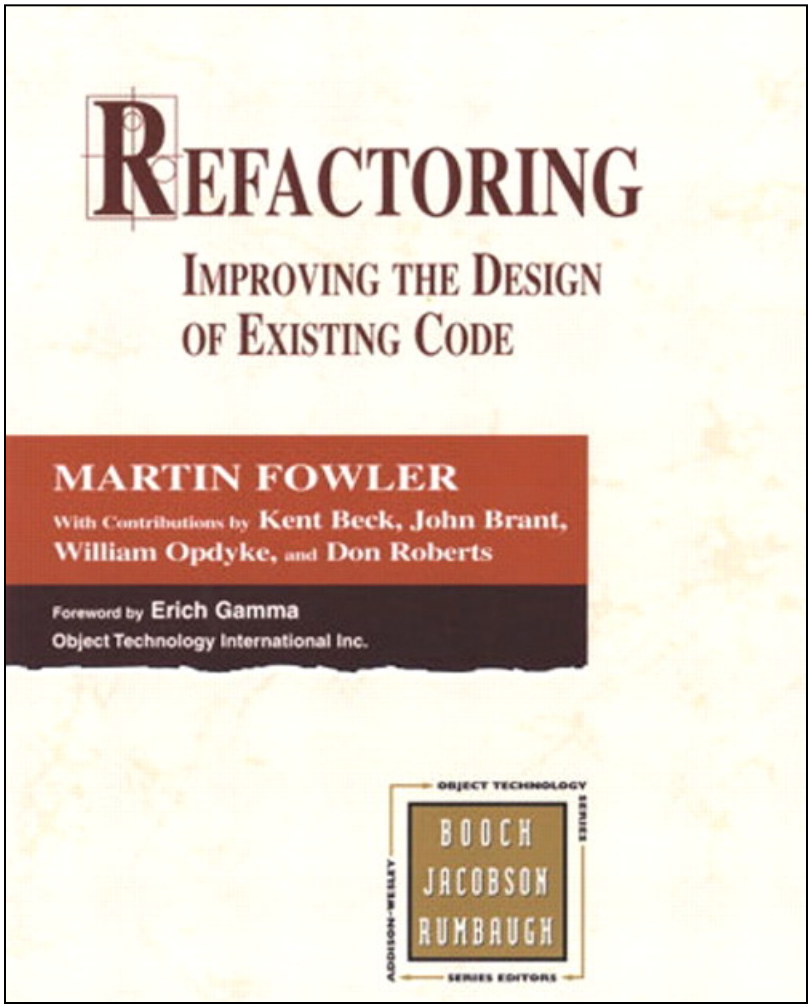
- Comments are of course a sweet smell, but they should not be used as deodorant
- When you feel the need to write a comment, first try to refactor the code so that any comment becomes superfluous
- Can also use
  - **Extract Method**
  - **Rename Method**
  - **Introduce Assertion**

More code smells

- Primitive obsession
- Message Chains
- Middle man
- Inappropriate intimacy
- Alternative classes with different interfaces
- Incomplete library class
- Data class

Reading

- Read all about code smells in Martin Fowler's refactoring book



Class exercise

- Let's look at the refactoring package in the course github
- The purpose of the system is to implement a system that keeps track of customers that rent movies
- For each customer, the system can produce a statement with each customer's charges and frequent renter points
- Your job:
  - Study the test case to see how the classes are used
  - Suggest ways to improve the system's design
  - Hint: Look for some of the code smells in this slide set

Lab Task

- Choose one application of refactoring to demonstrate
- You can demonstrate by performing the refactoring on the fly or by comparing the revisions before and after the refactoring
- You must provide a justification for the refactoring