Refactoring & Code Smells

CSCI 4448/5448: Object-Oriented Analysis & Design

Lecture 35

Acknowledgement & Materials Copyright

- I'd like to start by acknowledging Dr. Ken Anderson
- Ken is a Professor and the Chair of the Department of Computer Science
- Ken taught OOAD on several occasions, and has graciously allowed me to use his copyrighted material for this instance of the class
- Although I will modify the materials to update and personalize this class, the original materials this class is based on are all copyrighted © Kenneth M. Anderson; the materials are used with his consent; and this use in no way challenges his copyright

Next Steps

- New discussion topic is up, please post your responses to topics for Participation grading!
- We'll review Project 7 in class tomorrow
- Due Wednesday 7/14
  - Project 6 with demos – sign up sheet for demo slots at
    - https://docs.google.com/document/d/1Rnga64KMEVP3--sCLLhGuJXNQeODPFmsS7hT4HeJdQQ/edit?usp=sharing
  - Quiz 5 – active now
- Due Monday 7/19
  - Graduate Pecha Kucha – presented in class by teams on 7/20 and 7/21 – sign up sheet at
    - https://docs.google.com/document/d/193mP7K5zSR6FiYGgS0QOAYOJgsesI1HXZL1Aas3WijI/edit?usp=sharing
  - Bonus points for attendance during Pecha Kucha sessions on 7/20, 7/21
- Due Wednesday 7/21
  - Project 7 with recorded demo
  - Graduate Final Research Presentation – turned in for review, not presented in person
  - Quiz 6
- Final Exam – optional, available on Friday 7/23 to Saturday 7/24
- Upcoming lectures/activities: ORMs, Refactoring, Dependency Injection…
- How to find Bruce:
  - Post the question on the class Piazza site (often, if you have a question, others might as well)
  - Find me during office hours on Zoom at https://cuboulder.zoom.us/j/3844137608 on Tuesday 4-5 PM, Wednesday 10-11 AM, Thursday 3-4 PM
  - Make an appointment to see me via Zoom at https://brucem.appointlet.com
  - Email me at bruce.r.montgomery@colorado.edu

Goals of the Lecture

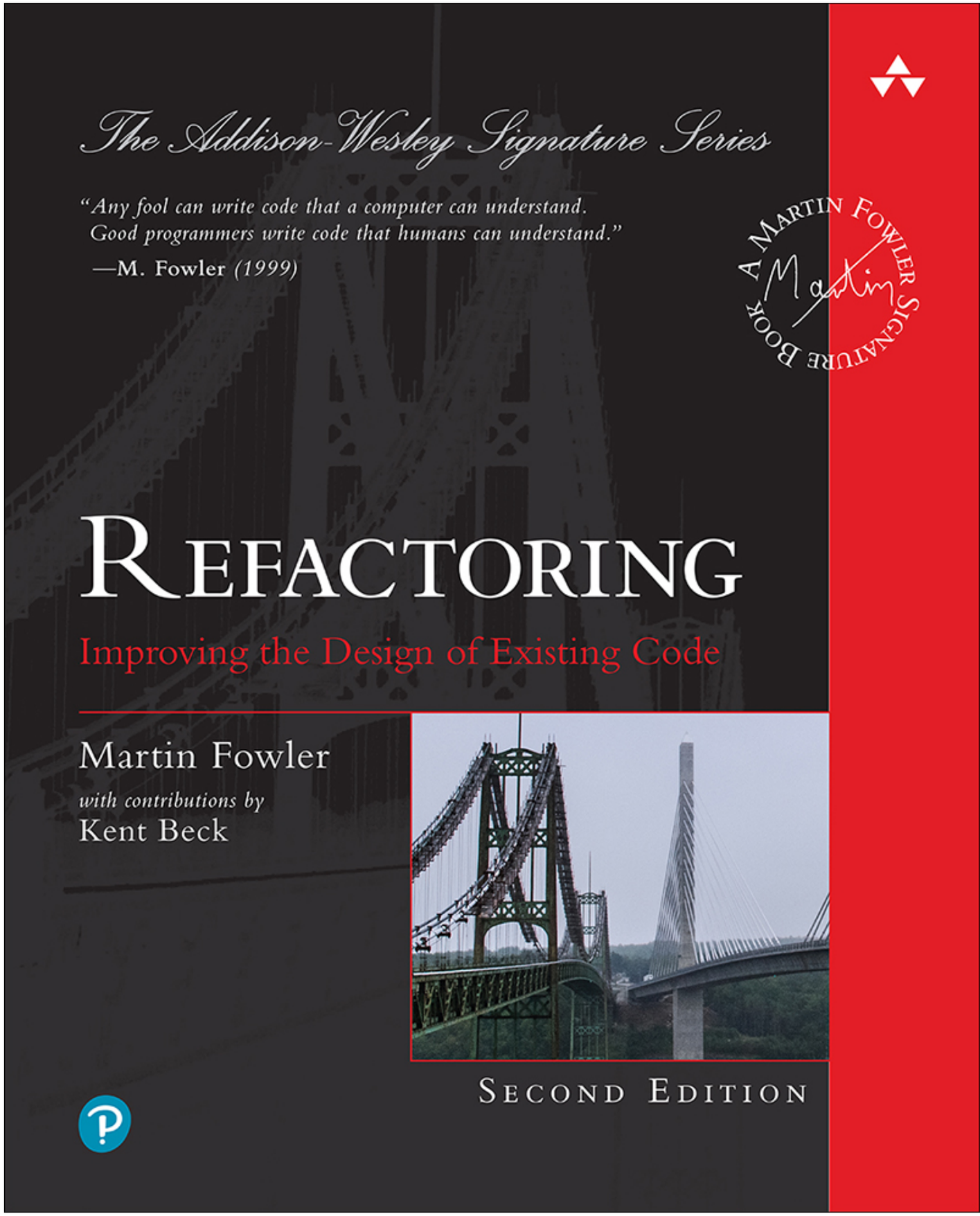
- Introduce Refactoring and Fowler's Refactoring Book, a key reference
- Consider when to Refactor
- Look at typical Code Smells
- Consider different categories of Refactoring elements

Refactoring Origins

- 1984: “Factoring” is described in Brodie’s “Thinking Forth” as “organizing code into useful fragments” which “occurs during detailed design and implementation”.
- 1990: Bill Opdyke coins the term “refactoring” in an ACM SIGPLAN paper with Ralph Johnson, “Refactoring: An aid in designing application frameworks and evolving object-oriented systems”
- 1992: A comprehensive description of “refactoring” is presented in Opdyke’s “Refactoring object-oriented frameworks”
- 1999: The practice of “refactoring”, incorporated a few years earlier into Extreme Programming, is popularized by Martin Fowler’s book
- 2001: Refactoring “crosses the Rubicon“, an expression of Martin Fowler describing the wide availability of automated aids to refactoring in IDEs for the language Java (https://martinfowler.com/articles/refactoringRubicon.html)
- From https://www.agilealliance.org/glossary/refactoring

Source for today's lecture

- Refactoring: Improve the Design of Existing Code, Martin Fowler (with Kent Beck), Second edition 2018, Addison-Wesley
- First edition in 2000
- Fowler also wrote UML Distilled, Patterns of Enterprise Application Architecture, and many other famous CS books
- https://refactoring.com/ - there is a web edition of the book, it's also on O'Reilly Safari



Refactoring, the book

- His own description:
  - Refactoring is a controlled technique for improving the design of an existing code base
  - Its essence is applying a series of small behavior-preserving transformations, each of which is "too small to be worth doing“
  - However the cumulative effect of each of these transformations is quite significant
  - By doing them in small steps you reduce the risk of introducing errors
  - You also avoid having the system broken while you are carrying out the restructuring - which allows you to gradually refactor a system over an extended period of time
- Focus on refactoring, code smells, the role of testing
- Includes details on some 70 refactorings, what they are, why you should do them, how to do them safely

Key points from the introduction

- When you have to add a feature to a program but the code is not structured in a convenient way, first refactor the program to make it easy to add the feature, then add the feature
- Before you start refactoring, make sure you have a solid suite of tests; these tests must be self-checking
- Refactoring changes the programs in small steps, so if you make a mistake, it is easy to find where the bug is.
- “Any fool can write code that a computer can understand, good programmers write code that humans can understand.”
- When programming, follow the camping rule: always leave the code base healthier than when you found it
- The true test of good code is how easy it is to change it

Principles in Refactoring

- Refactoring…
  - Improves the design of software
  - Makes the software easier to understand
  - Helps find bug
  - Helps make programming faster
  - Make it easier to add a feature
  - Litter-pickup
- Planned vs. Opportunistic Refactoring
- When to Refactor?
  - Rule of Three – third time you do something similar, refactor
  - If code is a mess, but doesn’t require modification, leave it
  - If code needs to be rewritten not refactored (judgement call)

Problems with Refactoring

- Management perceptions
- Slowing down new features
  - Although, the whole purpose of refactoring is to make us program faster, producing more value with less effort
- Published Interfaces (especially in use by customers)
- Branches
  - Not a problem if small scope, integrated frequently
- Testing
- Legacy Code
  - See Working Effectively with Legacy Code (Feathers)
    - Summary – get the code under test
- Databases and Migration Scripts
- Code Ownership
  - Before you refactor someone else's code, (if you can) talk to them first
  - <Insert cautionary tale(s)>

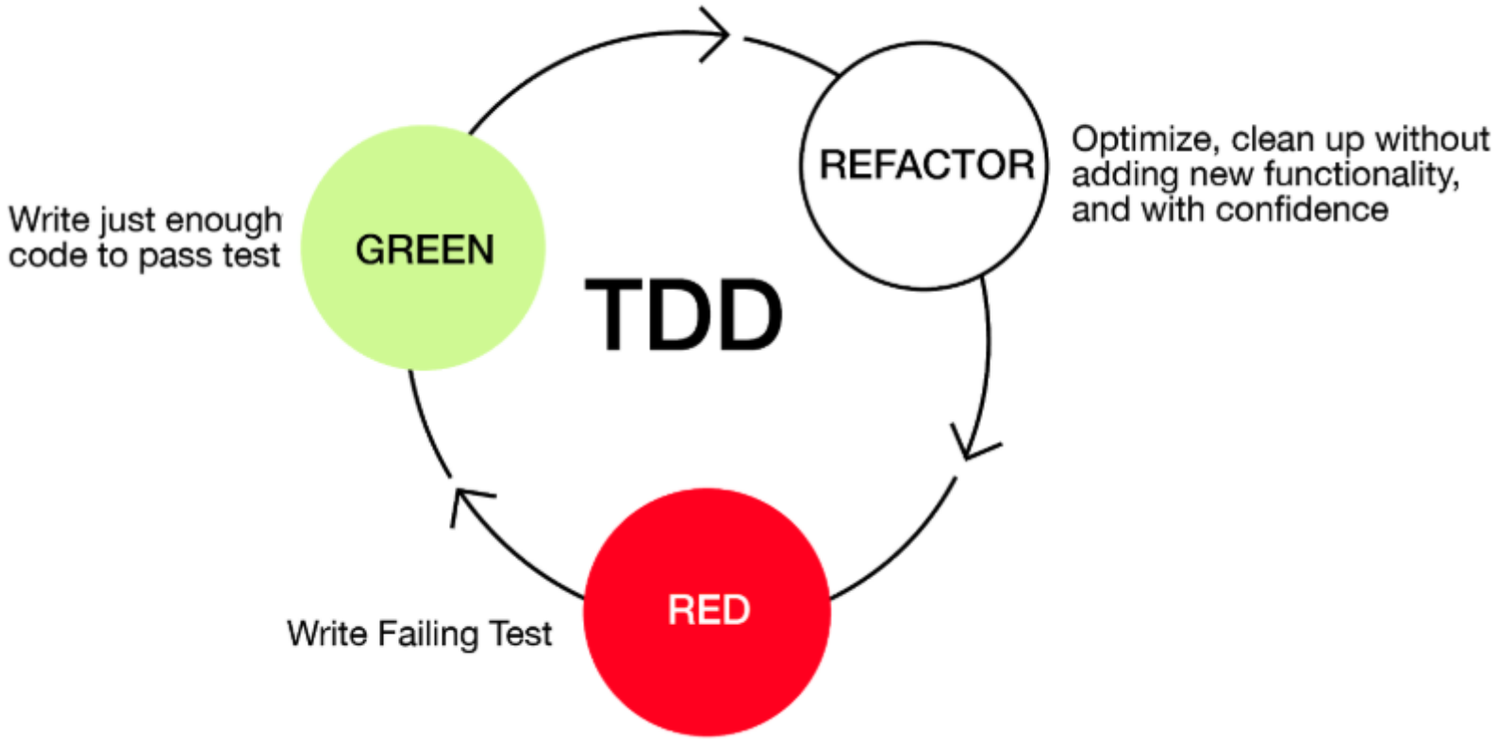
Bad Smells in Code

Fowler developed this initial list with Kent Beck (Extreme Programming)

- Mysterious Name
- Duplicated Code
- Long Function
- Long Parameter List
- Global Data
- Mutable Data
- Divergent Change (one module changed in different ways for different reasons)
- Shotgun Surgery (lots of edits to lots of classes for a single change)
- Feature Envy (using someone else's methods a lot)
- Refused Bequest (poor method inheritance)
- Comments (as deodorant)
- Data Clumps
- Primitive Obsession
- Repeated Switches
- Loops
- Lazy Element (largely unused)
- Speculative Generality (YAGNI)
- Temporary Field
- Message Chains
- Middle Man
- Insider Trading
- Large Class
- Alternative Class with Different Interfaces
- Data Class

- Another collection (divided between in classes and between classes) at: https://blog.codinghorror.com/code-smells/

Building Tests



- Test-code-refactor cycle (Test-Driven Development)
- Goal: Make sure all tests are fully automatic and that they check their own results
- A suite of tests is a powerful bug detector that decreases time to find bugs
- Always make sure a test will fail when it should
- Run tests frequently; run those exercising the code you're working on at least every few minutes; run all tests at least daily
- It is better to write and run incomplete tests than not to run complete tests
- Think of the boundary conditions under which things might go wrong and concentrate your tests there
- Don't let the fear that testing can't catch all bugs stop you from writing tests that catch most bugs
- When you get a bug report, start by writing a unit test that exposes the bug (regression tests)

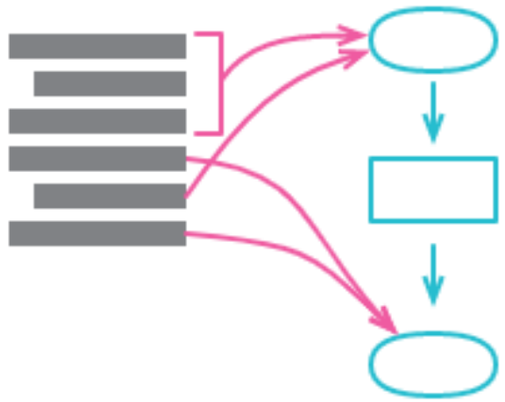
First Set of Most Common Refactorings

Each example in the book (and in part, on the refactoring.com website) provides

- method name
- image representing action
- simple before/after code
- motivation
- mechanics
- example

- Extract Function – move a code fragment into a function named after its purpose
- Inline Function – a group of badly factored functions are moved into one function
- Extract Variable – replace an expression with a new variable
- Inline Variable – drop a variable and use its RHS directly
- Change Function Declaration – names and parameters should be clear
- Encapsulate Variable – restrict visibility and access of variables
- Rename Variable – name should be meaningful
- Introduce Parameter Object – replace multiple function parameters with a single object
- Combine Functions Into Class – group like methods
- Combine Functions Into Transform – produces all derived values in one object
- Split Phase – split functionality into logical modules

Split Phase



- From refactoring.com; also a link there to various web editions of the book
- https://refactoring.com/catalog/splitPhase.html

```
const orderData = orderString.split(/\s+/);
const productPrice = priceList[orderData[0].split("-")[1]];
const orderPrice = parseInt(orderData[1]) * productPrice;
```

```
const orderRecord = parseOrder(order);
const orderPrice = price(orderRecord, priceList);

function parseOrder(aString) {
  const values =  aString.split(/\s+/);
  return ({
    productID: values[0].split("-")[1],
    quantity: parseInt(values[1]),
  });
}
function price(order, priceList) {
  return order.quantity * priceList[order.productID];
}
```

Encapsulation Refactorings

- Encapsulate Record – replace a record with a data class
- Encapsulate Collection – combine collection data with CRUD methods
- Replace Primitive With Object – move simple data items to classes
- Replace Temp With Query – extract assignment to a variable into a method
- Extract Class – single responsibility for data/methods
- Inline Class – refactoring an unneeded class into other code
- Hide Delegate – move delegate access away from clients
- Remove Middle Man – take out an intermediate class
- Substitute Algorithm – provide simpler algorithm for existing function

Moving Features Refactoring

- Move Method – move a method elsewhere
- Move Field – move data elsewhere
- Move Statements Into Function – move repeating code to a function
- Move Statements to Callers – move function code out to where used
- Replace Inline Code with Function Call – as described
- Slide Statements – rearrange code for clarity
- Split Loop – make one loop two if really doing two different things
- Replace Loop With Pipeline – allow methods to perform tasks
- Remove Dead Code – as described

Organizing Data Refactoring

- Split Variable – don’t use one variable for two things
- Rename Field – more descriptive variable names
- Replace Derived Variable with Query – remove variables that can be calculated
- Change Reference to Value – use immutable data that can easily be shared
- Change Value to Reference – use mutable data that should come from a source

Simplifying Conditionals Refactoring

- Decompose Conditional – make functions for complex clauses
- Consolidate Conditional – combine conditional checks
- Replace Nested Conditional with Guard Clauses – restructure if then else complexity
- Replace Conditional with Polymorphism – as described
- Introduce Special Case – create object for special-case element
- Introduce Assertion – add conditional that should always be true

Refactoring APIs

- Separate Query from Modifier – data request and action should be separate functions
- Parameterize Function – add a parameter that is needed
- Remove Flag Argument – use specific methods for specific parameters
- Preserve Whole Object – pass object as parameter rather than fields
- Replace Parameter With Query – don’t pass mix of object and fields
- Replace Query with Parameter – pass only needed data
- Remove Setting Method – for data that can’t be set
- Replace Constructor with Factory – as described
- Replace Function with Command – use command objects
- Replace Command with Function – provide simple function as needed

Refactoring Inheritance

- Pull Up Method – bring method up from subclass
- Pull Up Field – bring data up from subclass
- Pull Up Constructor – bring constructor up to parent class
- Push Down Method – move method down to subclass
- Push Down Field – move data down to subclass
- Replace Type Code with Subclasses – use polymorphism
- Remove Subclass – delete unneeded subclass
- Extract Superclass – make parent for similar classes
- Collapse Hierarchy – pull subclass into parent class
- Replace Subclass with Delegate – prefer composition/delegation over inheritance
- Replace Superclass with Delegate – similar to previous

Refactoring: Does it work?

- Refactoring is a popular practice, and common sense would say cleaner code has myriad benefits, but…
- Proving the benefits (much like using OOAD) shows a gap between research and common practice
- Studying the Effect of Refactorings: a Complexity Metrics Perspective, a 2010 study, finds surprisingly little correlation between refactoring and a decrease in cyclomatic complexity
- This may be impacted by mixing refactoring with addition of functionality, bug fixes, or other code changes…

- From https://www.agilealliance.org/glossary/refactoring