Refactoring:

Techniques for Software Evolution

Dennis Mancl

dmancl@acm.org

MSWX

MSWX ◊ Mancl ◊
Software ◊ Experts ◊ http://manclswx.com

Refactoring

In this talk, we will discuss software development and software engineering

- Software development = the nuts and bolts of building software products
- Software engineering = developing applications at a reasonable cost with good quality

Refactoring is a technique for making controlled modifications to existing code:

Refactoring is the process of making improvements to the structure of code without changing its functionality.

We are going to edit existing code, and when we are done, it will still work!

"Without changing functionality" -- is a pretty extreme definition... most people will alternate doing a little bit of "refactoring" followed by a little bit of "changing functionality"

Thinking about the changes we make to software

- Suppose we write tax preparation software
  - Tax tables change every year
  - The rules on tax deductions; new data about the user; formulas for computing tax return fields
  - Changes to the filing process = the process for printing forms; the process for submitting forms electronically
- Suppose we write software to edit and upload videos
  - Our software will need to interface with different video recording apps
  - Each social media site has different rules about format and content

Customers want "new things" every year (or every week!)

Thinking about the changes we make to software

- I used to work on telecom software… every “product release” has a combination of:
  - Old features that *must* work
  - New features that the customers are clamoring for…

Managers and customers say… don’t break the old stuff, but we want the new stuff to be great!

Why software changes

- Wait a minute… Software doesn’t wear out! Why do we ever need to change it?
  - Bugs / errors / defects
  - Adding a few new features… new requirements
  - Porting to work with new devices / new version of operating system
- 3 types of changes
  - corrective
  - perfective
  - adaptive

[B. P. Lientz and E. B. Swanson, *Software Maintenance Management*, 1980]

A software system is always changing and improving, unless it is dead and buried!

We want to have safe ways to make the changes. And we need an organized process to do it!

Refactoring is one of our tools…

Who refactors? There are 4 major “camps”

- camp 1 - software evolution
  - selective legacy code improvement (== make it easier to add new code)
- camp 2 - agile development
  - rapid code-test-refactor cycle
- camp 3 - patterns community
  - introduce design patterns into legacy designs
- camp 4 - legacy code analysis
  - refactor to understand legacy code

Each camp has different “goals”
**But* most of the refactoring techniques are the same!*

1. Software evolution camp

“Maintenance programmers”

- In many cases, *not* the original product developers
- But these are some of the smartest people in the software business!

The economics of legacy software:

- Legacy code is old software, but it is still making you $$$
- Customers rely on it -- in fact, they might rely on some of the bugs
- Legacy code = the developers did a lot of work to come up with a good design, and it would take a lot of effort to design new code from scratch...

So… we learn how to repair, adapt, and extend the code

Refactoring process

The refactoring process involves:

- making <u>small changes</u> that improve the code's structure
- but the changes do not change the functionality in the code
- and it should be easy to test that the code still works

We are "cleaning up the code"

- Making it easier for us (and other people) to work with
- Easier to fix errors and add new functionality

What we do:

- Renaming, minor restructuring, splitting up big functions, adding new "internal interfaces"
- And… we keep testing constantly
- (More soon about unit tests)

Maintenance programmers have been doing this kind of work forever…

But the first "refactoring tools" started to appear in 1990.

William F. Opdyke and Ralph E. Johnson (September 1990). "Refactoring: An Aid in Designing Application Frameworks and Evolving Object-Oriented Systems."

Can we make changes safely?

Refactoring process

The refactoring process – for legacy code improvement – is “focused”

- Limited refactoring … only in the parts of the code base where you plan to make changes

We “clean up” our mistakes

- In our first version, our database uses customer information in many parts of the system code
- But in our second version, we modified the database structure…
- Maybe we stored every record by customer name, but then we added a customer id (just in case two customers have similar names)
- Throughout the application code… we can see some code that performs “search by name” and code that performs “search by id”

- We might “refactor” our system – to reduce the duplicated code…
  - Create a “search by customer name” function and a “search by customer id” function [increase the clarity of the code, it becomes easier to change…]

“Mistake” = lots of duplicated code throughout the application!

**and* lack of clarity in the code*

In the refactored code, it is easy to see when we do “search by name” or “search by id”

2. Agile development camp

Agile development =

Small team development, rapid iteration, building very small features over a period of a few weeks

Building code in short iterations means "lots of rewriting"

- A "big feature" might be written in multiple stages (across multiple iterations)
- The code base is constantly changing – but everyone is compiling and testing the code all the time
- "Continuous Integration" – build and test every day (or every hour)

Refactoring is supported by unit tests – small automated tests:

- The existence of a large set of automated unit tests is a necessary part of the agile process

Refactoring to support software change -> refactoring to support "rapid change"

Essential:

- *All code is in a change control system (Subversion, Git, Mercurial)*
- *Rapid automated build*
- *Automated test suite*

Unit tests define the "low-level" functionality = all of the code-level decisions

In Agile, unit tests and refactoring go hand-in-hand

Automated unit tests are evaluating the software after each modification…

Unit tests are written by developers…

- Tests help the developers: the can “code in confidence” - because they can run the unit tests quickly and find coding blunders early
- The process of Test Driven Development (TDD) is the most extreme way to ensure that there are always unit tests

Goal: Good code quality at all times!

- Since everything is being developed quickly, developers need discipline to avoid sloppy code
- And refactoring is part of that discipline
- Refactoring work is part of every iteration

Common Unit Test Tools/Frameworks:

- *Java ⇒* ***JUnit***
- *Javascript ⇒* ***Jest*** *or* ***Mocha***
- *Python ⇒* ***PyUnit*** *or* ***pytest***
- *C++ ⇒* ***Boost.Test***

Lots of good online tutorials for these unit test frameworks

Code smells

Agile developers talk about "code smells"

- Every code smell is a place in the code where the original developer wasn't thinking about keeping the code "clean"

➢ Focus your refactoring based on "things that smell bad"

➢ The refactoring process is driven by the "intuition" of the developers… only you can answer these questions:

Instead of asking "Where are we making lots of changes?" – Agile developers consider Code Smells

When is a function too big?

How much duplicated code is OK?

When does a function have too many arguments?

Which comments are too confusing?

When is a Boolean conditional expression too complex?

Where do I need more unit tests?

It's up to you to decide!

Don't refactor something just because the "experts" say that you *must* follow their coding rules…

Code smells

Look within your functions. Have you seen these problems?

- Too many parameters
  - hard to read, complex to test
  - need to rethink the function - split it up?
- Long function
- Excessively long identifiers
  - difficult to read
- Excessively short identifiers
  - impossible to understand
  - make the name of a variable reflect its function in the code
- Excessive comments
- Excessively long line of code
  - difficult to read
- Dead code

Some lists of code smells…

http://wiki.c2.com/?CodeSmell

https://en.wikipedia.org/wiki/Code_smell

https://blog.codinghorror.com/code-smells

Code smells

Look across your application’s functions. Do these problems stand out?

- Mysterious Name
  - names of functions and classes be clear
  - why? … to help developers understand its purpose (“what does this function do?”)
- Duplicated code
  - it makes the code very brittle
  - the code might break if the multiple copies are modified inconsistently
  - “Extract function” or “Extract class” to fix it
- Uncontrolled side effects
  - many applications “break” easily … because of poor implementation choices

Code smells

Look at your classes. Are they clean and simple?

- Large class
  - you may have a “god class” (a class that tries to do everything)
- Close dependency between two classes
  - type 1 = Feature envy:
    - class A always seems to be requesting services from class B
  - type 2 = Inappropriate intimacy:
    - class A depends too much on the internal implementation details of class B
- Excessive use of literals
  - too many magic numbers or magic strings
  - use symbolic names instead (named constants)
    - ➢ **for (card_index = 1 to 52)** versus **for (card_index = 1 to CARDS_IN_DECK)**
- Data clump
  - a cluster of data that “wants to be a class”
  - maybe a group of function parameters that appear in multiple functions
  - maybe a group of data fields that are often being “operated on” at the same time

Some simple refactorings

There is a large catalog of simple refactorings

Most of these code smells can be improved with selected refactorings:

- Extract Function
- Extract Class
- Change Function Declaration
- Rename Variable
- Rename Field
- Encapsulate Field
- Decompose Conditional
- Remove Comments

Check out this book:

- **Refactoring** by Martin Fowler
- "Improving the Design of Existing Code"
- 7 chapters of refactorings…

Extract Function

```
function printOwing(invoice) {
  let outstanding = 0;

  console.log("***********************");
  console.log("**** Customer Owes ****");
  console.log("***********************");

  // calculate outstanding
  for (const o of invoice.orders) {
    outstanding += o.amount;
  }

  // record due date
  const today = Clock.today;
  invoice.dueDate = new Date(today.getFullYear(), today.getMonth(), today.getDate() + 30);

  //print details
  console.log(`name: ${invoice.customer}`);
  console.log(`amount: ${outstanding}`);
  console.log(`due: ${invoice.dueDate.toLocaleDateString()}`);
}
```

```
***********************
**** Customer Owes ****
***********************
name: Acme Widget, Inc.
amount: 3500.12
due: February 19, 2022
```

Print a customer invoice, computing the total value of orders and the due date

Extract Function

```
function printOwing(invoice) {
  let outstanding = 0;

  console.log("***********************");
  console.log("**** Customer Owes ****");
  console.log("***********************");

  // calculate outstanding
  for (const o of invoice.orders) {
    outstanding += o.amount;
  }

  // record due date
  const today = Clock.today;
  invoice.dueDate = new Date(today.getFullYear(), today.getMonth(), today.getDate() + 30);

  //print details
  console.log(`name: ${invoice.customer}`);
  console.log(`amount: ${outstanding}`);
  console.log(`due: ${invoice.dueDate.toLocaleDateString()}`);
}
```

Create a printBanner() function from this code

Print a customer invoice, computing the total value of orders and the due date

Create a printDetails() function from this code

Extract Function

```
function printOwing(invoice) {
  let outstanding = 0;

  printBanner();

  // calculate outstanding
  for (const o of invoice.orders) {
    outstanding += o.amount;
  }

  // record due date
  const today = Clock.today;
  invoice.dueDate = new Date(
       today.getFullYear(),
       today.getMonth(),
       today.getDate() + 30);

  printDetails(invoice, outstanding);
}
```

```
function printBanner() {
  console.log("***********************");
  console.log("**** Customer Owes ****");
  console.log("***********************");
}
```

```
function printDetails(invoice, outstanding) {
  console.log(`name: ${invoice.customer}`);
  console.log(`amount: ${outstanding}`);
  console.log(`due: ${invoice.dueDate.
       toLocaleDateString()}`);
}
```

Note: *This is a very humble refactoring… It doesn't require a lot of analysis or testing… *but* it can make a big improvement in the structure of the code!*

Decompose Conditional

```
if (!aDate.isBefore(plan.summerStart) && !aDate.isAfter(plan.summerEnd))
  charge = quantity * plan.summerRate;
else
  charge = quantity * plan.regularRate + plan.regularServiceCharge;
```

The system computes a different charge in the summer…

```
if (summer(plan))
  charge = summerCharge(plan, quantity);
else
  charge = regularCharge(plan, quantity);
```

Build simple “helper functions” to simplify a complex conditional expression

```
function summer(plan) {
  return !aDate.isBefore(plan.summerStart) && !aDate.isAfter(plan.summerEnd);
}
function summerCharge(plan, quantity) {
  return quantity * plan.summerRate;
}
function regularCharge(plan, quantity) {
  return quantity * plan.regularRate + plan.regularServiceCharge;
}
```

Automated refactoring

Many software development environments include tools to support "automated refactoring"

For example, if you want to rename a variable...

- click on the declaration of the variable in your local function or class defintion
- choose "rename" from the refactoring menu, type in new variable name
- the tool will update all of the places that variable is used

"Extract function" to remove duplicated code might not be completely automated

- select the code to move into a new function, choose "extract function," type in function name and arguments
- the tool inserts the new function with the correct syntax
- and the tool inserts a function call in place of the code you selected
- *but* the tool might not be able to find and replace the other duplicates

An unusual fact!

Disappearing code:

- Sometimes in an agile project you will have days when you do a lot of refactoring,
- and many of the refactoring steps that reduce duplicated code will create an interesting problem
- the count of source lines in the system will shrink
- “Oh, how productive were you today?” “Well, I developed -50 lines of code today!”

One reason for investing effort in refactoring - to reduce “technical debt”

- clean code is easier to modify than messy code
- periodic refactoring helps to improve the team’s “velocity”

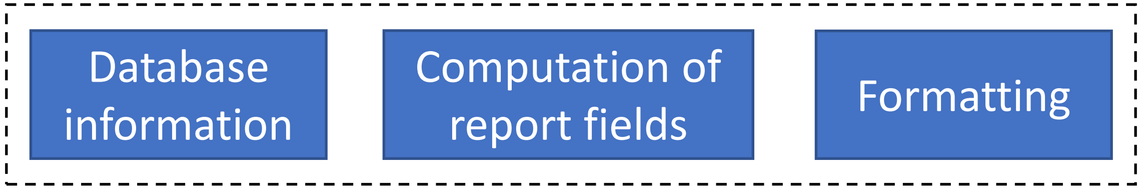
Caution! Refactor carefully

Can you take refactoring too far? Yes.

The focus will be on small pieces of the code

- We can make code “cleaner”
- *but* it is essential to respect the architecture and the algorithms of the system as a whole

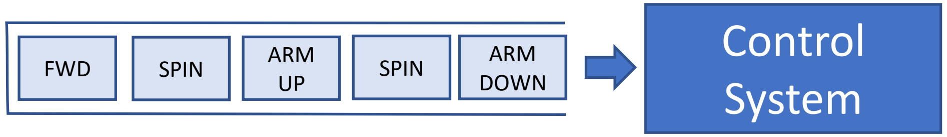
For example... a Billing System is mostly about “creating documents” (classic 3-tier architecture)



- Refactoring is extremely useful, because the “document” is structured to have many small pieces – and refactoring can clean up each small piece

A Robot System is mostly about “control”

- The fundamental architecture is either “command-response” or “managing a state machine”
- Creating subcommands or “sub-state-machines” is OK
- *but* the refactoring work must preserve the visible form of “command” or “state machine” structure



Other code improvements

There are more ambitious things you can do with refactoring...

My favorite... replace custom code with calls to standard library functions and classes

- particularly useful in languages with rich libraries - Java and Python
- the use of standard libraries can shrink the total code size, increase understandability, and sometimes improve performance
- can this be taken too far? absolutely!

It is also useful to refactor to “add more tests”…

Other code improvements

Refactoring to add unit tests to a legacy system without unit tests

- there is a process to find or create “seams” in the code
- places where you can add a simple unit test to exercise one or more functions in the code
- best explanation: “Working Effectively with Legacy Code” by Michael Feathers
- Michael says “get that old code under unit-test control”

3. Patterns community camp

“Refactoring to Patterns” by Joshua Kerievsky

- an excellent guide to some higher-level refactorings
- they involve some of the standard Design Patterns
- and they address certain code smells and design smells

The goal is to clean up the existing implementation – find a “pattern” that resolves a “design smell” – make a series of changes to incorporate the pattern

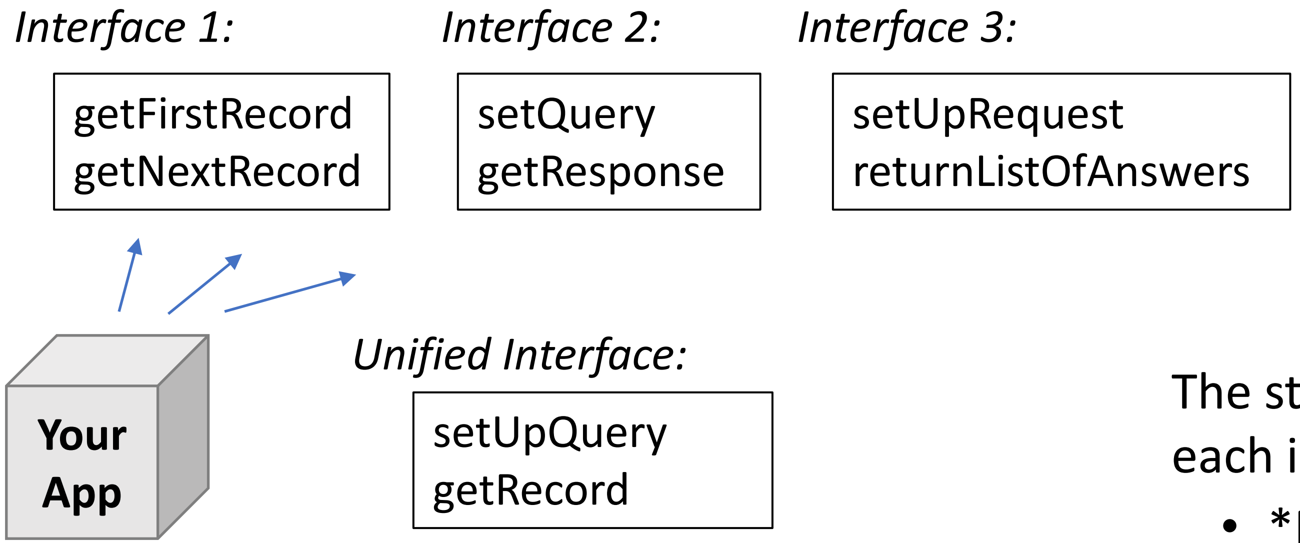
The refactoring to patterns process is also “step-by-step” -- reworking the code to make it align with one of the standard design patterns

When you are done, the resulting code will be cleaner and easier to document -- you can point to the standard patterns literature

Example of “refactoring to patterns”

One common “design smell” is related to one of the Code Smells:

- Alternative Classes with Different Interfaces



Interface 1:

getFirstRecord
getNextRecord

Interface 2:

setQuery
getResponse

Interface 3:

setUpRequest
returnListOfAnswers

Unified Interface:

setUpQuery
getRecord

Existing application receives data from three data sources… and each data source has a different interface

The standard “refactoring” solution: Refactor each interface to make them look the same.

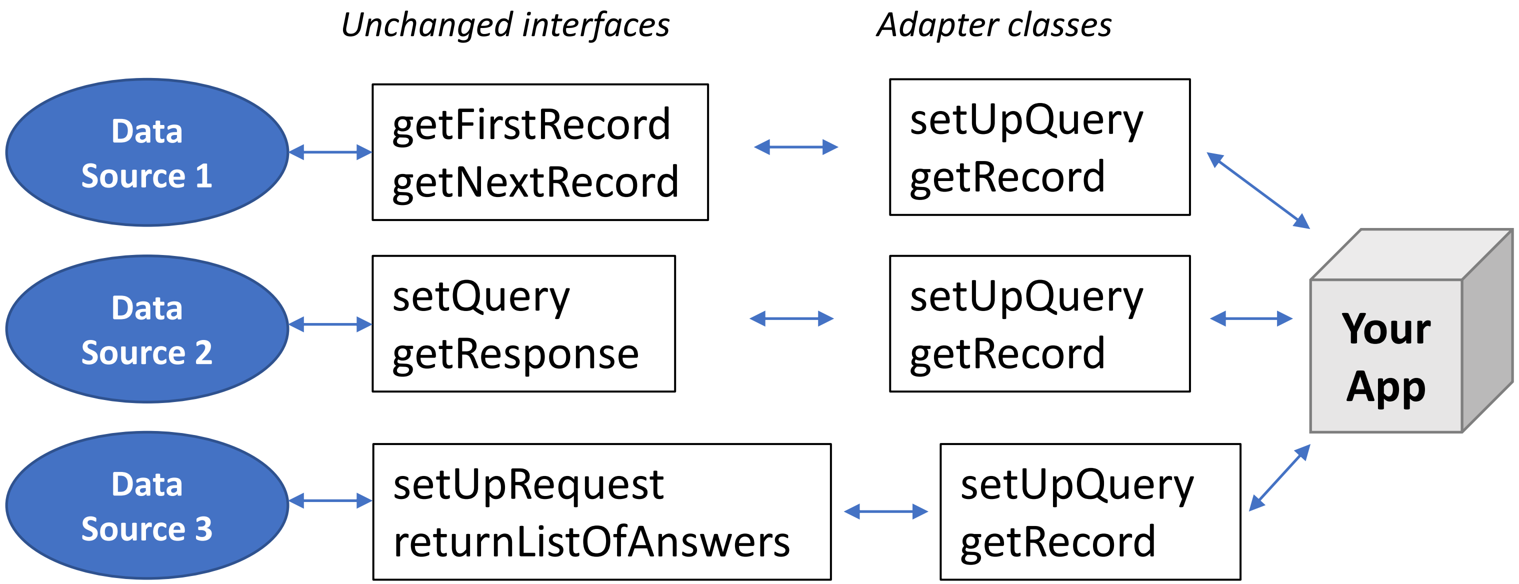
- *But* this might not be possible
- Third-party libraries, source code not available, code owned by another group

Example of “refactoring to patterns”

A pattern-based solution: use the Adapter Pattern

- from the **Design Patterns** book

An Adapter is a small “wrapper class” – it contains a pointer to the original interface object, and it “translates” each application-level request to the appropriate format



The “refactoring to patterns” solution:

- Implement three different “adapter classes”
- App uses the adapters
- Each adapter calls functions in one of the original interfaces

4. Legacy code analysis camp

Use refactoring to explore legacy code:

The code is the best documentation of a legacy software system -- much better than any design documents.

- design documents are often out of date or just wrong
- the code always "tells the truth"

But you can't understand everything at once

- initial exploration should focus on understanding the modules, execution paths, communication paths, and databases
- detailed exploration may include "exploratory refactoring"
- we make some small changes -- new "probes" into the middle of key functions and classes
- use simple refactorings (like Rename Attribute, Rename Method, and Extract Method) and "ask questions" to discover the essential features of the design

The goal is to "learn" – and the refactoring work may actually be thrown away after you understand the legacy system!

Book: Object Oriented Reengineering Patterns by Serge Demeyer, Stephane Ducasse, and Oscar Nierstrasz

Free download!!!
http://scg.unibe.ch/download/oorp/OORP.pdf

Chapter titles-

- First Contact Patterns
- Initial Understanding
- Detailed Model Capture
- Tests
- Migration Strategies
- Duplicated Code
- Redistribute Responsibilities
- Introduce Polymorphism

A summary of the 4 “camps”

- camp 1 - software evolution
  - They use refactoring selectively – to prepare for building new features
- camp 2 - agile development
  - They do some refactoring in every iteration… keeps the code clean
- camp 3 - patterns community
  - Selective refactoring to improve the software design
- camp 4 - legacy code analysis
  - Some limited “exploratory refactoring” helps answer questions about the legacy code

Summary

You should learn more about refactoring

- Do some refactoring to improve the structure of your own code
- Try refactoring the code you are "maintaining"
- Add unit tests to your legacy code – to support safe refactoring
- Both manual refactoring and automated refactoring are pretty easy

Goal: better quality, easier to make code changes

More resources:
manclswx.com/projects/refactoring.html

Advice

There are a lot of good ideas about refactoring…

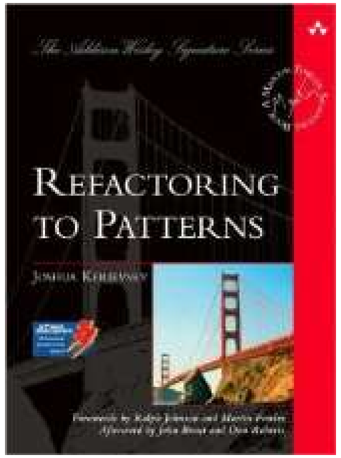
  - Read some of the books and articles about refactoring techniques
  - Walk through some of the examples (such as Chapter 1 of Martin Fowler’s book) – just to get a taste of how refactoring works

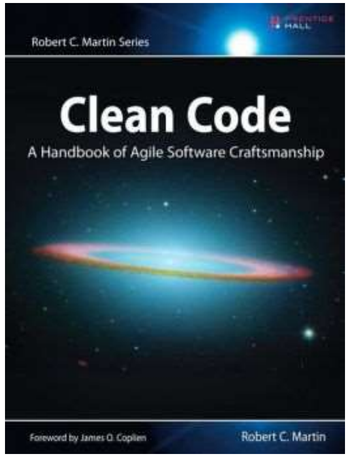
- BUT…
  - Refactoring examples don’t always make sense…
  - You might not *really understand* refactoring until you apply it yourself
  - So try it! Do some minor work on some of your own “old code”

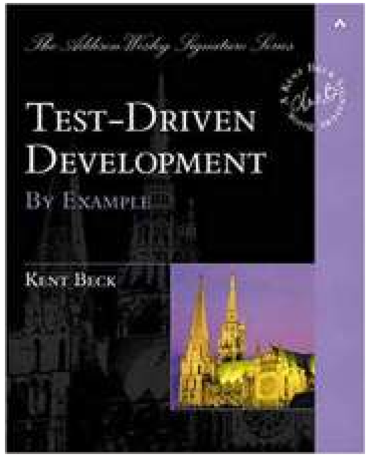
Where to learn more

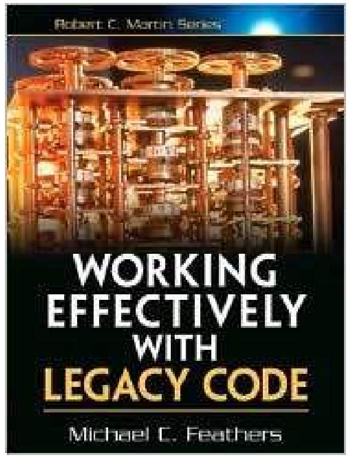
Books

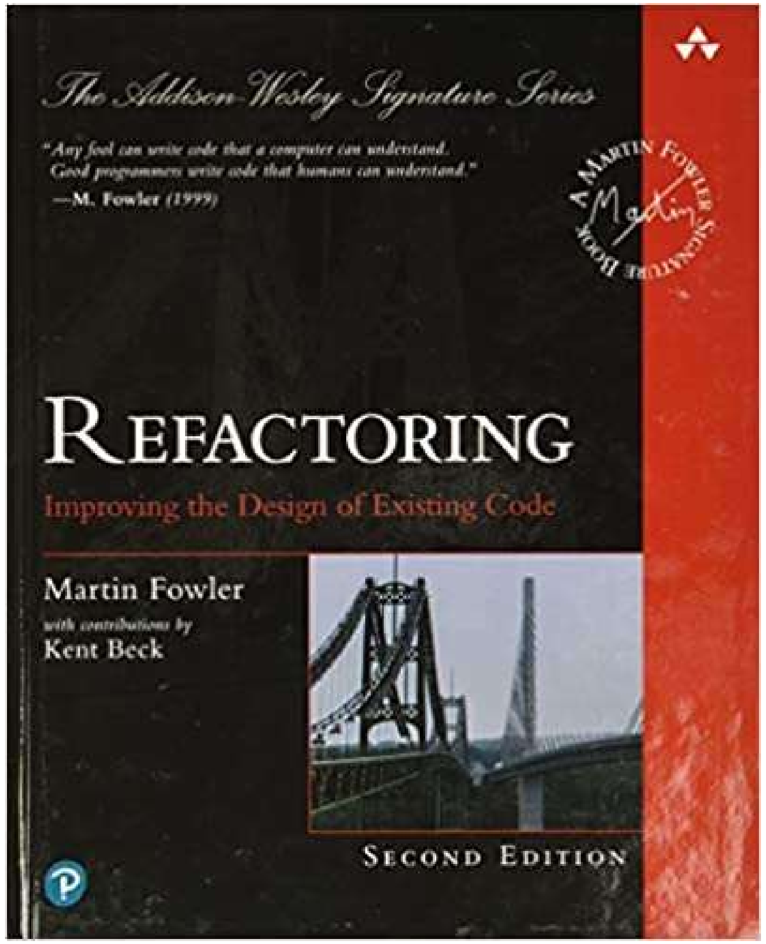
- **Refactoring** by Martin Fowler (2nd edition, 2019)
- **Refactoring to Patterns** by Joshua Kerievsky
- **Clean Code** and **The Clean Coder** by Robert C. Martin
- **Test-Driven Development By Example** by Kent Beck (many books give good examples of TDD!)
- **Working Effectively With Legacy Code** by Michael Feathers




Robert C. Martin Series
Clean Code
A Handbook of Agile Software Craftsmanship
Foreword by James O. Coplien
Robert C. Martin






The Addison-Wesley Signature Series
"Any fool can write code that a computer can understand. Good programmers write code that humans can understand." —M. Fowler (1999)
A Martin Fowler Signature Book
Refactoring
Improving the Design of Existing Code
Martin Fowler
with contributions by
Kent Beck
Second Edition

Free Download -- Chapter 1:
https://martinfowler.com/books/refactoring.html

Questions?

More resources:
manclswx.com/projects/refactoring.html

Dennis Mancl

dmancl@acm.org

MSWX ◊ Mancl ◊
Software ◊ Experts ◊ http://manclswx.com