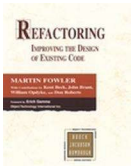
REFACTORING AND CODE SMELLS

Acknowledgements

- Material in this presentation was drawn from



Martin Fowler, *Refactoring: Improving the Design of Existing Code*

Definition

- Process of altering source code so as to leave its functionality unchanged
- Why refactor?
  - Improve maintainability, especially enhancement
- When to refactor?
  - Considered as soon as code writing begins
  - Essential part of most agile approaches
  - Refactor only with valid unit tests in place – need to ensure that refactoring does not break working code

What is it and what is it not?

Refactoring is…

- Small, behaviour-preserving, incremental and safe steps
- Improving the design of existing code
- Making it more understandable and/or flexible
- Breaking a large method into smaller, more focused methods
- Renaming variables and parameters to be meaningful
- Moving a responsibility from one class to a more appropriate one
- Creating an interface, based on the methods in one class, and making that class implement the new interface

Refactoring is NOT…

- An excuse to go back and "fill in the blanks" in your application
- Improving (adding) error handling
- Adding logging
- Cramming in another feature
- Enhancing the test coverage

Why Refactor?

- Prevent "design decay"
- Clean up messes in the code
- Simplify the code
- Increase readability and understandability => increase maintainability and extendabillity
- Find bugs
- Reduce debugging time
- Built-in learning about the application
- Redoing things is fundamental to every creative process

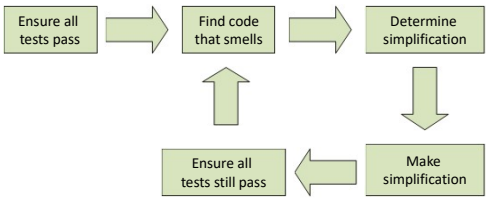
Design Decay

- Design decay occurs in software systems largely where there is an absence of an upfront design
- But also arises in cases where an upfront design exists but the system has been evolved without the design also being revised/evolved.
- Design decay describes a condition where code may be functionally correct and satisfying customers today but where the underlying implementation has become inefficient from a maintenance and evolution perspective.
- To address this issue we look for so called *code smells*, and attempt to resolve them.

How to refactor

- Make sure your tests pass
- Find some code that "smells"
- Determine how to simplify this code
- Make the simplifications
- Run tests to ensure things still work correctly
- Repeat the simplify/test cycle until the smell is gone

Refactoring Flow



Where to refactor

- **Anywhere that needs it**, provided:
  - Tests exist and currently pass for the code to be refactored
  - Someone else is not concurrently working in the same code
  - The customer agrees that the area is worth the time and money to refactor

  Q. What if the customer does not agree or will not pay?

When to refactor

- "All the time"
- Rule of Three
- When you add functionality
- When you learn something about the code
- When you fix a bug
- When the code smells

When not to refactor

- **When the tests aren't passing**
- When you should just rewrite the code
- When you have impending deadlines (Cunningham's idea of unfinished refactoring as **debt**)

Problems with refactoring

- Taken too far, refactoring can lead to incessant tinkering with the code, trying to make it perfect
- Refactoring code when the tests don't work or tests when the application doesn't work leads to potentially dangerous situations
- Databases can be difficult to refactor
- Refactoring published interfaces can cause problems for the code that uses those interfaces

Why developers are reluctant to refactor

- Lack of understanding
- Short-term focus
- Not paid for overhead tasks like refactoring
- Fear of breaking current program

Refactorings (Fowler)

- Add Parameter
- Change Bidirectional Association to Unidirectional
- Change Reference to Value
- Change Unidirectional Association to Bidirectional
- Change Value to Reference
- Collapse Hierarchy
- Consolidate Conditional Expression
- Consolidate Duplicate Conditional Fragments
- Convert Procedural Design to Objects
- Decompose Conditional
- Duplicate Observed Data
- Encapsulate Collection
- Encapsulate Downcast
- Encapsulate Field
- Extract Class
- Extract Hierarchy
- Extract Interface
- Extract Method
- Extract Subclass
- Extract Superclass
- Form Template Method
- Hide Delegate
- Hide Method
- Inline Class
- Inline Method

Refactorings (Fowler)

- Inline Temp
- Introduce Assertion
- Introduce Explaining Variable
- Introduce Foreign Method
- Introduce Local Extension
- Introduce Null Object
- Introduce Parameter Object
- Move Field
- Move Method
- Parameterize Method
- Preserve Whole Object
- Pull Up Constructor Body
- Pull Up Field
- Pull Up Method
- Push Down Field
- Push Down Method
- Remove Assignments to Parameters
- Remove Control Flag
- Remove Middle Man
- Remove Parameter
- Remove Setting Method
- Rename Method
- Replace Array with Object
- Replace Conditional with Polymorphism
- Replace Constructor with Factory Method
- Replace Data Value with Object
- Replace Delegation with Inheritance

Refactorings (Fowler)

- Replace Error Code with Exception
- Replace Exception with Test
- Replace Inheritance with Delegation
- Replace Magic Number with Symbolic Constant
- Replace Method with Method Object
- Replace Nested Conditional with Guard Clauses
- Replace Parameter with Explicit Methods
- Replace Parameter with Method
- Replace Record with Data Class
- Replace Subclass with Fields
- Replace Temp with Query
- Replace Type Code with Class
- Replace Type Code with State/Strategy
- Replace Type Code with Subclasses
- Self Encapsulate Field
- Separate Domain from Presentation
- Separate Query from Modifier
- Split Temporary Variable
- Substitute Algorithm
- Tease Apart Inheritance

Consolidate Conditional Expression

- Multiple conditionals can be extracted into method
- Don't do if conditions are really independent
- Example

BEFORE

```
double diasabilityAmount() {
    if (_seniority < 2) return 0;
    if (_monthsDisabled > 12) return 0;
    if (_isPartTime) return 0;
    if (_isVeteran) return 50;
    // Calculate disability amount
```

AFTER

```
double diasabilityAmount() {
    if (isNotEligibleForDisability())
     return 0;
    if (_isVeteran) return 50;
    // Calculate disability amount
```

Extract Class

- Remove a piece of a class and make it a separate class
- Done when class is too big to understand easily or behavior is not narrowly defined enough
- Indicated by having subsets of data & methods that go together, are changed together, or are dependent on each other

Extract Method

- Pull code out into a separate method when the original method is long or complex
- Name the new method so as to make the original method clearer
- Each method should have just one task

```
void printOwing() {
  printBanner();

  //print details
  System.out.println ("name:  " + _name);
  System.out.println ("amount " +
getOutstanding());
}
```

```
void printOwing() {
  printBanner();
  printDetails(getOutstanding());
}

void printDetails (double outstanding) {
  System.out.println ("name:  " + _name);
  System.out.println ("amount " + outstanding);
}
```

Replace Error Code with Exception

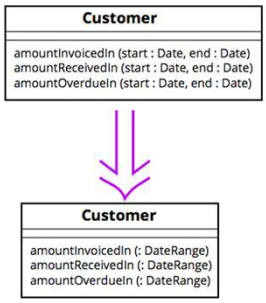
- A method returns a special code to indicate an error.
- Throw an exception instead.

```
int withdraw(int amount) {
  if (amount > _balance)
    return -1;
  else {
    _balance -= amount;
    return 0;
  }
}
```

```
void withdraw(int amount) throws
BalanceException {
  if (amount > _balance) throw new
BalanceException();
  _balance -= amount;
}
```

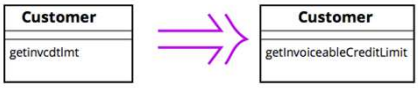
Introduce Parameter Object

- Replace a group of parameters that go together with an object
- Makes long parameter lists shorter & easier to understand
- Can indicate functionality to move to new class



Rename Method

- Method names should clearly communicate the one task that the method performs
- If you are having trouble naming the method, it might be doing too much. Try extracting other methods first



Replace Magic Number with Symbolic Constant

- Replace hard-coded literal values with constants
- Avoids duplication and shotgun surgery

```
double potentialEnergy(double mass, double height) {
  return mass * height * 9.81;
}
```

```
double potentialEnergy(double mass, double height) {
  return mass * GRAVITATIONAL_CONSTANT * height;
}
static final double GRAVITATIONAL_CONSTANT = 9.81;
```

Replace Nested Conditional with Guard Clauses

- In some conditionals, both paths are normal behavior & the code just needs to pick one
- Other conditionals represent uncommon behavior
- Use if/else with the normal behavior paths & guard clauses with uncommon behavior

```
double getPayAmount() {
  double result;
  if (_isDead) result = deadAmount();
  else {
    if (_isSeparated) result = separatedAmount();
    else {
      if (_isRetired) result = retiredAmount();
      else result = normalPayAmount();
    };
  }
  return result;
};
```

```
double getPayAmount() {
  if (_isDead) return deadAmount();
  if (_isSeparated) return separatedAmount();
  if (_isRetired) return retiredAmount();
  return normalPayAmount();
};
```

CODE SMELL

Code smells

- A code smell is a hint that something is wrong in your code
- Use the smell to track down the problem
- Differing view points:
  - Pragmatist: consider code smells on a case by case basis
  - Purist: all code smells should be avoided without exception
- http://c2.com/xp/CodeSmell.html

Comments

- Often used as deodorant for other smells
- Not necessarily bad in and of themselves but may indicate areas where the code is not as clear as it could be
- Refactorings
  - Extract Method

Duplicated Code

- Code repeated in multiple places
- Refactorings
  - Extract Method
  - Extract Class
  - Pull Up Method

Data Clumps

- Sets of variables usually passed together in multiple places
- Refactorings
  - Extract Class
  - Introduce Parameter Object

Inappropriate Intimacy

- Classes using too many things that should be private in other classes
- Refactorings
  - Move Method
  - Move Field
  - Change Bidirectional Association to Unidirectional

Large Class

- A class with too many instance variables or too much code
- Refactorings
  - Extract Class
  - Extract Subclass
  - Extract Interface
  - Replace Data Value with Object

Long Method

- Methods with many statements, loops, or variables
- Astels defines long as > 10, Fowler doesn't say
- Refactorings
  - Extract Method
  - Decompose Conditional

Long Parameter List

- Many parameters passed into a method
- Refactorings
  - Replace Parameter with Method
  - Introduce Parameter Object

For Testers Only

- Methods that exist in production code solely for the use of the tests
- Refactorings
  - Extract Subclass
  - Isolate testing code

How to identify code smells?

- Experience
- Regular code review
- Tools that generate metrics, such as:
  - cyclomatic complexity

What is Cyclomatic Complexity?

- This metric was introduced by Thomas McCabe and measures the structural complexity of a method.
- Example
  - Suppose you've got a particular implementation class that's become huge in size or too complex to maintain. Or, you've got a single class acting as the control class for a whole business layer, but there's too much business logic embedded within that one class. Or suppose again that you've been handed a class containing too much code duplication.
  - These situations are what are referred to as "**complexity**"

What is Cyclomatic Complexity?

- At its most basic, Cyclomatic Complexity (CC) = number of decision points + 1
- Decision points are conditional statements such as if/else, while etc.
- Consider the following example:

```
public void isValidSearchCriteria(
SearchCriteria s )
{
    if( s != null )
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

- In the above example, CC=2+1=3 (one if + one else + 1).

What is Cyclomatic Complexity?

- **Cyclomatic complexity has enormous impact on the testability and maintainability of code.**
- As the example shows, if you want to test the `isValidSearchCriteria()` method, you have to write three tests to verify it:
  - one for the method itself, and two for the decision points within the method.
- Clearly, if the CC value increases, and there is an increasing number of decision points within any method, it means more effort to test the method.

What is Cyclomatic Complexity?

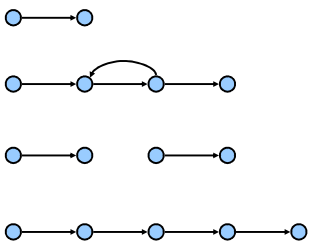
- The following table summarizes the impact of CC values in terms of testing and maintenance of the code:

CC Value	Risk
1-10	Low risk program
11-20	Moderate risk
21-50	High risk
>50	Most complex and highly unstable method

McCabe's Cyclomatic Complexity

- A measurement of the intricacy of a program module based on the number of repetitive cycles or loops that are made in the program logic.
- It is used as a general measure of complexity for software quality control as well as to determine the number of testing procedures.
- Defined as: M = L - N + 2P
  - L = number of links in the flowgraph
  - N = number of nodes in the flowgraph
  - P = number of disconnected parts of the flowgraph.

Examples of Cyclomatic Complexity



L=1, N=2, P=1
M=1-2+2=1

L=4, N=4, P=1
M=4-4+2=2

L=2, N=4, P=2
M=2-4+4=2

L=4, N=5, P=1
M=4-5+2=1

Other Metrics

- Many may more…

mean_relative_height_in_inheritance_tree	improved ratio of abstract classes.
afferent_couplings	Number of afferent couplings that means the number of classes outside of this package that depend on classes inside of this package.
average_method_weight	average cyclomatic complexity.
base_class_overriding_ratio	ratio of overridden to total members of baseclass.
base_class_usage_ratio	ratio of used to total inheritance-specific members of baseclass.
changing_classes	number of classes calling this method.
changing_methods	number of methods calling this method.
class_instability	degree of class instability.
co	normalized count of the number of edges of G of lcom4.
coupling_dispersion	number of classes called divided by coupling_intensity.
coupling_intensity	number of distinct methods called.
cyclomatic_complexity	Number of linear indipendent path through the control flow.
distance	distance from the main line.
depth_of_inheritance_tree	depth of inheritance tree.
temporal_degree_of_interest	temporal degree of interest (T-DOI).

https://sewiki.iai.uni-bonn.de/research/cultivate/smells_and_metrics

Tool Support

- Tools to identify bad smells and control metrics in a software:
- PMD, Borland Toguether, PRODEOOS, DECOR, Aspect, LCLINT, JLINT, Extended Static Checker, SmallLint, FindBugs, Saber, Analyst4J, CheckStyle, FxCop, Hammurapi, SemmleCode, Crocopat, Blast, Mops, POM framework, SORMASA, JDeodorant, Java Source Metric, InCode, InsRefactor, CodeBizard, Recorder API, JBuilder, Duploc and JSmell

Discussion / Reflection

- Should refactoring be a continuous activity?
- Is it better to have periodic refactoring or should there be an insistence on continuous refactoring that is built into the velocity?