Refactoring

Arnav Gupta

April 22, 2024

Contents

1 Bugs

Bugs often repeat and can be obvious/dumb, so blatant, well-understood, and easy to find if know what to look for.

OO languages are complex, so high potential for misuse.

Simple static code analysis can find many bugs.

Can use `FindBugs` on JDK1.

2 Software Refactoring

Preventative software maintenance is changes aiming to improve the current or future maintainability and reliability of the software system.

Necessary due to software aging, so refactoring brings back to original stable state.

Refactoring: process of changing a software system in such a way that it does not alter the *external* behaviour of the code, yet improves its *internal*

structure

- disciplined way to clean up code that minimizes chances of introducing bugs
- when refactoring, this improves the design of the code after it has been written

Refactoring activities are:

1. identify places where software should be refactored (code smells)
2. determine which refactoring(s) should be applied
3. guarantee that applied refactoring(s) preserve behaviour
4. apply refactoring(s)
5. assess effect of refactoring on design quality
6. maintain consistency between refactored code and other software artifacts (ex. documentation, design docs, tests, etc.)

Floss refactoring: frequent refactoring interleaved with other kinds of program changes

Root canal refactoring: infrequent periods of refactoring, during which programmers perform few if any other kinds of program changes

Floss refactoring is for maintaining healthy code, root canal refactoring is for correcting unhealthy code.

Refactoring OO frameworks started at UIUC where Bill Opdyke introduced the concept of refactorings as behaviour-preserving programming restructing operations. Also introduced the concept of preconditions as conditions that should be examined to determine whether a refactoring can be applied safely.

For duplicated code, if there is the same code structure in 1+ place, find a way to unify them. This is a code smell.

Simon and Lewerentz used metrics based refactoring, so they defined a metric based on Jaccard distance to quantify cohesion between attributes and methods. Calculated distances are visualized in a 3D perspective using VRML, and the developer manually IDs refactoring opportunities.

Entity set for a method is the considered method itself, all directly used methods, and all directly used attributes.

Entity set for an attribute is the considered attribute itself and all methods using it.

Detection strategy: a composition of various metric rules combined with AND/OR operators into a single rule

Metric rules: metrics that should comply with proper threshold values

Composition of metric rules expresses a violation of a **design heuristic**

To transform an informal design rule in a detection strategy, take a design-related informal rule, analyze it to a detection technique, and select metrics to form a detection strategy.

Marginal data filters can be semantical (based on a number), which can be relative or absolute, or statistical.

Interval data filters are composed of 2 marginal filters, with semantical limit specifiers of opposite polarities.

DECOR: a method for specification and detection of code and design smells based on a DSL (domain-specific language)

DETEX: a detection technique that instantiates DECOR

Antipatterns can be inter-class or intra-class, as well as structural, lexical, or measurable.

Detection algorithm for an anti-pattern is automatically generated from the rule card.

2.1 Long Method

The longer a method is, the more difficult it is to understand.

Larger modules are associated with higher error-proneness and change-proneness

Extracting code fragments having a distinct functionality into new methods can improve understandability.

Long method is one with low cohesion.

Slice-based metrics can be used to measure the level of cohesion within a module (method/function). Defined cohesion metrics are computed based on slice profiles.

To find tightness:

- let SL_i be the slice obtained for variable $v_i \in V_O$
- SL_{int} be the intersection of SL_i over all $v_i \in V_O$
- tightness of a method is $|SL_{int}|$ over the length of the method
- expresss the raio of the number of statements which are common to all slices over the module length

To find overlap:

$$Overlap(M) = \frac{1}{|V_O|} \sum_{i=1}^{|V_O|} \frac{|SL_{int}|}{|SL_i|}$$

which gives the average ratio of the number of statements which are common to all slices to the size of each slice

All slice-based cohesion metrics range from 0 to 1. Higher tightness and overlap means a module is more cohesive.

Trying to decompose a highly cohesive module would introduce excessive code duplication between the original module and the newly extracted modules.

2.2 Feature Envy

Method more interested in a class other than the one it belongs to. Violates design principle that behaviour should be allocated along with data being accessed. Usually envied data is accessed through a parameter of the method or field of the class.

Data class is a class containing mostly attributes and few or no methods. Violates the same design principle.

Be suspicious of:

- classes whose instance variables are manipulated by other classes
- classes with many public fields

2.3 God/Large Class

Class with multiple responsibilites, violates Single Responsibility principle.

Be suspicious of:

- large classes treating many other classes as data classes

- classes with names containing system, subsystem, manager, driver, or controller
- classes with unrelated sets of methods working on separate instance variables (low cohesion)

A cohesion graph has every method (other than ctors) represented in a node. Two methods connected with an undirected edge if they access a common instance variable or if one method calls another. Edges due to common field accesses have more weight than those due to calls. To decompose a class, find disconnected components in the graph or cut the graph using the max-flow min-cut theorem.

2.4 Duplicated Code

Worst code smell. All duplicate instances in source code have to be found and updated. Inconsistent updates may lead to errors in the program.

Clone detection defines 4 types of clones:

- **type 1**: code fragments are identical except for variations in whitespace, layout, and comments
  - may not be refactorable if clones:
    - return 1+ variable
    - return variables with different types
    - contain conditional return statements
    - contain branching statements (break, continue) without the corresponding loop
- **type 2**: code fragments are structurally and syntactically identical except for variations in identifiers, literals, types, in addition to type 1 differences
  - refactor by generalizing types, parameterizing object creation, or parameterizing number literal
- **type 3**: code fragments are copies with further modifications, statements can be changed, added, or removed in addition to type 2 differences

    - refactor by moving statements without changing behaviour or abstracting behaviour

- **type 4**: 2+ code fragments perform the same computation, but are implemented through different syntactic variants
    - ex. insertion sort vs bubble sort

When refactoring duplicate code:

1. if the duplicated code is in the same class, apply extract method refactoring
2. if the duplicated code is in different subclasses of the same superclass, apply extract & pull up method refactorings
3. if the duplicated code is in different unrelated classes and there are several duplicated instance methods, move them to a new common superclass, and if they are static, move them to a utility class