ERRS: Current Research

Isaac Griffith

April 8, 2010

1 Problem

Legacy code tends to be hard to understand and simultaneously tends to have redundant and poorly engineered code. There exists many techniques and technologies which can aid a developer in correcting these issues during development, but the time money, and understanding required to refactor older legacy systems is hard to come by. There also exists a gap in the technology which aids in the maintenance and utilization of legacy systems.

Another problem is that, although automated refactoring tools exist, almost all of them only refactor based on what the user desires (e.g., they act as a function processing text, and require the user's (developer's) interaction to ensure they function correctly), and there are no fully autonomous refactoring tools that alleviate the developer of the tedious details. This current methodology, although useful (for modifying current systems) still leaves the understanding and contextual part up to the developer, hence requiring full knowledge of the code (and its problems) up to the developer. Thus, for older systems where this knowledge may not be readily available, and where extraction of code and understanding is rapidly required, a gap exists. A system is intelligent enough to extract the necessary code, while simultaneously maintaining the knowledge and context surrounding it, and is able to successfully apply correct refatoring and design patterns to render the system easier to understand while maintaining its overall outward function.

2 Preliminary Research on Refactoring

- **Code Smells** ≡ Qualitative indications of need for refactoring. See: http://wiki.java.net/bin/view/p
- **Refactoring**; See: http://refactoring.com/catalog/index.html for a catalog of different refactoring techniques.
- Design Pattern Refactoring
- Anti-patterns

3 Use of Code Smells

- Code smells indicate a deeper problem in a program.

- Each smell is refactored out in small, controlled steps, and the resulting design is examined to see if anymore code smells exist indicating the need of further refactoring.

4 Metrics

Quality Metrics

- Complexity → Cyclomatic Complexity, Fan-in
- Maintainability → Design factors
- Understandability → Percent internal comments, Variable naming conventions
- Reusability → Percent reused components
- Documentation → Readability Index

Product Metrics

The goal of the project is to modify existing code in order to increase Understanding, Maintainability, and Re-use.

Metrics that can be used for these three subgoals are:

- U,M – Complexity → Cyclomatic Complexity
- U,M – Size → Lines of Code (LOC)
- U,M – Documentation → Comment Percentage
- M,R – WMC → Weighted Methods per Class
- R – RFC → Response For a Class
- U,M,R – LCOM → Lack of Cohesion
- U,M,R – CBO → Coupling Between Object Classes

How To Measure Each Metric

Measuring Complexity: Cyclomatic Complexity

$$Complexity = (Edges - Nodes) + 2$$

Measuring Size

- LOC: (Lines of Code) count all lines in source files
- NCNB: (Non-comment, Non-blank) counts all lines that are not comments and not blanks.
- EXEC: (Executable Statements) counts executable statements regardless of number of physical lines of code.

Comment Percentage

Both on-line (with code) and stand-alone comments

$$\%Comments = \frac{Comments}{totalLinesOfCode - blankLines}$$

Measuring Weighted Methods per Class (WMC)

- Count of methods in each class
- Sum of complexities (by cyclomatic complexity) of the methods

Measuring Lack of Cohesion (LCOM)

Number of methods within a class that reference a given instance variable.

Fan-in Fan-out Complexity

Maintains a count of the number of data flows from a component plus the number of global data structures the program updates. The data flow count includes procedure parameters and procedure calls from within a module.

$$Complexity = Length * (Fan - in * Fan - out)^2$$

Length is any measure of length such as LOC or alternatively Cyclomatic Complexity.

Measuring Coupling Between Object Classes (CBO)

A count of the number of other classes to which a class is couples. Measured by counting the number of distinct non-inheritance related class hierarchies on which a class depends.

5 Relevant Code Smells

Smells Between Classes

Data Class → Classes with fields and getters and setters and nothing else

- Move in behavior with *Move Method*

Inappropriate Intimacy → Two classes are overly intertwined
Use the following refactorings:

- *Move Method*
- *Move Field*
- *Change Bidirectional Association to Unidirectional Association*
- *Extract Class*
- *Hide Delegate*
- *Replace Inheritance with Delegation*

Feature Envy → Often a method that seems more interested in a class other than the one its in. In general, try to put a method in the class that contains most of the data the method needs.

- *Move Method*, may need to use *Extract Method* first, then *Move Method*.

Message Chains → This is the case in which a client has to use one object to get another, and then use that one to get to another, etc. Any change to the intermediate relationships causes the client to have to change.

- Use *Extract Method* and then *Move Method* to move it down the chain.

Shotgun Surgery → The opposite of Divergent Change. A change results in the need to make a lot of little changes in several classes.

- Use *Move Method* and *Move Field* to put all the changes into a single class.

Parallel Inheritance Hierarchies → A special case of *Shotgun Surgery*. Every time you make a subclass of one class, you also have to make a subclass of another.

- Use *Move Method* and *Move Field* to combine the hierarchies into one.

6 Refactoring

Move Class

- *Problem:* You have a class that is in a package that contains other classes that it is not related to in function.
- *Solution:* Move the class to a more relevant package, or create a new package if required for future use.

Move Field

- *Problem:* A field is, or will be, used by another class more than the class on which it is defined.

- *Solution:* Create a new Field in the target class, and change all its users

Move Method

- *Problem:* A method is, or will be, using or used by more features of another class than the class on which it is defined
- *Solution:* Create a new method with a similar body in the class that uses it most. Either turn the old method into a simple delegation, or remove it altogether.

Extract Method

- *Problem:* You have a code fragment that can be grouped together
- *Solution:* Turn the fragment into a method whose name explains the purpose of the method.

Collapse Hierarchy

- *Problem:* A superclass and subclass are not very different
- *Solution:* Merge the two classes together

Extract Class

- *Problem:* You have one class doing work that should be done by two
- *Solution:* Create a new class and move the relevant fields and methods from the old class into the new class

Extract Subclass

- *Problem:* A class has features that are used only in some instances
- *Solution:* Create a subclass for that subset of features

Inline Class

- *Problem:* A class isn't doing very much
- *Solution:* Move all its features into another class and delete it.

Introduce Parameter Object

- *Problem:* You have a group of parameters that naturally go together
- *Solution:* Replace them with an object

Pull-Up Field

- *Problem:* Two subclasses have the same field
- *Solution:* Move the field into the superclass