**Refactoring: Code Smells**

**Refactoring**

1. Refactoring:

* “A change to the system that leaves its behavior unchanged, but enhances some nonfunctional quality – simplicity, flexibility, understandability, performance” (Kent Beck, *Extreme Programming Explained*, page 179)
* “A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior” (Martin Fowler, *Refactoring*, page 53)

1. Martin Fowler wrote the seminal book on refactoring: *Refactoring – Improving the Design of Existing Code.*
2. Fowler defines a *code smell* as a piece of code (fragment of code, method, class, architecture, etc) that causes you to pause and and think, “…is this the best way to accomplish \_\_\_\_\_?”
3. If a piece of code exhibits a code smell, then it is a candidate for *refactoring*.
4. Before beginning refactoring, it is imperative that you have tests so that the refactored code at least generates results that are the same as before the refactoring.
5. “Refactoring keeps you ready for change by keeping you comfortable with changing your code”

(Ken Auer and Roy Miller, *Extreme Programming Applied*, page 189)

1. Refactoring flow:

Ensure all tests pass

Loop until no code smells:

Find code that smells

Determine simplification

Make simplification

Ensure all tests still pass

1. When not to refactor:
2. When tests don’t pass
3. When you should just rewrite the code
4. When you have impending deadlines
5. Comments are often used as deodorant for code smells.
6. jDeodorant is an Eclipse plug-in that identifies bad smells and resolves them by applying appropriate refactorings.

<https://marketplace.eclipse.org/content/jdeodorant>

**Videos**

1. Rewatch these clips from Video P4L5 – Refactoring:

2 - Introduction

4 – Reasons to Refactor

6 – Types of Refactorings

7 – Collapse Hierarchy

8 – Consolidate Conditional Expression

9 – Decompose Conditionals

10 – Extract class

11 – Incline class

12 – Extract method

13 – Refactoring Demo

18 – Bad Smells

19 – Bad Smells Examples

**Extract Method**

1. “Almost all the time the problems come from methods that are too long. Long methods are troublesome because they often contain lots of information, which gets buried by the complex logic that usually gets dragged in.” – Martin Fowler
2. Code Smells:
3. Duplicated code – much of the time the code is not exactly duplicated, but very similar, maybe a parameter or two can differentiate them.
4. Long method – no hard and fast rules, but over 20 lines is usually a bad sign, less than 10 is good.
5. Switch statements – the cases in a switch statement contain different logic
6. Solution: You have a code fragment that can be grouped together to turn the fragment into a method whose name explains the purpose of the method.
7. Benefits:
8. An extracted method may be able to be resused.
9. By using a descriptive name for the extracted method it makes the higher-level code easier to read and easier to maintain over time. It encourages self-documenting code through good organization.
10. Debugging is easier, especially after the refactored code has been modified because we can have unit tests on the extracted methods, providing a more efficient way to detect errors.
11. Encourages best coding practices by emphasizing cohesion by using discrete, reusable methods.

**Example 1 – Before Refactoring**

Code Smell – Long method, about 30 lines of actual code statements.

**public** **void** assess() {

**if**(expectedOutput.size()!=actualOutput.size()) {

**throw** **new** RuntimeException("expectedOutput and actualOutput not the same size");

}

isCorrect = **new** ArrayList<>();

numCorrect = 0;

numTotal = 0;

**for**(String exp : expectedOutput) {

**if**(exp.substring(0, 2).equals("%d")) {

String[] tokensExp = exp.split(" ");

**double** valExp = Double.*parseDouble*(tokensExp[1]);

String tolType = tokensExp[2];

**double** tol = Double.*parseDouble*(tokensExp[3]);

**double** errorMax=0.0;

**if**(tolType.equals("%tp")) {

errorMax = valExp \* tol/100.0;

}

**else** **if**(tolType.equals("%ta")) {

errorMax = tol;

}

String[] tokensAct = actualOutput.get(numTotal++).split(" ");

**double** valAct = Double.*parseDouble*(tokensAct[1]);

**double** errorAct = Math.*abs*(valExp-valAct);

**if**(errorAct<=errorMax) {

numCorrect++;

isCorrect.add(**true**);

}

**else** {

isCorrect.add(**false**);

}

}

**else** {

**if**(exp.equals(actualOutput.get(numTotal++))) {

numCorrect++;

isCorrect.add(**true**);

}

**else** {

isCorrect.add(**false**);

}

}

}

pointsEarned = (**double**)numCorrect/numTotal \* pointsMax;

}

**Example 1 – After *Extract Method* applied several times**

Now has exactly 10 lines of code and easy to read (assuming you know the context)

**public** **void** assess() {

compareExpAndActOutputSizes();

initializeInstanceVars();

**for**(**int** i=0; i<expectedOutput.size(); i++) {

String exp = expectedOutput.get(i);

String act = actualOutput.get(i);

**if**(hasDoubleResult(exp)) {

assessDouble(exp,act);

}

**else** {

assessString(exp,act);

}

numTotal++;

}

pointsEarned = (**double**)numCorrect/numTotal \* pointsMax;

}

Several of the extracted methods above are further refactored with an extracted method:

**private** **void** assessDouble(String exp, String act) {

**double** errorAct = getActError(exp, act);

**double** errorMax = getMaxError(exp);

**if**(errorAct<=errorMax) {

numCorrect++;

isCorrect.add(**true**);

}

**else** {

isCorrect.add(**false**);

}

}

**Example 2 – After *Extract Method* applied several times**

**public** **static** **void** main(String[] args) {

TestEngine testEngine = **new** TestEngine();

**if**(testEngine.shouldGenerateExpectedResults) {

testEngine.generateExpectedResults();

}

**else** {

testEngine.generateActualResults();

}

}

**public** **void** generateActualResults() {

buildExpectedGradeReport ( readExpectedResults(expectedResultsFile));

ArrayList<ArrayList<String>> actualTestResults = generateTestResults();

addActualResults(actualTestResults);

gradeReport.assessTests();

**if**(shouldSaveStudentReport) {

saveResults(gradeReport.toString(), studentReportFile);

}

}

**public** **void** generateExpectedResults() {

ArrayList<ArrayList<String>> testResults = generateTestResults();

gradeReport = buildExpectedGradeReport(testResults);

String expectedOutput = gradeReport.expectedOutput();

saveResults(expectedOutput, expectedResultsFile);

}

**Code Smell: Feature Envy**

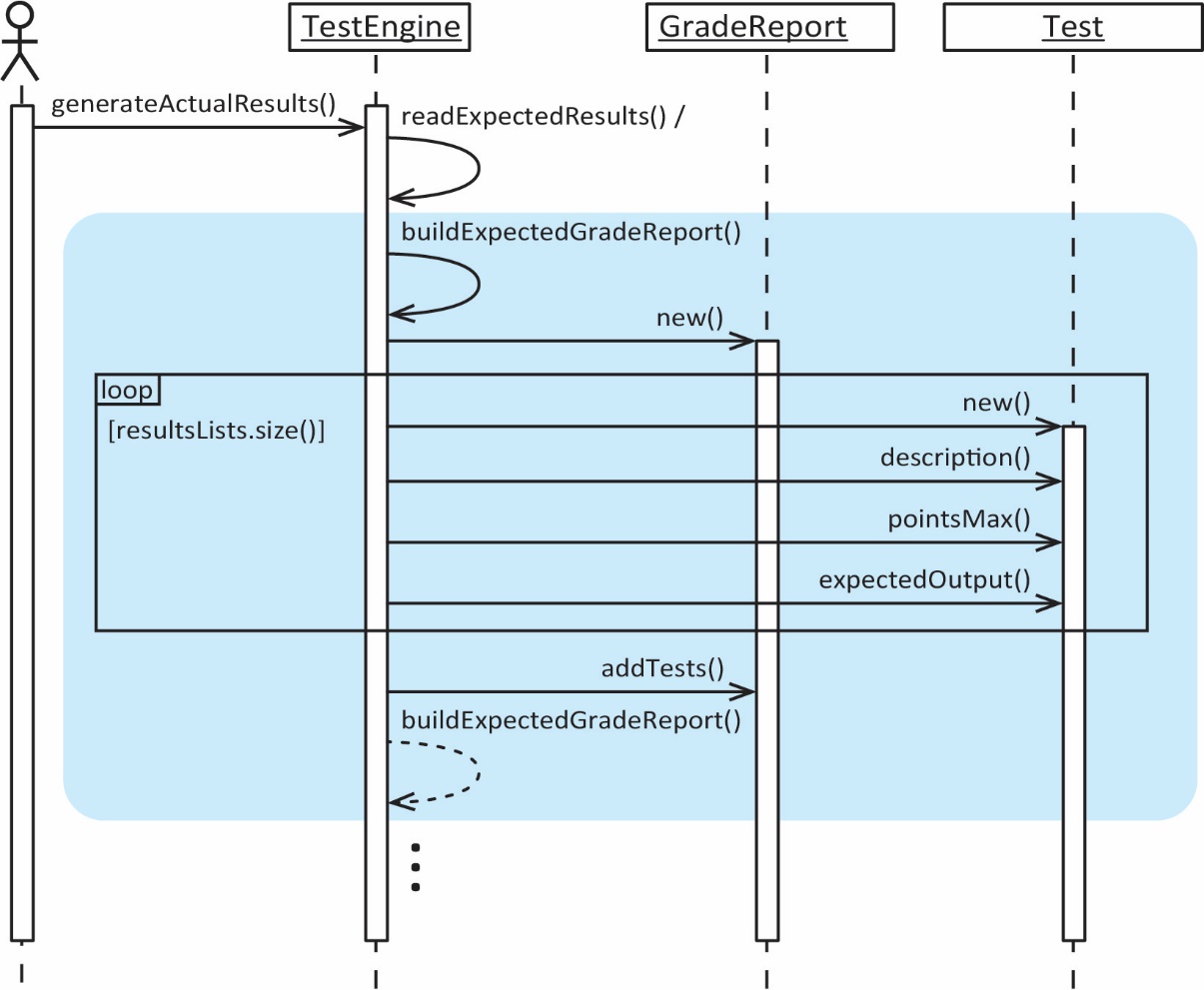
1. Feature envy

* A term used to describe a situation in which one object gets at the fields of another object in order to perform some sort of computation or make a decision, rather than asking the object to do the computation itself.
* A method accesses the data of another object more than its own data.

1. Solution: Use either the Extract Method, Move Method, or Move Field refactoring.

**Example 3 – Before Refactoring**

The *buildExpectedGradeReport* method has serious feature envy! It is envious of GradeReport and Test.



**private** GradeReport buildExpectedGradeReport(ArrayList<ArrayList<String>> resultsLists) {

gradeReport = **new** GradeReport();

**for**(ArrayList<String> results : resultsLists) {

Test test = **new** Test();

test.description = results.remove(0);

test.pointsMax = Double.*parseDouble*(results.remove(0));

test.expectedOutput.addAll(results);

gradeReport.tests.add(test);

}

**return** gradeReport;

}

In this refactoring, in some sense we are doing *move method* refactoring. However, we don’t actually move the method, we get rid of it completely . We do this by farming the work to the *GradeReport* constructor and the *Test* constructor:

**public** GradeReport(ArrayList<ArrayList<String>> resultsLists) {

**for**(ArrayList<String> results : resultsLists) {

gradeReport.tests.add(**new** Test(results));

}

}

**public** Test(ArrayList<String> expectedResults) {

**this**.description = expectedResults.remove(0);

**this**.pointsMax = Double.*parseDouble*(expectedResults.remove(0));

**this**.expectedResults = expectedResults;

}

