Product Metrics for Automatic Identification of "Bad Smell" Design Problems in Java Source-Code

Matthew James Munro
University of Strathclyde
Glasgow UK
Matthew.Munro@cis.strath.ac.uk

Abstract

Refactoring can have a direct influence on reducing the cost of software maintenance through changing the internal structure of the source-code to improve the overall design that helps the present and future programmers evolve and understand a system. Bad smells are a set of design problems with refactoring identified as a solution. Locating these bad smells has been described as more a human intuition than an exact science.

This paper addresses the issue of identifying the characteristics of a bad smell through the use of a set of software metrics. Then by using a pre-defined set of interpretation rules to interpret the software metric results applied to Java source-code, the software engineer can be provided with significant guidance as to the location of bad smells.

These issues are addressed in a number of ways. Firstly, a precise definition of bad smells is given from the informal descriptions given by the originators Fowler and Beck. The characteristics of the bad smells have been used to define a set of measurements and interpretation rules for a subset of the bad smells. A prototype tool has been implemented to enable the evaluation of the interpretation rules in two case studies.

Keywords: *Refactoring, Object-Oriented Designs, Software Metrics.*

1. Introduction

The design of source-code has become an increasingly important part of the overall development of software. Refactoring changes the internal code structure of an Object-Oriented (O-O) system without affecting the overall behaviour of the system to improve the quality of the design [6]. Refactoring has a role to play in reverse and re-engineering systems by making semantic-preserving transformations of code into a form that the software engineer finds easier to understand.

The agile software development process eXtreme Programming (XP) devised by Beck [1] is an incremental approach to software design. The design exists as a storyboard of events that encapsulate small requirements of the problem that are then integrated into the code at each build. Refactoring is an integrated part of XP to help integrate new functionality into the design at each build.

Refactoring is starting to become an integrated part of other software development processes to improve the design, help make design changes, integrate new functionality, and help understand the underlying design concepts.

Fowler [6] has produced a catalogue of 72 possible refactorings, describing the whole process so methods can be applied manually. Each refactorings complexity is broken down into five more manageable sections to help understand the motivation and mechanics how to apply the low-level transformation to source-code.

The process of refactoring has three distinct stages to its application: identify where to apply a refactoring, choose an appropriate refactoring as a solution and apply the refactoring. Current software tools and Fowler's description of 72 refactorings only consider the final stage of applying refactoring methods automatically and manually. Knowing where an appropriate place and which refactorings to apply in a system is arguably quite a challenge because the motivation for its application can vary.

One particular motivation is to improve the design of a software system through locating problems in the design and using refactoring as a solution. Fowler and Beck [6] define 22 colloquially named bad smells that describe a design problem that have a number of related refactorings that can change the structure of a system to help improve the design. However locating bad smells currently involves manually inspecting source-code, which quickly becomes unfeasible as the size of the system increases. Providing an automatic support for the detection of bad smells becomes quite appealing.

The motivation for this paper is to enhance the well-established refactoring process of identifying where to apply refactorings in a system. The focus will be on automatically identifying bad smell design problems in Java source-code. To achieve the goal a prototype tool is developed that applies a set of software metrics on Java systems and the results are interpreted to identify problems in the design (i.e. bad smells).

The rest of the paper is organised as follows: Section 2 discusses related work. Section 3 defines a template to segregate Fowler and Beck's [6] informal description of each bad smell to make it easier to understand the core design issues of the problem. Section 4 identifies measurements and interpretation rules for each bad smell, Section 5 describes how a semi-automated system is developed to enable a software engineer to interpret the measurements and identify possible candidates for refactoring. Section 6 evaluates the effectiveness of the identification of the bad smells in source-code. Section 7 concludes.

2. Related Work

The nature of applying refactorings is very much language specific and there is an increasing number of specialised software tools to handle various programming languages and any of the three stages of the refactoring process. This section first describes the functionalities of some of the software tools that can apply any of the refactoring process and secondly identifies current relevant literature relating to the application of refactoring.

2.1. Refactoring Support

There are a number of different IDE tools from well established commercial products (xRefactor [23], jFactor [9], jRefactory [11], jBuilder [8], Together [22]); or from open-source (Eclipse [5]); or from academia (Refactoring browser [2]). All of these tools help to automate applying refactorings, where the user is interactively involved in the process.

The current trend is for software tools is to give the developer support for locating potential areas in source-code that would benefit from being refactored. PMD [17] scans Java source-code for violations of rules that are related to design problems. PMD has the functionality to develop rules that can specifically encapsulate the requirements of a design problem.

Together [22] provide audits that are rules which source-code should conform to and metrics to analyse a systems source-code to help enforce company standards and conventions, or to improve the quality of features or a systems design. Violations of audits or thresholds from metric results are highlighted and can be examined manually to decide if the problem should be corrected. Also CodePro [4] is similar with over 500 audit rules and metrics to obtain information about a system.

Structural Analysis for Java (SA4J) [21] analyses structural dependencies of Java applications in order to measure their stability. It detects structural anti-patterns and provides dependency web browsing for detailed exploration. There is a "what-if" analysis available to access the impact of change on functionality of an application and offers guidelines for package refactoring.

The PROblem DEtector O-O System (PRODEOOS) [18] detects design flaws in C++ and Java systems. The tool uses the analysis based upon the detection strategy developed by Marinescu [12] that helps to quantify the description of a design flaw using software metrics.

2.2. Current Literature

The most major up to date publication on the whole field of refactoring is a survey by Mens and Tourwé [14].

Chatzigeorgiou et al. [3] evaluate an O-O design in terms of responsibility distribution among classes. The Hyperlink Induced Topic Search (HITS) algorithm (a method of link analysis) is extended to identify the number of discrete messages exchanged between classes. The extended HITS algorithm calculates an *authority* value representing the number of classes that send a message to a class and a *hub* value that represents the number of classes a class sends messages to. Modelling these attributes for a class on a collaborations diagram can identify a possible *god class* design heuristic. A class with a high *hub* value requests services from the rest of the system and is considered a behavioural *god class*. Whereas a high *authority* value means a class receives a number of messages and can be considered to be a data structure *god class*.

Marinescu defines detection strategies that relate to a specific design flaw. A detection strategy has four sequences of steps: analysis of the problem, selection of metrics, detection of candidates and examination of candidates. An identified problem taken from the literature is analysed to quantify the informal description. Using the quantitative description a selection of metrics are chosen that best match the problem's characteristics, and it is here where the detection strategy is expressed using the identified metrics. The last stage examines the results that the detection strategy identified using the proposed, and whether refinements are required [12].

Marinescu and Raţiu [13] use the Factor-Strategy Model (FSM) that quantifies a systems design conformance to a set of design principles, rules and heuristics. The FSM uses the *detection strategies* defined by Marinescu [12] to identify design problems that can be connected with quality attributes of a systems design.

2.3. Summary

Marinescu's [12] research defines detection strategies that start to encapsulate the requirements for automatically locating 14 design flaws within source-code. However, there is no justification for choosing the metrics, thresholds and the combination of metrics and thresholds defined in the detection strategies. It may be inappropriate to use an absolute filtering mechanism to interpret the metric results as they may be too restrictive on the results. It is vital to know if the chosen metrics and thresholds encapsulate the underlying design problem correctly.

A refinement of Marinescu's [12] detection strategies would need to include justification for choosing the metrics for a design problem with additional empirical evidence to show different techniques for interpreting metric results. Where required, a specific software measurement may have to be designed to encapsulate a bad smell characteristic. Any unique measurement technique designed requires a rigorous definition to exclude ambiguous interpretation.

The approach taken in the work presented in this paper use a combination of conventional metrics and some new metrics to identify ten bad smells. This paper presents the identification of two (*Lazy Class* and *Temporary Field*) of these ten bad smells as they are straightforward design problems and emphasis is made on the description of the automatic detection procedure and empirical evidence from two case studies.

The measurement techniques used in this paper are restricted to only consider a single software build as the assumption is made that the source-code of a system can be obtained. Older builds of a system's source-code may not be obtainable as this depends how the software was developed.

3. Bad Smells

Fowler and Beck are the originators of the colloquially named bad smell design problems and present the problems in an informal essay style to guide a human developer manually to locate within a system. However, the informal description of bad smells lacks enough information that then makes it difficult to identify automatically. A more formal definition for each of the bad smell design problems is required so the important characteristics can be related to attributes which can then be measured.

A framework template has been defined to segregate Fowler and Beck's [6] informal description of the bad smells. The framework template has three main parts as described in Figure 1.

Bad Smell Name	A quote from Fowler and Beck's [6] informal description of the problem.
Characteristics	Main characteristics identified from the informal description, and identifying any correspondence that occurs from other literature.
Design Heuristics	Identifying any design heuristics from the literature relevant to the problem.

Figure 1. Bad Smell Description Framework Template

Each bad smell starts with a quote-summarising Fowler and Beck's informal description of the design problem. Then the unique characteristics of the problem are described and where possible other interpretations from the literature relating to the characteristics are identified. Design heuristics taken from the literature relating to the characteristics are identified. The main authors used are: Riel [19], Gibbon [7], Johnson and Foote [10], and Parnas [16].

Some of the bad smell descriptions lack substantial information to realistically fulfil all the subsections of this template. In such cases only the Design Heuristics are not included.

3.1. Lazy Class

Bad Smell Name *A class that isn't doing very much should be eliminated [6].*

Characteristics Any class has a development and maintenance cost. Therefore any class should justify its cost and any that do not should be eliminated. Fowler and Beck suggest that a *Lazy Class* could occur when a class is implemented with the future in mind where the new functionality is never added [6].

However, another reason why a class could have *Lazy Class* attributes is that it may have been down-sized through refactoring [6]. Fowler and Beck's observation is interesting that using refactoring methods to improve the design of a system can inadvertently introduce a bad smell. Some bad smells make classes smaller so that they perform simple tasks and as a by-product may produce classes with *Lazy Class* characteristics.

Design Heuristics These are classes that do not do very much but should not be confused with a *Data Class*.

3.2. Temporary Field

Bad Smell Name *Sometimes you see an object in which an instance variable is set only in certain circumstances [6].*

Characteristics Fowler and Beck mention a class's source-code is difficult to understand when instance variables are only used in certain places [6]. Fowler and Beck give some insight why instance variables are created and only used in certain methods. One reason is because a complicated algorithm may require several variables and the developer wanted to avoid passing around a large parameter list [6]. Another case is when a programmer has not bothered to make a variable local. An extreme case of this can be a Graphical User Interface (GUI) where there exist a large number of instance variables to hold the visualisations.

In summary the characteristics are: instance variables of a class that are only used in certain places and a where a small set of instance variables are referenced by a small set of methods.

Design Heuristics If *Temporary Field* bad smell exists then it could identify that there are a number of instance variables defined in a class which Fowler and Beck [6] suggest to be an attribute of the *Large Class* bad smell.

4. Automatic Detection of Bad Smells

This section describes a process that uses the characteristics identified for each bad smell to help automate locating the design problem in source-code.

Fowler and Beck state that locating bad smells is more human intuition than an exact science. This paper will build on contributions from other researchers to devise a set of software metrics that can be used to identify a subset of bad smells in source-code. An interpretation rule framework has been defined and described in Figure 2.

Bad Smell Name	A quote from Fowler and Beck's [6] informal description of the problem.
Measurement/ Process	Describe possible measurement techniques that when applied to Java source-code can help identify the design problem.
Interpretation	The interpretation indicates a set of rules on how the metrics can be used to identify possible candidates (and non-candidates).

Figure 2. Bad Smell Interpretation Rule Framework Template

The metrics identified as part of the measurement/ process are a combination of conventional metrics and new metrics. New metrics are sometimes required to be developed to measure a specific characteristic. Also the interpretation will discus possible results that can be obtained. A true-positive or a true-negative candidate indicates that the rules work, whereas a false-positive indicates that the rules select a candidate that is not a bad smell. A false-negative indicates that the rules do not work in that they do not identify a candidate that is a bad smell.

There may exist many bad smells in one system and some bad smells may be used as a characteristic for another to exist. These issues are considered to be out of scope for this paper and are not discussed.

The ordering of the result of the rules and metrics are an important part of the interpretation process.

4.1. Lazy Class

Bad Smell Name *A class that isn't doing very much should be eliminated [6].*

Measurement/ Process The main problem is how to interpret if a class is doing enough work. Counting the number of methods (NOM), in conjunction with the classes Weighted Methods per Class (WMC) and LOC in a class is a reasonable approach.

A possible opposite link with the *Large Class* bad smell, would mean using the same metrics but with the inverse interpretation. This follows on from using Fowler and Beck's informal descriptions that a *Large Class* has too much functionality and a *Lazy Class* has too little functionality. Using this notion the coupling (CBO) measure can also be considered.

Interpretation

```
if
    $NOM = 0$
    then PRINT YES
else if
    $(LOC < LOC_{Median}) AND (\frac{WMC}{NOM} <= 2)$
    then PRINT YES
else if
    $(CBO < CBO_{Median}) AND (DIT > 1)$
    then PRINT YES
else
    PRINT NO.
```

Using the inverse interpretation of the *Large Class* rules are inappropriate to identify a *Lazy Class* as they only consider a class with low LOC, NOM, and complexity (WMC) is inadequate to give an overall interpretation that the operations of a class can be considered minimal. However still using the same metrics from *Large Class* interpretation rules but considering the average method complexity within a class will help to give a perspective on the operations of a class.

A class with no methods is considered to be a *Lazy Class* because it appears to carry out no operations. The next part of the rule considers small classes with below median LOC and an average complexity for each method to be less than or equal to the value 2. This latter value was chosen as it allows some methods to have a simple *if*, *while* or *switch* statement but no other complexity hence can be considered as being lazy.

The final part of the rule considers how the class is connected to other classes and a CBO count less than the median of the population was chosen as a reasonable comparison because a *Lazy Class* should not have too many connections to other classes. The Depth of Inheritance Tree (DIT) comparison indicates that *Lazy Class*es tend to be subclasses.

It has already been acknowledged that it is difficult to recognise the actual functionality of methods in classes and hence difficult to access a class for being a *Lazy Class*. The interpretation rules will produce a number of false-positive results because of the conservative use of the median of the measures. Classes with high complexity will not be candidates and hence will produce true-negative results. The median values were chosen to restrict the scope by a half of the classes to be considered.

4.2. Temporary Field

Bad Smell Name *Sometimes you see an object in which an instance variable is set only in certain circumstances [6].*

Measurement/ Process A simple measure to encapsulate the characteristics of this design problem is to measure the number of methods that reference each instance variable defined in a class (IVMC).

Interpretation For each instance variables in a class:

```
if
    $(IVMC <= 1)$
    then PRINT YES
else
    PRINT NO
```

then the class contains a *Temporary Field*.

A zero value of IVMC means that an instance variable is not used within any method in a class. An IVMC result of one means that the instance variable is only referenced in one method in a class and should be considered to be a *Temporary Field*.

A candidate may be a false-positive and can result from the situation where one method only references an instance variable that is public and this may then be directly referenced from another class. A non-candidate may be a false-negative result where instance variables are referenced more than once in more than one method but still may be temporary variables that should have be declared as local variables.

5. Implementation

This section outlines the processes for identifying bad smells in Java source-code as defined in Section 4. It does not give a complete description of any of the tools but does serve to show how tools can be constructed and used, and highlight some of the issues with these tools.

There are two main processes on how to locate bad smells in source-code, namely manual and automatic. The manual option is used by software engineers who follow the Fowler and Beck [6] bad smell chapter and recognises that bad smells are located through inspection of the code. This is a slow and error prone activity. The automation through the use of software tools to apply software metrics to systems can help to reduce the errors in their calculation and allow for larger scale systems to be analysed in a quick and repetitive manner.

The overall architecture of the prototype tool developed is shown in Figure 3 where Java source-code is input into a metrics generation tool where the output is a set of metrics which are then input into the results generation tool. The final output is a set of possible candidate classes or methods for each of the bad smells. It is at this point that the software engineer will have to inspect the source-code to check for true-positive or false-positive results.

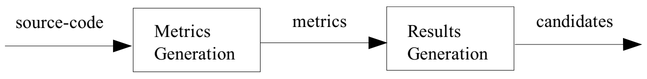

Figure 3. The Architecture of the Prototype Tool.

The metrics generation process analyses Java source-code through parsing it, then the metrics are calculated and then placed into a metrics repository. The metric framework plugin [15] for IBM's Integrated Development Environment (IDE) Eclipse [5] is extended to be the prototype tool. The metric plugin uses Eclipse's library methods to access an Abstract Syntax Tree (AST) from a parsed Java system that can be used to measure various attributes of a system that can be brought together as metrics. The metric results are stored in a repository that can be accessed through the tools metric-view table representation or exported as an XML file.

An in-house tool developed in at research group at the University of Strathclyde, known as TypEx [20] parses the XML metric result file produced from the Eclipse metric framework plugin and transform the information into a format more amenable to analysis. For example, a spreadsheet.

This section has described the implementation of the prototype tool. The whole process of applying, extracting and displaying metric information on a Java software system is shown in Figure 4. A system is analysed using the Eclipse metric framework where the results can be visualised both through the task-view or the metric-view table and eventually through the Excel spreadsheet.

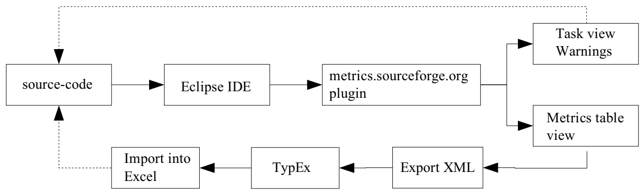

Figure 4. The Process to Extract Metric Measurement on Java source-code.

6. Evaluation

This section presents the results of an evaluation of the interpretation rules to automate the detection of the bad smells defined in Section 4, and implemented in Section 5.

Two case studies are presented. The first is a small simple hotel booking system (Hotel) written in Java and developed in house at the University of Strathclyde with 1,500 LOC, 13 classes, 124 methods all implemented within one package. The Hotel system allows the reservations and maintenance of hotel room bookings. The size of the Hotel system allows the results of the interpretation rules to be manually inspected. The second system is a medium sized Graph Tool system written in Java by a third party with 16,000 LOC, 730 methods and 84 classes. The Graph Tool system enables nodes and arcs of a graph to be inputted and displayed using simple layout algorithms. It also enables the graphs to be manually manipulated and nodes to be coloured and collapsed into composite nodes. The Graph Tool is used to give some indication as to how the method used scales up when applied to an unknown piece of software.

A framework has been devised to help evaluate the results from each case study. The framework is described in Figure 5.

Bad smell name	The name of the design problem to be evaluated.
Results	The metrics and interpretation rules identified in Section 4 relating to the design problem are applied.
Interpretation	The set of results are analysed using the true-false table from Figure 6, where a number is placed into each of the four boxes to represent how the results can be interpreted.

Figure 5. Bad Smell Evaluation Framework Template

In some cases it is impractical to show all the results from a system, as the volume is too large. In such cases the top results that are considered interesting are shown.

A full representation how all the results from each bad smell interpretation rule of a system can be interpreted are shown in a true-false table that is explained in Figure 6. A number is placed into each of the four squares in a true-false table corresponding to how all the results from a system can be interpreted after being manually inspecting the source-code with problem description outlined in Section 3. For example the Hotel system has 7 true-positive *Lazy Class*es.

However, it becomes unfeasible to manually assess all of the results manually for the Graph Tool medium sized system. In these cases the interpretation section from the framework and the truth-false table are not included.

An overview of the results is discussed to show how they relate to the design problem. This is achieved through manually inspecting each class or method shown as a true-positive and a selection of the false-positives results to ascertain if the interpretation rules have correctly identified the bad smell. Descriptions of some of the classes or methods classified in any part of the true-false table are justified to be worthy of such interpretation. The true-false table of results is discussed in relation to the strength of the interpretation rules.

6.1. Lazy Class

6.1.1. Small System (Hotel System)

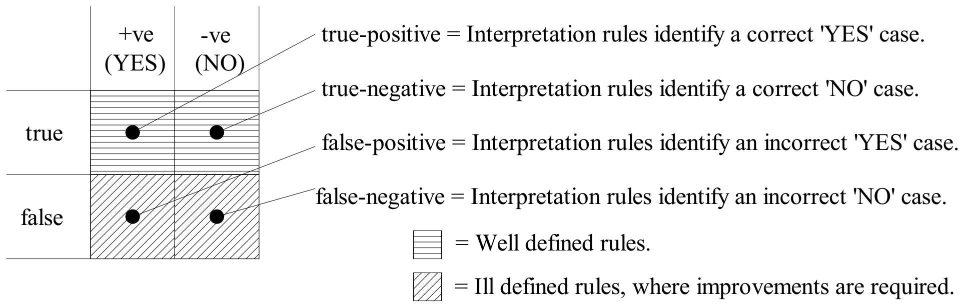
+ve (YES)
-ve (NO)
true
false
true-positive = Interpretation rules identify a correct 'YES' case.
true-negative = Interpretation rules identify a correct 'NO' case.
false-positive = Interpretation rules identify an incorrect 'YES' case.
false-negative = Interpretation rules identify an incorrect 'NO' case.
= Well defined rules.
= Ill defined rules, where improvements are required.

Figure 6. The different states in which an interpretation rule can take.

Lazy Class							
Mean	112.85	9.54	22.31	1.46	3.00		
SD	152.06	8.16	27.89	0.97	2.00		
Median	47.00	6.00	9.00	1.00	3.00		
Upper Quartile	86.00	9.00	16.00	1.00	4.00		
10th Percentile	21.00	4.00	4.60	1.00	0.20		
90th Percentile	297.80	24.00	72.80	2.80	5.80	**Yes Count**	8
95th Percentile	408.80	25.00	80.40	3.40	6.00		
Classes	**LOC**	**NOM**	**WMC**	**DIT**	**CBO**		**Interpretation**
BillItem	18	4	4	1	1		YES
HotelSystem	19	1	1	1	0		YES
Bill	29	6	7	1	0		YES
Delegate	30	6	8	2	4		YES
Guest	36	7	7	1	3		YES
Function	40	6	10	1	3		YES
FunctionDate	47	4	8	4	2		YES
HotelDate	53	5	9	3	2		YES
ConferenceRoom	74	6	16	1	5		NO
Reservation	86	9	15	1	3		NO
Room	177	25	44	1	4		NO
Hotel	328	20	81	1	6		NO
HotelUI	530	25	80	1	6		NO

Table 1. *Lazy Class* metric results from the Hotel System.

Results Table 1 shows the results at the class level and indicate 8 candidates (YES result).

Interpretation

	+ve	-ve
true	7	5
false	1	0

The one false-positive result is that for the *HotelSystem* class that contains one method (main) that is the entry point to the Hotel system.

The *BillItem* and *Guest* classes are essentially *Data Class*es and only have constructors and getter methods. They are thus also *Lazy Class*es according to the definition.

The remaining classes (*Bill*, *Delegate*, *Function*, *FunctionDate*, *HotelDate*) consist of primarily of constructor, setter, and getter methods and only one method that carries out any computation. Again, these are considered as true-positive results.

Evaluation For the small Hotel system there were a large number of *Lazy Class*es identified two of which were correctly identified as *Data Class*es. The false-positive result arose through the main class of the system.

6.1.2. Medium System (Graph Tool)

Lazy Class							
Mean	218.57	10.03	47.36	3.42	4.81		
SD	421.61	15.77	105.48	2.55	3.39		
Median	107.50	5.50	15.50	2.00	4.00		
Upper Quartile	197.75	8.75	36.50	6.00	7.75		
10th Percentile	12.90	1.00	1.30	1.00	1.00		
90th Percentile	480.20	20.40	121.70	7.00	10.00	**Yes Count**	34
95th Percentile	686.45	33.75	210.85	7.00	10.00		
Classes	**LOC**	**NOM**	**WMC**	**DIT**	**CBO**		**Interpretation**
Debug	4	0	0	1	0		YES
Field	4	1	1	1	0		YES
IntWrapper	8	1	1	1	5		YES
GraphTool	9	1	1	1	0		YES
BatchProcessor.BatchCommand	10	1	1	1	2		YES
LinkList.LinkNode	12	1	1	1	7		YES
CanvasScrollControl	12	2	2	1	2		YES
TokenString	12	3	3	1	1		YES
UnexpectedTokenException	15	3	3	3	2		YES
UnknownTokenException	15	3	3	3	2		YES
UniqueID	16	3	3	1	1		YES
ColoredSquare	17	4	4	1	3		YES
ConsoleWindow	28	2	3	5	3		YES
GraphToolHelp	31	1	2	6	0		YES
GroupNodeMenu	31	5	7	7	4		YES
GraphToolHelp.HtmlPane$PageLoader	36	2	4	1	2		YES
PreferencesDialog.ColumnLayout	36	5	7	1	3		YES
Statics	37	1	1	1	0		YES
EdgeDirectionMenu	40	6	10	7	4		YES

Table 2. *Lazy Class* metric results from the Graph Tool System.

Results The Table 2 shows the results at the class level and indicate 34 candidate classes (45.9%)(YES result) and the rest to be non-candidates (NO result).

Evaluation The *Debug* class is a true-positive since it only contains an instance variable. The name of the method identifies that it is used for testing purposes and would have little impact on the running functionality of Graph Tool. The class *Field* is an abstract class and can be considered to be a false-positive. The class *IntWrapper* has one instance variable and one constructor method and therefore is a true-positive as it does not contain any functionality.

The *GTColorMenu*, *SelectionMenu* and *CommonColorMenu* classes are part of Graph Tools GUI with a number of methods with condition statements and are thus considered to be true-negative results.

Ordering the classes with the lowest metric values helps to identify the smallest classes of Graph Tool to see of they are true-positives.

6.2. Temporary Field

6.2.1. Small System (Hotel System)

Temporary Field			
Mean	12 8		
59	4233		
Me60an	42SS		
D77edUpadr0e	. 2SS		
3SrQued enr0e	I 2SS	**Yes Count**	I
8SrQiued enr0ei	h2 S		
Instance Variables	**IVMC**		**Interpretation**
Pcre 2Q9 cpBe	3		I m5
Pcre 2Q5 0nYe	3		I m5
EcnHdent eo ccy 2asa0aBegat 000eG	I		FC
EcnHdent eo ccy 2y af 5ear0nY	I		FC
gpnt r0cn2Hat 000eGDGe6	I		FC
gpnt r0cn2cdYan0Ged	I		FC
o ccy 2dF py Bed	4		FC
Pcre 2y ey cd0a Pa	4		FC
RpeGr2YEad6	4		FC
RpeGr2YFay e	4		FC
RpeGr2Yo ccy G	4		FC
RpeGr2Y5 radr	4		FC
RpeGr2Y5 rc7	4		FC
gpnt r0cn2npy CH9 e eYareG	4		FC
gpnt r0cn2Gradr	4		FC
gpnt r0cn2Grc7	4		FC
N0 0rey 2BEcGr	4		FC
N0 0rey 2B9 eGt d07r0cn	4		FC
o ccy 2dvy7e	1		FC

Table 3. *Temporary Field* metric results from the Hotel System.

Results The Table 3 shows the results at the class level and indicates two candidate (YES result) instance variables that are contained in the same class.

Interpretation

	+ve	**-ve**
true	2	33
false	0	0

The instance variables *Hotel.hDouble* and *Hotel.hSingle* are only referenced once within the *Hotel* class both by the constructor method *Hotel(int noSingle, int noDouble)*. These are true-positive results indication that there is a problem in the *Hotel* class.

Evaluation The interpretation rule used has correctly identified a problem with *Temporary Field* in the *Hotel* class. It also indicated that the remaining instance variables are being used in the correct manner.

6.2.2. Medium System (Graph Tool)

Laz y CI GsYeoaun			
Mean	12 8		
59	1243		
Me60an	12SS		
D77edUpadr0e	82SS		
QSruit ed enr0e	Q2SS	**t aONN MWD**	4S
hSruit ed enr0e	. 2SS		
IWOD WTaYBI Gl r uaO	**IBp N**		**IWDaGyGaDI Do W**
t d0nrt dePedenl ect ane 2P0eBal e	S		mY5
Earl ut dHI eccHd2Earl uo HI I an62 HI I an6	Q		mY5
Earl ut dHI eccHd2Earl uo HI I an627adal eredc	Q		mY5
o yys 7r0Hnc9 0a Hg2 s 7r0Hnc	Q		mY5
o HncHeG 0n6HF 2cl dH t ane	Q		mY5
o HnreOrt dePedenl ect ane 2enaf es 7r0Hnc	Q		mY5
o HnreOrt dePedenl ect ane 2F 0n6HF s 7r0Hnc	Q		mY5
o HnreOrt dePedenl ect ane 2RHHI s 7r0Hnc	Q		mY5
o HnreOrG 0n6HF 2HPPNda7u0 c	Q		mY5
9 0c7 aOvdal e2f arl uv0eBal e	Q		mY5
9 0c7 aOvdal e2uHd0R5 I dH Ead	Q		mY5
9 0c7 aOvdal e2acrt d0nrNda7u9 0d	Q		mY5
9 0c7 aOvdal e2yedr5 I dH Ead	Q		mY5
Y6ge9 HI kMenp2cl H7e	Q		mY5
Y6geMenp2e6ge9 0del r0HnMenp	Q		mY5
Y6geMenp2e6gevdHI o HHdMenp	Q		mY5
Y6geMenp2e6gey0neo HHdMenp	Q		mY5
Y6geMenp2e6geTeOro HHdMenp	Q		mY5

Table 4. *Temporary Field* metric results from the Graph Tool System.

Results The Table 4 shows the results at the class level and indicates 80 candidate instance variables contained within 22 classes (YES result).

Evaluation The *PrintPreferencesPanel.fileName* instance variable is a true-positive as the IVMC value of zero means that it is not referenced in the body of the class and should be refactored out. Both the *BatchProcessor.BatchCommand.command* and *BatchProcessor.BatchCommand.parameters* instance variables are defined in the same class and initialised in the only method in the class which is a constructor. These are false-positives.

The *BatchProcessor.enum* instance variable is used in two method bodies that are not constructors and is thus a true-negative. The *CanvasScrollControl.canvas* instance variable is used in both two methods in the class, one is the constructor and the other is where the whole method body updates the state of the instance variables. The class *CanvasScroll* is a *Lazy Class* and the instance variable is in every method and is thus a true-negative result. The *CLLOptionsDialog.checkCllNormalizeNames* is referenced in two methods, one of which is a constructor. The other method checks the state of the instance variable. Again, this is a true-negative result.

Constructor methods should be excluded from IVMC calculation as they initialise the state of an instance variable.

7. Conclusion and Future Work

The research in this paper has addressed the issue of automating the detection of bad smells in O-O Java systems. Fowler and Beck have informally described bad smells in code as bad or inconsistent parts of the design of an O-O system. Refactoring is a technique that is central to the XP development method and provides a set of code level transformations to maintain the design of a system as it rapidly evolves. Thus refactoring can be applied to bad smells.

This paper has presented the results of an evaluation of the automatic detection of two bad smells. Section 3 has defined a simple framework to extract the main characteristics of bad smell design problems. Section 4 define interpretation rules that use existing and extended metrics to identify candidates in the results. Section 5 described how a semi-automated system was developed to apply the metric part of a bad smell interpretation rule and how the results were exported into a spreadsheet so the rest of the interpretation rule could be applied. Section 6 shows the effectiveness of the method developed as part of this research was evaluated using two case studies.

The research in this paper can be further extended through applying the interpretation rules to non-academic systems that will have numerous software build information to see if bad smells exist when for instance they are suppose to be refactored out. In addition, analysis of a system that has been developed using the XP methodology to see if there are any bad smells that exist in the source-code design.

The design of software is a skill and is largely a matter of personal opinion. Interpreting the informal descriptions of the bad smells can be done in many ways because they are open to interpretation because of the informality of their definition. This then makes it difficult to produce concrete interpretation rules and this paper has made an attempt at defining some of them.

8. Acknowledgements

This work has been supported by the Department of Computer and Information Sciences at the University of Strathclyde from a three-year stipend.

References

[1] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[2] J. Brant and D. Roberts. Refactoring Browser, 1999. visited January 2002, http://st-www.cs.uiuc.edu/users/brant/Refactory/.

[3] A. Chatzigeorgious, S. Xanthos, and G. Stephanides. Evaluating Object-Oriented Designs with Link Analysis. In *International Conference on Software Engineering*, Edinburgh, June 2004.

[4] CodePro, 1997. visited January 2004, http://www.instantiations.com/codepro.

[5] Eclipse, IBM, 2001. visited June 2003, http://www.eclipse.org.

[6] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[7] C. Gibbon. *Heuristics for Object-Oriented Design*. PhD thesis, University of Nottingham, October 1997.

[8] JBuilder, 2002. Version 8, visited June 2002, http://www.borland.com/jbuilder/.

[9] jFactor, 2000. visited June 2002, http://www.instantiations.com/jFactor.

[10] R. Johnson and B. Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June/July 1988.

[11] Jrefactory, 2001. visited June 2002, http://jrefactory.sourceforge.net.

[12] R. Marinescu. *Measurement and Quality in Object-Oriented Design*. PhD thesis, "Politehnica" University of Timişoara, October 2002.

[13] R. Marinescu and D. Ratiu. Quantifying the Quality of Object-Oriented Design: the Factor-Strategy Model. In *11th Working Conference on Reverse Engineering*, Deft, November 2004. IEEE Computer.

[14] T. Mens and T. Tourwé. A Survey of Software Refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, February 2004.

[15] Metrics, 2002. visited June 2003, http://metrics.sourceforge.net.

[16] D. Parnas. Designing Software for Ease of Extension and Contraction. *IEEE Transactions on Software Engineering*, 5(2):128–137, March 1979.

[17] PMD, 2003. visited June 2003, http://pmd.sourceforge.net.

[18] PRODEOOS, 2002. visited October 2003, http://allergy02.cs.utt.oo/prodeoos.

[19] A. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.

[20] G. Russell. Typex, 2004. visited January 2004, http://typex.dev.java.net.

[21] SA4J, 2004. visited January 2004, http://www.alphaworks.ibm.com/tech/sa4j.

[22] Together, 2002. visited June 2002, http://www.togethersoft.com.

[23] Xrefactor, 2000. visited June 2002, http://www.xref-tech.com.