Parallel Inheritance Hierarchy: Detection from a Static View of the System

Raúl Marticorena[1], Carlos López[1], and Yania Crespo[2]

[1] University of Burgos, Area of Language and Informatic Systems
{rmartico, clopezno}@ubu.es
[2] University of Valladolid, Department of Computer Science
yania@infor.uva.es

Abstract. We expose a case study of a bad smell detection through metrics. In practice, bad smell detection emerges from human observations. Metrics allow to obtain an objective view of the software, so they must be used as instruments to detect bad smells.
Concretely, we focus in the bad smell: *Parallel Inheritance Hierarchy*, using a metric subset. Although it is not a serious bad smell, however its detection is difficult in large and medium size systems. Besides, it is usually necessary to have several versions of the system to detect its presence.
We define a process to manage the big amount of data extracted from a system to determine where exists this bad smell, only with an available version. The saving of time and effort in this process is showed as an advantage opposite to other solutions.

Key Words: bad smell, metrics, refactoring inference, software evolution.

1 Introduction

Bad smell detection can be driven by metrics or human intuitions. Meanwhile the former solution seems objective, the second solution is subjective and difficult to automate. In our current research work, we are looking for an objective process to determine *when* and *where* the software system presents a bad smell. The aim is to define a process with a certain language independence so the process could be applied to a big family of object-oriented languages.

To obtain this aim, we must select the more adequate metrics to each bad smell, explaining a process to detect it from metric values. Two strict conditions are fixed in the proposal. We can only use language independent metrics and we have only one static view of the system (only one version of the system is available for the analysis).

From different products as library classes and frameworks we extract class metrics for each one of them. Using these values, and analyzing them with data mining techniques, we try to deduce where exist bad smells, therefore we will get refactoring opportunities.

The remainder of this paper is structured as follows: Section 2 presents the state of the art of the current problem, from metrics and refactoring inference to works connecting both subjects. In Section 3 we describe the process to detect a concrete bad smell: *Parallel Inheritance Hierarchy*. In Section 4 we show a case study applying the process previously established, and analysing the obtained results. Finally, in Section 5, we finish with conclusions and future work.

2 State of the Art and Current Trends

Bad smell detection based on metrics is established in [4] but in some cases it is difficult to fix a metric set for each bad smell. Other works [1] use combinations of metrics and other symptoms through logic meta-programming. The suitable refactorings for each bad smell are described in [2].

One of the current trends is to study the changes among several versions of the same system to detect bad smells as *Shotgun Surgery*, *Divergent Changes* and *Parallel Inheritance Hierarchy* [2]. Although there are works which detect this kind of situation, they need to observe several versions of the system [5, 6] inferring their results from the evolution of several versions.

Our proposal improves the previous works because it only needs one version. From one system static view, we can detect the presence of the bad smell *Parallel Inheritance Hierarchy*.

3 Process Definition

We detail the process stages to determine parallel inheritance hierarchies.

1. Obtain the class metric values of DIT (Depth Inheritance Hierarchy) and NOC (Number Of Children) [3] for each one of the classes.
2. Apply data mining to associate classes with related values obtaining clusters.
3. Observe the medium values and standard deviation which characterize each class cluster.
4. From the selected clusters, it is applied a filter on the DIT and NOC metric values taking those classes with similar or equal values. We also take those classes whose names have the same prefix, as [2] suggests.
5. Inspect the selected classes and their children to confirm the presence of the bad smell.

The process is not completely automatic, so it needs the human interaction in step 5. As follows, we present three different cases in which we apply this process. We also comment the obtained results.

4 Case Study

We have selected three different projects in Java: two libraries (`JFreeChart` and `jcoverage`) and one framework (`JUnit`). Metrics has been collected using the

`Metrics` [3] plug-in for Eclipse. Lately, we use `Weka` [4] (version 3.4) as data mining tool to determine clusters and membership of each class.

Taking classes with equal values and similar prefixes, we achieve a manual inspection of the suspect classes.

4.1 Case 1: JFreeChart Class Library

From a `jfreechart_1.0.0_pre2` version, with 629 classes, we extract metrics for each class. We establish the use of the two metrics (DIT and NOC) to detect this bad smell. Depending on the depth of inheritance tree and the number of children, we use these values as indicators of parallel inheritance hierarchies existence. More concretely, we choose classes with a number of children greater than 1, so the inheritance hierarchies are obviously complex.

Collecting the metric values and applying clustering techniques with `Weka`, we found four clusters (see Table 1).

Table 1. `JFreeChart-1.0.0_pre2` - Clusters

Cluster	Num.Classes	%	Mean DIT	St.Dev	Mean NOC	St.Dev
0	410	65%	2.8592	0.5164	0	1.4839
1	64	10%	5.1989	0.7940	0.1642	0.3704
2	128	20%	1.0478	0.2198	0.0921	0.3133
3	**27**	**4%**	**1.9991**	**0.9162**	**4.0688**	**3.4295**

Studying the different mean values and standard deviation for each cluster, we only focus in classes taking into account the mean values of DIT and NOC. We are looking for classes at the top of the inheritance hierarchy (DIT between 1 to 3) with a medium number of children (NOC greater than 4 in this case).

The rest of the clusters contain classes with high depth and without children (Cluster 0), very deep with few children (Cluster 1) or low depth with few children (Cluster 2). These three last clusters do not seem suitable to find parallel hierarchies.

Therefore, we take Cluster 3 with its 27 classes. To find parallel inheritance hierarchies we establish that classes must have values of DIT and NOC very similar. Also we added the criteria that class names must have similar prefixes.

By means of this process, we have detected three parallel hierarchies. We show the root classes and their metric values:

- Hierarchy 1
  - `Tick` (DIT=1, NOC=2)

[3] Available at http://metrics.sourceforge.net/
[4] Available at http://www.cs.waikato.ac.nz/ ml/index.html

    - `TickUnit` (DIT=1, NOC=2)
- Hierarchy 2
    - `AbstractCategoryItemLabelGenerator` (DIT=1, NOC=4)
    - `AbstractPieItemLabelGenerator` (DIT=1, NOC=2)
    - `AbstractXYItemLabelGenerator` (DIT=1, NOC=2)
- Hierarchy 3
    - `RenderedrState` (DIT=1, NOC=3)
    - `Plot` (DIT=1, NOC=12)

In **Hierarchy 2**, the NOC value includes two inner classes that they must not be considered to find the bad smell. In **Hierarchy 3**, similarity has been obtained by similar prefixes. Besides, the other nine child classes of `Plot` have not descendants, the other three classes have an association one to one with descendants of `RendererState` class.

4.2 Case 2: jcoverage Class Library

We repeat the process with a class library for white-box testing: jcoverage-1.0.5. It contains a number of 89 classes. Taking again the DIT and NOC metrics we obtain only one cluster due to the similarity among classes. We repeat the clustering, adjusting the number of clusters to two (see Table 2).

Table 2. `jcoverage-1.0.5` - Clusters

Cluster	Num.Classes	%	Mean DIT	St.Dev	Mean NOC	St.Dev
0	**9**	**10%**	**1.6249**	**0.9918**	**2.1823**	**1.0775**
1	80	90%	1.8083	1.0555	0	0.9608

Cluster 1 brings together classes with few or none children, so we choose Cluster 0. We apply the prefix criteria locating 9 classes with a NOC mean value greater than 2. We inspect the code to reject the false positives. Finally we obtain:

- Hierarchy 1
    - `AbstractLine` (DIT=1, NOC=3)
    - `AbstractPage` (DIT=1, NOC=3)

4.3 Case 3: JUnit framework

We repeat the complete experiment. Again we are interested in classes with a NOC greater than 1 (complex hierarchies). We obtain the following clusters (see Table 3).

Table 3. `JUnit-3.8.1` - Clusters

Cluster	Num.Classes	%	Mean DIT	St.Dev	Mean NOC	St.Dev
0	29	63%	1.8397	0.8793	0	0.7200
1	12	26%	5.6438	0.4825	0.0001	0.0110
2	**5**	**11%**	**1.5712**	**0.7282**	**1.8568**	**0.6394**

We take the Cluster 2 with 5 classes. We look for classes with similar values and prefixes. There is a false positive between `TestCase` and `TestDecorator` with equal values of DIT and NOC. But the value NOC of `TestCase` includes the number of inner classes. There is not parallelism among their children. Probably, it also indicates that design patterns could generate false positives.

4.4 Strength and Weakness

Although the process shows some flaws, there are some main advantages. In the first case study, with a total of 629 classes, we have discarded the 96% of classes. In the second case study, we have discarded the 90% but we need to adjust the number of clusters. The reason could be the uniformity of the DIT and NOC values (greatest NOC value is 4 in the library).

In the last example, we discarded the 89% of classes, taking only 5 classes. JUnit is a framework very stable. It could be the main reason so we have not found any parallel hierarchy.

5 Conclusions and Future Works

The process presents a method to detect parallel inheritance hierarchies using a static view of a software system. We discard a big number of classes to be inspected using metrics, data mining and name convention filter and do not need several versions of the same system to detect this kind of bad smell.

If we try to achieve this process by human observation, it finally would be hard task. One of the main advantage of this proposal is the saving of time.

We must introduce new improvements. By example, we must not consider classes in the same branch of the inheritance hierarchy. All these kind of improvement will reduce the number of false positives and tune the process.

By other side, we must stand out a language independence. DIT and NOC metrics are applied to a big number of object-oriented languages so this process could be widely applied.

In our future works, we propose to increase the number of case studies, to determine for which domains (class library vs. frameworks) is a good solution.

References

1. Francisca Muñoz Bravo. *A Logic Meta-Programming Framework for Supporting the Refactoring Process*. PhD thesis, Vrije Universiteit Brussel, Belgium, 2003.
2. Martin Fowler. *Refactoring. Improving the Design of Existing Code*. Addison Wesley, 2000.
3. Martin Hitz and Behzad Montazeri. Chidamber and kemerer's metrics suite: A measurement theory perspective. *Software Engineering*, 22(4):267–271, 1996.
4. Mika Mantyla. *Bad Smells in Software - a Taxonomy and an Empirical Study*. PhD thesis, Helsinki University of Technology, 2003.
5. Radu Marinescu Tudor Gîrba, Stéphane Ducasse and Daniel Ratiu. Identifying entities that change together. In *Ninth IEEE Workshop on Empirical Studies of Software Maintenance*, 2004.
6. Zhenchang Xing and Eleni Stroulia. Data-mining in support of detecting class co-evolution. In *16th International Conference on Software Engineering and Knowledge Engineering*, pages 123–128. Banff, Alberta, Canada, June 20-24, 2004.