An Analysis of Anti-Micro-Patterns Effects on Fault-Proneness in Large Java Systems

Giuseppe Destefanis, Roberto Tonelli, Giulio Concas, Michele Marchesi
University of Cagliari, DIEE, Cagliari, 09123, Italy
{giuseppe.destefanis|roberto.tonelli|concas|michele}@diee.unica.it

ABSTRACT

Micro patterns are similar to design patterns, but are at a lower level of abstraction, closer to the implementation. Anti patterns are micro patterns not respecting the prescriptions of good Object Oriented programming practices. In this paper, we use the definitions introduced by Arcelli and Maggioni [3] in order to study the evolution of five particular micro patterns (anti patterns) in different releases of the Eclipse and NetBeans systems, and the correlations between anti patterns and faults. Our analysis confirms previous findings regarding the high coverage of micro patterns onto the system classes, and show that anti patterns not only represent bad Object Oriented programming practices, but may also be associated to the production of lower quality software, since they present a fault proneness significantly enhanced.

Categories and Subject Descriptors
D.2.7 [Software Engineering]: Distribution, Maintenance and Enhancement – Restructuring, reverse engineering and reengineering; D.2.8 [Software Engineering]: Metrics, Product metrics

General Terms
Metrics, Design, Object-Oriented Programming

Keywords
micro patterns, anti patterns, metrics, software faults.

1. INTRODUCTION

In Object Oriented programming, micro patterns can help to identify the portions of code that should be improved (for example those where encapsulation is not respected), and highlight other portions that make up good design practices. The design patterns, defined in the early nineties [2], were an important breakthrough at analysis and design level, but are difficult to be automatically supported at the coding level. There are tools claiming to help finding the usage of design patterns in code, but in practice they are used in a very limited way. On the contrary, micro patterns are defined at coding level, and are relatively easy to recognize automatically, thus being able to implement formal conditions on the structure of the class.

Gil and Maman [1] justified the introduction of micro patterns because design patterns are away from the implementation, and from effective utilization. A traceable pattern is a condition on the attributes, types, names, and/or the body of a module, whose components have a meaning that is shared, simple and automatically recognizable. Gil and Maman cataloged 27 micro patterns, automatically recognizable, that are related to a variety of programming practices in Java – from inheritance, to data encapsulation, to the emulation of typical practices of procedural programming [1]. They developed a tool that implements the algorithms needed for the detection of micro pattern using a language based on First Order Predicate Logic (FOPL). However, they did not provide any detail on the implementation of these algorithms, leaving a certain degree of freedom on the interpretation of the definitions. For example, the condition all methods must be public can be interpreted in a restrictive manner by extending it even to inherited methods of a class, or applying it in a less restrictive way, only to methods declared by the class.

Arcelli and Maggioni re-defined micro patterns according to the metrics number of methods (NOM) and numbers of attributes (NOA) of a given class, offering an alternative implementation [3]. In this way, their micro patterns are not on-off definitions, but range over a continuum of possibilities. Evaluating the number of methods and the number of attributes of a class, it is possible to introduce two parameters called ASR (Attribute Similarity Ratio) and MSR (Method Similarity Ratio), which are used to compute the percentage of attributes of a given class (or interface) that satisfy the specific conditions on the attributes of a given micro pattern – or the numbers of methods of a given class (or interface) that satisfy the specific conditions on the methods for a given micro pattern – respectively. They also considered another parameter, the GSR (Global Similarity Ratio), that evaluates the micro patterns considering both ASR and MSR, ranging between zero and one.

In this paper we aim to study and clarify the relationship between a subset of micro patterns and software defects – a field of study not covered in the literature on this topic – by analyzing software systems with a large number of classes.
At the same time, we aim to confirm the following results found in the literature: (1) three out of four classes match at least one micro pattern in the catalog [1]; (2) the definitions of micro patterns in terms of NOM and NOA can be validated. Specifically, we will show that some design principles – that can be related to the absence of specific micro patterns – yield better results than others, studying the relationship between micro patterns in terms of the number of faults introduced in the software.

2. DEFINITION OF THE ANTI-PATTERNS

Let us consider the five micro patterns called anti-patterns, because they are associated to bad programming practices: Pool, Cobol Like, Record, Pseudo Class, and Function Pointer. We want to investigate their relationships with software defects. Below we report these anti patterns as defined in [3] by means of the parameters MSR and ASR in terms of NOM and NOA. Arcelli and Maggioni judged the constraint that classes implementing the Pool and Record patterns declare no methods as too restrictive [3]. Thus they prefer to define an upper bound (U.B.) in order to consider valid instances of these patterns also classes defining at most 5 methods (U.B. set to 5).

Pool (P): a class which declares only static final fields, but no methods;

- MSR: If $\text{NOM} > \text{U.B.}$ then $\text{MSR} = 0$ else $\text{MSR} = 1 - \frac{\text{static fields}}{\text{NOA}}$
- ASR: in any case $\frac{\text{static fields}}{\text{NOA}}$

Cobol like (CL): a class with a single static method, but no instance members;

- MSR: If $(\text{static m.} = 1)$ and $(\text{NOM} > 1)$ then $\text{MSR} = 0$
  else $\text{MSR} = \frac{\text{static methods}}{\text{NOM}}$
- ASR: in any case $\frac{\text{static fields}}{\text{NOA}}$

Record (R): a class in which all fields are public, with no declared methods;

- MSR: If $\text{NOM} > \text{U.B.}$ then $\text{MSR} = 0$ else $\text{MSR} = 1 - \frac{\text{static fields}}{\text{U.B.}}$
- ASR: in any case $\frac{\text{public fields}}{\text{NOA}}$

Pseudo Class (PC): a class which can be rewritten as an interface: no concrete methods, only static fields

- MSR: In any case $\frac{\text{abstract m. + static m.}}{\text{NOM}}$
- ASR: in any case $\frac{\text{static fields}}{\text{NOA}}$

Function Pointer (FP): a class with a single public instance method, but with no fields;

- MSR: If $\text{public methods} = 1$ then $\text{MSR} = 1$ else $\text{MSR} = 0$
- ASR: If $\text{NOA} = 0$ then $\text{ASR} = 1$ else $\text{ASR} = 0$

We also analyzed the parameter that evaluates the micro pattern considering both ASR and MSR, the GSR (Global Similarity Ratio), defined as $\text{GSR} = \min(\text{ASR,MSR})$. GSR is a real number between zero (complete absence of the micro pattern) and one (presence of the micro pattern as defined in [1]). Intermediate values indicate a partial presence of the micro pattern.

3. EXPERIMENTAL RESULTS

We developed a Java tool in order to extract the data relative to the micro patterns distribution in a generic software system. The tool works in two steps: in the first step the source code is parsed, and a series of files containing information relative to the various classes, fields, methods, calls and so on are generated; in the second step this set of files is examined for computing the micro patterns. In order to test the hypothesis formulated by Gil and Maman, the analyzed dataset includes twenty Eclipse releases and three NetBeans releases. It hosts a total of 335545 classes – Gil and Maman used 70000 classes for their study –. Our results confirm their claims: Three out of four classes match at least one micro pattern in the catalog [1]. The coverage relative to Eclipse is about 80%, while in NetBeans it is about 86%, as reported in Figure 1 for all the 27 micro patterns. This means that these systems can be characterized almost entirely in terms of micro patterns.

3.1 Relationship with faults

Tables 1, 2, and 3 report the relationships among anti patterns and faults in Eclipse 2.1, 3.0 and 3.1. In these tables we consider only perfect matches between classes and anti patterns ($\text{GSR} = 1$). The anti pattern Pseudo Class appears only between 0% and 0.1% of classes in the 20 releases examined, thus we decided to neglect this anti pattern. The first row reports the number of classes exhibiting a given anti pattern and presenting at least one fault. The second row reports the number of classes exhibiting a given anti pattern with no faults. The third row reports the total number of classes into the system matching the anti pattern. This analysis shows that about the 10% of the total number of faults can be ascribed to classes containing anti patterns.

3.2 Correlation between Anti Patterns

The correlation between micro patterns provides information on the independence (or dependence) between a particular micro pattern and another. The results for the Pearson
The data shown tracked software defects only for three of the twenty Eclipse releases: 2.1, 3.0 and 3.1. It is interesting to compare these data for all the system classes with those obtained by our analysis for the anti patterns in Tables 1, 2, and 3. Perhaps the most interesting result regards the Cobol Like anti pattern. In fact, while the use of any anti pattern should be avoided according to Object Oriented programming practices, our results show that in the particular case of the Cobol Like anti pattern such programming practice produces software more fault prone. A comparison among the average fault presence in all the Eclipse classes and in the Cobol Like classes indicates an enhanced fault proneness in the latter. We use the following indicators: percentage of faulty classes; average fault number; percentage of faults contained in the corresponding percentage of the total system (similar to the Pareto 80-20 rule). The first indicator for Cobol Like classes is 53%, 66% and 56% in Eclipse 2.1, 3.0 and 3.1, respectively, while the values obtained for the overall system are 28%, 27% and 34%. The average fault number is 1.89, 3.19 and 1.93 for Cobol Like classes for the three Eclipse versions, while it is 0.669, 0.867 and 0.866 for the overall three versions. For the last indicator, we have 3.6% of faults contained in 0.013% of system classes in Eclipse 2.1, 3.63% of faults contained in 0.009% of system classes in Eclipse 3.0, and 3.47% of faults in 0.015% in Eclipse 3.1. We computed the statistical t-test and Mann-Whitney test on these three indicators, which confirmed the higher fault proneness for Cobol Like classes to a high degree of significance. With regards to other anti patterns, even if the indicators above do not show meaningful results for any increased fault proneness, the absolute values of faults for the corresponding classes suggest that the fault presence can be reduced by eliminating the anti pattern classes. For example, the use of a Pool class may cause the presence of faults which could be avoided with the proper use of an interface in its place.

5. Threats to Validity

We identified three main threats to the validity of our results. We examined 20 Eclipse versions and 3 NetBeans versions. Both are similar environments (IDE for Java programming), and thus may be not representative of all environments or programming languages. This constitutes a threat to the external validity of our findings. Eclipse and NetBeans are Open Source software. Commercial software is typically developed using different platforms and technologies, with strict deadlines and cost limitation, and by developers with different experiences. This might provide different micro pattern distributions, which is another threat for the external validity. Another threat regards the relationships among anti patterns and faults, which has been studied only for three Eclipse versions.

6. Conclusions

The objective of this research was to confirm the assumptions made by Gil and Maman about the coverage of micro patterns in a software system, and try to quantify the relationships between anti patterns and faults in a software system. We observed from empirical studies on three Eclipse releases that 10% of the faults in these systems belong to classes classified as anti patterns. By implementing a refactoring of the classes that correspond to the studied anti patterns (they should be small classes very simple to analyze), it might be possible to eliminate the 10% of faults acting on 13% of classes of the system. Note that these 13% classes are typically very simple - being without methods, or with just one method - and should anyway be refactored to comply with good OO style. Performing this refactoring should also enable to fix a small but not negligible percentage of all system faults. Finally, we found that the presence of anti patterns causes an increment of the fault proneness in the interested classes. For the future we will extend our analysis to investigate the stabilization of a micro pattern over time in order to obtain information on the maturity of a software system.

7. References


