UML Specification and Correction of Object-Oriented Anti-patterns

Maria Teresa Llano† and Rob Pooley
School of Mathematical and Computer Sciences
Heriot-Watt University
Edinburgh, United Kingdom
{mtl4,R.J.Pooley}@hw.ac.uk

Abstract—Nowadays, the detection and correction of software defects has become a very hard task for software engineers. Most importantly, the lack of standard specifications of these software defects along with the lack of tools for their detection, correction and verification forces developers to perform manual modifications; resulting not only in mistakes, but also in costs of time and resources. The work presented here is a study of the specification and correction of a particular type of software defect: Object-Oriented Anti-patterns. More specifically, we define a UML-based specification of anti-patterns and establish design transformations for their correction. Through this work, we expect to open up the possibility to automate the detection and correction of these kinds of software defects.

Keywords- Object-Oriented Anti-patterns; UML; Refactoring.

I. INTRODUCTION

UML (Unified Modelling Language) [8] has become one of the most important languages used worldwide to model OO (Object-Oriented) software systems. Its visual characteristics allow the representation of concrete components of the real world as abstract models that can be easily translated into OO programs. Moreover, UML models offer structures that can be subject to further analysis. For instance, the UML specification of design patterns [4] has been used in the development of techniques for their automatic detection, evolution and verification [2], [11]. This facilitates their understanding and usage, and reduces the amount of errors produced by manual modifications.

Anti-patterns [1] are design patterns whose purpose is to document common bad practices in software design. An anti-pattern explains how a solution that initially appears to be a good choice to solve a specific problem results in the creation of conflicts because of its implementation. The major advantage obtained from anti-patterns is that they equip developers with the knowledge needed to avoid or fix errors before writing any code. However, unlike OO design patterns, OO anti-patterns have not been specified as UML structures; instead, anti-patterns suffer from a lack of documentation that represents an obstacle in the development of techniques for their detection and correction, since there is no standard structure on which this analysis can be based. Recently, some studies have been carried out on techniques for the automatic detection and correction of anti-patterns at the code level [6]; however, no analysis has been done yet on anti-patterns at the design level. This implies that all implementation efforts must be done before it is possible to analyse anti-patterns in a software application.

We believe that by providing a UML specification of anti-patterns these problems can be overcome. A specification based on UML would offer software engineers a well known standard for the study of anti-patterns; furthermore, it would provide the necessary features for them to be subject to further analysis during the design stage while having the advantage of being a language-independent approach.

The rest of the paper is structured as follows. In Section II, the UML specification of the two anti-patterns studied in this paper will be presented (other anti-patterns were analysed in this work but are not presented because of lack of space). Then, in Section III, design transformations for their correction are proposed. Section IV gives some related work and in Section V we conclude with remarks on the main contributions of the paper and future work.

II. ANTI-PATTERNS SPECIFICATION

The analysis of the specification of anti-patterns showed that different UML diagrams are needed according to the type of problems modelled by each of them. Some anti-patterns indicated problems related to behaviour while others were related to the structure and relationships of classes.

As a result, we decided to use different UML diagrams to capture the specific problems presented by each anti-pattern. Activity diagrams are used to model problems related to behaviour while Class diagrams are used to model problems related to the structure of classes. We also enrich our specifications with the use of Stereotypes, which are an extension mechanism used in UML for the definition of specialisations of UML components [9]. By using stereotypes we are able to identify specific characteristics of anti-patterns, which results in more accurate specifications.

A. God class anti-pattern specification

The problem modelled with this anti-pattern is that one of the classes of the design (the god class) is responsible for all (or most of the) behaviour of an application while the rest of the classes (the data classes) are only responsible for

† Supported by EPSRC grant EP/F037058.
encapsulating data. These types of designs show a wrong distribution of responsibilities and, since the components of the god class are not cohesive, finding and fixing bugs becomes a tedious job [1]. Also, reusing the god class may add unnecessary complexity, can be inefficient or produce undesirable side effects.

In order to decide if a class is an instance of the god class anti-pattern it is necessary to analyse each of the methods of the class so as to discover if most of them are god methods (i.e. methods that obtain their data from classes different from the class they belong to). In [1], it is argued that a class can be defined as a god class if it is composed of 60 or more methods and attributes. Since related data and behaviour should be encapsulated in the same place, following good OO design practices, even if a class contains only one god method, this should be located in the class from which it obtains its data. As a consequence, instead of focusing on the number of components that forms a god class, we are focused on the specification and correction of god methods.

In [10], Riel argues that the presence of many accessor methods indicates that related data and behaviour are not being kept in one place and that private information is being given away to be processed in a different class. God methods make use of these accessor methods as means to gain access to the attributes of the data classes. Based on this observation, the UML specification of a god method consists of a method that manipulates the attributes of other classes through their accessor methods but that does not manipulate any data of its own class. The UML specification of a god method is shown in Fig. 1.

Figure 1. A god method.

The basic form of a god method is defined in Figure 1 by the diagram modelled with continuous lines. This shows that the god method accesses one attribute from the class DATACLASS through its accessor method and that there should not exist any link from the god method to the attributes of the god class. This feature is modelled with a link marked with a cross. A cross defines elements that should not exist in the design for it to be a match of the specification of the god method.

The specification just described corresponds to the basic form of a god method; however, in real life applications, a god method may obtain different data from the same class, from different classes or it can also call other methods in the design. These optional elements are represented with dotted lines in Figure 1. These variations depend on the context of the design; nevertheless, they must be reducible to the basic form described previously. Our hypothesis is that if a design cannot be reduced to its basic form, then the design is not a god method. The reduction process will not be treated here but some ideas for this purpose are outlined in Section V.

B. Poltergeist anti-pattern specification

The problem modelled with this anti-pattern is the creation of classes that have limited responsibility within the operation of a system. It can appear in a number of different forms but their idea is the same, poltergeist classes do not have an important role in an OO design. This anti-pattern is against good OO practices because it creates classes that do not represent objects of the system; on the contrary, they depict objects without a state [1]. Furthermore, poltergeist classes have a short life cycle and represent a waste of resources when they are used.

In [10], Riel classifies 5 different ways in which this anti-pattern may appear in an OO design. Next, we develop the UML specification for each of the 5 forms of the poltergeist anti-pattern proposed by Riel.

**Irrelevant classes:** An irrelevant class is a class that does not have any meaningful behaviour in the design. These types of classes are characterised for being composed only of get, set and/or print methods. The UML specification of this anti-pattern, shown in Figure 2, consists of classes that contain only accessor, mutator and/or print methods.

![IrrelevantClass](image)

Figure 2. An irrelevant class.

**Agent classes:** Agent classes are classes that are formed by methods that are only responsible for passing messages from one class to another, i.e., methods that offer redundant paths to access operations of other classes in the design.

As with the specification of the god class anti-pattern, agent classes need to be analysed from the point of view of the methods that compose them. In order to classify a class as an agent class it is necessary to show that each of
its methods performs only one action: passing a message from one class to another. The UML specification of this anti-pattern is shown in Figure 3.

As is observed in Figure 3, a method of an agent class cannot call more than one method and cannot use any instance variables (in fact an agent class should not have any instance variables). An agent method can only be used by a class (CONCRETECLASS1) to call one method of another class (CONCRETECLASS2). Optionally, an agent method may return data that is returned by the method it calls (as shown with the optional elements in dotted lines).

Out of scope classes: A class is classified as out of the scope of the system if it sends messages to other classes but it never receives any message back.

To detect an out of scope class it is also necessary to focus on the methods that compose it. The UML specification of an out of scope method consists in showing that the method is never invoked by other methods of the design. This is shown in Figure 4. This specification defines that there cannot exist calls from any method in the design to the methods of an out of scope class.

Operation classes: Operation classes are characterised for being composed of only one meaningful behaviour and for having a short life cycle. The main idea of an operation class is that an operation that should have been a method within a class has been turned into a class itself.

To represent this form of the poltergeist anti-pattern it is necessary to show that the class is constituted of only one method and that all its instances are Transient objects. This specification is shown in Figure 5.

III. ANTI-PATTERNS CORRECTION

The correction of the anti-patterns specified in the previous section consists of refactorings [3] that are applied to the design to either move, eliminate or create components (attributes, methods, etc.) in order to produce a right (or more appropriate) model.

The correction is presented as production rules, in which the LHS (Left Hand Side) of the rule is the anti-pattern and the RHS (Right Hand Side) its respective correction. Because of lack of space, for the god class anti-pattern only the correction part of the rule is presented.

A. God class anti-pattern correction

In order to correct this anti-pattern, it is necessary to encapsulate related data and behaviour in the same class, i.e., each of the god methods found in a god class has to be moved to the data classes where they belong.

A god method has to be moved to the data class from where it obtains the data it uses to perform its process.
the case of a god method that obtains data from different data
classes, the solution we propose is to locate the god method
in the data class from which it uses more data. However,
the class where the method should be relocated may vary
depending on the context of the application. This type of
decision requires a more specialised analysis (such as an
evaluation of resources at runtime, etc.); nevertheless, our
purpose is to work over the design of the system and so, the
final decision should be left to the developer.

Besides moving a god method to the appropriate data
class, the correction of this anti-pattern also implies the
preservation of the references to other methods that are
used by the god method. Therefore, the associations between
the god method and the resources it uses need to be kept
consistent as they were when the method was in the god
class. The correction of a god method is shown in Figure 7.

\[\text{Figure 7. Correction of a god method.}\]

\[\text{B. Poltergeist anti-pattern correction}\]

The poltergeist anti-pattern resides in the creation of
unnecessary classes in the design; therefore, its correction
consists in eliminating those classes and in relocating their
limited behaviour in a more suitable class.

\text{Correction of an irrelevant class:} Although the behaviour
of irrelevant classes is meaningless, the data that it may
contain is not. The correction of irrelevant classes consists
in both eliminating them from the design and placing the
data they contain (with the respective accessor, mutator
and print methods) in another class. To decide which class
is the most appropriate for the data to be moved into, the
potential classes in which the elements can be located must
be presented to the user (these would be the classes that
use the data of the irrelevant class), so that the user can
choose the most appropriate one. After a class has been
chosen, the transformation shown in Figure 8 is applied in
order to correct the anti-pattern.

\[\text{Figure 8. Correction of an irrelevant class.}\]

\text{Correction of an agent method:} In this case, the
correction consists in removing the agent method from the
design and replacing the communication it performs to be
done directly between the other two classes involved in
the anti-pattern. In other words, instead of using the agent
method to access the operation of \text{CONCRETECLASS2},
\text{CONCRETECLASS1} accesses this functionality directly.
The transformation applied for this correction is shown in
Figure 9. After all agent methods have been removed, the
agent class is eliminated from the design.

\[\text{Figure 9. Correction of an agent method.}\]

\text{Correction of an operation class:} The correction of this
anti-pattern consists in removing the operation class from
the design and moving its attributes and functionality to
a more suitable class. As with irrelevant classes, the user
must choose this class from the set of classes that use the
functionality of the operation class. After a class has been
chosen, the operation and attributes are moved into it as
shown in the transformation of Figure 10, and the operation
class is removed from the design.

\text{Correction of an out of scope class:} These types of
classes are completely unnecessary in the design. As their
behaviour is never used, their correction simply consists in
eliminating them and all their elements (i.e. attributes and
methods). No transformation is shown as the correction
The refactorings made by the correction of the poltergeist class. However, if the class is part of a god class anti-pattern, eliminated and its information would be moved to another have been located in the god class.

If it is only composed of print, get or set methods. However, class. As mentioned before, a class is considered irrelevant when correcting a poltergeist irrelevant anti-pattern would reinforce the god class. If the design is identified as a god class anti-pattern instead, the refactorings would add behaviour to the data class and this one would not longer be considered as an irrelevant class.

Another possible side effect is the creation of a poltergeist operation class when correcting a god class anti-pattern. The correction of a god class involves moving god methods to the classes where they obtain their data from. However, it is possible that a data class has been wrongly defined in the design. If this is the case, the god method would be moved to a wrong class, which may produce new anti-patterns such as a poltergeist operation class.

We believe that more of these types of side effects can be produced when correcting an anti-pattern. Further study will be needed so as to define all possible cases.

IV. RELATED WORK

In [1], Brown et al. studied anti-patterns in three different areas: software development, software architecture and software project management. They analysed the causes, consequences and solution for several anti-patterns in these three categories; however, no standard specification or correction is proposed. Furthermore, unlike our work, in [1] anti-patterns are not analysed at the design level.

Also, in [5], Moha and Guéhéneuc analysed different software defects. In their study they consider design defects, code smells and anti-patterns as Software Architectural Defects (SAD). They propose a classification of SAD into two categories, one that studies the structure of classes in a design, and another that studies anti-patterns and the bad use of design patterns. Although they propose this classification to state a base line for future automation of detection and correction of SAD, they do not develop an approach to doing this. Moreover, for us, it was of unique interest the study of anti-patterns; code smells and design defects, which are analysed in [5], were beyond our scope.

Finally, in [6], Moha et al. defined a technique for automating the correction of design defects (including anti-patterns) through the use of refactorings and (RCA) Relational Concept Analysis. RCA is a framework that allows the conceptual derivation of hierarchies within a group of objects based on the characteristics they share. In their work they use refactorings in order to apply the results of the RCA. However, it is not proposed a specification of the design defects they correct. Instead, algorithms looking for symptoms that indicate the presence of design defects are applied to the code. Also, unlike their methodology, our approach is focused on the application of changes at the design level, not at the code level, which offers the advantage of being more general since it is language independent.

V. CONCLUSIONS AND FUTURE WORK

The novel contribution of this paper is two fold. First, we have defined an approach to the UML specification of
OO anti-patterns, in particular, for the god class and the poltergeist anti-patterns. Our specification not only facilitates their understanding, but most importantly, it contributes to the future automation of their detection and correction through the use of a language-independent approach. The second main contribution is the transformations proposed for the correction of the anti-patterns studied here. These transformations, which are based on local and structural refactorings, offer standard guidelines to developers in the process of correcting an anti-pattern.

Although we have proposed an approach to the correction of anti-patterns, we believe that before attempting such corrections it is important to have confirmation from the developer. Assumptions are made when identifying or correcting certain anti-patterns; however, these assumptions that we recognise as the causes of the anti-pattern may be the behaviour the developer actually intended with the design, or may not be the most optimal correction for the anti-pattern. We propose these transformations as a guide for the improvement of the design; nevertheless the decision of applying the changes should be left to the user.

As future work, the UML specification and correction of other OO anti-patterns should be developed. Also, a study of all possible cases where the correction of anti-patterns has side effects is required. If negative side effects are encountered, analysing alternative paths should be considered.

Regarding the automatic detection and correction of anti-patterns, we have been studying a technique based on graph transformations using a spatial graph grammar. This approach has already been used for the evolution and verification of design patterns [11]. Such a grammar offers a marking mechanism to control the changes applied to a design. This is particularly important for the correction of anti-patterns since we need to ensure that after correcting an anti-pattern the behaviour of and relationships with the other elements in the design are preserved. Furthermore, graph transformations can be applied to an anti-pattern to reduce it to its basic form (as done with design patterns in [11]) in order to ensure that a design is in fact an instance of a particular anti-pattern. Also, techniques such as the ones studied in [7], where patterns are detected through the use of algorithms that look for similarities between a design and the design of the pattern, seem promising for the automatic detection of anti-patterns. We believe that this detection technique together with the graph transformations could offer a good alternative for the automatic detection, verification and correction of OO anti-patterns.

Finally, in order to carry out with the analysis of anti-patterns, a tool that integrates our approach should be implemented. This would help with the study of their automatic detection and correction.

ACKNOWLEDGMENT

The authors would like to thank Andrew Ireland, Gudmund Grov, Ewen Maclean and Julian Gutierrez for their helpful comments. We also want to thank the referees for their comments.

REFERENCES


