Identification of Refused Bequest Code Smells

Elvis Ligu, Alexander Chatzigeorgiou, Theodore Chaikalis, Nikolaos Ygeionomakis
Department of Applied Informatics
University of Macedonia
Thessaloniki, Greece
{mai1315, achat, chaikalis}@uom.edu.gr, nygeion@gmail.com

***Abstract*—Accumulated technical debt can be alleviated by means of refactoring application aiming at architectural improvement. A prerequisite for wide scale refactoring application is the automated identification of the corresponding refactoring opportunities, or code smells. One of the major architectural problems that has received limited attention is the so called 'Refused Bequest' which refers to inappropriate use of inheritance in object-oriented systems. This code smell occurs when subclasses do not take advantage of the inherited behavior, implying that replacement by delegation should be used instead. In this paper we propose a technique for the identification of Refused Bequest code smells whose major novelty lies in the intentional introduction of errors in the inherited methods. The essence of inheritance is evaluated by exercising the system's functionality through the corresponding unit tests in order to reveal whether inherited methods are actually employed by clients. Based on the results of this approach and other structural information, an indication of the smell strength on a 'thermometer' is obtained. The proposed approach has been implemented as an Eclipse plugin.**

Keywords—software maintenance; refactoring; code smell; Refused Bequest

I. INTRODUCTION

One of the most challenging activities, in terms of cost and effort, in the lifecycle of contemporary software systems is the process of maintenance, an inevitable consequence of software evolution. From the perspective of quality, as software systems evolve, their original architecture usually deteriorates due to the fact that design and implementation decisions are taken under time pressure. To confront software degradation over time, code and design refactorings can offer significant aid by improving the internal structure of a software system without changing its external behavior [2]. The application of a refactoring can eliminate specific architectural anomalies or principle violations, widely known as "code smells", and restore the code structure that exhibited a smell, to an acceptable level of quality. However, while the mechanics for the application of each refactoring have been defined in detail [2], the identification of code smells that should be refactored is a non-trivial, time-consuming and challenging activity. To this end, a number of automated tools for the identification of code smells and the facilitation of software maintainers have been developed [6], [10].

In the context of object-oriented systems, the notion of inheritance has been recognized as a key feature claimed to reduce the amount of software maintenance. However, inheritance is not a panacea, especially if it is applied incorrectly in cases where other forms of relationships would be more appropriate. The Refused Bequest code smell concerns an inheritance hierarchy where a subclass does not support the interface inherited from its parent class [2]. More precisely, this smell is present if the functionality inherited by the subclass is not utilized by its clients nor specialized by means of overriding. In other words, the relation between the superclass and the subclass does not constitute an "*is-a*" relationship. The appropriate refactoring is the "Replace Inheritance with Delegation" [2] which dictates to transform an inheritance relationship into composition where the subclass contains a reference to an object of the superclass and uses only the desired functionality. This refactoring is in agreement to the GoF suggestion "*Favor Composition over Inheritance*" [3]. It can be deduced that the Refused Bequest bad smell cannot emerge in abstract classes or interfaces.

This paper proposes a methodology for the identification of the Refused Bequest smell that employs static source code analysis for the identification of suspicious hierarchies and dynamic unit test execution for the determination of subclasses that actually exhibit the smell. Identified smells are sorted according to their intensity based on criteria such as the number of overridden methods, the invocation of superclass methods and the results from test execution. Smell interpretation is facilitated by a "Smell Thermometer" which depicts graphically the intensity of the smell. The approach has been implemented as an extension on the JDeodorant Eclipse plug-in [4] and is evaluated on an open-source project.

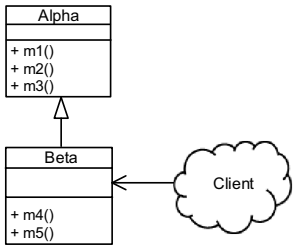
II. KEY CONCEPT

The key idea behind the proposed identification technique lies in the detection of whether a subclass in a given hierarchy actually "*wants to support the interface of the superclass*" [2]. Refusing an inherited interface, in the sense that clients of the subclass do not invoke any of the inherited functionality (but rather access only new functionality) is a relatively clear sign of Refused Bequest. There are numerous factors that come into play and indicate whether the use of inheritance is justified or not, but the notion of "refusing" the inherited behavior implies that the particular generalization does not have the properties of an "*is-a*" relationship.

This is a property that in general is hard to assess without relying on human expertise and thus is difficult to automate.

1063-6773/13 $26.00 © 2013 IEEE
DOI 10.1109/ICSM.2013.55

However, we could rely on the potential invocations of subclass' methods from the rest of the classes to detect whether inherited methods (and consequently the interface of the superclass) are actually exploited by the subclass. This concept is illustrated with the help of Fig. 1 where inherited and additional methods of `Beta` can be accessed by the Client.




```
class Client {
   private Beta ref;
   public void foo() {
      . . .
      ref = new Beta();
      ref.mX();
//where X is any of 1..5
      . . .
   }
}
```

Fig. 1. Client accessing subclass methods.

According to the Dependency Inversion Principle [7] the proper scenario should be a client holding a reference to the superclass to exploit the benefits of polymorphism. However, in case the subclass contains additional methods, these can only be assessed by a client holding a reference of type `Beta`.

If the Client is to invoke only the additional methods of class `Beta` (`m4` and `m5`) without never calling the inherited superclass methods (`m1`, `m2` or `m3`), and the same holds for all clients of class `Beta`, it appears that the subclass somehow "denies" the inherited interface implying that generalization might not be appropriate (instances of `Beta` are not used as specializations of `Alpha` entities). The role of other parameters such as overriding and invocations of superclass methods through `super` will be discussed in the next section.

One way to detect whether superclass methods are actually invoked on subclass instances, is to override these methods in the subclasses and intentionally introduce an error in the corresponding implementation (such as a division by zero). If the corresponding method is invoked anywhere in the code base on instances of the subclasses, then, in case of overriding, the overridden methods will be invoked instead, causing an easily observable failure. If, despite the introduction of errors, the execution of all system scenarios does not lead to any failures, it can be concluded that the inherited superclass methods would not be actually used on any of the subclasses, providing a strong hint for the presence of Refused Bequest.

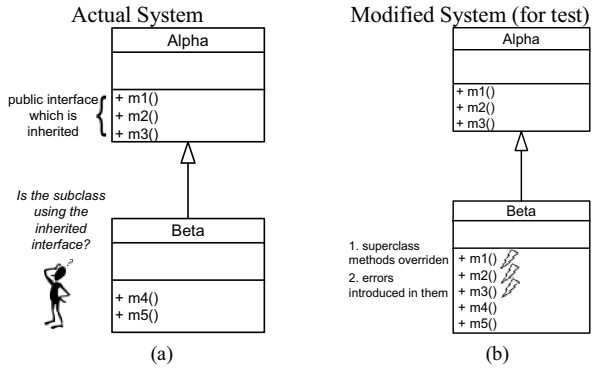


Fig. 2. (a) Inheritance relation under investigation, (b) Error insertion.

The proposed approach is illustrated in Fig. 2. Assuming that a subclass inherits behavior (Fig.2(a)), the question is whether the subclass actually uses the inherited interface. In Fig. 2(b) the non-overridden methods are now implemented in the subclass with errors deliberately introduced in them. If the execution of system functionality exhibits failures, it means that the overridden and faulty methods are invoked. In this case it can be concluded that in the initial system the inherited interface is not "Refused" and thus the smell is not present.

To exercise the system's functionality in order to reveal whether the inherited methods are invoked on instances of a subclass one could either rely on a) manually invoking all system functionality, b) executing specific methods that aim at demonstrating a large portion of the system's functionality and c) executing test cases within a project, assuming that a sufficient level of coverage is provided. We have relied on the third alternative since the execution of test cases can be automated and because for several open-source projects unit tests cover a large portion of the corresponding code base.

In contrast to other detection techniques which rely only on static analyses, the proposed approach can reveal dynamic information which is crucial for determining whether an inheritance relationship is appropriate or not. For example, polymorphic method invocations which totally justify the use of generalization, can only be detected by dynamic analysis.

III. SMELL THERMOMETER

As already mentioned, the proposed approach takes several factors into account to assess the smell intensity. Based on the findings the following classification is obtained:

a) Abstract Superclass or Interface

When a designer employs a generalization relationship and places an abstract class or an interface on the root of the hierarchy, his/her intention is rather clear: The goal is to apply the Dependency Inversion Principle and essentially to allow polymorphic behavior where the public interface of the base abstract class (or interface) is implemented by a corresponding subclass. In these cases it is theoretically impossible to encounter a Refused Bequest symptom, since the same benefit cannot be achieved by other means. In other words, it is clearly evident that the employed generalization is on purpose, well-designed and constitutes an "is-a" relationship. When the superclass is neither abstract nor an interface, the following cases can be distinguished.

b) Overriding and Failures

In this case one or more superclass methods have been overridden. Moreover, when exercising the system functionality failures emerged because of the introduced errors. Here, we have two clear indications that the presence of Refused Bequest is highly improbable. First, since the designer re-implemented methods which have been inherited to provide functionality that is specific to the subclass, it can be deduced that the goal is to enable polymorphism. Second, the presence of errors indicates that the inherited methods are invoked on instances of the subclass, i.e. the inherited functionality is actually employed. Thus, we can conclude that generalization is appropriately applied and cannot be replaced. The picture in the following cases is less clear.

c) Some overriding and No failures

A first indication that Refused Bequest might be present is the lack of failures in the executed test cases. This means that (assuming that the test cases provide complete coverage) no method in the entire system has invoked inherited methods on subclass instances. In other words, subclasses exhibit signs of refusing the inherited interface. However, if at the same time one or more of the superclass methods are overridden in the subclass, the symptom is alleviated. Overriding allows polymorphism, something which would not be possible in case inheritance is replaced by delegation. Therefore, we consider this situation as a mixed case where only limited signs of Refused Bequest can be diagnosed.

d) No overriding, some failures and invocation of super

When the subclasses in a generalization do not override superclass' methods, the inherited interface is no longer specialized by the descendants in the hierarchy. As a result the designer's intention deviates from the goal of enabling polymorphic behavior or conforming to the requirements of a design pattern such as State, Strategy or Template Method. On the other hand, the presence of failures which implies that the inherited methods are invoked on subclass instances, is a sign towards the opposite direction. Since the subclass does not override superclass methods, the only alternative left (for an hierarchy to be meaningful) is to introduce additional methods to the subclass. This particular case can be further categorized, depending on whether the introduced subclass methods invoke superclass methods through the `super` keyword. Although method invocations through `super` could be refactored in case inheritance is replaced by delegation [5], the presence of the `super` keyword is an indication of a certain degree of reuse. Consequently, we consider the presence of superclass method invocations as a (relatively weak) indication that inheritance might be appropriate, at least in comparison to the next case.

e) No overriding, some failures and No invocation of super

Here, the only difference to the previous case is the lack of superclass method invocations in the additional methods of the subclass. In combination with the lack of overridden methods, these characteristics imply an even stronger probability that a Refused Bequest symptom actually exists. In fact, only the occurrence of some failures due to the introduced errors points to the opposite direction.

f) No overriding, No failures

This case constitutes the stronger indication that the Refused Bequest smell exists in the examined hierarchy. Here, no superclass method is overridden, none of the inherited methods is actually used on instances of the subclass and none of the additional subclass methods contains a superclass method invocation. In other words it appears as if the subclass refuses any connection to its superclass and the corresponding generalization can hardly be characterized as an "*is-a*" relationship. No argument in favor of inheritance can be made and the hierarchy can be safely refactored to delegation [5].

All of the aforementioned symptoms according to which the strength of the Refused Bequest smell can be deduced, are summarized visually in the Smell Thermometer of Fig. 3. The higher the "temperature" gets, the stronger the smell is.

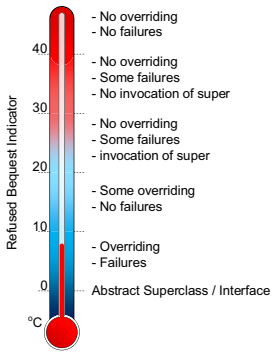


Fig. 3. Refused Bequest Thermometer.

To facilitate the identification of Refused Bequest smells we have extended the JDeodorant Eclipse plugin [4]. The plugin[1] enables the user to select either an entire Java project or a particular package, execute the identification and observe the findings, ordered by smell severity. The identification relies on the representation of the project under study as an Abstract Syntax Tree (AST) provided by the Eclipse JDT API.

IV. Case Study and Threats to Validity

The developed plugin has been applied on SweetHome3D (v.4.0) which is an open-source Java interior design application. Size properties are shown in Table I. The approach revealed one characteristic example shown in Fig. 5.

TABLE I. Overview of SweetHome3D Size Metrics

LOC	Number of Packages	Number of Classes	Number of Operations	Number of Inheritance Hierarchies	Number of test cases*
76730	9	460	3360	69	42

*These tests act as integration tests exercising a large portion of software features.

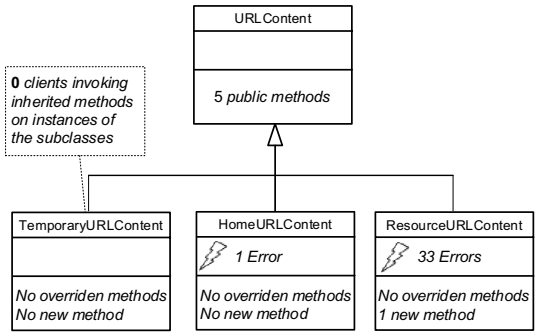


Fig. 4. No overriding, No failures case in SweetHome3D (Refused Bequest).

A `URLContent` enables the retrieval of resources from a Uniform Resource Locator. `TemporaryURLContent` is supposed to extend the `URLContent` parent class. However it provides no additional methods and consists of a single constructor and a single static method. No methods of the superclass are invoked through `super`. Overriding all 5 inherited methods and introducing errors into them has not led to any failure in the execution of JUnit test cases. Since there is no opportunity for polymorphic behavior and no use of the

[1] The plugin can be downloaded from http://java.uom.gr/ref_bequest/

inherited methods, it appears that this particular case exhibits the symptoms of a clear Refused Bequest. Even refactoring by means of delegation seems to make no sense for this case.

A similar, however slightly different observation can be made for the `HomeURLContent` subclass. Once again, no additional method is introduced. However when executing the test cases, one error was generated because of the flawed overriding of the superclass methods, implying that one of these methods has been invoked on a subclass instance. The symptom of Refused Bequest still exists since the rest of the superclass interface is refused by the subclass. In any case the intensity of the smell is lower than in the previous case.

Concerning the `ResourceURLContent` class, one additional method is introduced. More important is the fact that the introduction of errors in the overridden methods generated 33 failures, implying that the inherited interface is heavily used by clients of the subclass. As a result, Refused Bequest is hardly present. The only evidence that generalization is not exploited as much it could be, is the lack of any overridden methods (prohibiting essential polymorphism) and the lack of superclass method invocations.

The proposed detection process is based on the assumption that unit tests exercise thoroughly the system's functionality to reveal whether the inherited methods are actually invoked on instances of a subclass. This could impose a threat to construct validity which deals with how well the selected measures or tests can stand in for the concepts of interest. In particular, the exercised unit tests might not invoke the inherited methods on instances of the subclasses of interest and false positives might emerge. In other words, the introduced errors might not lead to test failures just because the unit tests have not been designed to cause the invocation of the corresponding methods and not because they are not actually utilized in the system. To mitigate this threat it is advised to perform the identification on projects with extensive test coverage, something which becomes typical for contemporary software projects.

V. Related Work

Although the frequency of the Refused Bequest smell is relatively low, it has been investigated by several researchers. Some works explicitly target the Refused Bequest smell, while others treat more general inheritance related problems. Zhang et al. [11] performed a literature review regarding approaches in the field of code smell detection and refactoring. According to their findings, 11 out of 39 relevant papers (28%), deal either with the identification or the removal of Refused Bequest. The need to analyze an hierarchy's clients to find out the original intention of a generalization has also been emphasized in [8]. Stefan Slinger [9] developed the CodeNose Eclipse plug-in which is capable of identifying the Refused Bequest smell by examining subclasses and the methods that they may override. Refused Bequest can also be identified by the detection strategy proposed by Marinescu [6] relying on a combination of selected code metrics and the definition of appropriate thresholds. Tourwe and Mens [10] employed logic meta-programming for the identification of *Inappropriate Interfaces*, which are unclear or incomplete interfaces hindering the evolution of hierarchies in a way that favors polymorphism. Arevalo et al. [1] in their smell identification approach, which is based on Formal Concept Analysis, refer to this type of smell as "Unanticipated Dependency Schemas". Kegel and Steimann [5] defined pre and post conditions for the application of the "Replace Inheritance with Delegation" refactoring to alleviate the Refused Bequest smell.

VI. Conclusions

Code smell detection is an extremely challenging maintenance task because it involves both static analysis for parsing structural code properties as well as dynamic examination of the context in which a particular code structure is being used. In this paper we have attempted to confront the problem of diagnosing Refused Bequest smells, employing a combination of static and dynamic analyses. The key contribution lies in the introduction of intentional errors in the non-overridden inherited methods, enabling designers to determine whether a subclass refuses the inherited interface or not. Measuring symptom severity on a smell thermometer can highlight suspect hierarchies that warrant further attention. We believe that the concept of intentionally introduced errors and the inspection of the resulting software behavior by means of test case execution can be generalized for the detection of other architectural problems.

Acknowledgment

This research has been co-financed by the European Union (European Social Fund – ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) – Research Funding Program: Thalis – Athens University of Economics and Business - SOFTWARE ENGINEERING RESEARCH PLATFORM.

References

[1] G. Arévalo, S. Ducasse, and O. Nierstrasz, "Discovering unanticipated dependency schemas in class hierarchies," *9th European Conference on Software Maintenance and Reengineering*, UK, March 2005, pp. 62-71.

[2] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[4] JDeodorant, http://www.jdeodorant.com, June 2013.

[5] H. Kegel, and F. Steimann, "Systematically refactoring inheritance to delegation in java," *30th IEEE/ACM Int. Conference on Software Engineering*, Leipzig, Germany, May 2008, pp. 431-440.

[6] R. Marinescu, "Detection strategies: metrics-based rules for detecting design flaws," *20th IEEE Int. Conference on Software Maintenance*, Chicago Illinois, USA, September 2004, pp. 350–359.

[7] R.C. Martin, *Agile Software Development: Principles, Patterns and Practices*. Prentice Hall, 2003.

[8] P. F. Mihancea, "Towards a Client Driven Characterization of Class Hierarchies", *14th IEEE Int. Conference on Program Comprehension.* Athens, Greece, June 2006, pp. 285-294.

[9] S. Slinger, "Code Smell Detection in Eclipse," Ph.D. Thesis, Delft University of Technology, March 2005.

[10] T. Tourwe and T. Mens, "Identifying Refactoring Opportunities using Logic meta programming," *7th European Conference on Software Maintenance and Reengineering*, Italy, March 2003, pp. 91-100

[11] M. Zhang, T. Hall, and N. Baddoo, "Code Bad Smells: a review of current knowledge," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 23, no. 3, pp. 179–202, 2011.