Using FCA to Suggest Refactorings to Correct Design Defects

Naouel Moha, Jihene Rezgui, Yann-Gaël Guéhéneuc,
Petko Valtchev, and Ghizlane El Boussaidi

GEODES, Department of Informatics and Operations Research
University of Montreal, Quebec, Canada
{mohanaou,rezguiji,guehene,valtchev,elboussg}@iro.umontreal.ca

Abstract. Design defects are poor design choices resulting in a hard-to-maintain software, hence their detection and correction are key steps of a disciplined software process aimed at yielding high-quality software artifacts. While modern structure- and metric-based techniques enable precise detection of design defects, the correction of the discovered defects, e.g., by means of refactorings, remains a manual, hence error-prone, activity. As many of the refactorings amount to re-distributing class members over a (possibly extended) set of classes, formal concept analysis (FCA) has been successfully applied in the past as a formal framework for refactoring exploration. Here we propose a novel approach for defect removal in object-oriented programs that combines the effectiveness of metrics with the theoretical strength of FCA. A case study of a specific defect, the *Blob*, drawn from the Azureus project illustrates our approach.

Keywords: Design Defects, Formal Concept Analysis, Refactoring.

1 Introduction

Design defects are bad solutions to recurring design problems in object-oriented programming. The activities of detection and correction of design defects are essential to improve the quality of programs and to ease their maintenance and evolution. Indeed, design defects have a strong negative impact on quality characteristics such as evolvability and maintainability [4]. A program without design defects is easier to understand and change and thus has lower maintenance costs.

However, the detection and correction of design defects are time-consuming and error-prone activities because of lack of (semi-)automated techniques and tools. Although approaches exist to detect design defects, using metrics [7] for example, to the best of our knowledge, no approach attempts to correct design defects (semi-)automatically. Huchard and Leblanc [6] use formal concept analysis (FCA) to suggest restructurations of class hierarchies to maximise the sharing of data structure and code through fields and methods and remove code smells from the program. Arévalo *et al.* applied FCA to identify implicit dependencies among classes in program models [1]. They build models from source code and extract contexts from the models. Concepts and lattices generated from the contexts with the ConAn engine are filtered out to build a set of views at different

levels of abstraction. These two approaches provide interesting results but none attempts *to suggest refactorings to correct design defects.*

We propose to apply FCA on a suitable representation of a program to suggest appropriate refactorings for certain design defects. A refactoring is a change performed on the source code of a program to improve its internal structure without changing its external behaviour [4]. In particular, we examine the benefits of FCA and concept lattices for the correction of a very common design defect, the Blob [2, p. 73–83]. It is generally accepted that a Blob reflects procedural thinking during the design of an object-oriented program. It manifests through a large class monopolising the computation, surrounded by a number of smaller data classes, which embed a lot of attributes and few or no methods.

Design defects are the results of *bad* practices that transgress *good* object-oriented principles. Thus, we use the degree of satisfaction of those principles before and after the refactorings as a measure of progress. Technically speaking, we rely on quantification of *coupling* and *cohesion*, which are among the most widely acknowledged software quality characteristics, key for the target maintainability factor. The cohesion of a class reflects *how closely the methods are related to the instance variables in the class* [3]. A low cohesion score witnesses a cohesive class whereas a value close to 1 indicates a lack of cohesion and suggests the class might better be split into parts. The coupling of a class is defined as the degree of its reliance on services provided by other classes [3], i.e. it counts the classes to which a class is coupled. A well-designed program exhibits *high* average cohesion and *low* average coupling, but it is well known that these criteria are antinomic hence a trade-off is usually sought.

Our intuition is that design defects resulting in high coupling and low cohesion could be improved by redistributing class members among existing classes (with possibly new classes) to increase cohesion and–or decrease coupling. FCA provides a particularly suitable framework for helping in redistributing class members because it can discover strongly related sets of individuals wrt. shared properties and hence supports the search of cohesive subsets of class members.

2   Combining Metrics and FCA to Correct Design Defects

2.1   Running Example

We illustrate our approach using Azureus version 2.3.0.6, a peer-to-peer program [8] that contains 143 Blobs for 1,449 classes (191,963 lines of code) and show that FCA can suggest relevant refactorings to improve the program. We choose Azureus because it has been heavily changed and maintained since its first release in July 2003. The addition of new features, optimisations, and bugs fixes have introduced design defects. We choose to illustrate our approach with the Blob because it impacts negatively the two important quality characteristics: such classes show low cohesion and high coupling. We notice that the underlying classes that constitute the Blobs in Azureus are difficult to understand, maintain, and reuse because they have a large number of fields and methods. For example, the class `DHTTransportUDPImpl` in the package `com.aelitis.azureus.core.-dht.transport.udp.impl`, which implements a distributed sloppy hash table

(DHT) for storing peer contact information over UDP, has an atypically large size. It declares 52 fields and 71 methods for 3,211 lines of code. It has a medium-to-high cohesion of 0.542 and a high coupling of 41 (8th highest value among 1,449 classes). The data classes that surround this large class are: `Average, HashWrapper` in package `org.gudy.azureus2.core3.util` and `IpFilterManagerFactory` in package `org.gudy.azureus2.core3.ipfilter`.

2.2 Our Approach in a Nutshell

Figure 1 depicts our approach for the identification of refactorings to correct design defects in general and the Blob in particular. The diagram shows the activities of detection of design defects and correction of user-validated defects.
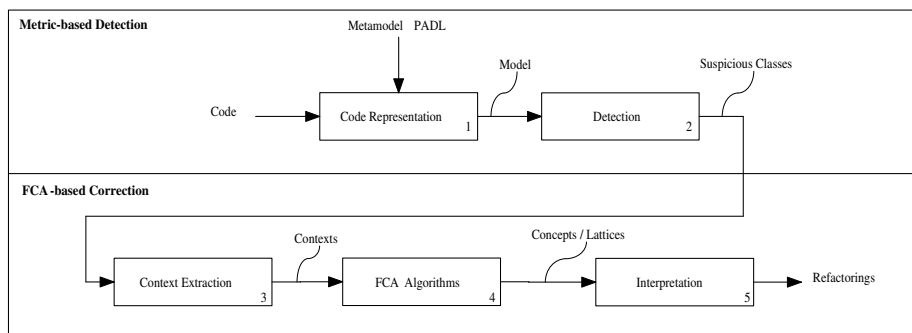

Fig. 1. Detection and FCA-based Correction of Design Defects.

First, we build a model of the source code which is simpler to manipulate than the raw source code and therefore eases the subsequent activities of detection and correction. The model is instantiated from a meta-model to describe object-oriented programs.

Next, we apply well-known algorithms based on metrics and–or structural data on this model to single out suspicious classes having potential design defects. We automatically extract contexts related to these classes (and to their methods and fields) from the model of the source code. These contexts are built to enable the detection of related methods, fields, and classes (see Section 2.3).

Then, the contexts are fed into a FCA engine which generates concept lattices. We explore the lattice structure (order) and interpret the discovered concepts to clarify the relationships among members of the suspect classes and their links to the rest of the program. Both concepts and order are analysed to suggest refactorings to recreate the discovered related sets of elements.

2.3 Encoding Blobs into Formal Contexts

To correct Blob design defects, we need to identify cohesive sets of methods and, possibly, fields with respect to three criteria: usage of fields, calls to other methods, and reliance on data classes. Hence, our individuals can be either methods or fields, our properties are substitutable to fields, methods, or data classes and our incidence relations represent associations, method invocations, or use-relationships.

Table 1. Context $\mathcal{K}_1$ linking methods of the large class to fields of the class.

	(a0) alien_average	(a1) alien_fv_average	(a2) bad_ip_bloom_filter	(a3) bootstrap_node	(a4) external_address	(a5) last_address_change	(a6) last_alien_count	(a7) last_alien_fv_count	(a8) listeners	(a9) local_contact	(a10) logger	(a11) other_non_routable_total	(a12) other_routable_total	(a13) packet_handler	(a14) reachable	(a15) reachable_accurate	(a16) recent_reports	(a17) request_handler	(a18) request_timeout	(a19) stats	(a20) stats_start_time	(a21) store_timeout	(a22) STATS_INIT_PERIOD	(a23) STATS_PERIOD
(m0) checkAddress()			×																					
(m1) externalAddressChange()						×																		
(m2) getAddressChange()					×																			
(m3) process()				×						×	×			×				×	×		×			
(m4) sendFindNode()										×				×						×	×			
(m5) sendFindValue()										×				×						×	×			
(m6) sendStore()										×				×							×		×	
(m7) setRequestHandler()																		×						
(m8) testInstanceIDChange()										×														
(m9) testTransportIDChange()					×					×														
(m10) updateContactStatus()												×	×											
(m11) updateStats()	×	×					×	×	×					×	×						×	×		×	×

Context 1. We defined three contexts. In the first formal context, $\mathcal{K}_1$, individuals are methods of a suspect large class and properties are fields of that class. The incidence relation is the *method-uses-field* relationship. The context aims at identifying methods using the same sets of fields and fields used by cohesive sets of methods. It allows to assess the cohesion of a class because methods sharing the same fields are, by definition, cohesive. In the second formal context, $\mathcal{K}_2$, both individuals and properties are methods of the suspect large class, while the incidence is the *method-invocation* relationship. This context highlights subsets of cohesive methods, because methods invoking the same set of other methods are highly cohesive. In the third formal context, $\mathcal{K}_3$, individuals are methods and fields of the large class and properties are the surrounding data classes. This context represents the *use-relationship* and allows to assess the coupling between the large class and its data classes. We can identify which methods and fields of the large class should be moved together to some data class.

In the first formal context, $\mathcal{K}_1$, individuals are methods of a suspect large class and properties are fields of that class. The incidence relation is the *method-uses-field* relationship. The context aims at identifying methods using the same sets of fields and fields used by cohesive sets of methods. It allows to assess the cohesion of a class because methods sharing the same fields are, by definition, cohesive.

Table 1 illustrates the context drawn from the large class `DHTTransportUDP-Impl` in Azureus. It shows the methods (individuals in rows) and their use-relationship links with fields (properties in columns) of the large class. Codes are provided that are used when presenting lattices in the next paragraphs.

2.4 Interpretation of Lattice Structure

We build lattices from the contexts $\mathcal{K}_1$, $\mathcal{K}_2$, and $\mathcal{K}_3$, respectively. We use these lattices to interpret the inner structure of the large class and then to suggest refactorings. More specifically, we look for specific configurations of concepts that reflect the presence of cohesive and (un)coupled sets. Intuitively, shared usages of fields and calls of methods is a sign of cohesion whereas coupling is

directly expressed by the reliance of a class member on a data class. We define the following interpretation rules.

Rule 1. *[Collection of cohesive and independent subsets.]* If a set of concepts has only the lattice supremum (top) as a successor and only the infimum (bottom) as a predecessor (*pancake lattice*), then they all represent cohesive and disjoint subsets of the individuals. For instance, in Figure 2, we interpret the concepts in the area 2 (on the right of the oblique line) as sets of elements that, whenever put together, form a low-cohesion group. Indeed, there is no collaboration (*i.e.*, no shared fields) between the individuals in different concepts.

Rule 2. *[A large cohesive subset.]* If a sub-structure of the lattice has many concepts that form a network with all their meets and joins (different from the supremum and the infimum of the lattice), then that structure represents a cohesive set of individuals. Such a situation is depicted in Figure 2, on the left of the oblique line (zone 1).
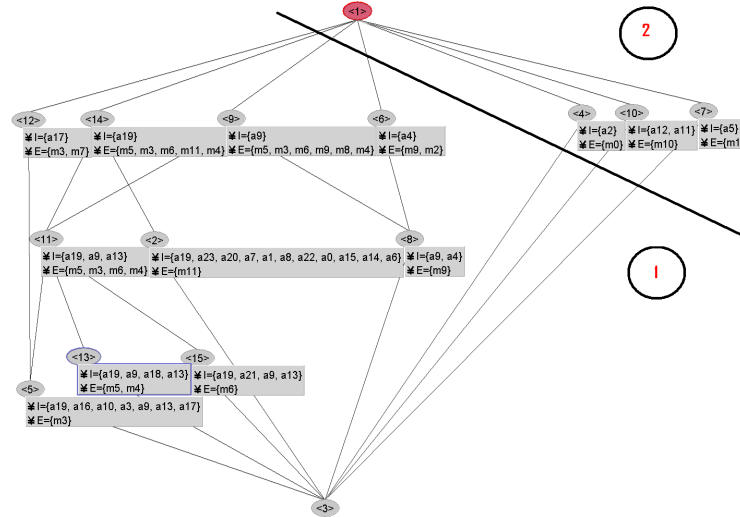

Fig. 2. Concept lattice of the methods × fields context.

Lattice 1. Recall that the lattice in Figure 2 represents the method-uses-attribute relationship. By applying **Rules 1,2**, we obtain four concept sets representing cohesive subsets of methods and fields in the large class:

$$\{ \{ \mathcal{C}_5, \mathcal{C}_{13}, \mathcal{C}_{15}, \mathcal{C}_{11}, \mathcal{C}_2, \mathcal{C}_8, \mathcal{C}_{12}, \mathcal{C}_{14}, \mathcal{C}_9, \mathcal{C}_6\}, \{\mathcal{C}_4\}, \{\mathcal{C}_{10}\}, \{\mathcal{C}_7\} \}$$

Combination of lattices. Following the interpretation of the lattices, we split the large class into two ways. First, we move disjoint and cohesive subsets of methods and–or fields that are related to a data class in that data class. Second, we organise cohesive subsets unrelated to data classes in separate classes.

Refactorings. Before the refactorings, class `DHTTransportUDPImpl` had a cohesion of 0.542 and a coupling of 41. After the refactorings, the cohesion of the classes is maximum of 0.278 and the coupling has reduced to 34, which shows a better compliance to good object-oriented design principles and highlights the interest of our approach.

Implementation. We use PADL to model source code and Galicia to construct and visualize the lattices. PADL is the meta-model at the heart of the Ptidej open-source tool suite (*Pattern Trace Identification, Detection, and Enhancement in Java*) [5]. Galicia is a multi-tool open-source platform for creating, visualizing, and storing lattices. Both tools communicate by means of XML files describing data and results. Thus, an add-on to Ptidej generates contexts in the XML format of Galicia, which are then transformed by the tool into lattices and shown on screen for exploration.

3   Conclusion

Design defects are the results of bad practices that transgress good object-oriented design principles. A low coupling and a high cohesion are among the most recognised design principles to assess the quality of programs, in particular their maintainability and evolvability.

We propose an approach based on the joint use of metrics and FCA to suggest corrections to design defects in object-oriented programs. FCA provides a sketch of the target design by grouping methods and fields into cohesive sets which, once turned into separate classes, represent a better trade-off between coupling and cohesion. Our approach can be systematically generalised to other design defects characterised by a high coupling and a low cohesion.

In the long term, we plan to develop fully automatic correction mechanisms and to propose an integrated tool platform to support FCA-based refactorings. A refinement of the proposed rules for lattice structure interpretation will also be developed, allowing for more subtle, possibly numerical, decision criteria for cohesive sets of concepts.

References

1. G. Arévalo, S. Ducasse, and O. Nierstrasz. Lessons learned in applying formal concept analysis. In *ICFCA*, vol.3403 of LNAI, p95–112. Springer Verlag, 2005.
2. W. J. Brown, R. C. Malveau, W. H. Brown, H. W. McCormick III, and Thomas J. Mowbray. *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis.* John Wiley and Sons, 1st edition, 1998.
3. N. Fenton and S. L. Pfleeger. *Software metrics (2nd ed.): a rigorous and practical approach.* PWS Publishing Co., Boston, MA, USA, 1997.
4. M. Fowler. *Refactoring – Improving the Design of Existing Code.* Addison-Wesley, 1st edition, 1999.
5. Y. Guéhéneuc. A reverse engineering tool for precise class diagrams. 14th *IBM Centers for Advanced Studies Conference*, p28–41. ACM Press, 2004.
6. M. Huchard and H. Leblanc. Computing interfaces in java. In *ASE*, p317–320, 2000.
7. R. Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *ICSM*, p350–359. IEEE Computer Society Press, 2004.
8. Open source project. Azureus, 2003. `http://azureus.sourceforge.net/`.