Automated Strategy Based Restructuring of Object Oriented Code

Adrian Trifu
FZI Forschungszentrum Informatik, Karlsruhe
trifu@fzi.de

Abstract

Software decay is a phenomenon that plagues all software systems in general, and object oriented systems in particular. Existing approaches fail to effectively address this problem because of their informal nature. We overcome the main deficiencies of other approaches, with the help of two innovations: encapsulation of correlated structural anomalies and machine processable patterns for restructuring. Our method allows unprecedented levels of automation in the decision making process involved in restructuring large object oriented systems.

1 Introduction

Since the adoption of the object oriented paradigm on a large scale in the software industry, companies (especially early adopters) have been facing increasing problems related to design drift [5], also known as software decay [3]. The causes are numerous, and have been analyzed thoroughly in the literature [5]. Besides its obvious scientific importance, the fight against design drift also has an important economic aspect. Restructuring decaying legacy systems is still a mostly manual, extremely costly and risky process. Finding ways to increase predictability, reliability as well as raising the level of automation, will lead to a sharp decrease in maintenance costs for all companies that own such systems.

Existing approaches fail to address this problem, because of two main reasons. The first reason is the existence of a conceptual gap between what we can measure in a system (symptoms), and the causing factors (the problems). Most analysis techniques rely on software metrics or compositions of metrics that can bring certain abnormal *attributes* of design artifacts to light. While these approaches may give a suggestive picture about the general state in which a system is, it is insufficient to tell us what needs to be done to improve the system. In other words, we know that something with a certain part of our system is wrong (e.g. a class is large, non-cohesive with lots of code duplication), but we neither know what exactly caused the identified anomalies, nor how to improve the situation.

The second main deficiency of other approaches is the lack of machine-usable and reusable reengineering expertise. Once we have successfully carried out a series of complex transformations that lead us to a better design, we do not have the possibility to formally record and later (perhaps automatically) reuse our knowledge. In other words, the process *is and remains* purely manual, to be carried out by experts. What is needed is a sort of formal patterns that describe what to do when problems occur in the design.

In the following section, we present a new method that successfully addresses these problems. Finally, we give a short overview of related research and conclude, by providing some insight into future work.

2 Strategy-based restructuring method

If we use a medical analogy, the key to an effective treatment is an exact diagnosis of the disease, based on its characteristic symptoms. Structural problems in object oriented systems are "diseases" that manifest themselves through a characteristic constellation of structural anomalies, also known as "bad smells" [3]. This clear distinction between problem and symptom is new in the literature. Until now, terms such as "bad smells", "design flaws" [6] and "structural problems" [2] have been generally used interchangeably. However, we make a clear distinction between *structural problems* and *structural anomalies*, which we define as follows: structural problems are the concretization of a design decision made by the designer/developer, which contradicts some commonly accepted design practices, with respect to well determined, reoccurring situations. Structural anomalies on the other hand, are the manifestation of structural problems in the source code of the system (the bad smells). They usually occur together in constellations that suggest the presence of a common underlying problem. For example, the problem that we call *abused inheritance* refers to an abusive use of the inheritance mechanism with the purpose of reusing the implementation of the superclass. Typical bad smells, or anomalies induced by this problem include: refused bequest [7], high inter-subsystem couplings, overlooked abstraction, etc. The characteristic anomaly fingerprint of a problem can contain mandatory, as well as optional anomalies.

We have currently defined seven structural problems and their corresponding anomaly fingerprints, used for their diagnosis: *conceptualization abuse, collapsed class hierarchy, collapsed method hierarchy, orphan sibling interfaces, abused inheritance, outsourced functionality* and *ignored abstraction.*

Having well defined (concrete) problems, we can describe their solutions much more easily and precisely as for bad smells. We do this in so called *restructuring strategies*. In order to shield restructuring strategies from the

(at this stage) unnecessary complexities related to correctness of transformations, as well as programming language particularities, they are specified in terms of an abstract model [11]. In other words, a restructuring strategy is a formally specified algorithm that operates on an abstract model in order to determine a sequence of refactorings that eliminate a problem instance. For example, the problem called *conceptualization abuse*, means that two or more, non-cohesive concepts have been packed into a single class of the system. The anomaly fingerprint for this problem includes the presence of the anomalies: God class (also known as *the blob*), several possible data classes (satellite classes of the god class), feature envy in methods of the god class, etc. The restructuring strategy for *conceptualization abuse* first determines envious methods in the abusive class, and moves them to other appropriate host classes. It then computes the cohesive clusters of data and behavior (concepts) and extracts corresponding classes out of the central class. All of these steps are described in terms of transformations on an abstract model of the system. When the problem is successfully removed on the model, a list of "real" refactorings is generated and transmitted to a code transformation tool, called Inject/J [10], which under user supervision, transforms the source code of the system.

The mappings between the anomaly fingerprint and the corresponding structural problem is determined by the diagnosis strategy, and the mapping between problem and solution is given by a *restructuring strategy*. The two types of strategies encapsulate all necessary knowledge to diagnose and respectively eliminate structural problems in object oriented systems.

3 Related work

Significant attempts towards automatic detection of structural anomalies have been made in [2, 6]. Concerning code refactorings as the low level mechanism to perform safe source code transformations we refer the reader to [3]. With regard to the so called "bad smells" and solutions defined in [3], our approach distinguishes itself in the fact that we treat bad smells as symptoms for higher level problems and that we describe formal, machine-processable "recipes" for the correction of these problems.

Along the same lines go [9] and [12], where a number of best practices in reengineering large object oriented systems is identified and formulated in the form of reengineering patterns and anti-patterns respectively. They have the disadvantage that they are strictly informal, because their purpose is to disseminate best practices, and not to be a support for automated restructuring.

In [4], opportunities of inserting design patterns to places in the source code where such patterns are missing, or present in a distorted form, are searched for. Although widely accepted as good practice, design patterns are not mandatory, therefore their absence is not necessarily problematic.

A more recent optimization approach for structure improvement, with good first results, can be found in [8]. The technique is based on applying genetic programming on a model in order to "evolve" the structure of a system. By operating on a simplified model, the process runs without human intervention. The result is a recommended, optimal structure of the model (with respect to the cost function defined), which needs to be implemented in the real system.

Significant work concerning tool support for automated introduction of design patterns has been done in [1]. Concerning low-level tool support for generic code transformations, we refer the reader to [10].

4 Conclusion

We presented a new method for the restructuring of large object oriented systems. The method builds on two important innovations: the encapsulation of constellations of structural anomalies into structural problems, and formally specified, reusable restructuring strategies that effectively support the decision making process of restructuring activities by employing modelling techniques. This method is a significant improvement over previous purely symptomatic approaches, which relied on informal descriptions of bad smells. We currently have seven restructuring strategies specified and significant parts of the infrastructure implemented. Their fingerprint definitions rely on about 25 different structural anomalies, taken from literature. Our current main priority for the near future is to bring the implementation into a state that allows us to validate the approach on real systems.

References

[1] M. Ó. Cinnéide and P. Nixon. A methodology for the automated introduction of design patterns. In *Proceedings of the ICSM*, 1999.

[2] O. Ciupke. *Problemidentifikation in objektorientierten Softwarestrukturen*. PhD thesis, Uni. Karlsruhe, 2001.

[3] M. Fowler. *Refactoring. Improving the Design of Existing Code*. Addison-Wesley, 1999.

[4] Y.-G. Guéhéneuc and H. Albin-Amiot. Using design patterns and constraints to automate the detection and correction of inter-class design defects. In *Proceedings of the TOOLS 39*, pages 296–305, 2001.

[5] Holger Baer et al. The FAMOOS object–oriented reengineering handbook, 1999.

[6] R. Marinescu. *Measurement and Quality in Object-Oriented Design*. PhD thesis, "Politehnica" University of Timişoara, 2002.

[7] A. Riel. *Object–Oriented Design Heuristics*. Addison–Wesley, first edition, 1996.

[8] O. Seng and G. Pache. Search based structure improvement. In *Proceeding of SET*, 2004.

[9] Serge Demeyer et al. *Object Oriented Reengineering Patterns*. Morgan Kaufmann, 2003.

[10] The Inject/J team. The Inject/J website. http://injectj.sourceforge.net.

[11] M. Trifu, P. Szulman, and V. Kuttruff. Das QBench Systemmetamodel. Technical report, FZI Forschungszentrum Informatik, Karlsruhe, 2004.

[12] William J. Brown et al. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, 1998.