Analyzing Dependencies to Solve Low-Level Problems*

Tamar Richner and Robb Nebbe
Software Composition Group, Institut für Informatik (IAM)
Universität Bern, Neubrückstrasse 10, 3012 Berne, Switzerland
(richner,nebbe)@iam.unibe.ch
http://iamwww.unibe.ch/∼(richner,nebbe)/

Summary

We have identified two levels of restructuring in the re-engineering of object- oriented legacy systems: high-level restructuring is concerned with improving the overall architecture of the system, whereas low-level restructuring deals with repairing local problems which are symptoms of bad style. We propose to characterize these low-level problems as patterns of dependencies between classes as an aid in detecting and resolving them. In this paper we briefly present low-level problems and give two examples of how these can be characterized as specific dependency patterns.

In the FAMOOS project we observed the following low-level problems in the industrial case studies:

misuse of inheritance: inheritance is used as a way to add missing behaviour to a superclass, instead of modeling the problem domain. This results in deep and narrow inheritance hierarchies.

missing inheritance: code duplication is used instead of subclassing, and long case statements are used instead of method dispatching.

misplaced operations: operations are defined outside of the class to which they should belong.

violation of encapsulation: classes frequently give access to private data, through the C++ friend mechanism.

missing encapsulation: classes are used as a structuring mechanism to encapsulate unrelated functions.

Solving these problems requires restructuring the code to a new, functionally equivalent system, analogous to the normalization of relational databases: the organization of the information is improved without changing the information content. Since most low-level problems can be characterized without domain-specific knowledge, their detection could be automated, and their resolution at least partly automated.

Low-level problems often manifest themselves as the presence of undesirable dependencies among classes. A tangle of dependencies in the code impedes understanding and maintenance of the software, especially for large projects. It is therefore important to minimize dependencies to the essential ones required to reflect the software design, and to avoid

*This research is supported by ESPRIT IV project no. 21975, and by Swiss National Science Foundation grant MHV 21-41671.94 (to T.R.)

circular dependencies which are an obstacle to modular compilation and testing [Sou94] [Lak96].

We thus characterize the low-level problems as patterns of dependencies among classes. These patterns of dependencies could be detected through query modules on a representation of the code. Furthermore, each pattern (or rather anti-pattern), in defining a problem, also suggests a solution. We give two examples:

Missing Enapsulation

Dependency Pattern: a class A has several unrelated clients, each using only a part of A's interface.

Symptom of: missing encapsulation

Solution: the interface of A is partitioned into groups of methods used by different kinds of clients. Class A is then factored out into several separate classes.

Misplaced Operations

Dependency Pattern: Circular dependencies between two classes

Symptom of: misplaced operations

Solution: create a new class which links the two dependent classes and whose methods break the circular dependencies

Similarly, other dependency anti-patterns are symptoms of misuse of inheritance (partial dependencies between clients and parts of the inheritance tree), missing inheritance (similar clients use different inheritance trees) and violation of encapsulation. We can thus characterize the different kinds of low-level problems as patterns of dependencies between classes. These dependency anti-patterns can then be detected in the code and the appropriate solution propagated using semi-automatic refactoring operations [JO93].

There are several issues to be addressed in implementing such low-level restructuring operations. First, we must refine the characterizations and catalogue the kind of information required to define each of these dependency patterns. We expect that a static analysis of the code would suffice. A second issue is which representation is best suited for query modules or recognizers [HYR96] to detect these patterns: a graph structure of dependencies may be a more appropriate representation than the abstract syntax tree. Thirdly, it remains open to what extent solutions can be propagated automatically using refactoring operations.

More general issues have to do with the benefit that we derive from 'repairing' low-level problems in re-engineering large industrial applications, where such static dependencies are only a small part of the problem. Ideally, we would like to be able to evaluate the importance of a problem to determine if it is worth solving or worth detecting in the first place.

References

[HYR96] D. Harris, A. Yeh, and H. Reubenstein. Extracting architectural features from source code. *Automated Software Engineering*, 3(1):109–139, 1996.

[JO93] R. Johnson and W. Opdyke. Refactoring and aggregation. In *Proceedings of ISOTAS '93 LNCS 742*, pp. 264–278, 1993.

[Lak96] J. Lakos. *Large-Scale C++ Software Design*. Addison-Wesley, 1996.

[Sou94] J. Soukup. *Taming C++*. Addison-Wesley, 1994.