16 Combining Clustering with Pattern Matching for Architecture Recovery of OO Systems

Markus Bauer
Mircea Trifu
FZI Forschungszentrum Informatik, Karlsruhe, Germany
{bauer,mtrifu}@fzi.de

16.1 Introduction

This work is concerned with recovering the architecture of an object-oriented software system based on information extracted from the source code. Over the years, several automatic techniques to recover subsystem decompositions have been proposed, such as clustering techniques or pattern-matching techniques, however none of them has the desired precision. The resulting decompositions are either not meaningful to a software engineer or they cover only pieces of the whole system ([3], [4]).

16.2 The Approach

In this paper, we propose an approach that combines clustering with pattern-matching techniques to recover complete and meaningful decompositions. Pattern-matching is used to identify architectural clues — small structural patterns that provide semantic information about the dependencies found between a system's entities. These clues are then used to compute an adaptive inter-class similarity measure which is then used by a clustering algorithm to produce the final system decomposition. In essence, the proposed approach tries to capture as much as it can from the original structure and then fill in the rest of the puzzle by imposing a suitable structure so as to minimize the coupling between the resulting subsystems and maximize their internal cohesion. Coupling and cohesion are expressed in terms of inter-class relationships and usage.

Our approach consists of five phases: *Fact extraction*, *Architectural clue gathering*, *Couplings adaptation*, *Compaction* and *Clustering*.

The purpose of the *Fact extraction* phase is to construct an object model of the source code in order to ease access to the information contained in it. The underlying metamodel[1] contains all the major syntactic elements and the interactions specifiable in a typical OO language such as: classes, methods, attributes, inheritance, aggregation, access, call, etc.

In the second phase, called *Architectural clue gathering*, the source model is decorated with semantic information. The information is incorporated in the model as annotations called *architectural clues*. One must point out that the information added in this phase is not essentially new. It is extracted from the already constructed source model by a set of *structural pattern recognizers*. As architectural clues we use *method types* — a classification of methods based on their semantic role, *library classes*, as well as seven GoF design patterns: *Template method*, *Abstract factory*, *Strategy*, *Composite*, *Proxy*, *Adapter* and *Facade*.

According to their semantic role, methods are classified based on three criteria: *Kind* (Abstract, Constructor, Constant, Empty, Accessor, Template, Factory, Delegating, Alias, Normal), *Inheritance Statute* (Implementing, Extending, Overriding, Adding, New) and *Usage* (Initialization, Public Interface, Protected Interface, Implementation).

Detection of library classes followed by coupling adaptation is our solution to the problem of omnipresent entities faced by other clustering approaches. We recognize library classes based on the number of clients that use them.

Recognizing design patterns can provide invaluable information about subsystem structure. Their presence usually points to a group of classes that belong together. For example, the presence of a *Composite* shows a strong coupling between the composite class and the aggregated component class.

In the *Couplings adaptation* phase, the annotated source model is reduced to a multigraph structure having classes as nodes and *coupling metrics* as edge values. For each of the syntactic interactions extracted in the first phase, a specific coupling metric is computed to show the strength of that particular type of interaction. Architectural clues are used to put each interaction in a wider context and adapt its corresponding metric value according to its semantic role in that context. There are six types of couplings that we consider: *Inheritance coupling*, *Aggregation coupling*, *Association coupling*, *Access coupling*, *Call coupling* and *Indirect coupling*. Following our previous example, the presence of a *Composite* pattern results in higher coupling metric values for the inheritance, aggregation as well as all the delegating calls between the composite class and the aggregated component class.

Indirect coupling expresses the coupling given by common usage. If two classes are constantly used together, it is likely that they are somewhat related even if no other direct relationship exists between them. To determine if two classes are used together, we consider only the calls to their public methods. If a method body contains calls to methods belonging to several classes, then between each pair of called classes, there is an indirect coupling.

Using indirect coupling was already suggested in the literature[2], however it has not been exploited yet. We

[1]For our meta-model we use MeMoJ: a metrics oriented meta-model for structural analysis of Java code developed at "Politehnica" University of Timişoara by Radu Marinescu, Daniel Raţiu and Mircea Trifu.

[2]A similar idea was proposed by Koschke in [2]

have found that indirect coupling is especially effective in grouping library classes together.

In the *Compaction* phase, the multigraph structure is reduced to a simple undirected graph. First we compute a weighted sum of the above mentioned couplings which we call *directed similarity* and then assign the maximum of the two directed similarities to the resulting undirected edge of the graph.

In the last phase, the undirected graph is clustered using a two-pass MST-like clustering algorithm to produce the final subsystem decomposition.

Further details can be found in [1].

16.3 Evaluation

We have developed an evaluation environment called ACT (Adaptive Clustering Testbed). ACT was written in Java, on top of MeMoJ and using RECODER[3] as fact extractor. Using ACT, we have made a comparative study of both the architecture-aware adaptive clustering technique and a conventional non-adaptive clustering technique.

The comparative study is based on two criteria: *Accuracy* and *Optimality*.

A recovered subsystem decomposition is considered accurate if it is “meaningful” to a software engineer. This means that the resulting subsystems should contain only semantically related architectural components and that all the semantically related architectural components should be in a single subsystem. We assess both techniques by comparing their resulting decompositions to specific reference decompositions (the original package structure and the ideal CRP structure) using the MoJo metric (see [5]). Further details and the argumentation for choosing these particular reference decompositions can be found in [1].

Optimality is measured using two metrics we have defined: *average cohesion* of the subsystems and *average coupling* between the subsystems of a given decomposition.

We have applied the above mentioned evaluation procedure on two case studies: *the Java AWT library* and *the SSHTools project* for three different parameters of the clustering algorithm.

For the *Java AWT library*, measurements show an average increase in accuracy (decrease of the MoJo value) of 19% for the architecture-aware adaptive clustering technique when comparing the decompositions to the original package structure and an average increase of 57% when comparing them to the ideal CRP structure. The average cohesion of the subsystems increased by 12% in the case of our approach. As for the average coupling values, they were slightly higher for our approach in 2 out of 3 experiments, but this is due to the fact that the non-adaptive approach created a much smaller number of large clusters thus turning many of the inter-cluster dependencies into intra-cluster dependencies.

In the case of the *SSHTools project*, the measurements revealed exactly the same thing as the ones made on the AWT library. The MoJo values show an average increase in accuracy for our approach of 23% when comparing the decompositions with the original package structure and an increase of 64% when comparing the decompositions with the ideal CRP structure. In the case of architecture-aware adaptive clustering, the optimality measurements show an average increase of 3% of the average cohesion metric and an average decrease of 10% of the average coupling metric.

The results presented in this section clearly show that architecture-aware clustering provides significantly better results than non-adaptive techniques both in terms of optimality and especially accuracy.

16.4 Conclusion

Our paper contributes to the software architecture recovery research by combining the strengths of clustering-based and pattern-based techniques. It proposes an approach which benefits from architectural clues that may be seen as traces of the high-level design of a system, the original software developers had in mind in early days of the system’s life span. These clues are used to guide an adaptive clustering process to recover that architecture.

Additionally, we have introduced a new indirect coupling metric for measuring the strength of coupling given by common usage and, to our knowledge, we are the first to use it to effectively cluster together library code, thus providing an elegant solution to the problem of omnipresent entities encountered in other clustering approaches.

We feel that our results of using architecture-aware adaptive clustering are very encouraging and we believe that further research in that direction is fully justified.

Bibliography

[1] M. Bauer and M. Trifu. Architecture-aware adaptive clustering of OO systems. In *Proceedings of the Eighth CSMR*, pages 3–14. IEEE, 2004.

[2] R. Koschke. *Atomic Architectural Component Recovery for Program Understanding and Evolution*. PhD thesis, Institute of Informatics, University of Stuttgart, Oct 1999.

[3] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E. R. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *Proceedings of the Sixth IWPC*, pages 45–52. IEEE, 1998.

[4] K. Sartipi and K. Kontogiannis. A graph pattern matching approach to software architecture recovery. In *Proceedings of the ICSM*, pages 408–419. IEEE, 2001.

[5] V. Tzerpos and R. C. Holt. Mojo: A distance metric for software clustering. In *Proceedings of the Sixth WCRE*, pages 187–193. IEEE, 1999.

[3]RECODER is an open source Java framework for source code meta-programming jointly developed at FZI Forschungszentrum Informatik Karlsruhe and the University of Karlsruhe.