

Architecture-Aware Adaptive Clustering of OO Systems

Markus Bauer
Forschungszentrum Informatik
Karlsruhe, Germany
bauer@fzi.de

Mircea Trifu
“Politehnica” University of Timișoara
Computer Science Department
Timișoara, Romania
mircea@cs.utt.ro

Abstract

The recovery of software architecture is a first important step towards re-engineering a software system. Architecture recovery usually involves clustering. The problem with current clustering techniques is that they decide exclusively based on syntactic dependencies instead of looking at higher-level semantic information. As a result, the recovered architecture is not always meaningful to a human software engineer.

In this paper, we propose an approach that combines clustering with pattern-matching techniques to recover meaningful decompositions. Pattern-matching is used to identify architectural clues — small structural patterns that provide semantic information to allow for a rating of the dependencies found between a system’s entities. These clues are used to compute an adaptive inter-class similarity measure which is then used by a clustering algorithm to produce the final system decomposition.

1 Introduction

With modern software systems, activities related to evolution become increasingly important. On one hand, there is the necessary repair-work for correcting bugs, and on the other hand, there is the enhancement-work for adding new functionality in order to fulfill new requirements. Adding new functionality to an existing software is a very delicate procedure. It takes a lot of expertise and careful revision of the architecture each time a new piece of functionality, that was not anticipated before, is added. However, anticipating future enhancements and providing hooks for their seamless integration without significant overhead may sometimes be impossible either because of time constraints or simply because some enhancements cannot be foreseen. As a result, software begins to “age” [15], its architecture begins to degrade as it is littered with new functionality. To make matters worse, documentation is almost never kept up-to-date

for every corrected bug, patch or hack. This makes subsequent enhancements very difficult to integrate in the current architecture, which is by now the biggest mystery in the whole project. Eventually, there is a point in time when taking shortcuts and adding hacks is easier than writing well-designed code. This is when you know that the software needs to be re-engineered.

As defined in the literature [3], *re-engineering* consists of *reverse engineering* a software system, *restructuring* the design to improve certain aspects and finally *forward engineering* it to a better system. *Reverse engineering* is the process of extracting design information from source code. *Architecture recovery* is a part of *reverse engineering* concerned with identifying architectural *components* such as subsystems, modules, objects as well as their interrelationships called *connectors*. The main purpose of architecture recovery is *program understanding* — a necessary precursor to restructuring a software system. This work is concerned with recovering the architecture of an object-oriented software system based on information extracted from the source code. The considered components of the architecture are subsystems and the connectors are the relationships between them. 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. Our goal is to bring automatically recovered subsystem decompositions closer to what a software engineer expert would produce by reverse engineering a system manually. In essence, we propose a method to produce complete and meaningful decompositions.

2 The Approach

Every software system has an architecture, which we cannot ignore if we want to understand the system. Even the worst systems, whose high-level architecture has degraded

beyond recognition due to changes, or was inexistent in the first place, have at least a set of practices or structural patterns that are used consistently in the different parts of the system. However, this information is not self-evident from the syntactic interactions found in the source code. In most cases, it is semantic design information that is not directly captured by the source code, and that should have been included in the system's documentation. It is a well-known fact that for most software systems the documentation is obsolete and the only reliable source of information about the system is the source code itself. Ultimately, we are faced with the task of understanding a system from the low-level, obfuscated representation given by the source code. When faced with such a task, a human expert would start by finding a suitable subsystem decomposition that would facilitate understanding. In order to do that, he would try to identify as much as possible from the original structure, if there is one, and then, based on his experience, fill in the rest of the puzzle, bringing structure where there is none.

In this paper, we propose to do the same thing automatically. We will recover as much as possible from the original decomposition and then, impose 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. The experience of a human expert concerning system structuring is captured in the form of architectural patterns or styles. These can be seen as blueprints for structuring a system and they provide some help to developers to derive a suitable system decomposition [2].

As architectural patterns are rather informal in their specification, they are extremely difficult, if not impossible, to effectively identify in the source code. It may even happen that a software system was using a particular pattern, but due to subsequent "enhancements", parts of the system no longer conform to that pattern. In this case, looking directly for architectural patterns is not a very good idea. Instead, we propose to do what every good detective would do. We gather *architectural clues* — small structural patterns which usually appear as parts of architectural patterns and provide hints to the existing subsystem structure. Figure 1 shows the most important architectural patterns and how our clues relate to them.

Based on these architectural clues, six inter-class adaptive coupling strengths are computed. The first five of them: *Inheritance coupling*, *Aggregation coupling*, *Association coupling*, *Access coupling* and *Call coupling* correspond to the five major syntactic interactions present in OO software. The strengths of these couplings are computed in an adaptive manner based on the context suggested by the architectural clues. The adaption process takes into account the roles these clues typically play in the implementation of architectural patterns. The last one is *Indirect coupling*

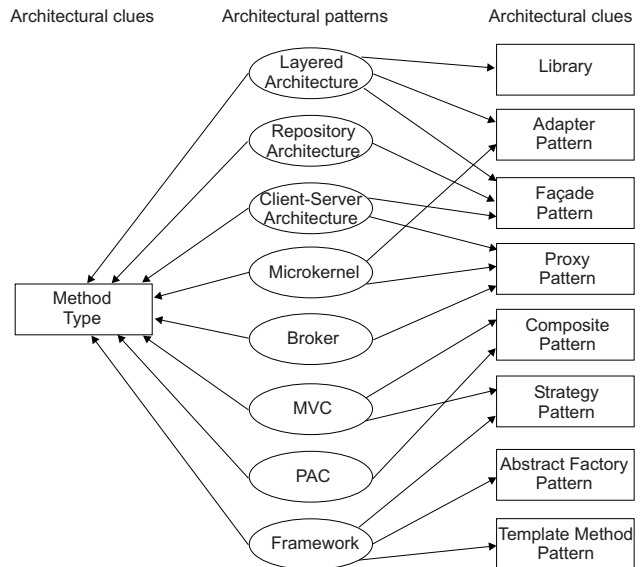


Figure 1. Architectural patterns and clues

which is used to group libraries together. These couplings are then pieced together to form a similarity measure between classes. This similarity measure is further used by a clustering algorithm to produce the final subsystem decomposition.

Figure 2 shows a schematic representation of the proposed five-phase approach. The rectangles represent the phases: *Fact extraction*, *Architectural clue gathering*, *Couplings adaptation*, *Compaction* and *Clustering*; while the ellipses show the different representations of the software system along the process, starting from the source code, until the final set of clusters.

2.1 Fact Extraction and Source Code Model

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. As shown in figure 3, the underlying meta-model 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.¹

In order to facilitate the discussion about detecting architectural clues, we used a Prolog-like formalism similar to the one presented in [10] and [4]. We can safely assume that the information contained in the source model has been expressed in the form of Prolog facts and added to the Prolog knowledge-base. Every class in the meta-model has a corresponding predicate. The list of predicates relevant to our discussion is given below:

¹Our meta-model is a variant of the FAMIX model [20].

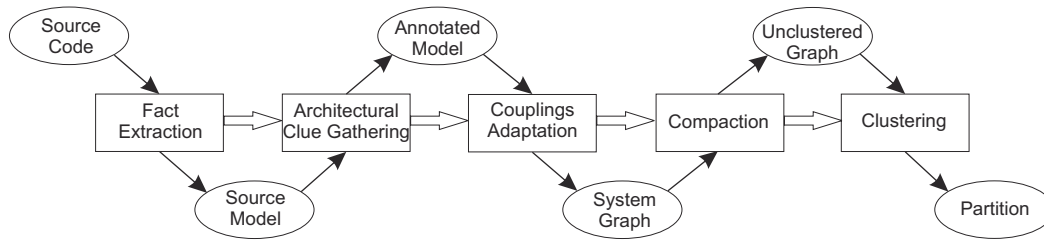


Figure 2. A schematic representation of architecture-aware adaptive clustering

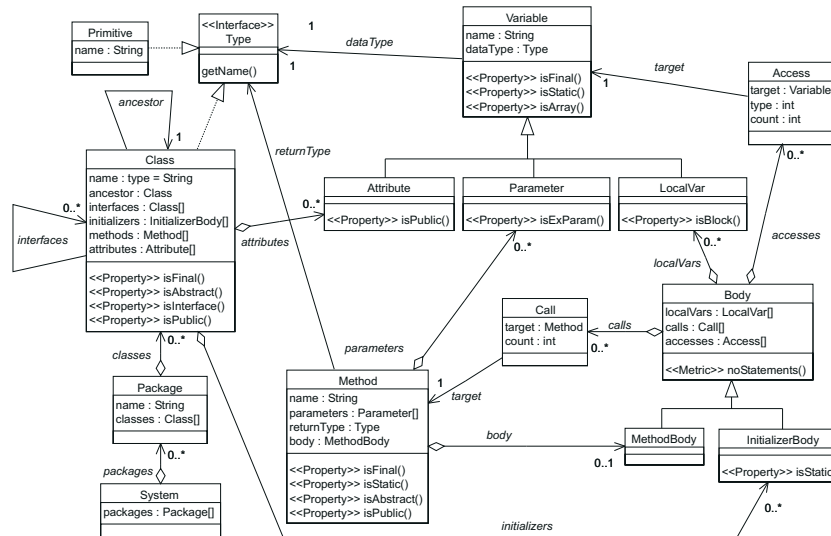


Figure 3. Meta-model

```

class(Class, Name, Abstract, Final, Interface,
      Visibility).
attribute(Class, Attribute, Name, Type,
          Static, Final, Visibility).
method(Class, Method, Name, Params, ReturnType,
        Abstract, Constructor, Static, Final,
        Visibility).
variable(Variable, Name, Type, Static, Final).
methodBody(Method, Body).
initializerBody(Class, Body).
call(Call, Body, Method, Count).
access(Access, Body, Variable, Count, AccessType).
bodyMetrics(Body, MetricNameList, ValueList).
subType(SubClass, SuperClass).
  
```

Most of the arguments are self-explanatory. *Visibility* refers to the member's access specifier. We used the term visibility in order to avoid confusion with the *access* predicate or its arguments. *AccessType* can have one of the values: *Read* or *Write* meaning read-only access or read-write access. Note that subsequent accesses to the same variable from within the same body are not recorded in the source model as separate accesses. The access is recorded only once and an associated counter holds the actual number of

accesses. The same is true for calls. Distinction between classes and interfaces is made by means of the *Interface* binary flag. *Abstract*, *Static*, *Final* and *Constructor* are also binary flags, having the expected meaning.

2.2 Architectural Clue Gathering

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 in the system, *library classes* as well as seven GoF design patterns[7]: *Template method*, *Abstract factory*, *Strategy*, *Composite*, *Proxy*, *Adapter* and *Facade*.

2.2.1 Method Classification

In order to get a idea about the semantic role of a particular method, a thorough investigation is required. Methods are classified according to three criteria: *Kind*, *Inheritance Statute* and *Usage*. The classifications given by the last two criteria are extensions of the ones given in [11]. The *kind* of a method gives an indication about what the method does. Currently, there are ten possible values for this criterion:

- *Abstract*
- *Constructor*
- *Constant* — a method that always returns a constant value
- *Empty* — a method with an empty body (no statements)
- *Accessor* (also known as getter or setter) — a method whose sole purpose is to facilitate access to a member attribute
- *Template* methods mark instances of the *Template method* design pattern
- *Factory* methods mark instances of the *Factory method* design pattern
- *Delegating* — a method whose sole purpose is to forward the call to another object
- *Alias* — a method which forwards the call to another method of the same class
- *Normal*

Abstract and *Constructor* methods are directly identifiable from the facts represented in the source model. For the other kinds of methods, a set of checking rules have been written. The rule for identifying template methods is shown below as an example. We have used a Prolog-like formalism which we describe in subsection 2.1. A template method may be identified by its calls to abstract or empty methods of the same class.

```
checkTemplate(Method) :-
    methodBody(Method, Body),
    call(_, Body, Met, _),
    method(Class, Method, _, _, _, _, _, _, _),
    method(Class, Met, _, _, _, _, _, _, _),
    ( methodKind(Met, Abstract);
      methodKind(Met, Empty) ).
```

The second criterion for classifying methods is the *Inheritance Statute*. This criterion divides methods in the following categories:

- *Implementing* — implementation of an abstract method

- *Extending* — a method that overrides a concrete method from a superclass and calls it as part of its own implementation
- *Overriding* — a method that overrides a concrete method from a superclass and does not call any of the methods of the superclass
- *Adding* — a newly added method that calls methods from a superclass as part of its implementation
- *New* — a newly added method that does not call methods from any of its superclasses

The *Inheritance statute* of a method is very useful in identifying frameworks as well as GoF patterns that involve inheritance such as *Composite*, *Abstract Factory* or *Strategy*.

Finally, the last criterion is called *Usage*. Methods are classified according to their usage by other methods in:

- *Initialization* — either constructors or methods whose name begins with “init”
- *Public Interface* — public methods, which are not called from within the same class,
- *Protected Interface* — same as *Public Interface*, except their Visibility is either *Protected* or *Package*
- *Implementation* — the rest

The three categories a particular method belongs to according to the above mentioned criteria are recorded and collectively added to the source model in the form of an annotation called *method type* to that particular method. In the annotated model, every method has a *method type* annotation attached to it.

2.2.2 Library Detection

One of the main problems with conventional clustering approaches is the so called “library problem”. Due to the fact that intensive use of a library may introduce strong coupling relations between the classes that use the library and the classes used from the library, a conventional clustering technique which is unaware of the fact that a software library is involved would most probably cluster together the classes involved. The same problem arises when the software system is structured using layered architecture. Conventional clustering methods do not effectively recover layers, instead they slice them and tend to cluster together classes that belong to several layers, several levels of abstraction. In the most unfortunate case of a nonuniform usage of the interface exported by the layer below (different classes from the layer above using different groups of

classes from the layer below), instead of recovering horizontal layers, a clustering-based approach is likely to recover vertical stripes. To a software engineer who knows that the system was structured into layers, the recovered decomposition is completely meaningless and misleading. As part of our architecture-aware adaptive clustering, we propose to identify possible library classes and mark them appropriately so that a proper adaptation of the coupling metrics can be performed in the *Couplings adaptation* phase. To do so, we have introduced a special annotation to the source model called *library marker* which we attach to every class that is discovered as being a library class. Library classes are discovered by a *library recognizer* which, in essence, counts the number of different classes that contain calls to one of the public interface methods of the suspected library class. If this number is greater than a given threshold, the suspect is considered a library class.

2.2.3 Pattern Detection

The architectural clues that we have covered so far represent useful semantic information about program entities, but sometimes they are too fine grained to offer relevant hints to the issue of subsystem decomposition. This is why we felt the need to identify higher-level clues which could give clear indications about subsystem decomposition. Such higher-level clues are design patterns. Most design patterns are used in key places where flexibility and future extension is needed to divide the responsibility of a complex task between its constituents so that separate aspects are kept separate. Such patterns are used inside a subsystem and their presence usually points to a group of classes that belong together. All the delegation patterns that we can identify: *Proxy*, *Adapter*, *Facade* and *Composite* plus the *Abstract factory* pattern fit into this category. However, there are a number of design patterns such as *Template method* and *Strategy* which are used to mark subsystem boundaries. *Template method* is generally used in conjunction with frameworks, while *Strategy* is used to separate a complex operation from the class that uses it. With the exception of *Template method* which is identified as part of the *method type* architectural clue, each design pattern is identified by its own *pattern recognizer*. As in the case of library classes, pattern constituents are attached a special annotation called *pattern marker*. There are separate markers for each design pattern and each such marker provides quick navigation access between its constituents. An example of such a *pattern recognizer* is given for the *Proxy* pattern. The idea behind a proxy is to create a placeholder that envelops a concrete object and delegates most of the tasks to the contained object. The uses for such a pattern are numerous, ranging from security checks to caching. As the proxy object was intended as a placeholder for a concrete object, its

interface must be the same, therefore its identification requires counting the delegations to methods having the same signature as the delegating one. In order to distinguish an instance of *Proxy* from a *Composite*, one must ensure that none of these delegating methods contain loops. Such a delegation would clearly point to a *Composite* pattern.

```
proxyRecognizer(Class, rDelegating, Target) :-
    findall(Met, ( method(Class, Method, Name,
                        Params, _, _, _, _, _, _),
                  methodKind(Method, Delegating),
                  methodBody(Method, Body),
                  call(_, Body, Met, _),
                  method(_, Met, Name, Param, _,
                        _, _, NoStatic, _, _) ),
            TargetMetList),
    findall(Met2, ( method(Class, Method, _, _, _,
                        _, _, _, _, _),
                  methodBody(Method, Body2),
                  call(_, Body2, Met2, _) ),
            CalledMetList),
    findall(Cls, ( present(Met3, TargetMetList),
                  method(Cls, Met3, _, _, _, _,
                        _, _, _) ),
            ClassList),
    present(Met4, TargetMetList),
    methodBody(Met4, Body4),
    bodyMetrics(Body4, [NoLoops], [0]),
    length(ClassList, 1),
    length(TargetMetList, x),
    length(CalledMetList, y),
    x/y >= rDelegating,
    present(Target, ClassList).
```

A class is considered a *proxy* for another class if the ratio between the number of delegating methods to same signature methods of a target class and the total number of methods in the candidate proxy is higher than a predefined threshold value, which we refer to as *rDelegating*.

A surprising side-effect of using the above given set of rules for *Proxy* identification is that the presence of another well-known design pattern is sometimes reported. This other pattern is the *Decorator* pattern. Since the two patterns are very similar in structure and usage, it is very difficult to differentiate between them. The problem is not particularly annoying since both patterns require the same actions to be taken with respect to clustering.

In addition to the already recovered clues, a set of user-specified clues can be added to the *annotated model*.

2.3 Couplings Adaptation

The third phase of the process, *Couplings adaptation*, is probably the most important one. It transforms the annotated model into a simplified multi-graph² structure called *system graph*, having classes as nodes and *coupling metrics*

²A multi-graph is a structure similar to a graph, except that multiple edges between two nodes are permitted.

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 between the source and destination classes. Architectural clues are used to put each interaction in a wider context and adapt its corresponding metric 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*
- *Indirect coupling*

For each of them a separate metric is computed for each pair of classes where the dependency occurs. As in the process of subsystem decomposition recovery only the classes are relevant, the loss of information, resulted by moving from the annotated source model to the system graph structure that contains only classes and coupling values, has no effect on the outcome. As an example, we examine the coupling adaptation rule for the *Aggregation coupling*:

The rule checks with the help of the annotated source model, whether the aggregated class and the containing class belong to one or more of the identified design patterns. If so, for each such pattern a fixed weight is added to the final aggregation strength according to table 1. If the two classes do not belong to such a pattern, three other cases are considered: interface aggregation, class aggregation and class aggregation with object creation. In each of these last three cases a different weight is assigned to the final aggregation strength.

Table 1. Context and weights used for the aggregation coupling adaption

Aggregation context	Weight
Facade	6
Composite	6
Proxy	5
Adapter	5
Factory	3
Strategy	3
Class aggreg. with object creation	2
Class aggreg.	1
Interface aggreg.	0.5

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 methods. If from inside a method body, there are calls to methods belonging to several classes, then between each pair of called classes, there is an indirect coupling. The contribution of each calling class to the indirect coupling between two classes is equal to the number of methods from the calling class containing calls to both called classes, divided by the total number of methods in the calling class. The formula is given below:

$$IndirectCoupling(A, B) = \sum_C \frac{noMethods}{methods(C)}$$

where C is the calling class, $noMethods$ is the number of methods from the calling class that contain calls to methods from both A and B , and $methods(C)$ is the total number of methods in class C .

The idea to compute the strength of the Indirect coupling in this fashion was sparked by the *Common Reuse Principle* — a well-known principle for subsystem granularity [14]. The principle states that a system is well structured if the following rule stands: whenever we reuse one of its subsystems, we reuse all the classes from that subsystem. Now, we cannot say anything about how subsystems will be reused, but if two classes from a subsystem are reused together, it is very likely that they are also used together in the original system. Using indirect coupling was already suggested in the literature³, but it has not yet been exploited. We have found that indirect coupling is especially effective in grouping library classes together.

2.4 Compaction

In the *Compaction* phase, the multi-graph structure is compacted to a normal graph so that it can be clustered using a graph clustering algorithm. The compaction is done by replacing all edges between a source and a destination node with only one edge. The cost associated with this single edge is a weighted sum of the above mentioned coupling metrics plus a custom coupling value. The role of this last, custom coupling is to allow the user to manually specify pairs of classes that should be clustered together. We use the following weighted sum, which we call directed similarity:

$$\begin{aligned}
 dsim(A, B) = & wInh * InheritanceCoupling(A, B) \\
 & + wAgg * AggregationCoupling(A, B) \\
 & + wAssoc * AssociationCoupling(A, B) \\
 & + wAccess * AccessCoupling(A, B) \\
 & + wCall * CallCoupling(A, B) \\
 & + wInd * IndirectCoupling(A, B) \\
 & + wCust * CustomCoupling(A, B)
 \end{aligned}$$

³A similar idea was proposed by Koschke in [9]

where A and B are classes, and $wInh$, $wAgg$, $wAssoc$, $wAccess$, $wCall$, $wInd$ and $wCust$ are user-adjustable weights given for each type of coupling.

Note, that the resulting graph is a directed graph. However, most of the clustering algorithms that operate on graphs (including ours) require an undirected graph. In order to move to an undirected graph, we use the maximum of the two directed similarities between two classes A and B as the similarity measure for the clustering.

$$sim(A, B) = max(dsim(A, B), dsim(B, A))$$

2.5 Clustering

The last phase of the process is called *Clustering*. In this phase, we use a two-pass modified MST⁴ clustering algorithm. Experiments with the original MST algorithm [21] have shown that the heuristic for identifying “long” edges is not very well suited for clustering software systems. Using the original heuristic, the algorithm tends to create a few large clusters containing many classes and having low internal cohesion, while several other classes remain separate, forming their own single-class clusters. Furthermore, using dissimilarities instead of the already available similarities requires additional normalizations and transformations.

Our two-pass algorithm was designed to overcome the above mentioned drawbacks. The first pass is the actual clustering algorithm, while the second pass is intended to assign the remaining unclustered classes to the cluster they are the “closest” to, rather than leave them to form their separate single-class clusters. The “closest” cluster of an unclustered class, means the cluster that contains the class adjacent to the edge with the highest value of all the edges incident to the unclustered class. This procedure is known in the literature as *orphan adoption* [25].

The clustering algorithm starts with a partition having each class in its own single-class clusters. The edges of the minimal spanning tree for the undirected graph are considered in descending order of their values. For each such edge, if it is not “too short”, two clusters are joined forming a new partition. As only the edges of the minimal spanning tree are considered and some of them are “too short”, the final partition will have more than one cluster. The algorithm is called *modified MST (MMST)* because we have replaced the original heuristic for “long” edges with the following one. An edge is considered too “short” if its value is lower than either of the products between the average value of all the edges of each of the two clusters that are joined by adding the edge and a *closeness factor*, supplied by the user as a parameter to the clustering algorithm. In essence, this factor represents the percentage of the average edge value

⁴A description of the original MST can be found in [27] and [21].

of a cluster that will be tolerated for the value of the new edge.

Note that our algorithm works directly with similarity values, instead of dissimilarities. Replacing the original heuristic had a positive impact on the internal cohesion and size of the resulting clusters, because the decision to join two clusters with an edge is taken based not only on local information about the adjacent nodes. Our algorithm produces small to medium-sized clusters, suitable for program comprehension.

3 Evaluation

In order to support the evaluation of the proposed approach, we have created an environment called ACT (Adaptive Clustering Testbed) which covers all of the five phases presented above. ACT was written in Java, on top of MeMoJ⁵ and using RECODER⁶ 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. For this reason, we have implemented our own non-adaptive clustering technique. To ensure objectivity, for both techniques we have used the same set of direct couplings for inheritance, aggregation, association, access and call, which we compute using the same formulas. We also use the same modified MST algorithm. There are only two differences. One is that, in the case of non-adaptive clustering, all the adaption rules are skipped, thus no adaptation based on architectural clues is performed. The other difference is that, in the case of architecture-aware adaptive clustering, we have used, in addition to the above mentioned couplings, our indirect coupling which we omitted from the formula of $dsim(A, B)$ in the other case.

The comparative study is based on two criteria: **Accuracy** and **Optimality**. **Scalability** is proven by means of time and memory measurements.

- **Accuracy.** A recovered subsystem decomposition is 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 compare the resulting decompositions produced by both the architecture-aware adaptive clustering and the conventional non-adaptive clustering with both the original package structure and the ideal CRP structure using

⁵MeMoJ is a metrics oriented meta-model for structural analysis developed at “Politehnica” University of Timișoara by Radu Marinescu, Daniel Rațiu and Mircea Trifu.

⁶RECODER is an open source Java framework for source code meta-programming developed at the University of Karlsruhe.

the MoJo⁷ metric. The ideal CRP structure is the ideal decomposition from the point of view of the Common Reuse Principle [14]. In order to obtain this decomposition, we applied our MMST algorithm on a set of similarities that were computed using only the indirect coupling. The MoJo metric counts the minimum number of basic operations (moves and joins) that must be performed to transform one decomposition to another. In essence, this metric shows how similar two decompositions are. It is clear, from the above mentioned description, that the lower the value of the MoJo metric, the more similar the two decompositions. Similarity to the original package structure means that subsystems contain only semantically related classes. We base this affirmation on the assumption that the original package structure was designed to reflect groups of semantically related classes. Similarity to the ideal CRP structure means that all the semantically related classes are in a single subsystem because they are consistently used together. In addition to these measurements, we also rely on manual inspection of the decompositions to prove that the architecture-aware adaptive clustering produces more accurate decompositions than its non-adaptive counterpart.

- **Optimality.** Most of the other clustering approaches are evaluated using some sort of optimality metric which shows that the resulting decompositions exhibit desirable attributes of the subsystem: high internal cohesion and low external coupling. In our case, optimality is just a secondary criterion. We wanted to prove that the superior accuracy of our approach is not achieved at the expense of optimality. To evaluate our approach based on this criterion, we have defined two metrics: *average cohesion* of the subsystems and *average coupling* between the subsystems of a given decomposition. The formulas to compute these metrics are given below:

$$avgCohesion(D) = \frac{\sum_{\substack{S_i \in D \\ |S_i| > 1}} \frac{noInternalEdges(S_i)}{\frac{|S_i|^2 - |S_i|}{2}}}{|D|^*}$$

$$avgCoupling(D) = \frac{\sum_{\substack{S_i, S_j \in D \\ i < j}} \frac{noExternalEdges(S_i, S_j)}{|S_i| * |S_j|}}{\frac{|D|^2 - |D|}{2}}$$

where D is a decomposition, $|D|$ is the number of subsystems in decomposition D , S_i is the i^{th} subsystem in D , $|S_i|$ is the number of classes in

subsystem S_i , $noInternalEdges(S_i)$ is the number of undirected edges between the classes of S_i and $noExternalEdges(S_i, S_j)$ is the number of undirected edges between classes from S_i and classes from S_j . Note that when computing the average cohesion, we do not consider single-class clusters as the internal cohesion of such clusters is undefined. $|D|^*$ is the number of subsystems that are not single-class subsystems in decomposition D . Also, $avgCoupling(D)$ is not defined for decompositions that contain a single cluster.

Both the **Accuracy** and **Optimality** related measurements were done for three different values of the *closeness factor* given as a parameter to the MMST clustering algorithm.

We have applied the above mentioned evaluation procedure on two case studies: *the Java AWT library* and *the SSH-Tools project*. The following subsection presents the results obtained for the former.

3.1 The Java AWT Library

The Java AWT Library is a collection of classes for creating lightweight user interfaces and for painting graphics and images. It is part of the standard Java platform. It is structured into 14 relatively large packages.

Table 2 presents time and size measurements for the Java AWT library.

From this table, we can clearly see that the only time and memory consuming phase is the fact extraction phase. Still, the size of the source model (38 MBytes) and the execution time (under 5 minutes) are reasonable for a project so large (more than 140,000 lines of code).

Next, table 3 presents the accuracy related measurements for the Java AWT library. The accuracy and optimality table headings contain the following abbreviations: **Pack** - the original package structure, **CRP** - the ideal CRP structure, **NA** - the decomposition produced by the non-adaptive clustering, and **A** - the decomposition produced by the architecture-aware adaptive clustering.

The results clearly show that the architecture-aware adaptive clustering produces more accurate decompositions than its non-adaptive counterpart. In the case of architecture-aware adaptive clustering, the values of the MoJo metric, although lower than in the case of non-adaptive clustering, are rather high for the comparison with the original package structure. This is due to the difference in average cluster size. Our MMST tends to create small clusters, while the original package structure contains a small number of large clusters. The problem will be discussed in more detail later on in this section.

The manual inspection of the clusterings revealed some very interesting results. We have compared only the cluster-

⁷For a description of the MoJo metric, see [24].

Table 2. Scalability related measurements for the Java AWT library

Java AWT Library	
Number of source files	345
Number of lines of code	142,242
Number of classes	482
Number of methods	5,534
Fact extraction	
Execution time	283,097 msec
Source model size	38,810,512 bytes
Number of model objects	56,698
Architectural clue gathering	
Execution time	5,888 msec
Annotated model size	39,123,448 bytes
Number of library classes	20
Number of GoF patterns	30
Couplings adaptation and Compaction	
Execution time	1,713 msec
System graph size	1,686,816 bytes
Number of nodes	482
Number of edges	2,519
Clustering	
Mean execution time	624 msec

ings produced for one value of the closeness factor, namely 0.75.

The decomposition produced by the architecture-aware adaptive clustering contained very few misplaced classes. The decomposition looked as if the original packages were chopped into smaller chunks. For example classes such as `Menu`, `MenuItem`, `MenuContainer`, `MenuShortcut` were effectively separated from the rest of the classes in the `java.awt` package and clustered together. The same thing happened for `TextArea`, `TextField` and `TextComponent` from the `java.awt` package or the classes from the `java.awt.event` or the `java.awt.peer` packages. Classes that are strongly related were separated from their package and clustered together. This could not be observed in the case of non-adaptive clustering.

Probably the most convincing result is the effective separation of the classes belonging to the `java.awt.event` and the `java.awt.peer` packages from their corresponding AWT components from `java.awt`, based on their semantic role as well as their level of abstraction. The architecture-aware adaptive clustering successfully separated 39 of the 44 classes of the `java.awt.event` package from their corresponding AWT components, while the non-adaptive version separated only 2. As for the `java.awt.peer` package, the architecture-aware adaptive clustering separated 22 of the 27 classes from this pack-

age, while the non-adaptive clustering separated only 16.

Table 4 presents the optimality related measurements. The abbreviations used to name decompositions are the same as those in table 3.

As the table shows, our architecture-aware adaptive clustering produced decompositions with higher average cohesion than the non-adaptive one. The higher values for the average coupling in two of the cases can be explained if we look at the corresponding average cluster size from table 3. The higher average cluster size of the non-adaptive decomposition suggests that some of the external edges between clusters from the architecture-aware adaptive decomposition became internal edges in the non-adaptive one, thus lowering the average coupling. All the values from the accuracy and optimality related measurements for both case studies as well as the manual inspection confirm that our architecture-aware adaptive clustering produces roughly the same decomposition consisting of many small clusters, even if we vary the closeness factor of the MST algorithm. This can be explained by the fact that the adaptation of couplings has a sharpening effect on the similarity values. Based on the context suggested by architectural clues, the value of a certain coupling can be amplified or attenuated so that the final set of similarity values reflects semantic closeness. We consider this stability of the resulting decomposition for various values of the closeness factor an advantage of our approach. One might say that we lose the ability to control the granularity of the clusters by varying this parameter. Not true, because using a low closeness factor does not control the granularity of the clusters. Our results have shown that a low closeness factor results in clustering together classes that do not belong together. This is due to the fact that the MMST algorithm decides to merge two clusters based on a single edge rather than all the edges between them. When the clusters are small (few classes) this assumption is reasonable, however as clusters become larger, deciding based on a single edge is ridiculous, to say the least. The correct way to obtain higher granularity clusters is to run the algorithm several times. Each run, except the first one, will use the clusters from the previous run as nodes and composite similarities as edges. Composite similarities may be computed as the sum of all the edges between two clusters, divided by the number of possible edges between the two clusters.

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

Table 3. Accuracy related measurements for the Java AWT library

Closeness factor	MoJo(Pack,-)		MoJo(CRP,-)		Average cluster size			
	NA	A	NA	A	Pack	CRP	NA	A
0.60	245	158	164	87	34.42	6.34	11.75	4.30
0.75	196	164	224	84	34.42	6.17	5.60	3.95
0.90	180	171	212	80	34.42	6.17	4.34	3.98

Table 4. Optimality related measurements for the Java AWT library

Closeness factor	Average cohesion				Average coupling			
	Pack	CRP	NA	A	Pack	CRP	NA	A
0.60	0.242	0.700	0.674	0.760	0.003	0.005	0.003	0.007
0.75	0.242	0.699	0.688	0.804	0.003	0.005	0.006	0.007
0.90	0.242	0.699	0.744	0.800	0.003	0.005	0.007	0.007

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. Manual inspection has also revealed that the accuracy of the ideal CRP structure is worse than the accuracy of the decomposition produced by our architecture-aware adaptive clustering. This was to be expected as the indirect coupling used to derive it has a rather narrow view of the system. Nevertheless, this does not make the comparison with the ideal CRP structure less relevant. As a result, we must conclude that these better decompositions are the result of both the adaptation based on architectural clues and the indirect coupling. We can safely say that our technique is a real step forward in program understanding at subsystem level.

4 Related Work

Current approaches for automatic architecture recovery fall into one of two categories: *clustering-based techniques* and *pattern-based techniques* [18].

4.1 Clustering-based Techniques

Clustering-based techniques [26] use a bottom up approach and generate architectural components by gradually grouping related system entities together using a similarity measure.

Wiggerts [26], Anquetil and Lethbridge [1], Tzerpos and Holt [23] are among the first to systematically explore the application of clustering algorithms for system modularization and architecture recovery. In their papers, they present the general state of the art on clustering algorithms, conduct experiments with these techniques in the context of software architecture recovery and analyze benefits and limitations of these techniques.

Mancoridis, Mitchell and others [13, 5, 12] treat clustering as an optimization problem: Find the system decomposition that optimizes a system modularization quality function which expresses, that the clusters of the decomposition should have high internal cohesion (intra-connectivity) and low external coupling (inter-connectivity). To solve the optimization problem, a genetic algorithm is used. A tool, *Bunch*, implements these ideas. An extensive set of case-studies is presented to prove the suitability of the approach for software architecture recovery. Later, they refine their approach by introducing techniques dealing with *omnipresent modules* (e.g. libraries) and incremental software structure maintenance (by *orphan adaption*), and ways to incorporate developer-know-how into the clustering process.

Most of the research on clustering techniques for architecture recovery is focused on legacy code written in procedural programming languages. Some of the techniques above can also be applied to object-oriented systems.

Rayside [16] applies an agglomerative clustering algorithm to group classes of an object-oriented system into modules. The clustering process is guided by a composite metric based on a weighted sum of inter-object relationships. Trifu [21] experiments with several different clustering algorithms as well as with dominance analysis. He establishes a four stage methodology for clustering software systems consisting of parsing and analysis, compaction, clustering and result interpretation. The clustering is directed by a similarity metric which takes four types of inter-class dependencies into account: inheritance, calls, variable accesses and type dependencies. Our approach uses many ideas presented in his work. In particular, we employ the same formula for measuring the strength of the call coupling between two classes. It incorporates a *scattering factor* which captures the distribution of calls over caller and called methods. Brito e Abreu [6] experiments with 7 clustering algorithms and 6 similarity metrics (which incorpo-

rate 13 coupling categories like inheritance, parameter in operation, message recipient).

All approaches use syntactic interactions as a basis for the clustering process. Syntactic interactions can give an accurate idea of how the classes or functions of a system use each other, but they fail to capture the rationale behind such a usage.

4.2 Pattern-based Techniques

Pattern-based techniques use a top-down approach. They rely on a pattern matching engine which searches for instances of user-modeled architectural patterns, also called conceptual architectures.

Sartipi and Kontogiannis [17, 18] employ such a technique. A software expert defines conceptual architectures expressed in a language called *AQL*. The source code of the system to be analyzed is parsed and transformed into a relational graph. The *AQL* description is also converted into a graph. The *AQL* graph contains composite nodes and composite edges. During a matching process, performed by an *A** search algorithm, the composite nodes are expanded to pattern regions and composite edges into edge-bundles and thus a relationship to the source graph is established. An experimental tool called *Alborz* was developed and tested on 6 open source case studies written in C.

Ciupke [4] uses a Prolog based query engine to identify typical design flaws in object-oriented systems. In his approach, a set of fact extractors analyze Java or C++ code and store structural information about the system in a Prolog fact base. The Prolog engine is then used to search for structural patterns that may point to common design flaws. The approach has been successfully applied to a number of industrial and open source case studies. The Prolog-based search technique scales surprisingly well for systems up to several million lines of code.

A vast number of authors propose pattern matching techniques for the detection of GoF design patterns.

Prechelt [10], for example, describes a tool called *Pat* which converts pattern descriptions, given as OMT diagrams into Prolog rules, and C++ code, extracted from the repository of the Paradigm Plus CASE tool, into Prolog facts. Then, a simple Prolog query detects all the instances of design pattern present in the analyzed code. It was tested on four case studies and it successfully detected: *Adapter*, *Bridge*, *Composite*, *Decorator* and *Proxy*. A comparable approach is presented in [19]. The approach detects the *Composite*, *Strategy* and *Bridge* patterns. Structural information extracted from Java source code is stored as a graph having classes, interfaces and methods as nodes and the following relations as edges: implements, extends, references, owns, calls and downcasts. Design patterns are formalized based on their structural relations using a formalism similar

to the one of set theory.

Heuzeroth et al. [8] combine static and dynamic analysis to identify interaction patterns. In the first stage, static analysis is used to produce a set of candidate patterns. Then, using dynamic analysis, communication protocols and constraints between objects as well as multiplicity of relations are checked to ensure a small number of false positives. The approach was successfully applied to recover *Observer*, *Composite*, *Mediator*, *Chain of responsibility* and *Visitor*.

Pattern-based techniques are very well suited to detect structural relationships. However, all pattern matching approaches detect only a very limited number of previously specified patterns. As a consequence, a pure pattern-matching approach will not be able to produce a complete decomposition of a software system into subsystems.

5 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. There are numerous ways to improve our technique. The fact extraction phase could be extended to support more detailed information about the interdependencies between the system's classes, e.g. cast expressions could be considered. The clue gathering phase could be refined to support additional architectural clues (e.g. the detection of the *Observer* pattern). The adaption phase is still subject to ongoing research. Proper weights have to be defined for each coupling situation. In the clustering phase, we plan to experiment with more sophisticated algorithms. Furthermore, we want to investigate how to turn a flat system decomposition (as it results for example from our MMST algorithm) into a hierarchical decomposition with nested subsystems. An idea for doing so was briefly suggested in subsection 3.1.

Another interesting line of research is, whether we can use our architecture-aware, adaptive clustering technique for quality assessments. Since our algorithms try to recover an ideal, clean subsystem structure, a comparison with the

system's current structure could lead to interesting insights about its quality.

References

- [1] N. Anquetil and T. C. Lethbridge. Experiments with clustering as a software remodularization method. In *Proceedings of the Sixth Working Conference on Reverse Engineering*, pages 235–255. IEEE, 1999.
- [2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture. A System of Patterns*, volume 1. John Wiley & Sons, 1996.
- [3] E. J. Chikofsky and J. H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, Jan 1990.
- [4] O. Ciupke. Automatic detection of design problems in object-oriented reengineering. In *Proceedings of the IEEE Conference on Technology of Object-Oriented Languages and Systems*, pages 18–32. IEEE, 1999.
- [5] D. Doval, S. Mancoridis, and B. S. Mitchell. Automatic clustering of software systems using a genetic algorithm. In *Proceedings of the IEEE Conference on Software Technology and Engineering Practice*, pages 73–81. IEEE, 1999.
- [6] F. B. e Abreu, G. calo Pereira, and P. Sousa. A coupling-guided cluster analysis approach to reengineer the modularity of object-oriented systems. In *Proceedings of the Fourth European Conference on Software Maintenance and Reengineering*, pages 13–22. IEEE, 2000.
- [7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.
- [8] D. Heuzeroth, T. Holl, G. Högström, and W. Löwe. Automatic design pattern detection. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, pages 94–103. IEEE, 2003.
- [9] R. Koschke. *Atomic Architectural Component Recovery for Program Understanding and Evolution*. PhD thesis, Institute of Informatics, University of Stuttgart, Oct 1999.
- [10] C. Krämer and L. Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In *Proceedings of the Third Working Conference on Reverse Engineering*, pages 208–215. IEEE, 1996.
- [11] M. Lanza and S. Ducasse. A categorization of classes based on the visualization of their internal structure: The class blueprint. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 300–311. ACM, 2001.
- [12] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 50–59. IEEE, 1999.
- [13] 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 International Workshop on Program Comprehension*, pages 45–52. IEEE, 1998.
- [14] R. C. Martin. Granularity. *C++ Report*, Nov/Dec 1996.
- [15] D. L. Parnas. Software aging. In *Proceedings of the Sixteenth International Conference on Software Engineering*, pages 279–287. IEEE, 1994.
- [16] D. Rayside, S. Reuss, E. Hedges, and K. Kontogiannis. The effect of call graph construction algorithms for object-oriented programs on automatic clustering. In *Proceedings of the Eighth International Workshop on Program Comprehension*, pages 191–200. IEEE, 2000.
- [17] K. Sartipi and K. Kontogiannis. A graph pattern matching approach to software architecture recovery. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 408–419. IEEE, 2001.
- [18] K. Sartipi and K. Kontogiannis. Pattern-based software architecture recovery. In *Proceedings of the Second ASERC Workshop on Software Architecture*. ASERC, 2003.
- [19] J. Seemann and J. W. von Gudenberg. Pattern-based design recovery of java software. In *Proceedings of the Sixth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 10–16. ACM, 1998.
- [20] S. Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, Institute of Informatics and Applied Mathematics, University of Bern, Sept 2001.
- [21] A. Trifu. Using cluster analysis in the architecture recovery of object-oriented systems. Master's thesis, Computer Science Department, "Politehnica" University of Timișoara, Sep 2001.
- [22] M. Trifu. Architecture-aware, adaptive clustering of object-oriented systems. Master's thesis, Computer Science Department, "Politehnica" University of Timișoara, Sep 2003.
- [23] V. Tzerpos and R. Holt. Software botryology. automatic clustering of software systems. In *Proceedings of the Ninth International Workshop on Database and Expert Systems Applications*, pages 811–818. IEEE, 1998.
- [24] V. Tzerpos and R. C. Holt. Mojo: A distance metric for software clustering. In *Proceedings of the Sixth Working Conference on Reverse Engineering*, pages 187–193. IEEE, 1999.
- [25] V. Tzerpos and R. C. Holt. Acdc: An algorithm for comprehension-driven clustering. In *Proceedings of the Seventh Working Conference on Reverse Engineering*, pages 258–267. IEEE, 2000.
- [26] T. A. Wiggerts. Using clustering algorithms in legacy systems remodularization. In *Proceedings of the Fourth Working Conference on Reverse Engineering*, pages 33–43. IEEE, 1997.
- [27] C. T. Zahn. Graph theoretical methods for detecting and describing gestalt clusters. *IEEE Transactions on Computers*, 20(1), Jan 1971.