Reducing the Class Coupling of Legacy Code by a Metrics-Based Relocation of Class Members

Marvin Ferber[1], Sascha Hunold[1], Björn Krellner[2],
Thomas Rauber[1], Thomas Reichel[2], and Gudula Rünger[2]

[1] University of Bayreuth, Germany,
{marvin.ferber,hunold,rauber}@uni-bayreuth.de
[2] Chemnitz University of Technology, Germany,
{bjk,thomr,ruenger}@cs.tu-chemnitz.de

Abstract. With the rapid growth of the complexity of software systems, the problem of integrating and maintaining legacy software is more relevant than ever. To overcome this problem, many methods for refactoring legacy code have already been proposed such as renaming classes or extracting interfaces. To perform a real modularization, methods have to be moved between classes. However, moving a single method is often not possible due to code dependencies.
In this article we present an approach to modularize legacy software by moving multiple related class members. It is shown how to identify groups of class members with similar concerns. We present two different code patterns that the related members and their dependent classes must match to allow a relocation of the related members. We also demonstrate how our pattern-based approach for automated modularization of legacy software can be applied to two open source projects.
Keywords: legacy software, class decoupling, pattern-based code refactoring, code metrics

1 Introduction

Many software systems that undergo evolution for several years can become legacy when the costs and time to maintain the software grows incrementally. Programmers handle this problem known as code decay [1] with continuous refactoring and system restructuring. Apart from the need for software analysis tools for an extensive system understanding [2], there is also the need for transformation tools that support the programmer while restructuring the legacy code. An important restructuring task is the modularization (decoupling) of classes, which helps to reduce the systems' complexity [3] and to adapt legacy systems to distributed environments [4].

In a coarse-grained modularization in object-oriented languages the basic elements are classes, which can be moved between packages. The operations to move classes between packages and preserve syntactic and semantic correctness are supported by most modern IDEs. To reduce class coupling, and to improve class encapsulation and cohesion between class members, a finer-grained modularization is desirable. This kind of modularization includes the relocation of member methods and variables into more appropriate classes and the splitting of oversized classes in order to separate inner class

concerns or services [5]. Moving member methods and variables without preserving correctness and appropriate test cases is an error-prone task [6]. Since programmers are often not familiar with the software and the dependencies of relocated methods or variables in detail, they are not aware of possible side effects. Fully automated transformations that preserve the behaviour of the software are desirable but not available in general.

In this article we focus on refactoring and modularizing legacy classes by moving methods between classes. Since the methods that should be moved may depend on other members of the considered class, e.g., variables or other methods, all related members have to be moved to preserve code integrity. We introduce the term *MemberGroup* as a set of related class members. A *MemberGroup* represents an encapsulated concern of a class and is therefore independent from other class members. Because *MemberGroups* do not have dependencies to other members of the containing class, they can be relocated to other classes to reduce inter-class dependencies.

The article is structured as follows. In the next section we introduce the related groups of member variables and methods called *MemberGroups*. Section 3 outlines two *MemberGroup* patterns that can be relocated automatically. Preconditions and transformation rules for the relocation task are also given. In Section 4 we discuss metrics to support the relocation task in order to reduce class dependencies. We show that the analysis and relocation of *MemberGroups* can be integrated into reengineering toolsets like TRANSFORMR [7]. We investigate the impact of the relocation of *MemberGroups* on two open source applications and show the achieved improvements in Section 5. Related work is discussed in Section 6 and Section 7 concludes the article.

2 *MemberGroups* in Classes

In this section we introduce the term *MemberGroup* as set of member variables and methods that represent an encapsulated concern of a class. To define such a group we describe a software system as a graph structure. Table 1 summarizes the nodes and edges that are necessary to describe all dependencies between the members of all classes of a software as a dependency graph.

The function $dep(u,v)$ between two members u and v represents a dependency between u and v in the source code. If there is a dependency $u \to v$ or a dependency $v \to u$ then $dep(u,v)$ returns True. To take transitive dependencies into account, we introduce the function $dep^*_{\mathcal{U}}(u,v)$, which indicates a *dep*-relation between u and v over a subset of members $\mathcal{U} \subseteq \mathcal{M}$. Therefore $dep^*_{\mathcal{U}}$ represents an undirected path in the dependency graph of the software system. With the definitions of Table 1 we define a *MemberGroup* $MG(m)$ in a class C starting with a method $m \in Meth(C)$ as follows:

$$\begin{aligned} MG(m) = \{m\} &\cup \{v \in Meth(C) \cup Var(C) \mid dep^*_{Meth(C)\cup Var(C)}(m,v) = \text{True}\} \\ &\cup \{u \in Con(C) \mid \exists\, t \in MG(m) \text{ with } dep(u,t) = \text{True}\}\,. \end{aligned} \tag{1}$$

A *MemberGroup* $MG(m)$ consists of the method m and contains all other member variables and methods of class C that depend on m. Additionally, all constructors belong to $MG(m)$ if they have a dependency to a member of the existing group $MG(m)$. As

Table 1. Elements of the graph structure used to model the software system.

Description	Definition
Set of all classes	$\Delta = \{C_0, \ldots, C_n\}$
Set of all constructors of a class C	$Con(C)$
Set of all methods of a class C	$Meth(C)$
Set of all member variables of a class C	$Var(C)$
Set of all members of a class C	$M(C) = Meth(C) \cup Var(C) \cup Con(C)$
Set of all members of a project	$\mathcal{M} = \bigcup_{C \in \Delta} M(C)$
Read/write/call reference from u to v (between arbitrary members u and v)	$u \rightarrow v$
Dependency of class members	$dep(u, v) = (u \rightarrow v) \vee (v \rightarrow u)$
Transitive dependency of class members over a subset of all members $\mathcal{U} \subseteq \mathcal{M}$	$dep^*_{\mathcal{U}}(u, v)$

a result a constructor can belong to more than one *MemberGroup* if the constructor initializes member variables or calls methods that belong to distinct *MemberGroups*.

The members of a *MemberGroup* do not have dependencies to other members of the same class and represent a separate concern (or functionality) of the class. The internal state (value of the member variables) of a *MemberGroup* can only be modified using members of the *MemberGroup*. To relocate an entire *MemberGroup*, several conditions have to be met, which are discussed in the next section.

3 Relocation of *MemberGroups*

We describe two *MemberGroup* patterns, which can be applied in order to improve modularization of the source code. The relocation of "misplaced" *MemberGroups* is based on transformation rules, which are different for each pattern. Since it is not possible to relocate every arbitrary *MemberGroup*, some preconditions are necessary to restrict a *MemberGroup* before applying transformation rules. The *common* pattern (Section 3.1) specifies a *MemberGroup* that is not restricted in its composition but in its usage inside the software system. In contrast to the *common* pattern the *strong* pattern (Section 3.2) characterizes a *MemberGroup* that is restricted in its structure. To describe the patterns, the target classes for the *MemberGroup* relocation, and the transformation rules, we define several terms outlined in Table 2, which are used in the following sections.

The object-oriented concept of inheritance can lead to further restrictions because of possible dependencies between members of the ancestor class and members of the specialized class. A *MemberGroup* can not be relocated if members of this *MemberGroup* are overridden by members of descendant classes or if they contain dependencies to ancestor classes. This is similar to methods in a *MemberGroup* that realize an interface. Other obstructive dependencies can be found between members of an inner class and a *MemberGroup*. In this case the restricted visibility of the inner class has

Table 2. Functions and sets of classes to define *MemberGroup* transformation rules.

Description	Definition
Type (class) of a variable v	$type(v)$
Visibility of a member u	$vis(u)$ (*public* or *private*)
Set of all types of parameters of a method m	$M_{par}(m)$
Set of classes which have a member variable of type S	$M_{has}(S)$
Set of classes in $\Delta \setminus \{S\}$ which each reference S and are referenced by all classes of the set Ψ	$M_{ref}(\Psi, S)$
Set of possible target classes for *MemberGroup* MG(m)	$M_{tar}(MG(m))$

to be taken into account when relocating the *MemberGroup* to another class. Even if relocation strategies in these contexts might be available, we focus on the relocation of members without these constraints in this article. The absence of dependencies caused by inheritance, interfaces, and inner classes concerning a *MemberGroup* are preconditions for the relocation of both patterns. Advanced programming techniques such as multithreading or reflection are not considered in this article.

To improve the modularization, a *MemberGroup* is relocated to a target class. After introducing the patterns a possible set of target classes is presented. These classes have to be determined according to the specific restrictions of the *MemberGroup* pattern. The transformation process is done by a transformation function `moveMG`, which automatically relocates a *MemberGroup* from a source class to a target class if all prerequisites for the relocation are satisfied.

3.1 Moving *Common MemberGroups*

To improve the modularization, a target class T has to be found to which a *common MemberGroup* in class S can be relocated safely, i.e., preserving the behavior of the software system. Therefore, the software system has to be inspected to locate possible target classes. A reference to a target class must exist in all classes that use the *common MemberGroup* (call methods or use member variables of the *MemberGroup*) of S. We assume that the source class S and the target class T are available as member variables in all classes that use the *common MemberGroup*. This excludes possible source and target classes that are used as local variables or as parameters in a method call. All classes that contain a member variable of type S and use a *common MemberGroup* in S are denoted as set $M_{has}(S)$:

$$M_{has}(S) = \{c \in \Delta \mid \exists x \in Var(c) \text{ with } type(x) = S\} \ . \tag{2}$$

$M_{has}(S)$ is further inspected to find possible target classes that have to be member variables of all classes of $M_{has}(S)$. The set $M_{ref}(\Psi, S)$ with Ψ as a subset of Δ contains classes that are used as types of member variables in all classes of Ψ and are not equal to S.

$$M_{ref}(\Psi, S) = \{t \in \Delta \mid t \neq S \text{ and } (\forall v \in \Psi \ \exists x \in Var(v) \text{ with } type(x) = t)\} \ . \tag{3}$$

The set $M_{tar}(MG(m))$ defines the set of possible target classes for the relocation of a *common MemberGroup* $MG(m)$ located in the source class S. According to Equations (2) and (3) we define:

$$M_{tar}(MG(m)) = M_{ref}(M_{has}(S), S)\,. \tag{4}$$

A matching pair of member variables, one of type S and one of type T must be found to relocate the *common MemberGroup*. There can be multiple occurrences and instantiations of the source and target types within the containing class. Thus, different cases have to be taken into account. For the number of occurrences of the source and target types we can distinguish four cases. $\mathcal{V}(1{:}1)$ specifies the case in which there is one occurrence of a member variable of the source and one of the target type in the containing class. In the $\mathcal{V}(1{:}n)$ case a member variable of the source type appears once and the target type occurs multiple times as member variable in the containing class (e.g., in an array). The cases $\mathcal{V}(n{:}1)$ and $\mathcal{V}(n{:}m)$ are accordingly defined.

In a related way the number of instantiations of member variables of the source and target types have to be taken into account. Multiple instantiations of the source class cause reinitializations of the member variables of the *MemberGroup*. When relocating a *MemberGroup* its behaviour has to be preserved as well, e.g., a reinitialization of the source class must cause a reinitialization of the relocated *MemberGroup* in the target class. In an $\mathcal{I}(1{:}1)$ relation both member variables of type S and T are instantiated once (e.g., in the constructor of the containing class). In an $\mathcal{I}(1{:}n)$ relation T is instantiated at least twice. The cases $\mathcal{I}(n{:}1)$ and $\mathcal{I}(n{:}m)$ also exist.

Each individual case requires different relocation strategies to preserve the behaviour of the software. In the following we consider the case in which the source and the target types occur only once in all classes that use the *common MemberGroup* ($\mathcal{V}(1{:}1)$). Additionally, we assume that both member variables are instantiated only once ($\mathcal{I}(1{:}1)$).

One of the target classes that meet all preconditions is selected. We use code metrics to obtain a suitable candidate, which is described in Section 4. We summarize our preconditions for the relocation of a *common MemberGroup* from a class S to a class T as follows:

a) If a constructor is part of the *MemberGroup*, it must be possible to move the *MemberGroup*-dependent statements to the constructor of the target class T. Therefore these statements must not depend on other variables not available in the constructor of class T.
b) The source and target classes must be member variables in all the *MemberGroup* referencing classes and are permitted to occur only once ($\mathcal{V}(1{:}1)$).
c) Both variables of type S and T are supposed to be instantiated once (e.g., in the constructor of the referencing class) to meet the $\mathcal{I}(1{:}1)$ requirement.

The algorithm `moveCommonMG` relocates a *common MemberGroup MG* from a given source class S to a target class T. The reference to our software graph (project P) is required to modify all references from the source class to the target class in all classes of $M_{has}(S)$.

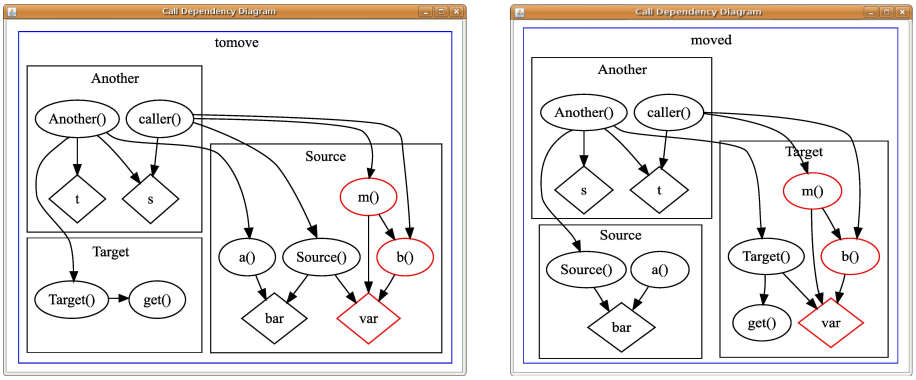

Fig. 1. TRANSFORMR call diagrams. Ellipses denote member methods and rhombi denote member variables. Left: Relocatable *common MemberGroup* {`m(Target), b(), var`} in class Source. Right: Relocated *MemberGroup* into `Target`.

```
1  function moveCommonMG(source S, membergroup MG, target T,
                         project P)
2  begin
3    resolveNameClashes(MG, T);
4    for each member M in MG:
5      if M in constructors(S):
6        moveConstructorStatements(S, M, T)
7      else:
8        moveMember(S, M, T)
9    replaceMemberGroupReferences(P, MG, S, T);
10 end
```

The transformation process consists of the following steps. As a pre-transformation step all name clashes between the target class T and the *common MemberGroup* MG, which will be relocated to T, are resolved (line 3). Therefore the function detects equal identifiers, uniquely renames them, and updates referencing statements. Although renaming is necessary to avoid name clashes, a huge number of automatic renamings may lead to unreadable code. To avoid this effect, human interaction can be integrated in the renaming task. Afterwards, all constructor statements (line 6) and members (line 8) of the *common MemberGroup* are moved to the target class by the methods `moveConstructorStatements` and `moveMember`. Finally, all references to members of the *common MemberGroup* of the original class S are updated to references to the relocated members in the extended class T by the method `replaceMemberGroupReferences` (line 9). Fig. 1 shows the relocation of the *common MemberGroup* {`m(), b(), var`} from class `Source` to class `Target`.

3.2 Moving *Strong MemberGroups*

The *strong MemberGroup* pattern presented in [8] defines a special type of *MemberGroups* that consist of exactly one public method m and a set of private members.

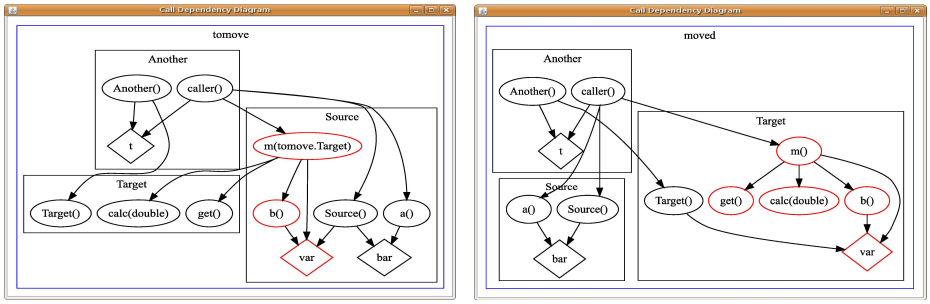

Fig. 2. TRANSFORMR call diagrams. Left: Movable *strong MemberGroup* {`m(Target)`, `b()`, `var`} with exactly one publicly used method and the target class as parameter. Right: Relocated *MemberGroup* with decoupled classes `Source` and `Target`.

Fig. 2 shows an example of a *strong MemberGroup*, emphasized in class `Source`. A *strong MemberGroup* can be built upon the (*common*) *MemberGroup* definition of Equation (1) defined as follows:

$$\begin{aligned} MG_{strong}(m) = \{m : vis(m) = public\} \\ \cup \{v \in \{MG(m) \setminus Con(C)\} \mid v \neq m : vis(v) = private\}. \end{aligned} \tag{5}$$

Possible target classes are taken from the parameters of the method m. The motivation for the relocation is a high cohesion between the *strong MemberGroup* and the class of the parameter as well as the removal of the class dependency between the source class and the class of the parameter. As a precondition for the possible target classes we require the type of the relevant parameter not to be an interface. The target classes $M_{tar}(MG_{strong}(m))$ for the relocation of the *strong* pattern are given as:

$$\begin{aligned} M_{tar}(MG_{strong}(m)) = \{p \mid p \in M_{par}(m) \text{ and} \\ p \text{ is not an interface or a primitive type}\}. \end{aligned} \tag{6}$$

In the *strong MemberGroup* pattern there is only one occurrence of variables of the source and target type involved ($\mathcal{V}(1{:}1)$). To relocate a *strong MemberGroup* to a target class in M_{tar} the following preconditions have to be fulfilled in order to preserve the behaviour of the software system.

a) Like in the *common MemberGroup* pattern, the dependent constructor statements of *strong MemberGroups* must be movable as well.
b) To keep the state of the *strong MemberGroup* inside the target class, it must be guaranteed that source and target classes are not reinstantiated ($\mathcal{I}(1{:}1)$).

If a *strong MemberGroup* pattern is found and all previous preconditions are fulfilled, we can relocate the *MemberGroup* $MG_{strong}(m)$ to the target class $T \in M_{tar}$ using the relocation function `moveStrongMG`:

```
1  function moveStrongMG(source S, membergroup MG, target T,
                         project P)
2  begin
3    resolveNameClashes(MG, T);
4    for each member M in MG:
5      if M in constructors(S):
6        moveConstructorStatements(S, M, T);
7      else:
8        moveMember(S, M, T);
9    replacePublicMethodParameterByTarget(MG, T)
10   replaceMemberGroupReferences(P, MG, S, T);
11   removePublicMethodParameter(P, MG, T);
12 end
```

Similar to the relocation algorithm for the *common MemberGroup* the function `moveStrongMG` resolves overlapping identifier names and moves the members and dependent constructor statements (lines 3–8). After the relocation of the *strong MemberGroup* the formerly used variable of type *T*, passed to the method *m* as parameter, is unnecessary. The occurrences of the parameter inside the *strong MemberGroup* are redirected to the target type (line 9). The relocation has to be propagated to the referencing classes of the public method *m*. All references to classes that used the *strong MemberGroup* inside the source class *S* are replaced by references to the target class (line 10). Finally, the formerly used parameter in the method *m* can be removed in all occurrences of the method *m* (line 11).

Fig. 3 displays the source code of class `Another` (illustrated in Fig. 2). The class is shown before (left) and after (right) the relocation of the *strong MemberGroup*. Line 8 contains the result of the relocation function: Method `m` and all other members of the *MemberGroup* are relocated to parameter class `Target` followed by the removal of the parameter of `m`. An additional improvement can be achieved if the reference to the source class becomes unused in a class that originally invoked the publicly used method *m* of the relocated *strong MemberGroup* (line 7). In this case an additional class dependency between the classes `Another` and `Source` can be removed, which leads to another modularization improvement.

```
1  public class Another {
2    private Target t;
3    public Another() {
4      t = new Target();
5    }
6    public void caller() {
7      Source s = new Source();
8      s.m(t);
9    }
10 }
```

```
public class Another {
  private Target t;
  public Another() {
    t = new Target();
  }
  public void caller() {
    /* Source s = new Source(); */
    t.m();
  }
}
```

Fig. 3. Source code of class `Another` following the example of Fig. 2. Left: Original source code. Right: Resulting source code after moving of `m` from `Source` into `Target` with additional unused reference to `Source` (line 7).

4 Metrics and Tools for Analyzing and Relocating *MemberGroups*

We leverage code metrics to select an appropriate target class from a set of possible classes for the relocation of a *MemberGroup* (considered in Section 3). These metrics calculate the system improvement after the transformation by measuring the coupling between classes. A coupling between two classes exists if there is at least one dependency between members of both.

The Coupling Intensity (CINT) [9] is used to determine the number of inter-class dependencies, which can be removed by relocating the *MemberGroup*. From a set of possible target classes the class with the highest value of CINT is used as target class. The CINT metric of the *MemberGroup* $MG(m)$ and the members of the target class $T \in M_{tar}$ is defined as follows:

$$CINT(MG(m), T) = \left| \{ u \to v \mid u \in MG(m) \text{ and } v \in M(T) \} \right| . \tag{7}$$

This metric does not measure the overall number of removed class couplings, but it is used to estimate the system improvement in advance. The overall number of resulting class couplings (see Equation (8)) depends on the actual dependencies to other classes of the source and target class. A class coupling between the source class and another class might be removed if only the *MemberGroup* references this class. But the coupling remains if the source class has additional references to this class. In the same manner the number of class couplings of the target class might be influenced.

In order to measure the improvement of decoupling and modularization of the whole software after applying the transformations, we use the Class Coupling metric (CC) (c.p. the Coupling Factor [10]), which indicates the number of dependent class pairs in relation to the maximum possible number of dependent class pairs. The CC is defined as follows whereas n denote the overall number of classes in the system:

$$\begin{aligned} CC &= \frac{\sum_{i=0}^{n-1} \sum_{j=0, j\neq i}^{n-1} c(C_i, C_j)}{n^2 - n} \quad \text{with} \\ c(C_i, C_j) &= \begin{cases} 1, & \text{dependency between classes } C_i \text{ and } C_j \,, \\ 0, & \text{otherwise} \,. \end{cases} \end{aligned} \tag{8}$$

We intend to search for "misplaced" *MemberGroups* using the defined patterns and try to relocate the *MemberGroups* to other classes that fit better. Considering this, we state that a lower class coupling is an improvement in separation of concerns and modularization of a legacy software system.

To support the programmer in different reengineering and restructuring tasks we developed the toolset TRANSFORMR [7]. It extracts the software graph from the source code, based on abstract syntax trees, and offers transformation operations. As part of this work, the TRANSFORMR toolkit was extended to support the analysis and transformation of *MemberGroups*. This new module of TRANSFORMR is called the *MemberGroup* browser, see Fig. 4. The browser lists all *MemberGroups* in a selected software element and restricts the resulting *MemberGroup* list with user-defined constraints. With these constraints *MemberGroups* of a software system that match the *strong* or

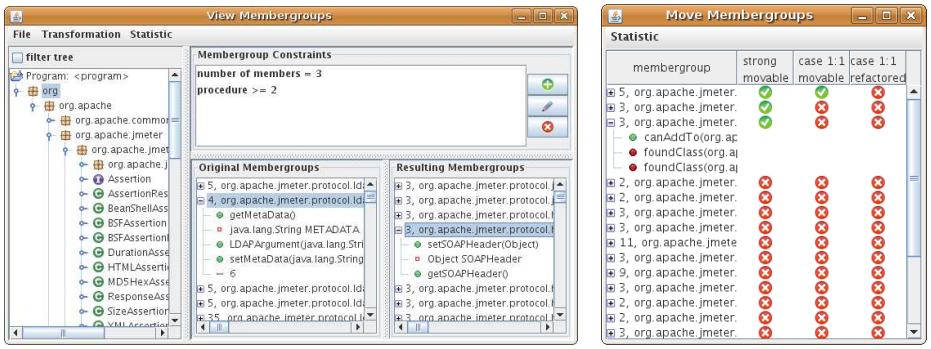

Fig. 4. Left: *MemberGroup* browser (in the tree view of the software), showing the list of all *MemberGroups* in the selected software graph node and two constraints limiting the *MemberGroup* list, arranged below. Right: Move *MemberGroup* dialog indicating the move possibilities.

the *common* pattern are determined and it is detected whether a *MemberGroup* is movable or not. In the next section the presented patterns and metrics are applied to two open source projects to test their usability and benefit in a real world evaluation.

5 Case Study

In a case study we applied our approach for *MemberGroup* detection and transformation on two open source projects written in Java: JMeter (ver. 2.3.2, 80 KLOC, 794 classes), a desktop application designed to test and measure performance of various network services, and jEdit (ver. 3.4, 95 KLOC, 940 classes), a cross platform text editor.

To measure the effect of applying the presented *MemberGroup* patterns on the two projects, a software graph was constructed using TRANSFORMR. Based on the graphs we calculated the original coupling metrics and tested the preconditions to move all *MemberGroups* found with the *MemberGroup* browser. A small number of movable *MemberGroups* was found on which we applied the transformations from Section 3 for each pattern. In order to gain information about the code improvement, we recomputed the CC metric and recorded the removed and newly added class dependencies for each pattern.

Table 3 lists the number of movable and not movable *MemberGroups* with the specific pattern for JMeter and jEdit. As we can see the number of movable *MemberGroups* is very low in relation to the number of the not movable groups. The main difficulty of the *strong* pattern is the absence of an appropriate parameter used as target class in the publicly used method. In about 90 % of the not movable *MemberGroups* the method had no parameters, the parameters were interfaces or primitive types, or were identified as external library classes. Additional 8 % of the not movable *MemberGroups* do not have any dependencies to the target class, so no improvement of the decoupling between source class and target class could be expected. A few not movable *MemberGroups* have members, which could not be moved because of inheritance restrictions. In case of the *common* pattern, the most frequent cause that a *MemberGroup* cannot be

Table 3. Number of movable and not movable *MemberGroups* in analyzed projects. The parenthesized values denote the identified but even so not movable *MemberGroups* (due to constraints not yet detectable by our toolset).

project, type of pattern	movable (false positives)	not movable
jEdit, *strong*	2 (15)	851
jEdit, *common*	3 (27)	1600
JMeter, *strong*	4 (8)	897
JMeter, *common*	6 (32)	1678

Table 4. Improvements in the number of pair couplings (p.c.). A lower number denotes coupling improvement and a higher number denotes a decline in the coupling,

project	original p.c.	*strong* p.c.	*common* p.c.	$\sum$
jEdit	4453	4455 (+0.04 %)	4449 (−0.08 %)	4451 (−0.04 %)
JMeter	3698	3694 (−0.1 %)	3697 (−0.03 %)	3693 (−0.14 %)

relocated was due to inheritance of members. Another reason for not movable *MemberGroups* was the assumption that the source and target classes are only used as member variables in all classes that use the *MemberGroup*. The source class of many not movable *MemberGroups* was additionally used as parameter or local variable, or the source class was never used as a member variable in all classes that use members of the *MemberGroup*. These causes sum up to about 95 % of all identified *common MemberGroups* in both investigated projects.

Since the toolset TRANSFORMR cannot yet detect whether a method realizes an interface or uses methods of an ancestor class of a library, some *MemberGroups* are incorrectly marked as movable (false positives in Table 3). This problem could be solved either by additionally inspecting all libraries with TRANSFORMR (possible for most open source projects) or by using sophisticated compiler tools (like Eclipse JDT) to detect the inheritance properties of methods.

Despite the previously mentioned problems, we found that we can reduce the code coupling by relocating *MemberGroups* with the outlined patterns. Table 4 shows the results with the number of pair couplings in both projects and the improvement of the CC metric after applying both patterns. There is only a small improvement of the coupling in both projects because we could only relocate a small number of *MemberGroups* (see Table 3). In summary all relocated *MemberGroups* lead to an improved modularization as dependencies between classes could be removed. Interestingly enough, the relocation leads to a higher coupling regarding the *strong* pattern for jEdit. This is due to couplings of the relocated *MemberGroup* that had to be added to the couplings of the target class. Unfortunately, in the source class these couplings could not be removed since the source class contains further dependencies to these classes. But even in this case we could decouple source and target classes. The results show that an automated relocation of *MemberGroups* can help to decouple software.

6 Related Work

Restructuring and refactoring to improve the quality of software in the context of software evolution has been intensively studied. Fowler et al. introduce many refactorings and design patterns for object-oriented languages as solutions for common mistakes in code style in order to make the software easier to understand and cheaper to modify [6]. Tests support the proposed manual changes to verify the correctness afterwards. A case study in a close-to industrial environment [11] also indicates that "refactoring not only increases aspects of software quality, but also improves productivity". Mens and Tourwé describe the need for automatic support but also state the limits and drawbacks of automated restructuring (e.g., untrustworthy comments, meaningless identifier names) [3]. They present transformation operations and discuss the behaviour preservation including the consistency of unit test results. In [5] an approach with evolutionary algorithm usage is described. By simulating refactorings with its pre- and postconditions classifications are done. The variation, which improves the class structure best, is chosen. With an open source case study the potential of the method is shown. The so-called search-based determination of refactorings is introduced in detail in [12]. A framework, which uses object-oriented metrics for detecting error-prone code for particular transformations, is proposed in [13]. After analyzing the impact that various transformations have on functional and non-functional software metrics, and by modeling the dependencies between design and code features, the potential to enhance the target qualities and requirements (*soft-goals*) for the new system can be shown. Automatic refactorings may also benefit from considering of antipatterns [14]. Brown et al. describe (in contrast to the various how-to-publications) common mistakes in every step of software engineering procedures and show better alternatives. These problem information can be used for metrics as well as for the manual choice of refactoring tasks. The points-to analysis [15] was initially developed for structured programming languages. It still has its eligibility for object-oriented languages, as references-to analysis [16]. The (dynamic) knowledge of referencing objects can support complex refactorings. Some of our current restrictions may be eased by using reference analysis.

7 Conclusions

This article presents a pattern-based approach to relocate sets of member variables and methods to automatically improve the modularization of legacy software systems. The key element of the described modularization of software is the so-called *MemberGroup*, which represents a separate concern or a service in a class. The *MemberGroup* is independent of the rest of a class and can therefore be relocated entirely to a target class. This relocation however has several constraints on the structure of a *MemberGroup* as well as on the referencing classes. We show which constraints have to be fulfilled in order to automatically apply the relocation operations on the legacy software without user interaction. We also demonstrate how the relocation of *MemberGroups* can be integrated into reengineering toolsets, such as TRANSFORMR. In a case study the approach was applied to two open source systems and the results have shown that the automatic relocation of *MemberGroups* can lead to less coupled legacy code (less code dependencies).

In future work we intend to enhance the *common MemberGroup* pattern with additional relocation strategies to rise the number of possible *MemberGroup* relocations. The integration of inheritance and language specific constructs like inner classes in the *MemberGroup* definition is also a promising goal for additional options to modularize.

Acknowledgment

The transformation approach described in this article as well as the associated toolkit are part of the results of the joint research project called TransBS funded by the German Federal Ministry of Education and Research.

References

1. Eick, S.G., Graves, T.L., Karr, A.F., Marron, J.S., Mockus, A.: Does Code Decay? Assessing the Evidence from Change Management Data. IEEE Transactions on Software Engineering 27(1), 1–12 (2001)
2. Binkley, D.: Source Code Analysis: A Road Map. In: Future of Software Engineering (FOSE), pp. 104–119. IEEE Comp. Soc. Washington, DC, USA (2007)
3. Mens, T., Tourwé, T.: A Survey of Software Refactoring. IEEE Transactions on Software Engineering 30(2), 126–139 (2004)
4. Serrano, M.A., Carver, D.L., de Oca, C.M.: Reengineering Legacy Systems for Distributed Environments. Journal of Systems and Software 64(1), 37–55 (2002)
5. Seng, O., Stammel, J., Burkhart, D.: Search-Based Determination of Refactorings for Improving the Class Structure of Object-Oriented Systems. In: Proc. of the 8th Annual Conf. on Genetic and Evolutionary Computation (GECCO), pp. 1909–1916. ACM, New York, NY, USA (2006)
6. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional, Massachusetts (1999)
7. Hunold, S., Korch, M., Krellner, B., Rauber, T., Reichel, T., Rünger, G.: Transformation of Legacy Software into Client/Server Applications through Pattern-Based Rearchitecturing. In: Proc. of the 32nd IEEE International Computer Software and Applications Conf. (COMPSAC), pp. 303–310 (2008)
8. Hunold, S., Krellner, B., Rauber, T., Reichel, T., Rünger, G.: Pattern-based Refactoring of Legacy Software Systems. In: Proc. of the 11th International Conference on Enterprise Information Systems (ICEIS), pp. 78–89 (2009)
9. Lanza, M., Marinescu, R., Ducasse, S.: Object-Oriented Metrics in Practice. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2006)
10. Brito e Abreu, F., Carapuça, R.: Object-Oriented Software Engineering: Measuring and Controlling the Development Process. In: Proc. of the 4th International Conference on Software Quality (ASQC), McLean, VA, USA (1994)
11. Moser, R., Abrahamsson, P., Pedrycz, W., Sillitti, A., Succi, G.: A Case Study on the Impact of Refactoring on Quality and Productivity in an Agile Team. In: 2nd IFIP TC 2 Central and East European Conf. on Software Engineering Techniques (CEE-SET), Revised Selected Papers, pp. 252–266 Poznan, Poland, Springer (2007)
12. O'Keeffe, M., í Cinnéide, M.: Search-based Refactoring for Software Maintenance. Journal of Systems and Software 81(4), 502–516 (2008)
13. Tahvildari, L., Kontogiannis, K.: Improving Design Quality Using Meta-pattern Transformations: a Metric-based Approach. Journal of Software Maintenance 16(4-5), 331–361 (2004)

14. Brown, W.J., Malveau, R.C., McCormick III, H.W., Mowbray, T.J.: Anti-Patterns: Refactoring Software, Architectures, and Projects in Crisis. First edn. John Wiley & Sons, Inc. (1998)
15. Rayside, D.: Points-To Analysis. Technical report, MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, USA (2005)
16. Ryder, B.G.: Dimensions of Precision in Reference Analysis of Object-oriented Programming Languages. In: Proc. of the 12th International Conf. on Compiler Construction, pp. 126–137. Warsaw, Poland, Springer, Heidelberg (2003)