Identification of Move Method Refactoring Opportunities

Nikolaos Tsantalis, *Student Member*, *IEEE*, and Alexander Chatzigeorgiou, *Member*, *IEEE*

Abstract—Placement of attributes/methods within classes in an object-oriented system is usually guided by conceptual criteria and aided by appropriate metrics. Moving state and behavior between classes can help reduce coupling and increase cohesion, but it is nontrivial to identify where such refactorings should be applied. In this paper, we propose a methodology for the identification of Move Method refactoring opportunities that constitute a way for solving many common Feature Envy bad smells. An algorithm that employs the notion of distance between system entities (attributes/methods) and classes extracts a list of behavior-preserving refactorings based on the examination of a set of preconditions. In practice, a software system may exhibit such problems in many different places. Therefore, our approach measures the effect of all refactoring suggestions based on a novel Entity Placement metric that quantifies how well entities have been placed in system classes. The proposed methodology can be regarded as a semi-automatic approach since the designer will eventually decide whether a suggested refactoring should be applied or not based on conceptual or other design quality criteria. The evaluation of the proposed approach has been performed considering qualitative, metric, conceptual, and efficiency aspects of the suggested refactorings in a number of open-source projects.

Index Terms—Move Method refactoring, Feature Envy, object-oriented design, Jaccard distance, design quality.

1 Introduction

According to several principles and laws of object-oriented design [18], [25], designers should always strive for low coupling and high cohesion. A number of empirical studies have investigated the relation of coupling and cohesion metrics with external quality indicators. Basili et al. [3] and Briand et al. [7] have shown that coupling metrics can serve as predictors of fault-prone classes. Briand et al. [8] and Chaumun et al. [12] have shown high positive correlation between the impact of changes (ripple effects, changeability) and coupling metrics. Brito e Abreu and Melo [11] have shown that Coupling Factor [10] has very high positive correlation with defect density and rework. Binkley and Schach [4] have shown that modules with low coupling (as measured by Coupling Dependency Metric) require less maintenance effort and have fewer maintenance faults and fewer runtime failures. Chidamber et al. [14] have shown that high levels of coupling and lack of cohesion are associated with lower productivity, greater rework, and greater design effort. Consequently, low coupling and high cohesion can be regarded as indicators of good design quality in terms of maintenance.

Coupling or cohesion problems manifest themselves in many different ways, with *Feature Envy* bad smell being the most common symptom. Feature Envy is a sign of violating the principle of grouping behavior with related data and occurs when a method is "more interested in a class other than the one it actually is in" [17]. Feature Envy problems can be solved in three ways [17]: 1) by moving a method to the class that it envies (Move Method refactoring); 2) by extracting a method fragment and then moving it to the class that it envies (Extract + Move Method refactoring); and 3) by moving an attribute to the class that envies it (Move Field refactoring). The correct application of the appropriate refactorings in a given system improves its design quality without altering its external behavior. However, the identification of methods, method fragments, or attributes that have to be moved to target classes is not always trivial since existing metrics may highlight coupling/cohesion problems but do not suggest specific refactoring opportunities.

Our methodology considers only Move Method refactorings as solutions to the Feature Envy design problem. Moving attributes (fields) from one class to another has not been considered, since this strategy would lead to contradicting refactoring suggestions with respect to the strategy of moving methods. Moreover, fields have stronger conceptual binding to the classes in which they are initially placed since they are less likely than methods to change once assigned to a class.

In this paper, the notion of distance between an entity (attribute or method) and a class is employed to support the automated identification of Feature Envy bad smells. To this end, an algorithm has been developed that extracts Move Method refactoring suggestions. For each method of the system, the algorithm forms a set of candidate target classes where the method can possibly be moved by examining the entities that it accesses from the system classes (system classes refer to the application or program under consideration excluding imported libraries or frameworks). Then, it iterates over the candidate target classes according to the number of accessed entities and the distance of the method from each candidate class. Eventually, it selects as the final

• *The authors are with the Department of Applied Informatics, University of Macedonia, 54006 Thessaloniki, Greece.*
E-mail: nikos@java.uom.gr, achat@uom.gr.

Manuscript received 15 Apr. 2008; revised 5 Dec. 2008; accepted 15 Dec. 2008; published online 6 Jan. 2009.
Recommended for acceptance by H. Ossher.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2008-04-0150.
Digital Object Identifier no. 10.1109/TSE.2009.1.

target class the first one that satisfies a certain list of preconditions related with the application of Move Method refactorings. The examination of preconditions guarantees that the extracted refactoring suggestions are applicable and preserve the behavior of the code.

Obviously, in large applications, several refactoring suggestions may be extracted hindering the designer to assess the effect of each refactoring opportunity. To this end, an Entity Placement metric is proposed to rank the refactoring suggestions according to their effect on the design. This metric is based on two principles: 1) The distances of the entities belonging to a class from the class itself should be the smallest possible (high cohesion) and 2) the distances of the entities not belonging to a class from that class should be as large as possible (low coupling).

The actual application of the refactoring suggestions on source code in order to calculate the Entity Placement metric value that the resulting systems will have can be very time-consuming, especially when the number of suggestions is large. The proposed methodology offers the advantage of evaluating the effect of a Move Method refactoring without actually applying it on source code. This is achieved by virtually moving methods and calculating the Entity Placement metric.

It should be emphasized that the proposed methodology is by no means a fully automatic approach. In other words, after the extraction of the refactoring suggestions, the designer is responsible for deciding whether a refactoring should be applied or not based on conceptual or other design quality criteria. For example, cases that require the designer's knowledge on the examined system are User Interface methods that should not be moved to classes holding data due to the Model-View-Controller pattern, test methods that should be not moved to the classes being tested (such cases are discussed in Section 5.1), and methods of a composing class that should not be moved to its contained classes due to composition relationships. The tool implementing the proposed methodology assists the designer in determining the reason behind selecting a specific target class over other possible target classes.

The evaluation of the proposed methodology consists of four parts. The first part contains a qualitative analysis of the refactoring suggestions extracted for an open-source project, along with some interesting insights obtained from the inspection and application of the suggestions. The second part studies the evolution of coupling and cohesion metrics when successively applying the refactoring suggestions extracted for two open-source projects. In the third part of the evaluation, an independent designer provides feedback concerning the conceptual integrity of the refactoring suggestions extracted for the system that he developed. The last part refers to the efficiency of the methodology based on the computation time required for the extraction of refactoring suggestions on various open-source projects.

The rest of the paper is organized as follows: Section 2 provides an overview of the related work. The proposed methodology is thoroughly analyzed in Section 3, and Section 4 presents the tool that implements it. The results of the evaluation are discussed in Section 5. Finally, we conclude in Section 6.

2 Related Work

According to Mens and Tourwé [26], the refactoring process consists of the following distinct activities:

1. Identify places where software should be refactored (known as bad smells) and determine which refactoring(s) should be applied.
2. Guarantee that the applied refactoring preserves behavior.
3. Apply the refactoring.
4. Assess the effect of the refactoring on quality characteristics of the software.
5. Maintain the consistency between the refactored code and other software artifacts (such as documentation, design documents, tests, etc.).

The proposed methodology covers activities 1-4 concerning Move Method refactorings.

Simon et al. [32] define a distance-based cohesion metric, which measures the cohesion between attributes and methods. This metric aims at identifying methods that use or are used by more features of another class than the class that they belong to, and attributes that are used by more methods of another class than the class that they belong to. The calculated distances are visualized in a three-dimensional perspective supporting the developer to manually identify refactoring opportunities. However, visual interpretation of distance in large systems can be a difficult and subjective task. The approach does not evaluate the effect of each refactoring on the architecture of the resulting system inhibiting the selection of those refactorings that will actually improve the design. Moreover, the case studies used for demonstrating their approach are small systems written by the authors with very obvious bad smells. Our approach is inspired by the work of Simon et al. in the sense that it also employs the Jaccard distance. However, the difference is that our approach defines the distance between an entity (attribute or method) and a class enabling the direct extraction of refactoring suggestions, while the approach of Simon et al. defines the distance between two entities and, thus, its output requires the application of clustering techniques or visual interpretation in order to extract Move refactoring suggestions to specific classes. To summarize, our methodology exhibits the following advantages:

1. It clearly indicates which methods and to which class they should be moved.
2. It suggests refactorings which are applicable and behavior-preserving by examining a list of preconditions.
3. It efficiently ranks multiple Move Method refactoring suggestions based on their positive influence on the design of the system.
4. It has been evaluated on large-scale open-source projects.

5. It has been fully automated and implemented as an Eclipse plug-in, allowing the developer to apply the suggested refactorings on the source code.

Tahvildari and Kontogiannis [33] use an object-oriented metrics suite consisting of complexity, coupling, and cohesion metrics to detect classes for which quality has deteriorated and re-engineer detected design flaws. In particular, they identify possible violations of design heuristics by assessing which classes of the system exhibit problematic metric values, and then select an appropriate metapattern that will potentially improve the corresponding metric values. A limitation of their approach is that it indicates the kind of the required transformation but does not specify on which specific methods, attributes, or classes this transformation should be applied (this process requires human interpretation). Moreover, in case of multiple potential suggestions, the approach does not evaluate their effect in order to rank them.

O'Keeffe and O'Cinneide [22] treat object-oriented design as a search problem in the space of alternative designs. For this purpose, they employ search algorithms, such as Hill Climbing and Simulated Annealing, using metrics from the QMOOD hierarchical design quality model [2] as a quality evaluation function that ranks the alternative designs. The refactorings used by the search algorithms to move through the space of alternative designs are only inheritance-related (Push Down Field/Method, Pull Up Field/Method, and Extract/Collapse Hierarchy).

Seng et al. [31] use a special model that examines a set of pre and postconditions in order to simulate the application of Move Method refactorings and a genetic algorithm to propose Move Method refactoring suggestions that improve the class structure of a system based on a fitness function. Their approach also includes an initial classification process which excludes from optimization, methods playing special roles in the system's design, such as getter and setter methods, collection accessors, delegation methods, state methods, factory methods, and methods participating in design patterns. The methodology proposed by Seng et al. has the following differences compared to our methodology:

1. It produces as output a sequence of refactorings that should be applied in order to reach an optimal system in terms of the employed fitness function. If the designer decides not to apply some of the suggested refactorings, then the resulting system might be worse than a system resulting from other sequences that have not been presented as solutions to the designer. Moreover, the application of the refactoring suggestions might lead to new refactoring opportunities (not originally present in the initial system), which are not taken into account in the proposed solution. On the contrary, a stepwise approach in which, after the application of each refactoring, the system is re-evaluated and a new list of refactorings that improve the current system is extracted (including any new refactoring opportunities that might have arisen), provides the possibility to the designer to assess the conceptual integrity of the suggestions at each step. Consequently, the designer is able to determine a sequence of refactoring applications that are conceptually sound and at the same time optimize certain software metrics.
2. It employs genetic algorithms that make random choices on mutation and crossover operations and, as a result, the outcome of each execution on the same input system may differ. Moreover, the outcome depends on initial parameter settings decided by the user. On the contrary, a deterministic approach which suggests refactorings based on Feature Envy criteria always results in the same solution for a certain system.
3. Its efficiency is limited by the following factors: a) It requires numerous generations in order to converge to a solution; b) the algorithm has to be executed several times (10 times in the example of the evaluation) in order to gather the common refactoring suggestions from all executions that will be reported as final results since each execution might lead to different results; and c) the algorithm includes in the optimization process all movable methods regardless of whether they suffer from Feature Envy problems or not.
4. It requires the definition of an arbitrary trapezoidal function for the normalization of certain metrics (such as WMC and NOM), a calibration run for optimizing each metric separately, and the specification of weights used in the definition of the employed fitness function. On the contrary, the Entity Placement metric does not rely on any arbitrary definition.

Concerning the evaluation of refactoring effect on design quality, the following approaches appear in the literature.

Kataoka et al. [21] propose a quantitative evaluation methodology to measure the maintainability enhancement effect of refactoring. They define three coupling metrics (return value, parameter, and shared variable coupling) in order to evaluate the refactoring effect. By comparing the metric values before and after the application of refactorings, they evaluate the degree of maintainability enhancement. The definition of each metric contains a coefficient that accounts for interclass coupling. The coefficient values are based on the specific characteristics of the system under study. However, the authors do not provide a systematic approach for estimating the coefficient values. Moreover, they did not include cohesion as a metric for evaluating the modification of maintainability caused by refactorings.

Du Bois et al. [15] theoretically analyze the best and worst-case impact of refactorings on coupling and cohesion dimensions. The refactorings they studied are Extract Method, Move Method, Replace Method with Method Object, Replace Data Value with Object, and Extract Class. According to the authors, moving a method that does not refer to local attributes or methods, or is called upon by only few local methods will increase cohesion. Additionally, moving a method that calls external methods more frequently than it is called will decrease import coupling. These observations are in agreement with the principles on which our methodology is based.

The cumulative effect of move refactorings (in the sense that their application eventually leads to a system where behavior and data are grouped together properly) could also be theoretically achieved by clustering techniques. However, the object-oriented clustering techniques found in the literature [27], [24] refer to the partitioning and

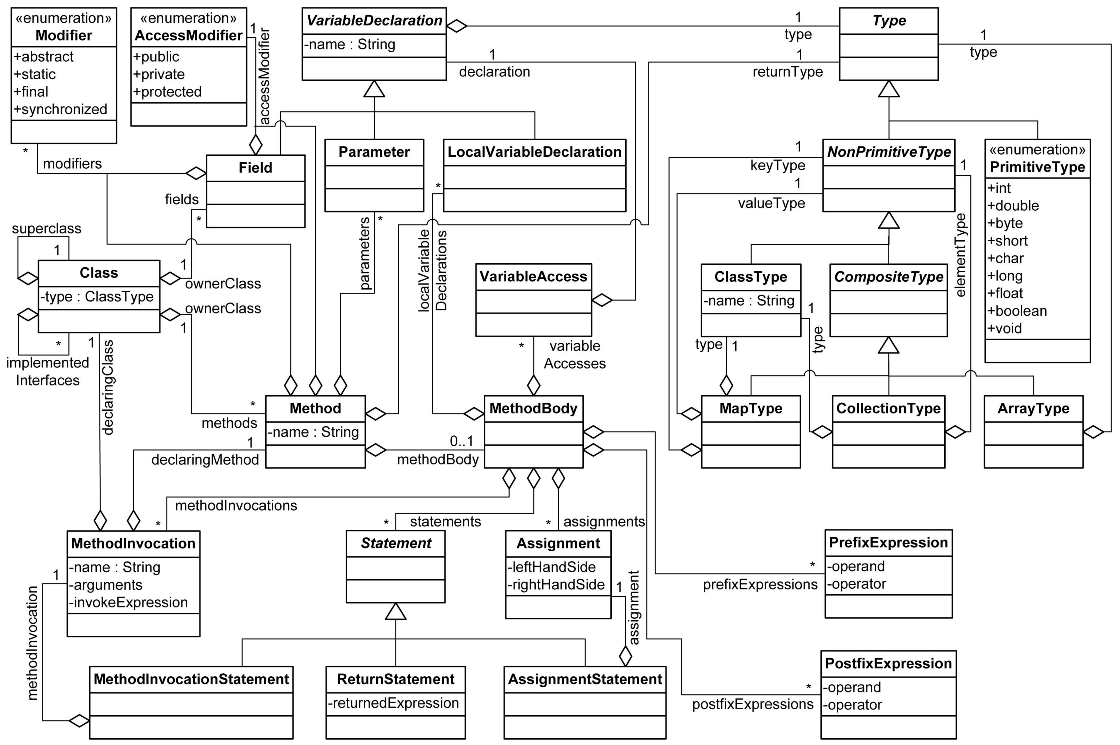


Fig. 1. UML model for the notation of the proposed methodology.

modularization of systems at class level rather than method level. Clustering techniques at method level could possibly lead to an optimal system in terms of coupling and cohesion. However, such techniques would present a solution that is an aggregate of multiple Move Method refactorings which the designer should accept or reject in its entirety. A stepwise approach, on the other hand, might not lead to an optimal solution but offers the advantage of gradual change of a system, allowing the designer to assess the conceptual integrity of the refactoring suggestions at each step.

3 METHODOLOGY

An object-oriented system is considered to be well-designed in terms of coupling and cohesion when its entities (attributes/methods) are grouped together according to their relevance. During analysis, relevance is usually evaluated on a conceptual basis. However, during design and implementation, relevance can be practically assessed considering the attributes and methods that a method accesses.

The notation required for the rest of the methodology is graphically illustrated in the UML class diagram of Fig. 1. The model represents the types, properties, and relationships which are necessary in order to identify Feature Envy bad smells for a given Java program.

Within the context of the above model, a program has as property the ClassTypes that it contains (denoted as program.classTypes), excluding imported library or framework class types.

3.1 Definition of Distance

A class in object-oriented programming consists of attributes and methods. Attributes may also be references to other classes of the system (i.e., attributes whose type is a system class), in order to provide access to the functionality of these classes. As a result, a method can access directly attributes and methods of the class that it belongs to and also attributes and methods of other classes through references. Likewise, an attribute can be accessed directly from methods of the class that it belongs to and also from methods of other classes that have a reference to that class.

For each entity (attribute/method), we define a set of the entities that it accesses (if it is a method) or the entities that it is accessed from (if it is an attribute).

The entity set of an attribute *attr* contains the following entities:

- the methods directly accessing *attr* that belong to the same class with *attr*;
- the methods accessing *attr* that belong to other classes of the system (accesses can be performed either through getter and setter invocations or in exceptional cases, directly when *attr* has public visibility).

The entity set of a method m contains the following entities:

- the directly accessed attributes that belong to the same class with m;
- the accessed attributes that belong to other classes of the system;
- the directly accessed methods that belong to the same class with m;
- the accessed methods through reference that belong to other classes of the system.

Apart from the entity sets of methods and attributes, the entity set of a class C is also defined and contains the following entities:

- all attributes that belong to class C;
- all methods that belong to class C.

For the formation of entity sets, the following rules should be taken into account. Rules are given in both a descriptive and a formal manner (auxiliary functions are defined in Appendix A).

1. Attributes that are references to classes of the system are not considered as entities nor added to the entity sets of other entities since such references are essentially a pipeline to the state or behavior of another class.

```
a. if $\exists f \in c$.fields where $f$.type $\in$ program.classTypes $\vee$
   ($elementType$ = elementTypeOfCollection($f$) $\neq$ null $\wedge$
   $elementType$ $\in$ program.classTypes)
   then do not add $f$ to the entity set of Class $c$
b. if $\exists$ $variable$ $\in m$.methodBody.variableAccesses
   where $variable$.declaration is
   Field $f$ $\wedge$ ($f$.type $\in$ program.classTypes $\vee$
   ($elementType$ = elementTypeOfCollection($f$) $\neq$ null $\wedge$
   $elementType$ $\in$ program.classTypes))
   then do not add $f$ to the entity set of Method $m$
```

2. Getter and setter methods are neither considered as entities nor added to the entity sets of methods and attributes, since they do not offer functionality except for access to attributes. However, the attributes to which they provide access are added to the entity sets. For an attribute that is a collection of objects, we consider as getters the methods that return an element at a specific position, or return an iterator/enumeration of the elements. As setters, we consider the methods that add an element to or replace an element of that collection.

```
a. if $\exists$ $m \in c$.methods where (isGetter($m$) $\neq$ null $\vee$
   isSetter($m$) $\neq$ null $\vee$ isCollectionGetter($m$) $\neq$ null $\vee$
   isCollectionSetter($m$) $\neq$ null)
   then do not add $m$ to the entity set of Class $c$
b. if $\exists$ $methodInv$ $\in m$.methodBody.methodInvocations
   where
   (Field $f$ = isGetter($methodInv$.declaringMethod) $\neq$ null $\vee$
   Field $f$ = isSetter($methodInv$.declaringMethod) $\neq$ null $\vee$
   Field $f$ = isCollectionGetter($methodInv$.
   declaringMethod) $\neq$ null $\vee$
   Field $f$ = isCollectionSetter($methodInv$.
   declaringMethod) $\neq$ null)
   then do not add $methodInv$.declaringMethod to the
   entity set of Method $m$
   if $f \neq$ null $\wedge$ ($f$.type $\notin$ program.classTypes $\vee$
   ($elementType$ = elementTypeOfCollection($f$) $\neq$ null
   $\wedge$ $elementType$ $\notin$ program.classTypes))
   then add $f$ to the entity set of Method $m$
```

3. Static attributes and methods are neither considered as entities nor added to the entity sets of methods and attributes, since they can be accessed or invoked from any method without having any reference to the class that they belong to. An instance method requires the existence of a reference to a target class in order to be moved to that class, and as a result, it cannot be moved to a class from which it accesses only static members.

```
a. if $\exists$ $f \in c$.fields where $f$.modifiers $\ni$ static
   then do not add $f$ to the entity set of Class $c$
b. if $\exists$ $m \in c$.methods where $m$.modifiers $\ni$ static
   then do not add $m$ to the entity set of Class $c$
c. if $\exists$ $variable$ $\in m$.methodBody.variableAccesses
   where $variable$.declaration is Field $f$ $\wedge$ $f$.modifiers $\ni$
   static
   then do not add $f$ to the entity set of Method $m$
d. if $\exists$ $methodInv$ $\in m$.methodBody.methodInvocations
   where $methodInv$.declaringMethod.modifiers $\ni$ static
   then do not add $methodInv$.declaringMethod to the
   entity set of Method $m$
```

4. Delegate methods are neither considered as entities nor added to the entity sets of methods, since they do not offer functionality except for delegating a responsibility to another method. However, the method to which they delegate is added to the entity sets. The treatment of delegations is recursive (in the case of a chain of delegations, only the final nondelegate method is considered).

```
a. if $\exists$ $m \in c$.methods where isDelegate($m$) $\neq$ null
   then do not add $m$ to the entity set of Class $c$
b. if $\exists$ $methodInv$ $\in m$.methodBody.methodInvocations
   where
   $nonDelegateMethod$ =
   finalNonDelegateMethod($methodInv$.declaringMethod) $\neq$
   null $\wedge$
   (Field $f$ = isGetter($nonDelegateMethod$) = null $\wedge$
   Field $f$ = isSetter($nonDelegateMethod$) = null $\wedge$
   Field $f$ = isCollectionGetter($nonDelegateMethod$) =
   null $\wedge$
   Field $f$ = isCollectionSetter($nonDelegateMethod$) = null)
   then add $nonDelegateMethod$ to the entity set of
   Method $m$
   if $f \neq$ null $\wedge$ ($f$.type $\notin$ program.classTypes $\vee$
   ($elementType$ = elementTypeOfCollection($f$) $\neq$ null
   $\wedge$ $elementType$ $\notin$ program.classTypes))
   then add $f$ to the entity set of Method $m$
```

5. In case of a recursive method, the method itself is not added to its entity set, since a self-invocation does not constitute a dependency with the class that the method belongs to.

```
if $\exists$ $methodInv$ $\in m$.methodBody.methodInvocations
where $methodInv$.declaringMethod = $m$
then do not add $methodInv$.declaringMethod to the entity
set of Method $m$
```

6. Access to attributes/methods of classes outside the system boundary (e.g., library classes) is not taken

into account. That is because, in our approach, the library classes are assumed to be fixed from the programmer's perspective and, therefore, are not subject to refactoring.

The similarity between a method and a class should be high when the number of common entities in their entity sets is large. In order to calculate the similarity of the entity sets, the *Jaccard similarity coefficient* is used. For two sets A and B, the Jaccard similarity coefficient is defined as the cardinality of the intersection divided by the cardinality of the union of the two sets

$$similarity(A, B) = \frac{|A \cap B|}{|A \cup B|}.$$

The *Jaccard distance* measures the dissimilarity between two sets. For two sets A and B, the Jaccard distance is defined as

$$distance(A, B) = \frac{|A \cup B| - |A \cap B|}{|A \cup B|} = 1 - \frac{|A \cap B|}{|A \cup B|} = 1 - similarity(A, B).$$

Let e be an entity of the system, C a class of the system, and S_x the entity set of entity or class x. The distance between an entity e and a class C is calculated as follows:

Definition 1. *If the entity e does not belong to the class C, the distance is the Jaccard distance of their entity sets*

$$distance(e, C) = 1 - \frac{|S_e \cap S_C|}{|S_e \cup S_C|}, where\ S_C = \bigcup_{e_i \in C} \{e_i\}.$$

Definition 2. *If the entity e belongs to the class C, e is not included in the entity set of class C*

$$distance(e, C) = 1 - \frac{|S_e \cap S'_C|}{|S_e \cup S'_C|}, where\ S'_C = S_C \backslash \{e\}.$$

In this way, we ensure that all distance values range over the interval [0, 1]. If we calculate the distance between a class and an entity that belongs to it without excluding e from the entity set of the class, the intersection of their entity sets can never be equal to their union and, thus, the distance could never obtain the value 0.

3.2 Move Method Refactoring Preconditions

According to Opdyke [28], each refactoring is associated with a set of preconditions which ensure that the behavior of a program will be preserved after the application of the refactoring. In order to describe the preconditions that should be satisfied for a Move Method refactoring in a formal manner, we define the following set of auxiliary functions:

```
boolean $matchingSignature$(Method $m_1$, Method $m_2$) $\equiv$
    ($m_1$.name $=$ $m_2$.name) $\wedge$
    ($m_1$.returnType $=$ $m_2$.returnType) $\wedge$
    (size of $m_1$.parameters $=$ size of $m_2$.parameters) $\wedge$
    for $i = 1$ to size of $m_1$.parameters
        $m_1$.parameters[$i$].type $=$ $m_2$.parameters[$i$].type
```

```
boolean $abstract$(Method $m$) $\equiv$
    $m$.ownerClass is interface $\vee$ $m$.modifiers $\ni$ abstract
```

```
(set of Field) $inheritedFields$(Class $c$) $\equiv$
    return $f \in$
    {inheritedFields($c$.superclass) $\cup$ $c$.superclass.fields}
    where $f$.accessModifier $\neq$ private
```

```
(set of Method) $inheritedMethods$(Class $c$) $\equiv$
    return $m \in$
    {inheritedMethods($c$.superclass) $\cup$ $c$.superclass.methods}
    where $m$.accessModifier $\neq$ private $\wedge \sim$ abstract($m$)
```

*In the case where class c does not explicitly inherit a superclass, then its superclass is `java.lang.Object.`

```
(set of Method)$abstractMethodsToBeOverriden$(Class $c$)$\equiv$
    return $m_1 \in$
    {abstractMethodsToBeOverriden($c$.superclass) $\cup$
    $c$.superclass.methods} where abstract($m_1$) $\wedge$
    ($\nexists\, m_2 \in c$.methods $\wedge$ matchingSignature($m_1, m_2$))
```

*An abstract method cannot be declared as final, static, or private.

```
(set of Method) $interfaceMethodsToBeImplemented$
(Class $c$) $\equiv$
    for $i = 1$ to size of $c$.implementedInterfaces
        return $m \in$ {interfaceMethodsToBeImplemented
        ($c$.implementedInterfaces[$i$]) $\cup$
        $c$.implementedInterfaces[$i$].methods}
```

*An interface may extend more than one interface.

The preconditions that should be satisfied for a Move Method refactoring are divided into three categories, namely, compilation preconditions which ensure that the code will compile correctly, behavior-preservation preconditions which ensure that the behavior of the code will be preserved, and quality preconditions which ensure that certain design quality properties will not be violated. In all of the precondition functions, the method parameter (m or m_1) refers to the method to be moved and the class parameter (t) refers to the target class.

3.2.1 Compilation Preconditions

1. The target class should not contain a method having the same signature with the moved method.

```
$noSimilarLocalMethodInTargetClass$(Method $m_1$,
Class $t$) $\equiv$
    $\nexists\, m_2 \in t$.methods $\wedge$ matchingSignature($m_1, m_2$)
```

   This issue can be resolved by renaming the moved method.

2. The method to be moved should not override an abstract method. Moving a method that overrides an abstract method would lead to compilation problems, since the overriding of abstract methods is obligatory for concrete classes.

```
$notOverridesAbstractMethod$(Method $m_1$) $\equiv$
    $\nexists\, m_2 \in$ {abstractMethodsToBeOverriden($m_1$.ownerClass)$\cup$
    interfaceMethodsToBeImplemented($m_1$.ownerClass)} $\wedge$
    matchingSignature($m_1, m_2$)
```

This issue can be resolved by keeping the original method as delegate to the moved method.

3. The method to be moved should not contain any super method invocations.
4. The target class should not be an interface, since interfaces contain only abstract methods and not concrete ones.

3.2.2 Behavior-Preservation Preconditions

1. The target class should not inherit a method having the same signature with the moved method. Moving a method which has the same signature with an inherited method of the target class would lead to the overriding of the inherited method, affecting the behavior of the target class and its subclasses.

$noSimilarInheritedMethodInTargetClass(\text{Method } m_1, \text{Class } t) \equiv$
$\nexists\, m_2 \in \text{inheritedMethods}(t) \wedge \text{matchingSignature}(m_1, m_2)$

This issue can be resolved by renaming the moved method.

2. The method to be moved should not override an inherited method. Moving a method that overrides a concrete method would affect the behavior of the source class and its subclasses since the source class would inherit the behavior of the method defined in its superclass.

$notOverridesInheritedMethod(\text{Method } m_1) \equiv$
$\nexists\, m_2 \in \text{inheritedMethods}(m_1.\text{ownerClass}) \wedge$
$\text{matchingSignature}(m_1, m_2)$

This issue can be resolved by keeping the original method as delegate to the moved method.

3. The method to be moved should have a reference to the target class either through its parameters or through source class fields (including inherited fields) of target class type. In order to preserve the behavior of the code, the methods originally invoking the method to be moved should be modified to invoke it through that particular reference after its move. On the contrary, a local variable of target class type declared inside the body of the method to be moved cannot serve as a reference to target class, since it is not accessible outside the method.

$validReferenceToTargetClass(\text{Method } m, \text{Class } t) \equiv$
$\exists\, variable \in m.\text{methodBody.variableAccesses where}$
$variable.\text{declaration} \in$
$\{m.\text{parameters} \cup m.\text{ownerClass.fields} \cup$
$\text{inheritedFields}(m.\text{ownerClass})\} \wedge$
$variable.\text{declaration.type} = t.\text{type}$

4. The method to be moved should not be synchronized. The synchronization mechanism of Java ensures that when one thread is executing a synchronized method of an object, all other threads that invoke synchronized methods of the same object suspend the execution until the first thread is done with the object. As a result, the move of a synchronized method could create concurrency problems to the objects of the source class.

3.2.3 Quality Preconditions

1. The method to be moved should not contain assignments of a source class field (including inherited fields). In that case, the assigned field cannot be passed as parameter to the moved method, since parameters are passed by value in Java, and as a result, the value of the field will not change after the invocation of the moved method. The alternative approach of passing a parameter of source class type to the moved method and invoking the setter method of the assigned field would increase the coupling between the source and target class, since the moved method would get coupled to the source class. Moreover, a method that changes the value of a field has stronger conceptual binding with the class to which the field belongs to compared to a method that simply accesses the value of the field.

$noSourceClassFieldAssignment(\text{Method } m) \equiv$
$\nexists\, assignment \in m.\text{methodBody.assignments}$
$\text{where } assignment.\text{leftHandSide} \in$
$\{m.\text{ownerClass.fields} \cup \text{inheritedFields}(m.\text{ownerClass})\} \wedge$
$\nexists\, postfixExpression \in m.\text{methodBody.postfixExpressions}$
$\text{where } postfixExpression.\text{operand} \in$
$\{m.\text{ownerClass.fields} \cup \text{inheritedFields}(m.\text{ownerClass})\} \wedge$
$(\nexists\, prefixExpression \in m.\text{methodBody.prefixExpressions}$
$\text{where } prefixExpression.\text{operand} \in$
$\{m.\text{ownerClass.fields} \cup \text{inheritedFields}(m.\text{ownerClass})\} \wedge$
$(prefixExpression.\text{operator} = \text{`}++\text{'} \vee$
$prefixExpression.\text{operator} = \text{`}--\text{'}))$

2. The method to be moved should have a one-to-one relationship with the target class. In this way, a method which participates in a one-to-many composition relationship cannot be suggested to be moved from the composing class (the source class that it originally belongs to) to the contained class (target class).

$one\text{-}to\text{-}oneRelationshipWithTargetClass(\text{Method } m, \text{Class } t) \equiv$
$\nexists\, variable \in m.\text{methodBody.variableAccesses where}$
$variable.\text{declaration} \in$
$\{m.\text{ownerClass.fields} \cup \text{inheritedFields}(m.\text{ownerClass}) \cup$
$m.\text{parameters} \cup$
$m.\text{methodBody.localVariableDeclarations}\} \wedge$
$((variable.\text{declaration.type is ArrayType } aType \wedge$
$aType.\text{type} = t.\text{type}) \vee$
$(elementType = \text{elementTypeOfCollection}(variable.$
$\text{declaration}) \neq \text{null} \wedge$
$elementType = t.\text{type}))$

3.3 Extraction of Move Method Refactoring Suggestions

The algorithm used for the extraction of Move Method refactoring suggestions is applied to all method entities of a system and consists of four main parts:

1. Identification of the set of candidate target classes T by examining the entity set of method m.

```
private TaskManager taskManager;
private LocationManager locationManager;

public boolean removeLocation(Location loc) {
  Task[] ts = taskManager.tasks();
  for (int i = 0; i < ts.length; i++) {
     if (ts[i].locationID() == loc.id())
          ts[i].setLocationID(
          locationManager.getLocationAnywhereInstance().id());
  }
  return locationManager.removeLocation(loc);
}
```

(a)

```
public class LocationManager {
  private ArrayList<Location> locations;

  public boolean removeLocation(Location l) {
     if (!locations.contains(l))
          return false;
     locations.remove(l);
     return true;
  }
}
```

(b)

Fig. 2. Example of method modifying a data structure of a candidate target class. (a) Method under examination. (b) Method invoked from candidate target class.

2. Sorting of set T according to the number of entities that method m accesses from each target class in descending order at the first level and according to the distance of method m from each target class in ascending order at the second level.
3. Examination of whether method m modifies a data structure in the candidate target classes.
4. Suggestion of moving method m to the first candidate target class that satisfies all of the preconditions, following the order of the sorted set T.

It should be noted that the Jaccard distance which is used for sorting the candidate target classes when the method under examination accesses an equal number of entities from two or more classes ensures that the candidate target classes will be examined in an order that promotes the classes having fewer entities. This property is desired since it leads to the decomposition of God classes [30] and the equal redistribution of functionality among the system classes. The notion of distance is also employed as a means to rank multiple refactoring suggestions, as will be explained in Section 3.4.

The third part of the algorithm aims at identifying cases where the method under examination modifies a data structure in a candidate target class by invoking an appropriate method of the target class and passing as argument one of its parameters. In such a case, we consider that the method under examination has a strong conceptual binding with the specific target class regardless of the number of entities that the method accesses from the candidate target classes. For example, in Fig. 2a, method `removeLocation(Location)` has three candidate target classes, namely, `TaskManager`, `LocationManager`, and `Location`. It accesses one entity from each candidate target class and none from the source class. It invokes method `removeLocation()` through field `locationManager` and passes parameter loc as argument to the invoked method. More importantly, method `removeLocation()` of class `LocationManager` (Fig. 2b) actually removes the passed argument from list `locations` that contains objects of Location class type. As a result, class `LocationManager` is considered a better choice for moving the method under examination compared to the other candidate target classes.

A formal description of the algorithm used for the extraction of Move Method refactoring suggestions is shown in Fig. 3.

Function *modifiesDataStructureInTargetClass*(Method m, Class t), which determines whether method m modifies a data structure in the candidate target class t, is formally described in Appendix B. Function *preconditionsSatisfied* (Method m, Class t) returns true if all preconditions of Section 3.2 are satisfied. In the case where method m accesses the same number of entities and has the same distance from two or more candidate target classes, then suggestions are extracted for all the classes for which the preconditions are satisfied.

3.4 Evaluation of Refactoring Effect on Design Quality

In a large software system, it is reasonable to expect that several Move Method refactoring suggestions will be extracted. In that case, we should be able to distinguish the most effective refactorings in terms of their impact on the design.

Our approach follows the widely accepted principle of low coupling and high cohesion [19]. To this end, the distances of the entities belonging to a class (inner entities) from the class itself should be the smallest possible (high

```
extractMoveMethodRefactoringSuggestions(Method m)
    T = {}
    S = entity set of m
    for i = 1 to size of S
        entity = S[i]
        T = T ∪ {entity.ownerClass}
    sort(T)
    suggestions = {}
    for i = 1 to size of T
        if (T[i] ≠ m.ownerClass ∧ modifiesDataStructureInTargetClass(m, T[i]) ∧
        preconditionsSatisfied(m, T[i]))
            suggestions = suggestions ∪ {moveMethodSuggestion(m→ T[i])}
    if suggestions ≠ ∅
        return suggestions
    else
        for i = 1 to size of T
            if T[i] = m.ownerClass
                return {}
            else if preconditionsSatisfied(m, T[i])
                return {moveMethodSuggestion(m→ T[i])}
    return {}
```

Fig. 3. Algorithm used for the extraction of Move Method refactoring suggestions.

cohesion). At the same time, the distances of the entities not belonging to a class (outer entities) from that class should be as large as possible (low coupling). This can be ensured by considering for each class the ratio of average inner to average outer entity distances. For each class, the closer to zero this ratio is, the safer it is is to conclude that inner entities have correctly been placed inside the class and outer entities to other classes. A formula that provides the above information for a class C is given by

$$EntityPlacement_C = \frac{\frac{\sum_{e_i \in C} distance(e_i, C)}{|entities \in C|}}{\frac{\sum_{e_j \notin C} distance(e_j, C)}{|entities \notin C|}},$$

where e denotes an entity of the system. In the special case where a class does not have inner entities, the above formula cannot be calculated.

The weighted metric for the entire system which considers the number of entities in each class is given by

$$EntityPlacement_{System} = \sum_{C_i} \frac{|entities \in C_i|}{|all\ entities|} EntityPlacement_{C_i}.$$

The lower the value of this metric is, the more effective the specific refactoring for the entire system is. The classes that do not have inner entities are not included in the above metric.

3.5 Virtual Application of Move Method Refactoring Suggestions

In order to evaluate which of the Move Method refactoring suggestions are the most effective ones, we could apply each one of them on source code and then recalculate the distances between the entities and the classes to measure the Entity Placement metric for each of the resulting systems. However, the actual application of the suggested refactorings on source code adds a significant overhead due to disk write operations (once for applying each refactoring and once for undoing it).

To overcome this problem, all suggested refactorings are virtually applied. This is achieved by updating the entity sets of the entities/classes which are involved in the move of the corresponding method and calculating the Entity Placement metric for the resulting entity sets.

The virtual move of a method from the source class to a target class is performed as follows:

1. The tag indicating to which class the method belongs is changed from source class to target class.
2. The entity sets of all methods accessing the method are updated according to the new tag.
3. The entity sets of all attributes that are being accessed by the method are updated according to the new tag.
4. The method is removed from the entity set of the source class.
5. The method is added to the entity set of the target class.

The distances which have to be recalculated after the virtual application of a refactoring are: 1) the distances from the source and the target class of the entities whose entity set has been affected from the virtual application (i.e., methods that access the moved method and fields being accessed from the moved method) and 2) the distances from the source and the target class of the entities whose entity set contains at least one entity of the source and/or the target class. The rest of the distances remain unchanged since the entity sets of the classes that do not participate in the refactoring are the same compared to the initial system.

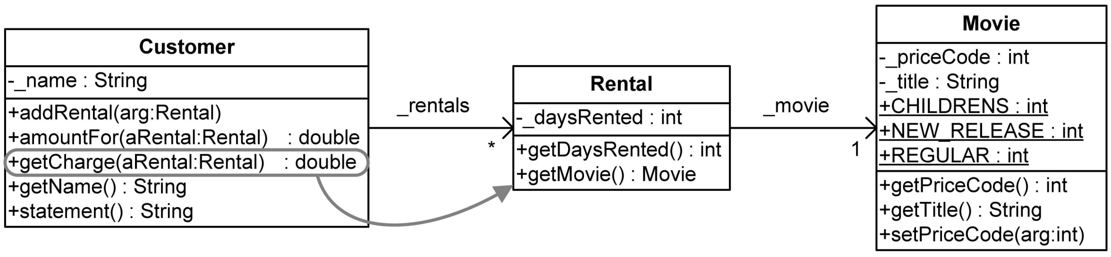


Fig. 4. UML class diagram of the Video Store before the application of the first Move Method refactoring.

The extracted refactoring suggestions are ranked in an ascending order according to the corresponding Entity Placement metric values. Eventually, all refactoring suggestions for which the resulting system has a lower Entity Placement value than the current system are considered as refactorings that can improve the design of the system.

3.6 Demonstration of the Methodology on a Refactoring Teaching Example

To demonstrate the application of the methodology, we have used a widely known example for refactorings, namely, Fowler's Video Store [17]. The initial version of the program is intentionally not well designed. Its design is gradually improved by applying successive refactorings. We have taken a snapshot of the evolving system exactly before the application of the first Move Method refactoring.

The UML class diagram of the snapshot that we examined is shown in Fig. 4. The arrow indicates the move of method `getCharge(Rental)` from class `Customer` to class `Rental`, as suggested by the author of the example.

To calculate the distances between the entities and the classes of the system, we need first to construct their entity sets, as shown in Table 1. The entity set of each entity contains the attributes and methods that it accesses (if it is a method) and the methods accessing it (if it is an attribute).

As can be observed from Table 1, the attributes that are references to classes of the system, namely, `Customer::_rentals` and `Rental::_movie` are not considered as entities and do not participate in the entity sets of other system entities. The getter and setter methods of the system, namely, `Customer::addRental(Rental)`, `Customer::getName()`, `Rental::getDaysRented()`, `Rental::getMovie()`, `Movie::getPriceCode()`, `Movie::getTitle()`, `Movie::setPriceCode()` are also not considered as entities. However, the attributes to which they provide access (`Customer::_name`, `Rental::_daysRented`, `Movie::_priceCode`, `Movie::_title`) are added to the entity sets of the system entities. The static attributes `Movie::CHILDRENS`, `Movie::NEW_RELEASE` and `Movie::REGULAR` are also not considered as entities and do not participate in the entity sets of the system entities accessing them. Finally, the method `Customer::amountFor(Rental)` that delegates to `Customer::getCharge(Rental)` is not considered as entity and its invocation from method `Customer::statement()` is replaced with the method that it delegates to.

To extract refactoring suggestions for method `Customer::getCharge(Rental)`, a set of candidate target classes T should be identified by examining its entity set.

The entity set of method `getCharge(Rental)` is

$$S_{getCharge()} = \{Movie::\_priceCode, Rental::\_daysRented\}$$

and, consequently, the set of candidate target classes for `getCharge(Rental)` is

$$T_{getCharge()} = \{Movie, Rental\}.$$

Since method `getCharge(Rental)` accesses an equal number of entities from both candidate target classes (i.e., entity *_priceCode* from *Movie* and *_daysRented* from *Rental*), the two candidate target classes will be sorted according to their distance from method `getCharge(Rental)`.

TABLE 1
Information Required for Extracting the Entity Sets of All System Entities

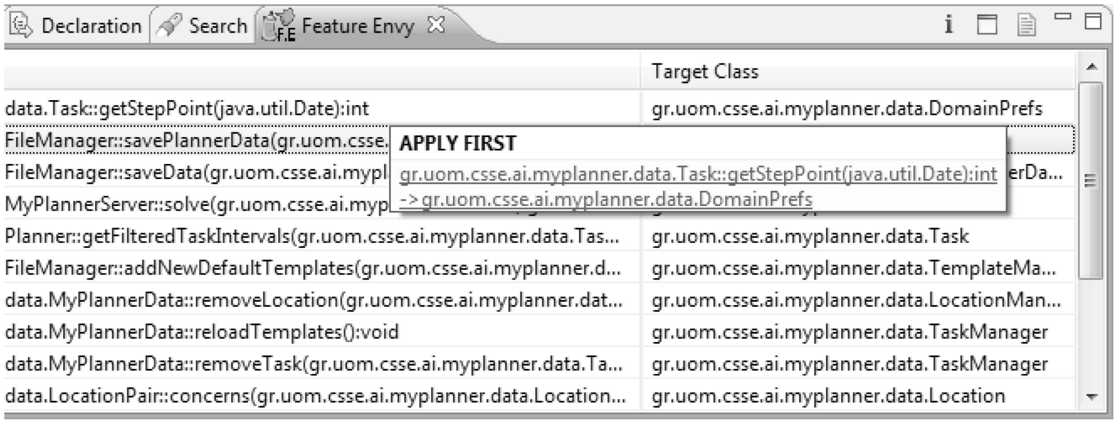
Entity name	Accessed attributes	Accessed methods	Accessing methods
Movie::_title	N/A	N/A	Customer::statement()

Movie::_priceCode	N/A	N/A	Customer::statement()
			Customer::getCharge(Rental)

Rental::_daysRented	N/A	N/A	Customer::statement()
			Customer::getCharge(Rental)

Customer::_name	N/A	N/A	Customer::statement()
Customer::statement()	Customer::_name Movie::_priceCode Rental::_daysRented Movie::_title	Customer:: getCharge(Rental)	N/A
Customer::getCharge(Rental)	Movie::_priceCode Rental::_daysRented	-	N/A

N/A: Not Applicable



	Target Class
data.Task::getStepPoint(java.util.Date):int	gr.uom.csse.ai.myplanner.data.DomainPrefs
FileManager::savePlannerData(gr.uom.csse.	
FileManager::saveData(gr.uom.csse.ai.mypl	erDa...
MyPlannerServer::solve(gr.uom.csse.ai.myp	
Planner::getFilteredTaskIntervals(gr.uom.csse.ai.myplanner.data.Tas...	gr.uom.csse.ai.myplanner.data.Task
FileManager::addNewDefaultTemplates(gr.uom.csse.ai.myplanner.d...	gr.uom.csse.ai.myplanner.data.TemplateMa...
data.MyPlannerData::removeLocation(gr.uom.csse.ai.myplanner.dat...	gr.uom.csse.ai.myplanner.data.LocationMan...
data.MyPlannerData::reloadTemplates():void	gr.uom.csse.ai.myplanner.data.TaskManager
data.MyPlannerData::removeTask(gr.uom.csse.ai.myplanner.data.Ta...	gr.uom.csse.ai.myplanner.data.TaskManager
data.LocationPair::concerns(gr.uom.csse.ai.myplanner.data.Location...	gr.uom.csse.ai.myplanner.data.Location

APPLY FIRST
gr.uom.csse.ai.myplanner.data.Task::getStepPoint(java.util.Date):int
->gr.uom.csse.ai.myplanner.data.DomainPrefs

Fig. 5. Tooltip indicating a dependency between two refactoring suggestions.

The entity sets of the candidate target classes are the following:

$$S_{Rental} = \{Rental::\_daysRented\},$$
$$S_{Movie} = \{Movie::\_title, Movie::\_priceCode\}.$$

The distances between method `Customer::getCharge(Rental)` and the candidate target classes are calculated as:

$$distance(getCharge(), \mathrm{Rental}) = 1 - \frac{|S_{getCharge()} \cap S_{Rental}|}{|S_{getCharge()} \cup S_{Rental}|} = 1 - \frac{1}{2} = 0.5,$$

$$distance(getCharge(), Movie) = 1 - \frac{|S_{getCharge()} \cap S_{Movie}|}{|S_{getCharge()} \cup S_{Movie}|} = 1 - \frac{1}{3} = 0.667.$$

The target class having the lowest distance from method `getCharge(Rental)` is `Rental` and, since all preconditions are satisfied with the specific target class, a Move Method refactoring suggestion is extracted, indicating the move of method `getCharge(Rental)` to class `Rental`. The second candidate target class `Movie` will not be examined by the algorithm since a Move Method refactoring suggestion has been already extracted. However, it should be noted that if class `Movie` was examined as target class, the preconditions would not be satisfied since method `getCharge(Rental)` has a local reference to class `Movie` which is not accessible outside the method.

4 JDEODORANT ECLIPSE PLUG-IN

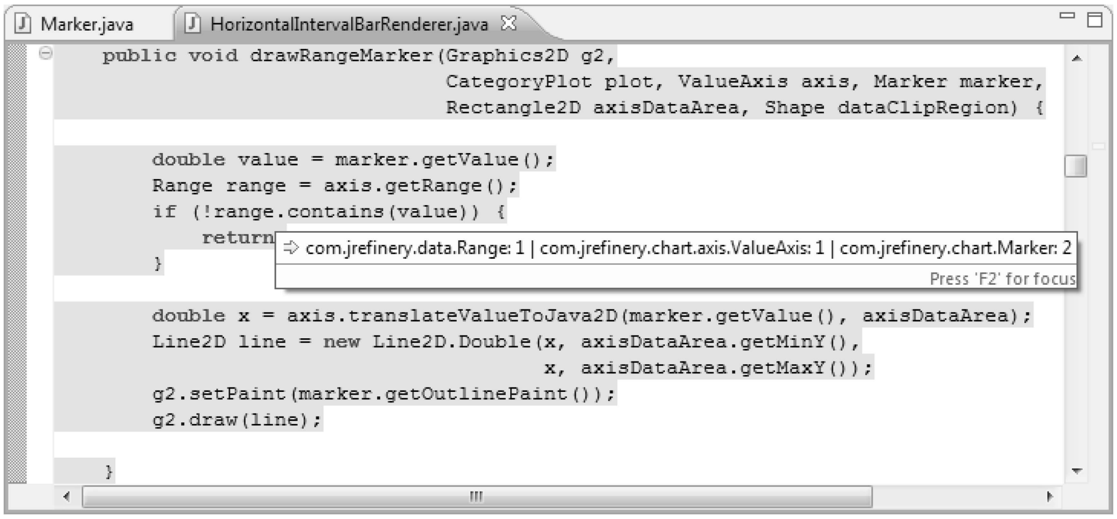
The proposed methodology has been implemented as an Eclipse plug-in [1], [16] that not only identifies Feature Envy bad smells but also allows the user to apply the refactorings that resolve them on source code. Moreover, the tool preevaluates the effect on design quality of all refactoring suggestions, assisting the user to determine the most effective sequence of refactoring applications. The plug-in employs the ASTParser of Eclipse Java Development Tools (JDT) to analyze the source code of Java projects and the ASTRewrite to apply the refactorings and provide undo functionality. JDeodorant offers some novel features concerning the application of Move Method refactorings:

1. It automatically determines whether the original method should be turned into a method that delegates to the moved one. The delegate method is necessary when other classes apart from the source class invoke the method to be moved and it prevents these classes from changing the way in which they invoke the moved method.
2. It automatically identifies dependencies between the refactoring suggestions and provides tooltip support aiding the user to resolve them (Fig. 5). For example, if the method associated with refactoring suggestion X invokes a method which is associated with another Move Method refactoring suggestion Y, a tooltip informs that suggestion Y (corresponding to the invoked method) should be applied before X (corresponding to the invoking method).
3. It automatically moves to the target class all of the private methods of the source class which are invoked only by the moved method.
4. When the user inspects a method which is suggested to be moved, the tool provides tooltip support indicating the number of members that it accesses from each class (Fig. 6). In this way, the user can more easily realize the Feature Envy problem.

5 EVALUATION

The proposed methodology has been evaluated in four ways:

1. To provide a qualitative analysis of the refactoring suggestions extracted by the proposed methodology, we have listed, categorized, and discussed the results for an open-source project.
2. To assess the effect on two aspects of design quality, namely, coupling and cohesion, we have measured their evolution when successively applying the suggested refactorings on two open-source projects.
3. To have an independent assessment that takes into account issues of conceptual integrity, we requested



```
public void drawRangeMarker(Graphics2D g2,
                            CategoryPlot plot, ValueAxis axis, Marker marker,
                            Rectangle2D axisDataArea, Shape dataClipRegion) {

    double value = marker.getValue();
    Range range = axis.getRange();
    if (!range.contains(value)) {
        return
    }

    double x = axis.translateValueToJava2D(marker.getValue(), axisDataArea);
    Line2D line = new Line2D.Double(x, axisDataArea.getMinY(),
                                    x, axisDataArea.getMaxY());
    g2.setPaint(marker.getOutlinePaint());
    g2.draw(line);

}
```

```
com.jrefinery.data.Range: 1 | com.jrefinery.chart.axis.ValueAxis: 1 | com.jrefinery.chart.Marker: 2
```

Fig. 6. Tooltip indicating the number of members that the highlighted method accesses from each class.

from a designer to provide feedback on the refactoring suggestions extracted by the proposed methodology for the system that he developed.

4. To evaluate the efficiency of the proposed methodology, we measured the computation time with regard to the size of various open-source projects.

For the purpose of evaluation, we have used open-source projects developed in Java, which are relatively active and mature, namely, JFreeChart, JEdit, JMol, and Diagram.

5.1 Qualitative Analysis

In order to investigate the kind of Move Method refactoring suggestions extracted by the proposed methodology, we have divided the suggestions into three main categories, according to the characteristics of the method to be moved:

1. The method does not access any entity from the source class.
2. The method accesses more entities from the target class than the source class.
3. The method accesses an equal number of entities from the source and target classes.

The suggestions belonging to the first category constitute relatively clear cases of Feature Envy. The second and third categories refer to cases where the method to be moved has dependency on fields and/or methods of the class that it belongs to. The philosophy behind the suggestions of the third category is that, when a method accesses the same number of fields/methods from the source and target classes, it should be placed to the smaller class (in terms of the total number of fields/methods) since smaller classes are more easily maintained [9]. Obviously, for all categories, the designer should take into account conceptual parameters in order to decide whether the refactoring should be applied or not.

The application of the proposed methodology to JFreeChart 0.9.6 resulted in 23 Move Method refactoring suggestions, leading to a system with lower Entity Placement metric value than the initial system. Table 2 contains the source class, method, and target class for each suggestion along with the number of members (fields/methods) that the method accesses from the source and target classes. The suggestions are sorted in ascending order according to the corresponding Entity Placement metric values.

To illustrate the soundness of the extracted suggestions, we analyze the first suggestion of Table 2. Method `drawRangeMarker()` shown in Fig. 7 does not access any field or method from class `HorizontalIntervalBarRenderer` that it belongs to. The method has six parameters in total from which two are not used at all inside the body of the method (`CategoryPlot plot`, `Shape dataClipRegion`), while two others correspond to Java API class types (`Graphics2D g2`, `Rectangle2D axisDataArea`) and, thus, their types cannot constitute valid target classes. Consequently, method `drawRangeMarker()` has two candidate target classes, namely, `ValueAxis` and `Marker`. It invokes two methods of class `Marker` through parameter `marker` and one method of class `ValueAxis` through parameter `axis` and, therefore, is suggested to be moved to class `Marker`. Moreover, class `Marker` is sufficiently smaller than class `ValueAxis` and constitutes a Data class [17] since it contains only fields and getter methods. This refactoring suggestion is a typical case of moving behavior close to data.

As can be observed from Table 2, in 13 out of 23 refactoring suggestions, the target class belongs to a different package than that of the source class. The suggestion of such kind of refactorings is desirable since their application may reduce the degree of package dependencies. Moreover, it is harder to identify such refactoring opportunities by manual inspection of the source code since they require the examination of classes that belong to different packages.

Table 3 shows the refactoring suggestions belonging to each category.

As can be observed from Table 3, about half of the suggestions (12 out of 23) refer to methods that do not access any field or method from the source class. Moreover, 10 out of 11 suggestions belonging to categories 2 and 3 refer to methods that access only fields (no methods) from the source class. In these cases, the accessed fields can be passed as parameters to the moved method

TABLE 2
Move Method Refactoring Suggestions for JFreeChart 0.9.6

id	Source class	Method	Target class	#accessed source members	#accessed target members
1	chart.renderer. HorizontalIntervalBarRenderer	drawRangeMarker	chart.Marker	0	2
2	chart.plot.ContourPlot	drawDomainMarker	chart.Marker	0	3
3	chart.plot.ContourPlot	drawRangeMarker	chart.Marker	0	3
4	chart.StandardLegend	createDrawableLegendItem	chart.LegendItem	1	2
5	chart.axis.DateAxis	previousStandardDate	chart.axis.DateTickUnit	0	3
6	chart.renderer. HorizontalShapeRenderer	drawRangeMarker	chart.Marker	0	2
7	chart.renderer. MinMaxCategoryRenderer	drawRangeMarker	chart.Marker	0	2
8	chart.renderer. VerticalIntervalBarRenderer	drawRangeMarker	chart.Marker	0	2
9	chart.MeterLegend	createLegendItem	chart.LegendItem	0	1
10	data.TimeSeriesCollection	getX	data.RegularTimePeriod	2	3
11	chart.StandardLegendItemLayout	doHorizontalLayout	chart.LegendItemCollection	1	1
12	chart.StandardLegendItemLayout	doVerticalLayout	chart.LegendItemCollection	1	1
13	chart.JFreeChart	drawTitle	chart.AbstractTitle	0	4
14	data.DynamicTimeSeriesCollection	getX	data.RegularTimePeriod	2	3
15	chart.MeterLegend	updateInformation	chart.plot.MeterPlot	0	3
16	chart.plot.PiePlot	getPaint	chart.renderer.PaintTable	2	2
17	chart.axis.DateAxis	nextStandardDate	chart.axis.DateTickUnit	1	2
18	chart.plot.PiePlot	getOutlineStroke	chart.renderer.StrokeTable	2	2
19	chart.plot.PiePlot	getOutlinePaint	chart.renderer.PaintTable	2	2
20	chart.demo.CompassDemo	adjustData	data.DefaultMeterDataset	0	2
21	chart.demo.ThermometerDemo	setMeterValue	data.DefaultMeterDataset	0	3
22	chart.renderer.AbstractRenderer	getSeriesPaint(int, int)	chart.renderer.PaintTable	2	2
23	chart.demo.CompassDemo	pick1PointerAction-Performed	chart.plot.CompassPlot	1	3

* all class names are preceded by package "com.jrefinery."

```
public void drawRangeMarker(Graphics2D g2,
      CategoryPlot plot, ValueAxis axis, Marker marker,
      Rectangle2D axisDataArea, Shape dataClipRegion) {

   double value = marker.getValue();
   Range range = axis.getRange();
   if (!range.contains(value)) {
      return;
   }
   double x = axis.translateValueToJava2D(marker.getValue(),
                                          axisDataArea);
   Line2D line = new Line2D.Double(x, axisDataArea.getMinY(),
                                   x, axisDataArea.getMaxY());
   g2.setPaint(marker.getOutlinePaint());
   g2.draw(line);
}
```

Fig. 7. Method `drawRangeMarker()` corresponding to the first extracted suggestion for JFreeChart 0.9.6.

resulting in a method that is no longer coupled to the source class. On the contrary, if a method invokes methods of the source class, a parameter of source class type should be added to the moved method in order to be able to invoke them after its move. In this case, the moved method remains coupled to the source class. The Entity Placement metric promotes suggestions where methods access only fields from the source class, since such refactorings lead to less coupled methods.

The analysis of the refactorings suggested by the proposed methodology offered some additional interesting insights:

1. By successively applying the suggested refactorings in JFreeChart 0.9.6, three cases of already existing duplicated code emerged. Specifically, the application of the suggestions 1 and 6 (Table 2) resulted in the move of two identical methods named `drawRangeMarker` to class `Marker` that

TABLE 3
Categorization of Refactoring Suggestions for JFreeChart 0.9.6

category	suggestion ids	#suggestions
1	{1,2,3,5,6,7,8,9,13,15,20,21}	12/23
2	{4,10,14,17,23}	5/23
3	{11,12,16,18,19,22}	6/23

not only had the same signature but also the same body. Two similar cases of duplicated code have been revealed by suggestions 7, 8 and 10, 14 (Table 2), respectively. Both pairs of suggestions had as result the move of identical methods (`drawRangeMarker`, `getX`) to a common target class (`Marker`, `RegularTimePeriod`, respectively). Obviously, it is easier for a designer to detect duplicate methods when they exist in the same class, rather than when they are scattered throughout different system classes.

2. The inspection of the refactoring suggestions in JEdit 4.3pre12 revealed that several Move Method suggestions were extracted due to the special handling of delegate methods by the proposed methodology. In the example of Fig. 8, method `lineComment()` of class `org.gjt.sp.jedit.textarea.TextArea` invokes methods `getLineText()` and `getLineStartOffset()` that delegate to methods of `JEditBuffer` through field `buffer`. Moreover, it accesses three methods `rangeLineComment()`, `getSelectedLines()`, `selectNone()`, and one field `caret` of class `TextArea`, while it invokes five methods of class `JEditBuffer` through field `buffer`. A methodology that does not properly handle delegate methods would erroneously consider that method `lineComment()` accesses six entities of class `TextArea` and five entities of class `JEditBuffer`, thus prohibiting the suggestion of moving the method to class `JEditBuffer`. On the other hand, a methodology that properly handles delegate methods would consider that method `lineComment()` accesses seven entities of class `JEditBuffer` and four entities of class `TextArea`.
3. The refactoring suggestions 20, 21, and 23 (Table 2) extracted for JFreeChart 0.9.6 refer to cases where the source class is a Graphical User Interface (GUI) class which extends class `JPanel` from Java Swing API (`CompassDemo`, `ThermometerDemo`). The corresponding source class methods (`adjustData`, `setMeterValue`, and `pick1PointerActionPerformed`) actually modify attributes (through setter methods) of source class fields which can be considered as references to classes holding data. These methods are invoked by ActionListeners which are implemented in the source class and are used to handle ActionEvents on various GUI components (such as buttons and combo boxes) placed on the user interface of the source class. The methodology suggests that the methods could be moved to the corresponding data classes (`DefaultMeterDataset` and `CompassPlot`). Although these suggestion can be considered as valid in terms of the number of accessed members, they are not conceptually sound since the methods which are related to UI functionality should be separate from the data classes that they may access (according to the Model-View-Controller pattern). This kind of suggestion can be avoided by applying the methodology separately on the various modules that the system under examination may consist of (e.g., domain classes, GUI classes, database classes, etc.). To this end, our tool offers to the designer the capability of applying the methodology on a specific package (including its subpackages) of the examined project. This issue could also be resolved by excluding from examination the methods which are invoked by implemented UI Listener methods (such as method `actionPerformed` of the `ActionListener` interface) using an appropriate precondition.
4. In several of the examined projects, we have observed that they contain test classes along with the application source code. A test class is responsible for testing whether the behavior of an application class is correct. It usually creates an instance of the class being tested and contains special methods that invoke methods of the tested class with a given input in order to compare the returned result with the expected one. Obviously, the suggestion of moving a test method to the class being tested is not conceptually sound. To this end, our tool automatically excludes from the analysis the classes that either extend class `junit.framework.TestCase` from JUnit 3.x API, or contain at least one method annotated with the `@Test` annotation (JUnit 4.x API).

5.2 Evaluation with Software Metrics

In this part of the evaluation, we have successively applied the most effective refactoring suggestions according to the Entity Placement metric value to open-source projects and studied the evolution of coupling and cohesion. The underlying assumption is that refactorings leading to systems with reduced coupling and increased cohesion have a positive effect on design quality. For the analysis, we have selected two projects, namely, JEdit 3.0 (425 classes) and JFreeChart 0.9.6 (459 classes).

5.2.1 Description of the Metrics Used in the Evaluation

There is a wide variety of coupling and cohesion metrics found in the literature [5], [6]. The criterion for choosing the appropriate metrics for the evaluation of the proposed methodology is that the metrics should be sensitive enough

```
protected JEditBuffer buffer;

public final String getLineText(int lineIndex) {
   return buffer.getLineText(lineIndex);
}

public int getLineStartOffset(int line) {
   return buffer.getLineStartOffset(line);
}

public void lineComment() {
   if(!buffer.isEditable()) {
       getToolkit().beep();
       return;
   }
   String comment =
       buffer.getContextSensitiveProperty(caret,"lineComment");
   if(comment == null || comment.length() == 0) {
       rangeLineComment();
       return;
   }
   comment += ' ';
   buffer.beginCompoundEdit();
   int[] lines = getSelectedLines();
   try{
       for(int i = 0; i < lines.length; i++) {
              String text = getLineText(lines[i]);
              buffer.insert(getLineStartOffset(lines[i])
              + StandardUtilities.getLeadingWhiteSpace(text),
                     comment);
       }
   }
   finally {
       buffer.endCompoundEdit();
   }
   selectNone();
}
```

Fig. 8. Handling of delegate methods in a refactoring suggestion for JEdit 4.3pre12.

to capture small code changes in an object-oriented system, such as the move of a method from one class to another.

We have employed the Message Passing Coupling (MPC) metric [23] for measuring the evolution of coupling. MPC for a class C is defined as the number of invocations of methods not implemented in class C by the methods of class C. Among the import coupling metrics that consider method-method interactions, MPC evaluates coupling employing the total number of method invocations, while the others measure the number of distinct methods invoked (e.g., RFC [13]). Other more coarse-grained metrics, such as CBO [13] and Coupling Factor [10], have not been considered because they estimate coupling based on the number of coupled classes and, therefore, their value might not change when a method is moved.

We have employed the redefined Connectivity metric by [5], which was originally proposed by [20], for measuring the evolution of cohesion. Connectivity for a class C is defined as the number of method pairs of class C, where one method invokes the other or both access a common attribute of class C, over the total number of method pairs of class C. Its difference with the other cohesion metrics is that it considers two methods m_1, m_2 to be cohesive, not only if they access a common attribute but also if m_1 invokes m_2 or vice versa. In the implementation of Connectivity metric, we have not considered as methods the constructors and the accessor (getter and setter) methods since the cohesion of a class is artificially increased if constructors are taken into account and decreased if accessor methods are taken into account [5].

While there is some degree of definitional relevance between Entity Placement and the aforementioned metrics, their major difference lies in the fact that the Jaccard distance (on which Entity Placement is based) is essentially a similarity metric, while MPC and Connectivity are based on an absolute count.

Concerning cohesion, Connectivity considers two methods either as cohesive or noncohesive, while a distance in the numerator of Entity Placement quantifies the degree of similarity between a method and the class to which it belongs. For example, a class might have a Connectivity value of 1 (absolute cohesion) because all of its methods invoke each other; however, in the case where these methods also invoke methods from other classes, the numerator of Entity Placement will reveal that the similarity of these methods to the class to which they belong is not absolute (i.e., the average distance is not zero).

Concerning coupling, MPC does not capture the "positive" coupling (expressed by messages being sent from a class to itself), while a distance in the denominator of Entity Placement quantifies for a given class also the similarity of foreign entities from the classes to which they belong. For

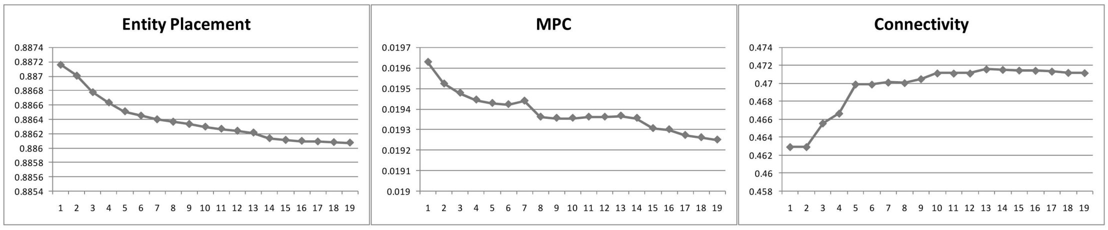


Fig. 9. Evolution of metrics for JEdit 3.0.

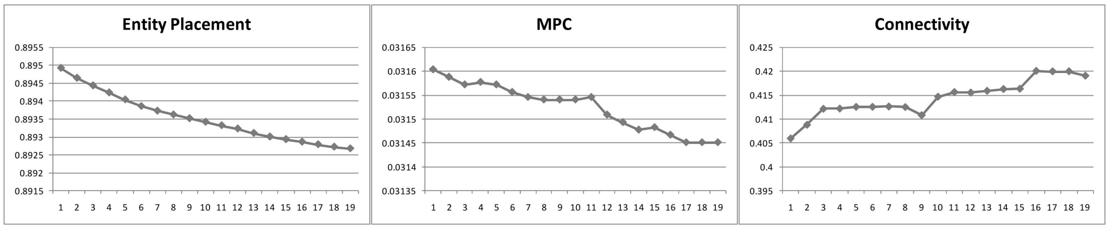


Fig. 10. Evolution of metrics for JFreeChart 0.9.6.

TABLE 4
Correlation between Entity Placement (EP) and MPC/Connectivity (Co)

Project	EP-MPC correlation	EP-Co correlation
JEdit 3.0	0.9572*	-0.9616*
JFreeChart 0.9.6	0.9483*	-0.9246*

* Correlation is significant at the 0.01 level (2-tailed)

example, the existence of a method that invokes only methods from its class is not being taken into account in the value of MPC, while it is considered (positively) in the denominator of Entity Placement. Moreover, the MPC metric does not have an upper limit representing the worst case, while the worst case for the denominator of Entity Placement occurs when, for a given class, all foreign entities access all entities from this class and none from the class to which they belong (i.e., the average distance is zero).

5.2.2 Results

The evolution of Entity Placement, MPC, and Connectivity for projects JEdit and JFreeChart is shown in Figs. 9 and 10, respectively. At each step, the refactoring corresponding to the lowest Entity Placement metric value has been applied. The x-axis represents the successive refactorings that have been performed.

As can be observed, the application of successive Move Method refactorings, which according to the methodology reduces the Entity Placement metric value, in general, reduces coupling and increases cohesion. The Pearson correlation coefficient between Entity Placement and Message Passing Coupling/Connectivity for the two projects is shown in Table 4.

The correlation between Entity Placement and coupling, as measured by MPC, is strongly positive and statistically significant for both projects. The correlation between Entity Placement and cohesion, as measured by Connectivity, is strongly negative and statistically significant for both projects. Thus, it can be argued that a measure of how well methods and attributes are placed in classes according to the Jaccard distance is a good criterion for ranking Move Method refactoring suggestions.

5.3 Independent Assessment

In this experiment, an independent designer assessed the conceptual integrity of the refactoring suggestions extracted by the proposed methodology for the system that he developed.

The project that has been examined is called SelfPlanner [29] and is an intelligent Web-based calendar application that plans the tasks of a user using an adaptation of the Squeaky Wheel Optimization framework. It is the outcome of a research project of the Artificial Intelligence Group at the Department of Applied Informatics, University of Macedonia, Greece. It consists of a planning engine developed in C++ and a client/server application developed in Java. The evaluation focused on the client/server application, since JDeodorant analyzes Java source code. The application (version 1.11) consists of 34 classes and 5,800 lines of code. The reasons for selecting the specific project are:

- It is a rather mature research project which has been constantly evolving for more than a year. Moreover, it has been subject to continuous adaptive maintenance due to constant requirement changes. Therefore, it is reasonable to expect that it offers several refactoring opportunities.
- The client/server part of the application was designed and developed by a single person. As a result, the developer that participated in the experiment had complete and deep knowledge of the system's architecture.
- The developer that participated in the experiment is an experienced programmer with knowledge of object-oriented design principles that enabled him to assess the refactoring suggestions extracted by

TABLE 5
Move Method Refactoring Suggestions for SelfPlanner

id	Source class	Method	Target class	#accessed source members	#accessed target members	designer's opinion
1	data.Task	getStepPoint	data.DomainPrefs	0	2	A
2	FileManager	savePlannerData	Planner	0	4	A
3	FileManager	saveData	data.MyPlannerData	0	1	D
4	MyPlannerServer	solve	Planner	0	3	A
5	Planner	getFilteredTaskIntervals	data.Task	2	4	A
6	FileManager	addNewDefaultTemplates	data.TemplateManager	0	2	A
7	data.MyPlannerData	removeLocation	data.LocationManager	0	1	A
8	data.MyPlannerData	reloadTemplates	data.TaskManager	0	1	A
9	data.MyPlannerData	removeTask	data.TaskManager	0	1	A
10	data.LocationPair	concerns	data.Location	0	1	D

* all class names are preceded by package "gr.uom.csse.ai.myplanner."
A: total agreement, D: conceptual disagreement

```
public void saveData(MyPlannerData data) throws IOException {
   data.getTemplateManager().resetTemplates();
   ObjectOutputStream out = new ObjectOutputStream(
      new FileOutputStream(data.user() + ".spdata", false));
   out.writeObject(data);
   out.close();
}
```

Fig. 11. Method corresponding to suggestion 3 for SelfPlanner.

```
private Location l1;
private Location l2;

public boolean concerns(Location loc1, Location loc2) {
   return ((l1.equals(loc1) && l2.equals(loc2)) ||
       (l1.equals(loc2) && l2.equals(loc1)));
}
```

Fig. 12. Method corresponding to suggestion 10 for SelfPlanner.

the proposed methodology and provide valuable feedback.

- The developer that participated in the experiment was able to dedicate a significant amount of time on studying and commenting on the refactoring suggestions extracted by the proposed methodology.

The refactoring suggestions extracted by the proposed methodology for SelfPlanner along with the opinion of the independent designer are shown in Table 5.

As can be observed from Table 5, the independent designer agreed in 8 out of 10 refactoring suggestions. Moreover, 9 out of 10 refactoring suggestions refer to methods that do not access any field or method from the source class, and therefore, constitute clear cases of Feature Envy. The method corresponding to suggestion 5 accesses two fields from the source class.

Method `saveData()` corresponding to suggestion 3 (Fig. 11) has one candidate target class, namely, `MyPlanner Data`. It invokes two methods of the target class through parameter `data` and its purpose is to save the data of the corresponding user as a serialized object. The independent designer supported that the methods which are exclusively related to file operations should be located in class `FileManager`, and thus, disagreed with this suggestion.

Method `concerns()` corresponding to suggestion 10 (Fig. 12) invokes method `equals()` of class `Location` through two different fields (`l1, l2`) that can serve both as reference to target class. By moving this method to class `Location`, one of the fields will be replaced with `this` reference and the other will be passed as the parameter, resulting in a method with three parameters of `Location` type. The independent designer stated that the method will become more complicated if it is moved.

5.4 Evaluation of Efficiency

The process which is required for the extraction of Move Method refactoring suggestions in a given system consists of the following steps:

1. Parsing of the system under study using the Abstract Syntax Tree (AST) Parser of Eclipse JDT.
2. Determination of system entities and construction of the corresponding entity sets.
3. Calculation of the distances between all system entities and system classes.
4. Application of the algorithm for the extraction of Move Method refactoring suggestions to all method entities of the system. Moreover, all extracted refactoring suggestions are virtually performed in order to

TABLE 6
Various Size Measures for the Examined Open-Source Projects

measures	Diagram 1.0.1	JMol 9.0	JEdit 3.0	JFreeChart 0.9.6
#classes	214	316	425	436
#examined methods	727	1435	1864	1847
LOC	23119	49668	71897	106840
#total suggestions	34	246	58	58

TABLE 7
CPU Times for Each Step Required for the Extraction of Refactoring Suggestions

step	Diagram 1.0.1	JMol 9.0	JEdit 3.0	JFreeChart 0.9.6
a	1150 ms	2150 ms	3780 ms	6300 ms
b	30 ms	100 ms	125 ms	80 ms
c	400 ms	2100 ms	3480 ms	3050 ms
d	4.7 sec	133.2 sec	47.1 sec	52 sec

* Measurements performed on Intel Core 2 Duo E6600 2.4 GHz, 2 GB DDR2 RAM

calculate the Entity Placement metric value that the system would have if they were actually applied.

Table 6 contains various size measures for four open-source projects. The measure of examined methods refers to the methods that constitute entities of the system under study. This means that the constructors, accessor methods, static methods, and delegate methods are not included in this measure. The measure of total suggestions also includes the refactoring suggestions having a higher Entity Placement value than the initial system.

Table 7 presents the required computation time for each step of the process.

As can be observed from Table 7, the most time-consuming part of the process is the virtual application of the extracted refactoring suggestions. The calculation of the Entity Placement metric value that the system will have after the virtual application of a refactoring suggestion requires a recalculation of distances between the entities and the classes which are affected by the move of the corresponding method. This part can be very time-consuming, since it involves the construction of the union and intersection between several entity sets.

The total CPU time required for the last step primarily depends on the number of the extracted refactoring suggestions. This is evident from the CPU time required for JMol, which has the largest number of refactoring suggestions compared to the other examined systems. The CPU time required for the last step is also affected by the size of the system since, in larger systems, more distances have to be calculated. The results satisfy the intuition, since performance is affected by the size of the underlying problem.

6 Conclusions

The cumulative effect of several simple refactoring steps and the tool support for their automated application has made the refactoring process a widely accepted technique for improving software design. However, identifying the places where refactoring should be applied is neither trivial nor supported by tools. In this paper, we have proposed a methodology for locating Feature Envy bad smells and evaluating the effect of the Move Method refactorings that resolve them.

The qualitative analysis of the refactoring suggestions for an open-source project revealed that they can be useful in assisting the designer to improve design quality. The study of coupling and cohesion evolution on two open-source projects has shown that the refactorings suggested by the methodology have a positive impact on both coupling and cohesion. The assessment by an independent designer of the suggested refactorings for a system that he developed indicated that the proposed methodology is capable of extracting conceptually sound suggestions. Finally, CPU time measurements have shown that the efficiency of the approach primarily depends on the number of extracted refactoring suggestions, and second, on the size of the system under study.

Appendix A

The following auxiliary functions examine whether a given method is an accessor or delegate method. The description of the functions assumes that such methods are written in a certain way, following the most common conventions.

```
(Field or null) $isGetter$(Method $m$) $\equiv$
  size of $m$.parameters $= 0 \wedge$
  size of $m$.methodBody.statements $= 1 \wedge$
  $\exists$ ReturnStatement $r \in m$.methodBody.statements where
  $r$.returnedExpression is VariableAccess $v \wedge v$.declaration
  is Field $f \wedge$
  $f$.type $= m$.returnType
  return $f$

(Field or null) $isSetter$(Method $m$) $\equiv$
  size of $m$.parameters $= 1 \wedge$
  size of $m$.methodBody.statements $= 1 \wedge$
  $\exists$ AssignmentStatement $a \in m$.methodBody.statements
  where
  ($a$.assignment.leftHandSide is VariableAccess $v_1 \wedge$
  $v_1$.declaration is Field $f$) $\wedge$
  ($a$.assignment.rightHandSide is VariableAccess $v_2 \wedge$
  $v_2$.declaration = $m$.parameters[0]) $\wedge$
  $f$.type = $m$.parameters[0].type
  return $f$

(Type or null) $elementTypeOfCollection$
(VariableDeclaration $d$) $\equiv$
  if $d$.type is CollectionType $cType \wedge cType$.type $\in$
  {Collection, List, AbstractCollection,
  AbstractList, ArrayList, LinkedList, Vector,
  Set, AbstractSet, HashSet, LinkedHashSet,
  SortedSet, TreeSet}
  return $cType$.elementType
  else if $d$.type is MapType $mType \wedge mType$.type $\in$ {Map,
  AbstractMap, HashMap, Hashtable, SortedMap,
  TreeMap, IdentityHashMap, WeakHashMap}
  return $mType$.valueType
```

*the element type is inferred by the generic type(s) of the field type, or by the type of the parameter of the collection setter methods corresponding to the field

```
(Field or null) $isCollectionGetter$(Method $m$) $\equiv$
  (size of $m$.parameters = 0 $\wedge$
  size of $m$.methodBody.statements = 1 $\wedge$
  $\exists$ ReturnStatement $r \in m$.methodBody.statements where
  $r$.returnedExpression is MethodInvocation $methodInv \wedge$
  $methodInv$.invokeExpression is VariableAccess $v \wedge$
  $v$.declaration is Field $f \wedge$ elementTypeOfCollection($f$) $\neq$
  null $\wedge$
  $methodInv$.name $\in$ {iterator, toArray, listIterator,
  elements, keySet, entrySet, values}
  return $f$) $\vee$
  (size of $m$.parameters = 1 $\wedge$
  size of $m$.methodBody.statements = 1 $\wedge$
  $\exists$ ReturnStatement $r \in m$.methodBody.statements where
  $r$.returnedExpression is MethodInvocation $methodInv \wedge$
  $methodInv$.invokeExpression is VariableAccess $v \wedge$
  $v$.declaration is Field$f \wedge$
  $elementType$ = elementTypeOfCollection($f$) $\neq$ null $\wedge$
  $methodInv$.name $\in$ {get, elementAt} $\wedge$ $elementType$ =
  $m$.returnType $\wedge$
  positionOfArgument($methodInv$, $m$.parameters[0]) $\neq -1$
  return $f$)

(Field or null) $isCollectionSetter$(Method $m$) $\equiv$
  size of $m$.parameters = 1 $\wedge$
  size of $m$.methodBody.statements = 1 $\wedge$
  $\exists$ MethodInvocationStatement $s \in m$.methodBody.
  statements where
  $s$.methodInvocation.invokeExpression is VariableAccess $v \wedge$
  $v$.declaration is Field$f \wedge$ elementTypeOfCollection($f$) $\neq$
  null $\wedge$
  $s$.methodInvocation.name $\in$ {add, remove, addAll,
  removeAll, retainAll, addElement, removeElement,
  put} $\wedge$
  positionOfArgument($s$.methodInvocation, $m$.
  parameters[0]) $\neq -1$
  return $f$

int $positionOfArgument$(MethodInvocation $inv$,
Parameter $param$) $\equiv$
  for $i = 1$ to size of $inv$.arguments
    if $inv$.arguments[$i$] is VariableAccess $arg \wedge$
    $arg$.declaration = $param$
      return $i$
  return $-1$

(Method or null) $isDelegate$(Method $m$) $\equiv$
  (size of $m$.methodBody.statements = 1 $\wedge$
  $\exists$ MethodInvocationStatement $s \in m$.methodBody.
  statements where
  $s$.methodInvocation.declaringClass.type $\in$ program.
  classTypes $\wedge$
  (($s$.methodInvocation.invokeExpression
  is VariableAccess $v \wedge v$.declaration $\in$
  {$m$.ownerClass.fields $\cup$ $m$.parameters $\cup$
  inheritedFields($m$.ownerClass)}) $\vee$
  ($s$.methodInvocation.invokeExpression is
  MethodInvocation $methodInv2 \wedge$
  Field $f$ = isGetter($methodInv2$.methodDeclaration) $\neq$
  null $\wedge$
  $f \in$ {$m$.ownerClass.fields $\cup$
  inheritedFields($m$.ownerClass)}) $\vee$
  $s$.methodInvocation.invokeExpression = null)
  return $s$.methodInvocation.declaringMethod) $\vee$
  (size of $m$.methodBody.statements = 1 $\wedge$
  $\exists$ ReturnStatement $r \in m$.methodBody.statements where
  $r$.returnedExpression is MethodInvocation $methodInv \wedge$
  $methodInv$.declaringClass.type $\in$ program.classTypes $\wedge$
  $methodInv$.declaringMethod.returnType =
  $m$.returnType $\wedge$
  (($methodInv$.invokeExpression is VariableAccess $v \wedge$
  $v$.declaration $\in$
  {$m$.ownerClass.fields $\cup$ $m$.parameters $\cup$
  inheritedFields($m$.ownerClass)}) $\vee$
  ($methodInv$.invokeExpression is MethodInvocation
  $methodInv2 \wedge$
  Field $f$ = isGetter($methodInv2$.methodDeclaration) $\neq$ null $\wedge$
  $f \in$ {$m$.ownerClass.fields $\cup$ inheritedFields
    ($m$.ownerClass)}) $\vee$
  $methodInv$.invokeExpression = null)
  return $methodInv$.declaringMethod)

(Method or null)$finalNonDelegateMethod$(Method $m$) $\equiv$
  $nonDelegateMethod = m$
  while($delegatedMethod$ =
  isDelegate($nonDelegateMethod$) $\neq$ null)
    $nonDelegateMethod = delegatedMethod$
    finalNonDelegateMethod($nonDelegateMethod$)
  if $nonDelegateMethod = m$.declaringMethod
    return null
  else
    return $nonDelegateMethod$
```

Appendix B

```
boolean $modifiesDataStructureInTargetClass$(Method $m$, Class $t$) $\equiv$
  for $i = 1$ to size of $m$.parameters
    $parameter = m$.parameters[$i$]
    $F =$ one-to-manyAssociationRelationships ($t$, $parameter$.type)
    if $F \neq \oslash$
      for $j = 1$ to size of $m$.methodBody.methodInvocations
        $methodInv = m$.methodBody.methodInvocations[$j$]
        if $methodInv$.declaringClass $= t \wedge$
        $pos =$ positionOfArgument($methodInv, parameter$)$\neq -1$
          $methodDecl = methodInv$.declaringMethod
          $methodDeclParam = methodDecl$.parameters[$pos$]
          for $k = 1$ to size of $F$
            Field $f = F[k]$
            if modifiesDataStructure($methodDecl, f$,
            $methodDeclParam$)
              return true

(set of Field) $one\text{-}to\text{-}manyAssociationRelationships$(Class $fromClass$, Type $toType$) $\equiv$
  return $f \in fromClass$.fields
  where($f$.type is ArrayType $aType \wedge aType$.type$=$
$toType$) $\vee$
  elementTypeOfCollection($f$) $= toType$

boolean $modifiesDataStructure$(Method $m$, Field $f$, Parameter $param$) $\equiv$
  if $f$.type is ArrayType $aType$ return
    $\exists$ $assignment \in m$.methodBody.assignments
    where $assignment$.leftHandSide is arrayAccess of
    VariableAccess $v_1 \wedge$
    $v_1$.declaration $= f \wedge aType$.type $= param$.type $\wedge$
    $assignment$.rightHandSide is VariableAcess $v_2 \wedge$
    $v_2$.declaration $= param$
else if $elementType =$ elementTypeOfCollection($f$) $\neq$ null
return
    $\exists$ $methodInv \in m$.methodBody.methodInvocations
    where $methodInv$.invokeExpression is VariableAccess $v_1 \wedge$
    $v_1$.declaration $= f \wedge$
    $methodInv$.name $\in$ {add, remove, addElement,
    removeElement, set, setElementAt,
    insertElementAt, put} $\wedge$
    positionOfArgument($methodInv, param$) $\neq -1 \wedge$
    $elementType = param$.type
```

* arrayAccess of variable *array* is expression *array[indexExpression]*

Acknowledgments

The authors would like to thank Anastasios Alexiadis and Ioannis Refanidis for their contribution to the independent assessment of the refactoring suggestions.

References

[1] "Bad Smell Identification for Software Refactoring," http://www.jdeodorant.org, 2007.

[2] J. Bansiya and C.G. Davis, "A Hierarchical Model for Object-Oriented Design Quality Assessment," *IEEE Trans. Software Eng.*, vol. 28, no. 1, pp. 4-17, Jan. 2002.

[3] V.R. Basili, L.C. Briand, and W.L. Melo, "A Validation of Object-Oriented Design Metrics as Quality Indicators," *IEEE Trans. Software Eng.*, vol. 22, no. 10, pp. 751-761, Oct. 1996.

[4] A.B. Binkley and S.R. Schach, "Validation of the Coupling Dependency Metric as a Predictor of Runtime Failures and Maintenance Measures," *Proc. 20th Int'l Conf. Software Eng.*, pp. 452-455, 1998.

[5] L.C. Briand, J.W. Daly, and J. Wust, "A Unified Framework for Cohesion Measurement in Object-Oriented Systems," *Empirical Software Eng.*, vol. 3, no. 1, pp. 65-117, 1998.

[6] L.C. Briand, J.W. Daly, and J.K. Wust, "A Unified Framework for Coupling Measurement in Object-Oriented Systems," *IEEE Trans. Software Eng.*, vol. 25, no. 1, pp. 91-121, Jan./Feb. 1999.

[7] L.C. Briand, J. Wust, S.V. Ikonomovski, and H. Lounis, "Investigating Quality Factors in Object-Oriented Designs: An Industrial Case Study," *Proc. 21st Int'l Conf. Software Eng.*, pp. 345-354, 1999.

[8] L.C. Briand, J. Wust, and H. Lounis, "Using Coupling Measurement for Impact Analysis in Object-Oriented Systems," *Proc. Int'l Conf. Software Maintenance*, pp. 475-482, 1999.

[9] L.C. Briand and J. Wust, "Modeling Development Effort in Object-Oriented Systems Using Design Properties," *IEEE Trans. Software Eng.*, vol. 27, no. 11, pp. 963-986, Nov. 2001.

[10] F. Brito e Abreu, "The MOOD Metrics Set," *Proc. Ninth European Conf. Object-Oriented Programming Workshop Metrics*, Aug. 1995.

[11] F. Brito e Abreu and W. Melo, "Evaluating the Impact of Object-Oriented Design on Software Quality," *Proc. Third Int'l Software Metrics Symp.*, pp. 90-99, 1996.

[12] M.A. Chaumun, H. Kabaili, R.K. Keller, F. Lustman, and G. Saint-Denis, "Design Properties and Object-Oriented Software Changeability," *Proc. Fourth European Conf. Software Maintenance and Reeng.*, pp. 45-54, 2000.

[13] S.R. Chidamber and C.F. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Trans. Software Eng.*, vol. 20, no. 6, pp. 476-493, June 1994.

[14] S.R. Chidamber, D.P. Darcy, and C.F. Kemerer, "Managerial Use of Metrics for Object-Oriented Software: An Exploratory Analysis," *IEEE Trans. Software Eng.*, vol. 24, no. 8, pp. 629-639, Aug. 1998.

[15] B. Du Bois, S. Demeyer, and J. Verelst, "Refactoring—Improving Coupling and Cohesion of Existing Code," *Proc. 11th Working Conf. Reverse Eng.* pp. 144-151, Nov. 2004.

[16] M. Fokaefs, N. Tsantalis, and A. Chatzigeorgiou, "JDeodorant: Identification and Removal of Feature Envy Bad Smells," *Proc. 23rd Int'l Conf. Software Maintenance*, pp. 519-520, Oct. 2007.

[17] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code.* Addison Wesley, 1999.

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, 1995.

[19] C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of Software Engineering,* second ed. Prentice Hall, 2003.

[20] M. Hitz and B. Montazeri, "Measuring Coupling and Cohesion in Object-Oriented Systems," *Proc. Int'l Symp. Applied Corporate Computing,* Oct. 1995.

[21] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya, "A Quantitative Evaluation of Maintainability Enhancement by Refactoring," *Proc. 18th IEEE Int'l Conf. Software Maintenance,* pp. 576-585, Oct. 2002.

[22] M. O'Keeffe and M. O'Cinneide, "Search-Based Software Maintenance," *Proc. 10th European Conf. Software Maintenance and Reeng.,* pp. 249-260, Mar. 2006.

[23] W. Li and S. Henry, "Object-Oriented Metrics that Predict Maintainability," *J. Systems and Software,* vol. 23, no. 2, pp. 111-122, 1993.

[24] O. Maqbool and H.A. Babri, "Hierarchical Clustering for Software Architecture Recovery," *IEEE Trans. Software Eng.,* vol. 33, no. 11, pp. 759-780, Nov. 2007.

[25] R.C. Martin, *Agile Software Development: Principles, Patterns and Practices.* Prentice Hall, 2003.

[26] T. Mens and T. Tourwé, "A Survey of Software Refactoring," *IEEE Trans. Software Eng.,* vol. 30, no. 2, pp. 126-139, Feb. 2004.

[27] B.S. Mitchell and S. Mancoridis, "On the Automatic Modularization of Software Systems Using the Bunch Tool," *IEEE Trans. Software Eng.,* vol. 32, no. 3, pp. 193-208, Mar. 2006.
[28] W.F. Opdyke, "Refactoring Object-Oriented Frameworks," PhD dissertation, Univ. of Illinois at Urbana-Champaign, 1992.
[29] I. Refanidis and A. Alexiadis, "SelfPlanner: An Intelligent Web-Based Calendar Application," *Proc. 17th Int'l Conf. Automated Planning and Scheduling Systems,* Sept. 2007.
[30] A.J. Riel, *Object-Oriented Design Heuristics.* Addison-Wesley, 1996.
[31] O. Seng, J. Stammel, and D. Burkhart, "Search-Based Determination of Refactorings for Improving the Class Structure of Object-Oriented Systems," *Proc. Eighth Ann. Conf. Genetic and Evolutionary Computation,* pp. 1909-1916, 2006.
[32] F. Simon, F. Steinbrückner, and C. Lewerentz, "Metrics Based Refactoring," *Proc. Fifth European Conf. Software Maintenance and Reeng.,* pp. 30-38, Mar. 2001.
[33] L. Tahvildari and K. Kontogiannis, "A Metric-Based Approach to Enhance Design Quality through Meta-Pattern Transformations," *Proc. Seventh European Conf. Software Maintenance and Reeng.,* pp. 183-192, Mar. 2003.

Nikolaos Tsantalis received the BS and MS degrees in applied informatics from the University of Macedonia, in 2004 and 2006, respectively. He is currently working toward the PhD degree in the Department of Applied Informatics at the University of Macedonia, Thessaloniki, Greece. His research interests include design patterns, refactorings, and object-oriented quality metrics. He is a student member of the IEEE and the IEEE Computer Society.

Alexander Chatzigeorgiou received the Diploma in electrical engineering and the PhD degree in computer science from the Aristotle University of Thessaloniki, Greece, in 1996 and 2000, respectively. From 1997 to 1999, he was with Intracom, Greece, as a telecommunications software designer. He is currently an assistant professor of software engineering in the Department of Applied Informatics at the University of Macedonia, Thessaloniki, Greece. His research interests include software metrics, object-oriented design, and software maintenance. He is a member of the IEEE and the IEEE Computer Society.