Automated refactoring to the Strategy design pattern

Aikaterini Christopoulou, E.A. Giakoumakis, Vassilis E. Zafeiris, Vasiliki Soukara

Abstract

Context: The automated identification of code fragments characterized by common design flaws (or “code smells”) that can be handled through refactoring, fosters refactoring activities, especially in large code bases where multiple developers are engaged without a detailed view on the whole system. Automated refactoring to design patterns enables significant contributions to design quality even from developers with little experience on the use of the required patterns.

Method: An algorithm is introduced for the automated identification of refactoring opportunities to the Strategy design pattern. Suggested refactorings comprise conditional statements that are characterized by analogies to the Strategy design pattern, in terms of the purpose and selection mode of strategies. Moreover, this work specifies the procedure for refactoring to Strategy the identified conditional statements. For special cases of these statements, a technique is proposed for total replacement of conditional logic with method calls of appropriate concrete Strategy instances. The identification algorithm and the refactoring procedure are implemented and integrated in the JDeodorant Eclipse plug-in. The method is evaluated on a set of Java projects, in terms of quality of the suggested refactorings and run-time efficiency. The relevance of the identified refactoring opportunities is verified by expert software engineers.

Results: The identification algorithm recalled, from the projects used during evaluation, many of the refactoring candidates that were identified by the expert software engineers. Its execution time on projects of varying size confirmed the run-time efficiency of this method.

Conclusion: The proposed method for automated refactoring to Strategy contributes to simplification of conditional statements. Moreover, it enhances system extensibility through the Strategy design pattern.

1. Introduction

Refactoring is the process of applying changes to the internal structure of software without changing its observable behaviour [1]. It focuses, according to Fowler [1], on making the source code more understandable, while lowering the cost of introducing modifications to it. These objectives denote the relevance of refactoring with system design improvement and, thus, its applicability to both system implementation and maintenance stages of the software life cycle. During system implementation, refactoring enables continuous design, i.e., alternation of coding and design modifications, a practice that is adopted in agile development methodologies, such as eXtreme Programming (XP), that do not emphasize up-front design [2]. On the other hand, refactoring prevents software quality degradation that emanates from software maintenance tasks. The support for automated execution of multiple, though simple, types of refactorings by widely-used IDEs (e.g., Eclipse, Netbeans, MS Visual Studio), is indicative of the refactoring activities' adoption by current developers.

The refactoring process comprises a set of activities such as identification of the candidate code fragments and the required refactoring(s), evaluation of behaviour preservation guarantees, application of the refactoring and assessment of its effects on software quality [3,4]. However, the refactoring automation provided by IDEs and specialized tools is, primarily, related to the application of code transformations, in a way that ensures code correctness and behaviour preservation while providing the capability of reverting them. However, the developer's role is still significant, since he/she is required to specify to the tool the target code fragments and the type of refactoring to apply, a task that highly depends on the skills and experience of the developer. Fowler [1] tried to systematically put refactoring into practice by specifying common design flaws (or “code smells”) in object oriented code, along with sequences of
primitive refactoring operations for their remedy, Kerievsky [5] focused on the improvement of system design quality through refactoring to patterns. His work specifies “bad smells” and refactoring procedures for their elimination through the use of appropriate design patterns. Despite the detailed presentation of “bad smells” in both works, their identification is often based on code semantics that makes the automation of this task rather challenging.

The automated identification of refactoring opportunities, i.e., code fragments characterized by “code smells” that can be handled through refactoring, fosters refactoring activities, especially in large code bases where multiple developers are engaged without a detailed view on the whole system. Automation in the introduction of design patterns enables significant contributions to design quality even from developers with little or no practical experience on the use of the required patterns. This work focuses on the automated identification of refactoring opportunities to the Strategy design pattern [6] and the elimination, through polymorphism, of respective “code smells” related to extensive use of complex conditional statements.

We propose an algorithm for the identification of conditional statements that emulate, in their use, the Strategy design pattern. These statements are characterized by certain analogies to the Strategy pattern, in terms of the purpose and selection mode of strategies [6]: (a) conditional branches include mutually exclusive, non-trivial computations, corresponding to alternative algorithm implementations represented by strategies, (b) branch selection is not controlled by the logic of the class declaring the conditional statement but, instead, by its clients, in proportion to Strategy selection in the Strategy pattern. Moreover, this work specifies the procedure for refactoring to Strategy the identified conditional statements. For special cases of these statements, we propose a technique for total replacement of conditional logic with method calls of appropriate concrete Strategy instances. The identification algorithm, as well as the refactoring procedure have been implemented and integrated in the JDeodorant Eclipse plug-in [7]. We have evaluated our approach, on a set of Java projects, in terms of the quality of identified refactoring opportunities and its execution time scalability. The evaluation resulted in the identification of a number of refactoring opportunities on each project, whose relevance was verified by two expert software engineers. Moreover, the execution time of the identification algorithm on projects of varying size confirms the execution efficiency of this method.

The rest of the paper is organized as follows: Section 2 presents relevant literature in the area of automated refactoring to patterns. Section 3 specifies the proposed algorithm for identification of refactoring opportunities to Strategy and the respective procedure for applying suggested refactorings. The evaluation of our method is included in Section 4 and the paper is concluded in Section 5 with open issues and possible extensions for future research.

2. Related work

This section provides an overview of methods for automated refactoring to design patterns. The reader may refer to the work of [4] for an extensive review in the research area of software refactoring. Among the earlier approaches for introducing automation in the application of design patterns, for source code maintenance and evolution, is the method proposed by Tokuda and Batory [8]. The method is based on the specification of primitive object-oriented transformations that can be automated with appropriate tool support. Moreover, it prescribes the introduction of a design pattern as a composition of a sequence of parameterized object-oriented transformations. The authors demonstrate their method through the introduction of the Abstract Factory design pattern in a simple case study.

Refactoring to design patterns is also treated as a series of mini-transformations in the methodology proposed by Cinneide and Nixon [9]. The methodology introduces a systematic way for enabling refactoring towards a specific design pattern that involves (a) selection of the target design pattern, (b) definition of the precursor, i.e., structural properties, expressing the intent of the pattern, of the source code that the pattern will be applied, (c) decomposition of the pattern in a series of mini-patterns, (d) specification of the refactoring as a composition of mini-transformations. A mini-transformation comprises pre-conditions, post-conditions, transformation steps and an argument over how the mini-transformation supports behaviour preservation after its application. The authors present the application of the methodology for the Factory Method pattern that is also implemented as a Java tool. The methodology is mostly focused on structure-rich, rather than behavioural patterns, and has been applied to the Abstract Factory, Builder and Singleton patterns.

An approach for automated identification of refactoring opportunities to the Abstract Factory design pattern in a Java code base has been proposed by Jeon et al. [10]. The authors propose a method for inferring refactoring opportunities, through employment of logic programming, and a refactoring Strategy for transforming the source code towards the target structure. Inference is based on extraction from Java code of the system design and its representation as a set of Prolog-like predicates that are then converted to Prolog facts. Moreover, design pattern inference rules are defined, for each target pattern for refactoring, that are then transformed to Prolog rules. Each inference rule has input components that correspond to the roles of the design pattern. The identification of refactoring opportunities takes place through issuing of Prolog queries. Jiebelein et al. [11] also used logical programming for the detection of certain “bad smells” specified by Kerievsky [5]. The approach involves transformation of the Java project’s Abstract Syntax Tree (AST) into Prolog facts through the JTransformer engine [12]. Problematic code fragments are identified through definition of appropriate Prolog rules. The work specifies a respective rule for detecting refactoring opportunities to the Composite design pattern.

As concerning behavioural patterns, Juillerat and Hirsbrunner specify an algorithm for automated refactoring to the Template Method design pattern [13,14]. The approach applies to a pair of methods of two different classes that share a common abstract class ancestor. The authors employ and extend existing methods for clone detection in order to identify the common and different statements of the compared methods. The differences are extracted to a new method for each class in a way that (a) extract method preconditions are satisfied for behaviour preservation and (b) the new methods have a common signature in order to be polymorphically invoked from the template method. The common parts of the initial methods are moved as a single template method to the common superclass of the refactored classes. Moreover, an additional abstract method is created in the superclass that has the same signature with the extracted methods. The findings of this work have been implemented as an Eclipse plug-in.

The approaches, described so far for refactoring to patterns, focus mainly on structural patterns (e.g. Abstract Factory, Builder, Composite) with the exception of the work of Juillerat and Hirsbrunner [13,14] that handle refactoring to the Template Method behavioural pattern. Our method is not directly comparable to these works, since refactoring to the Strategy design pattern is not in their scope. From a methodological perspective, relevant work can be categorized to (a) generic methods [8–11] (e.g. mini-transformations, logic inference) that can be applied with
appropriate parameterization to various patterns and (b) methods specialized for a specific pattern [13–15]. The proposed method (along with the work of [15] that will be described thereafter) belongs to the second category since it builds on the special structural and behavioural properties of the Strategy pattern.

Tsantalis and Chatzigeorgiou [15] also focus on behavioural patterns, through identification of refactoring opportunities towards the State/Strategy design patterns. Moreover, the authors propose a behaviour preserving transformation for applying the suggested refactorings. The identification of the candidate code fragments for refactoring is based on the syntactic analysis of the system's source code, construction and traversal of the corresponding AST. It focuses on two cases of refactoring for simplification of conditional expressions, as described by Fowler [1]: (a) replace type code with State/Strategy and (b) replace conditional logic with polymorphism. In the first case, a set of criteria is applied on a subset of the variables, defined inside the target class, that participate in conditional expressions. In the second case, code fragments that perform Runtime Type Identification (RTTI) are discovered in order to be replaced with invocation of polymorphic methods. The method has been implemented and integrated in the JDeodorant Eclipse plug-in [7], an open source project in which Tsantalis and Chatzigeorgiou participate as core members of the development team.

The method proposed in this work, for automated refactoring to the Strategy design pattern, is differentiated from the approach of [15] (or JDeodorant for brevity) in a series of aspects. First of all, the two methods analyze different parts of a conditional block in order to characterize it as a refactoring opportunity. While JDeodorant focuses on the conditional expressions of a conditional statement, our approach takes also into account properties of their branches (e.g., average number of statement per branch). The scope of applied checks for the evaluation of a given conditional statement, i.e., the code fragments that need to be analyzed in order to determine its suitability for refactoring to Strategy, is broader in this work as compared to JDeodorant. The latter targets its checks to the method that includes the candidate statement (intra-procedural analysis), while our approach extends its scope to methods that directly or indirectly invoke that method (inter-procedural analysis) and either belong to the declaring class or client classes (intra- and inter-class analysis). The analysis of these methods takes place by following the system's call graph in the inverse direction, starting from the method enclosing the candidate statement. JDeodorant employs syntactic analysis for the identification of the candidate code fragments for refactoring. Moreover, our approach employs control and data flow analysis that is based on the Program Dependence Graph (PDG) of each analyzed method. As concerning the application of refactoring to the suggested conditional statements, the proposed method performs (where applicable) total replacement of conditional statements with polymorphic invocation of methods on appropriate concrete Strategy implementations.

3. Automated refactoring to Strategy

3.1. State vs. Strategy design pattern

The State and Strategy design patterns, both members of the behavioural patterns family, are identical, as regarding their structure. Specifically, both patterns employ composition and polymorphism in order to circumvent the use of complex conditional logic. Fig. 1a and b highlight the structural similarity of the two patterns.

The Strategy pattern enables the use and interchange of different algorithms or implementations for a certain policy. It prescribes the definition of an abstract type (abstract class or interface), representing the policy contract (Strategy), and a series of different concrete subtypes (Concrete Strategies) that correspond to alternative implementations of the policy [6]. On the other hand, the State design pattern allows an object to alter its behaviour as a result of changes to its state. The object (instance of the Context class) appears to its clients as changing its class at runtime. The State pattern’s realization is based on the introduction of an abstract type (usually abstract class) that represents an Abstract State and defines methods corresponding to state-dependent operations of the Context class. Each discrete Context state is mapped to a concrete subtype of Abstract State (Concrete State) that provides the state-specific implementation of Abstract State’s methods and controls transitions to appropriate target states.

The similarity of these patterns, in terms of their structure and suitability for tackling complex conditional logic, often leads to their indistinguishable treatment by software designers. For instance, Fowler [1] proposes a series of code transformations for refactoring conditional logic to State/Strategy (termed as replace type code with State/Strategy) without emphasizing on a clear distinction between the two patterns. On the other hand, Kerievsky [5] brings out in his work the different purpose of these patterns by fully separating refactorings to the State pattern (replace state-altering conditionals with State) from refactorings to Strategy (replace conditional logic with Strategy).

A key difference between State and Strategy lies in their scope of use. While State is most appropriate for an object that changes its behaviour as it passes through different states, Strategy allows transparent interchange of alternative algorithms (strategies) for a certain aspect of an object’s behaviour. Note that although strategies are interchangeable, this does not hold for state implementations that are only relevant to a certain part of the Context object’s life-cycle. Algorithm selection in Strategy usually takes place during the initialization of a Context instance and does not change during its life-cycle. State transitions, though, occur throughout the life-cycle and independently for each Context instance [16].

Further differentiation among the two patterns is based on the degree of client involvement during their use, as quoted by Gamma et al. [6]: “Clients must be aware of different Strategies. The pattern has a potential drawback in that a client must understand how Strategies differ before it can select the appropriate one”. In other words, Strategy selection is delegated to the client or to a factory. As regarding the State pattern, the client interacts with the Context object without being aware of its available states or

---

**Fig. 1.** State and Strategy design patterns.
directly affecting state selection. However, as State transitions are controlled by state objects, a Concrete State subtype may be aware of the presence of other states, unlike alternative strategies in the Strategy pattern that are fully decoupled from each other.

As concerning the coupling of the types that introduce polymorphism (states or strategies) with the Context class, the State pattern is characterized by the highest coupling. Specifically, concrete state subtypes are aware of all aspects of Context's behaviour and usually have access to its State transition operations. On the other hand, strategies usually do not require access to the Context interface, while they are relevant to a single behaviour of the Context class.

3.2. Automated refactoring of conditional logic in JDeodorant

The work of [15] focuses on the identification of refactoring opportunities that introduce the State/Strategy pattern. The implementation of their approach is part of the JDeodorant Eclipse plug-in [7] that also supports full automation in the application of the identified refactorings. Identification of refactoring opportunities concerns two cases of refactoring for simplification of conditional expressions, as described by Fowler [1]:

- replace type code with State/Strategy,
- replace conditional logic with polymorphism.

As concerning the first case, Tsantalis and Chatzigeorgiou [15] propose a method for identification of conditional statements (referred to as state-checking code fragments) that are appropriate for applying refactoring to the State/Strategy pattern. The identification of state-checking code fragments is based on the evaluation of a set of criteria on certain types of variables (State Variables1 or type codes according to Fowler [1]) that participate in condition expressions of their branches. The concept is that a state-checking code fragment contains, in all branch conditions, equality comparisons of a certain State Variable (SV) against different constants. Each constant corresponds to a different state of the Context class. In the second case, a method is proposed for detection of code fragments that perform Runtime Type Identification (RTTI) and their consecutive replacement with invocations of polymorphic methods. An RTTI code fragment is a conditional statement that includes in all branch conditions instanceof or equivalent equality comparisons containing the same Supertype Variable (STV), i.e., a variable whose class is subclassed by other classes of the system. For instance, let \( v \) be a variable of type \( C \) and \( C_i, i = 1,2, \ldots \) be subclasses of \( C \), then the presence of \( v \text{ instanceof } C_i \) expressions in all branch conditions of a conditional statement results to its identification as RTTI code fragment.

The approach of [15], for identification of “replace type code with State/Strategy” refactoring opportunities, keeps pace with [1], as regarding the common handling of both State and Strategy patterns. Their method, that will be henceforth referred to as JDeodorant, employs syntactic analysis in order to figure out properties of the candidate conditional statements that match the common structure of the State/Strategy design patterns. As the identification procedure does not focus on the intent or behavioural aspects that differentiate these patterns, the suggested refactoring candidates may correspond to opportunities for introducing either (a) the State pattern, (b) the Strategy pattern, or (c) plain polymorphism. In practice, the refactoring opportunities identified by JDeodorant correspond mainly to the State design pattern. This is justified by the nature of the identification criteria and the orientation of the method towards the State pattern. Moreover, since the identification procedure focuses on the presence of named constants in branch conditions, the method misses refactoring candidates that do not use named constants for explicit representation of alternative states/strategies.

3.3. A method for automated refactoring to Strategy

We propose a method for automated refactoring to Strategy that supports identification of refactoring opportunities and transformation of the source code to comply with the Strategy design pattern. Our method complements JDeodorant, that focuses mainly on the State pattern, with capabilities for automated refactoring to the Strategy design pattern. The use of novel refactoring identification criteria, that also take into account behavioural properties of the Strategy design pattern during evaluation of the candidate conditional statements, differentiates our method from the approach of [15]. Specifically, the focus is on verifying that (a) branch selection in a conditional statement is not controlled by the Context class and (b) branch bodies correspond to non-trivial computations. These properties reflect the interchange of alternative strategies in the Strategy design pattern and their identification in conditional statements improves substantially the recall of refactoring opportunities to Strategy, as compared to JDeodorant. As concerning the proposed source code transformation for introducing the Strategy pattern, it supports, under certain conditions, the total replacement of the conditional statement with a polymorphic method call.

Our method searches for refactoring opportunities among the conditional statements declared in a given Context class and evaluates them on the basis of properties of their branch bodies and conditions. On the other hand, JDeodorant’s analysis of conditional statements is based exclusively on properties of branch conditions. Moreover, the scope of applied checks for the evaluation of a given conditional statement, i.e., the code fragments that need to be analyzed in order to determine its suitability for refactoring to Strategy, is broader in this work as compared to JDeodorant. The latter targets its checks to the method that includes the candidate conditional statement (intra-procedural analysis), while our approach extends its scope to methods that directly or indirectly invoke that method (inter-procedural analysis) and either belong to the Context class or client classes (intra- and inter-class analysis).

The processing of conditional statements in this method is based on static analysis, as well as control and data flow analysis relevant to variables in branch conditions and their values. The State Variables of the JDeodorant method are referred to, in this work, as Strategy Selection Variables (SSVs), since they represent a different, though overlapping, set of variables (fields and parameters of non-primitive type are also included, local variables are excluded) and serve a different purpose.

As mentioned above, the identification of refactoring opportunities is based on the role of the Context class and its clients in branch selection in the candidate conditional statements. This also contributes to the separation of Strategy from State refactoring opportunities. Specifically, if branch selection in a conditional statement is determined by clients of the Context class (class declaring the statement), then the code fragment is considered as candidate for refactoring to Strategy. This is in accord with selection of Concrete Strategies in the Strategy design pattern, as also highlighted in the comparison of the two patterns of Section 3.1. The proposed method evaluates branch selection in a conditional statement by identifying the classes that define (directly or indirectly) the values of SSVs in branch conditions.

3.3.1. Algorithm for identification of refactoring opportunities

This section introduces an algorithm for identification of refactoring opportunities towards the Strategy design pattern. The algo-

---

1 State Variables comprise (a) non-static fields, (b) parameters or (c) local variables declared before the state-checking code fragment. They correspond to primitive types or enumerations.
Algorithm operates each time on a single class of the system under maintenance, that will be, henceforth, referred to as Context class. This characterization denotes the role of the class in the Strategy design pattern, in case that a refactoring is successfully identified and applied.

The code fragments that are processed by the algorithm, in search of refactoring opportunities, are conditional statements (i.e., switch, if/else, if/else if/...) declared in Context methods. However, only a subset of conditional statements, providing hints for Strategy selection, are thoroughly evaluated. These statements are referred to as Strategy Selection Blocks (SSBs) and are determined by the decision function _isSsb_. The function receives a conditional statement as parameter and returns true in case that the statement is an SSB. The rules that comprise the decision logic of _isSsb_ and must apply for a conditional statement _s_ in order to be considered as SSB, are:

(a) _s_ must have at least two branches,
(b) _s_ must not be part of a static method or other static code block,
(c) _s_ must not be part of a method that overrides a method declared in a non-system class, e.g. methods of the language runtime libraries like equals, toString, hashCode for Java,
(d) the average number of statements per branch (excluding setter and object instantiation statements) for _s_ must be at least two, except for cases that branches contain invocations of Context methods.

The identification of an SSB as refactoring opportunity is contingent on characteristics of the variables that participate in the condition expressions of its branches. A determining factor is the presence of certain variables of the Context class that potentially contribute to Strategy selection and will be referred to as Strategy Selection Variables. Let _SSV_ be the set of these variables for the Context class that comprises: (a) member variables (or fields) (Strategy Selection Fields set, _SSF_ that their value assignment is not controlled by the Context, and (b) method parameters (Strategy Selection Parameter set, _SSP_ that their value is defined, either directly or indirectly, by Context class clients. It holds that _SSV_ = _SSF_ U _SSP_.

The elements of _SSF_ are member variables that are not assigned a value inside the Context class. An exception to this rule are member variables that are assigned a value inside the body of setter methods that are either invoked by: (a) clients of the Context class, (b) Context class methods that provide as setter parameter during invocation method parameters whose values originate from outside of the Context class (as determined by procedure get_parameter_trace that will be explained thereafter).

Fig. 2 presents a code fragment from class SerializationUtility that supports object serialization/deserialization across different formats. The conditional statement declared in serializeObject method is a valid SSB, since it satisfies all rules of the isSsb function. Member variable serializationMode is an SSB, as its value is not defined inside SerializationUtility. Note that SerializationUtility is characterized as an SSB despite its value assignment by setter method setSerializationMode. The reason is that the method is never invoked by its declaring class and, thus, its parameter value is always provided by third-parties. The same holds for parameter obj of serializeObject method that represents an SSP.

The identification of refactoring opportunities for a given class is performed by Algorithm 1. The algorithm generates, during its execution, a set of refactoring opportunities _R_, represented by pairs (_s,V_) where _s_ is an SSB and _V_ a set of decision variables of _s_ that perform branch selection and their value is not controlled by Context text. For brevity reasons, additional notation will be introduced prior to algorithm specification. Let _sm_ denote a conditional statement declared in method _m_ of the Context class. The set _DV(sm)_ for _sm_ includes variables that participate in condition expressions of its branches, while _DV(SSB)_ is a subset of _DV(sm)_ with variables that participate in condition expressions of all branches of _sm_. Moreover, let _LVm_, _Pm_ represent the sets of local variables and parameters of _m_ respectively. Unlike _SSVs_, the presence of local variables in condition expressions of the branches of an _SSB_ results to its rejection from refactoring opportunities. The reason is that local variables are always assigned a value inside the declaring method and, thus, their value is controlled by the Context.

The algorithm suggests a given _SSB sm_ as refactoring candidate in case that the condition expressions of all its branches contain at least one variable _v_ that is either: (a) a Strategy Selection Field, or (b) a Strategy Selection Parameter of method _m_, i.e., a parameter that (i) is not assigned a value (is not defined) in any execution path inside method _m_ that leads to _sm_ (function _is_defined_(m,s,v)) and (ii) is always bound to values provided by Context clients during method _m_ invocations (procedure get_parameter_trace(_m_,v)). The presence in _DV(sm)_ of any variable that is assigned a value by Context results to the rejection of _sm_ from refactoring candidates. With reference to the code fragment of Fig. 2, Algorithm 1 returns a single refactoring opportunity _R_ = {_s, serializationMode}, where _s_ is the delineated conditional statement. Note that although parameter _obj_ is an SSB, it is not returned by our algorithm since it does not participate in branch conditions of all branches of _s_.

![Fig. 2](image-url) Illustration of SSB, SSF and SSP instances in a code fragment.
Algorithm 1. Refactoring to Strategy Identification Algorithm

```plaintext
input : System class C
output: R = \{(m,v): m \in SSB, v variable\}
1. R = \{\}
2. foreach Conditional Statement sn do
   /* subscribe m denotes the method that includes statement sn */
   /* if the statement is not SSB */
   /* if decision variables include local variables */
   if (isDecisionVariable(sn)) then continue;
   /* for each variable in branch conditions of sn */
   foreach variable v in DB(sn) do
     /* if v is an SSB or an SSS */
     /* if v is defined(m,sn,v) = false */
     if v \not\in SSB \&\& \neg isDefined(m,sn,v) = false then continue;
     /* for each variable in branch conditions of sn */
   end
   V = \{\}
   foreach variable v \in DB(sn) do
     /* if v is an SSS */
     /* if v is present in conditions of all branches of sn */
     if v \in DB(sn) then V \leftarrow V \cup \{v\};
     else/* variable v is assigned a value inside class C */
       if v \not\in DB(sn) then V \leftarrow V \cup \{v\};
     end
   end
   V = \emptyset;
   break;
end
if V = \emptyset then R \leftarrow R \cup \{\{sn,V\}\};
end
return R;
```

The boolean function `is_defined (m,s,v)` determines whether a variable `v` is assigned a value at least one execution path between the start of method `m` and a statement `s` that is included in the body of `m`. The function employs the Program Dependence Graph (PDG) of method `m` and returns `true` if the PDG node corresponding to statement `s` has at least one incoming data dependence edge for variable `v`. The functionality of `is_defined` relies on the base PDG specification, initially proposed by Ferrante et al. [17], enhanced with various additions (e.g., support of break, continue, try/catch statements). A comprehensive presentation of that PDG model is included in the work of [18] that employs the PDG for automation of the extract method refactoring.

Algorithm 2. Procedure get_parameter_trace

```plaintext
input : Method m, parameter p
output: Tuple set T = \{(m',s,e): m' \in method, s \in statement inside m', e \in expression\}
1. T \leftarrow \emptyset;
2. foreach each caller mp of method m do
3.   foreach Statement s that invokes m in mp do
4.     if (s \not\in actual(p, m, s))
5.       if (\text{Methods } m, m' \in mp \text{ belong to different classes})
6.       if (\text{class}(m') \neq \text{class}(m))
7.         \text{T} \leftarrow \text{T} \cup \{(m',s,e)\};
8.       else
9.         t \leftarrow \emptyset;
10.        if (\text{m is invoked with a field value provided by client class})
11.       if (\text{a \in SSS then \text{t} \leftarrow \{(m,a)\}})
12.       if (\text{m is invoked with a parameter of } m \text{ that is not defined before its use in statement } s)
13.       if (\text{a \in P_m, \&\& is_defined}(m,a) = \text{false})
14.         \text{T} \leftarrow \text{T} \cup \{(m',a)\};
15.       else\text{ get_parameter_trace}(mp, s);
16.     if (\text{t} \neq \emptyset) then \text{T} \leftarrow \text{T} \cup \{(m',s,e)\};
17.   end
18. end
19. end
20. return T;
```

The `get_parameter_trace (m,p)` procedure is described by Algorithm 2. It represents a recursive procedure that (a) receives as input a method `m` of the `Context` class and a respective parameter `p` of that method and (b) returns a list of direct or indirect invocations of `m`, by methods of the `Context` class or its clients, along with the respective values supplied for parameter `p`. Its focus is to determine whether the value(s) bound to method parameters originate exclusively from clients of the `Context` class. In the positive case, the function returns a set `T` of tuples `(m,s,e)` where `m` is a method that calls `m` (directly or indirectly), `s` the calling statement and `e` the expression that is supplied as value for parameter `p`. In any case that the method `m` is called with a value for parameter `p` determined by `Context`, the procedure returns the empty set.

Fig. 3 presents the recursive invocation of the `get_parameter_trace` procedure on methods of the `RandomGenerator` class. Its execution confirms that the values passed for parameter `distr`, during invocations of the `generateRandomVariable` method, are always indirectly provided by clients of the `RandomGenerator` class (in this case by `VideoSource`). The dashed arrows in Fig. 3 illustrate how the procedure “traces” the sources of `distr` values. Based on the successful outcome of `get_parameter_trace`, our refactoring identification algorithm suggests the delineated switch statement as a candidate for refactoring to Strategy with parameter `distr` having the role of the `SSV`.

The procedure operates iteratively on each method `mp` that invokes `m` (line 2 of Algorithm 2). Let `C_m` be the set of all methods that call directly a given method `m`. The role of `mp` in the example of Fig. 3, is enacted by `generateRandomVariable`, while its callers `C_mp` can be derived from the respective call graph. For each statement `s` of `mp \in C_m` that invokes method `m`, the actual parameter `a` bound to parameter `p` is retrieved through function `actual (p,m,s)` (line 4). The function returns a variable name, in case that the actual parameter is a single variable expression. Otherwise, its return value is null. If `mp` is a `Context` method, the tuple `(mp,s,a)` is added to the set `T` only in case that the actual parameter `a` corresponds to a Strategy Selection Field of `Context` (line 9), i.e. its value is not determined by that class. In our case study example, the callers of `generateRandomVariable` are also methods of `RandomGenerator`, but provide as value for parameter `distr` one of their parameters. In such cases, `get_parameter_trace` is executed recursively for each caller `mp` in order to determine whether their respective parameter is, in its turn, controlled by `Context` clients (line 10). Fig. 3 shows the recursive invocations initiated by `get_parameter_trace` and the methods that they apply to.

In any case that a caller `mp` does not belong to `Context` class, the tuple `(mp,s,a)` is added without further processing to the set `T` (lines 5, 6). This condition holds at the execution of `get_parameter_trace` for methods `generateRandomVariable (char, float, float)` and `genRv (char, float, float)` respectively. Note that the class of a method `m` is denoted by the predicate `class (m)`.

If actual parameter `a` is a local variable, constant or parameter with non-primitive type the procedure returns the empty set. Note that if any recursive call to `get_parameter_trace` returns the empty set, i.e., the parameter under check is controlled by `Context`, then the procedure also returns the empty set, regardless of the existence of any elements in `T`.

The finiteness of `Algorithm 2` is not affected by the presence of circles in the call graph of `Context`, i.e., existence of recursive methods or circular invocations among `Context` methods. These
cases can be handled by marking the methods (call graph nodes) that have been used as parameters in getParameterTrace entry by storing a reference to method \textit{m} into a list structure. Each caller \textit{m} is looked up in that list prior to the invocation of getParameterTrace in line 10 of the algorithm. In case of successful look up, the procedure is not invoked again with \textit{m}, but, instead, its execution continues with the next iteration of the inner for-loop. These details are suppressed from Algorithm 2 for reasons of clarity to its presentation.

3.3.2. Applying the refactoring

**Base Refactoring Procedure** The suggestions for refactoring to Strategy, derived by the algorithm of Section 3.3.1, can be applied to the source code through a sequence of elementary refactoring operations. In certain cases, the code transformation involves the complete elimination of the conditional statement. This section specifies the procedure that transforms an SSB, declared inside method \textit{methodA} of a class named Context, to a code block that employs polymorphism based on the Strategy design pattern. Let \textit{b} be the SSB and SSV its respective set of Strategy selection variables. The refactoring to Strategy procedure involves the following steps:

1. Create the Abstract Strategy class \textit{AbstractXStrategy}. Let \textit{X} represent the name of the Context's behaviour with alternative implementations that the pattern will support.
2. Add an abstract method, also named \textit{methodA}, to \textit{AbstractXStrategy} representing the behaviour with alternative implementations.
3. Add formal parameters to \textit{methodA} that include:
   - the Context class, if the body of at least one branch of \textit{b} includes statements that reference fields or methods of the Context class or its super-class,
   - one parameter for each parameter or local variable that is defined outside \textit{b} and is read in statements of at least one branch of \textit{b}.
4. For each branch \textit{i} of \textit{b} create a \textit{ConcreteXStrategy} class that inherits from \textit{AbstractXStrategy} and
   - override \textit{methodA} in each subclass with the body of the respective branch of the SSB,
   - add the pair \((e_i, c_i)\) to a list \(L\), where \(e_i\) represents the branch condition expression and \(c_i\) the branch's corresponding concrete Strategy class. The role of the list \(L\) will be explained thereafter.
5. Add to Context a private member variable \textit{instXStrategy} of type \textit{AbstractXStrategy} that will hold the currently active concrete Strategy instance.
6. If the set SSV contains just a single Context.methodA parameter of primitive or enumerated type then
   - find the set \(T\) of invocations of \textit{methodA} with the help of getParameterTrace procedure,
   - if the variable expression part \(e_i\) of each tuple \(t_j \in T\) is a constant then Total Replacement of Conditional Logic in code fragment \textit{b} is performed and the refactoring procedure terminates. This step will be analyzed in detail in the next subsection.
7. Replace the body of each conditional branch \textit{i} of \textit{b} with the following statements:
   - assignment of a new \textit{ConcreteXStrategy} instance to \textit{instXStrategy},
   - invocation of the method \textit{instXStrategy.methodA}.

The application of this procedure to a refactoring opportunity identified in the open source project Pamvotis [19] will be described for illustrative purposes. Pamvotis simulates a network of wireless nodes that generate data traffic and coordinate their packet transmissions with the IEEE 802.11 protocol. The traffic, generated by each wireless node, has the form of a stream of packets. Each wireless node that generates data traffic and coordinates its packet transmissions with the IEEE 802.11 protocol. The traffic, generated by each wireless node, has the form of a stream of packets. Each wireless node that generates data traffic and coordinates its packet transmissions with the IEEE 802.11 protocol. The traffic, generated by each wireless node, has the form of a stream of packets.

Packet streams in Pamvotis are generated by instances of FTPSource, HTTPSource, VideoSource and GenericSource classes that inherit from abstract class Source. With regard to the first three classes, they implement widely accepted traffic models for FTP, HTTP and Video applications respectively. All Source subclasses depend on the RandomGenerator class that has the responsibility of generating random numbers according to various probability distributions (e.g., exponential, pareto, uniform). Fig. 4
presents a class diagram with the relationships of the classes that are relevant to our illustrative case study.

The refactoring opportunity identified by the proposed algorithm concerns the switch statement included in the generateRandomVariable method of RandomGenerator. The method's source code, before refactoring to Strategy, is depicted in the left part of Fig. 5. The role of the SSB in the switch statement is represented by the parameter distr that determines the random number generation algorithm during generateRandomVariable invocation. The domain of distr spans a set of character constants that correspond to different probability distributions. Each constant activates a different branch of the switch statement, implementing an appropriate pseudo-random generation algorithm for the probability distribution.

The left part of Fig. 5 presents relevant code fragments of RandomGenerator and VideoSource classes before refactoring. These classes play the roles of Context and Client in the Strategy design pattern that derives from refactoring and is depicted in the right part of the figure. The code that is produced by each step of the refactoring procedure is annotated with the respective step number. Specifically, AbstractRandomGen, that has the role of the abstract Strategy class, results from steps 1, 2, 3. During step 4, the body of each branch of the SSB is mapped to the implementation of a concrete Strategy implementation, e.g., the branch that corresponds to constant 'u' is refactored to Strategy class UniformRandomGen that inherits from AbstractRandomGen (step 4). After refactoring, the body of each branch is replaced by a call to an instance of an appropriate subclass of AbstractRandomGen (step 7). Note that step 6 is not included in Fig. 5 since it involves the elimination of the switch statement that is not applicable in the SSB under examination.

**Total Replacement of Conditional Logic**

The proposed refactoring method supports, under certain conditions, the Total Replacement of Conditional Logic (TRCL) in an SSB. TRCL support is restricted to SSBs that include in all branch conditions a single Strategy Selection Variable of primitive or enumerated type, represented by a specific Context methodA parameter p. TRCL applies in cases that the concrete Strategy implementation, that is used during each (direct or indirect) invocation of Context methodA by client classes, can be inferred at compile time. This is determined by ensuring that parameter p is bound to constant values during client calls of Context methodA. TRCL results to the transformation of an SSB to a simple non-conditional code fragment. This refactoring employs the list L, populated in step 4 of the Base Refactoring Procedure, as well as the tuple set T generated on step 6. On the basis of the previously introduced notation, the additional refactoring operations required during TRCL are:

(a) For each element \( t = (m_i, s_j, e_j) \in T \), where \( m_i \) represents the method of a client class that controls the value of parameter \( p \). \( s_j \) is the statement where the value is supplied through an invocation of a Context method and \( e_j \) is a constant expression representing the actual value of the parameter, do the following:

- Find the tuple \( l_i \in L \) whose expression \( e_i \) evaluates to true once parameter \( p \) is replaced by \( e_j \). Recall that exactly one element of \( L \) is selected, since conditional expressions of the branches of an SSB are mutually exclusive.
- Let \( \text{methodC}_0 \) be the Context method that is invoked in statement \( s_j \). Introduce in that statement an additional parameter to the \( \text{methodC}_0 \) invocation given by the expression new ConcreteXStrategy(\( t \)).

(b) Introduce an additional formal parameter to Context methodA with name \( x \) and type AbstractXStrategy. Assume that \( x \) is placed first in the list of methodA formal parameters.

(c) Replace the SSB inside Context methodA with the following two statements:

- assignment of \( x \) to member variable instXStrategy.
- invocation of the instXStrategy.methodA. The latter may be invoked in a return statement in case it has a non-void return type.

(d) Let \( ssb \) be the Context methodA parameter that has the role of a Strategy selection variable. If \( ssb \) is not referenced inside methodA eliminate it from the list of formal parameters. Moreover, eliminate the respective actual parameters in methodA calls.

(e) For each Context method methodC that invokes methodA do the following:

- apply step (b) to that method,
- introduce the new parameter \( x \) as a first argument in methodA invocations,
apply step (d) to methodC for the parameter that was supplied as a value for the ssv parameter in methodA calls,
apply step (e) recursively for each Context method that invokes methodC.

The application of TRCL to the SSB of Fig. 5 is presented in Fig. 6. Note that our identification algorithm does not suggest TRCL for the given conditional statement, as the values supplied by class GenericSource (Fig. 4) for the SSV distr are not constants.

The refactoring included in Fig. 6 is valid under the assumption that class GenericSource is removed from project Pamvotis.

The annotations in Fig. 6 correspond to the steps of TRCL procedure that introduce or eliminate the respective code fragments. Specifically, step (a) applies to invocations of method RandomGenerator.genRv by client classes VideoSource and HTTPSource (the latter is not included in figure and will no longer be referred to). It, initially, determines which AbstractRandomGen concrete implementation, included in list L, corresponds to each character literal that is bound to parameter distr on RandomGenerator.genRv method invocations. In the case of VideoSource class, the aforementioned concrete implementation is ParetoRandomGen. Step (a) completes with the introduction of a ParetoRandomGen object instantiation expression as the first parameter in...
3.3.3. Preconditions

The source code transformation of Section 3.3.2 comes with a set of preconditions that are evaluated on the candidate conditional statements for refactoring. Applying the transformation to a refactoring candidate that violates any of these preconditions results to erroneous source code that either has compilation errors or does not preserve the external behaviour of the system. The preconditions that are associated with the Base Refactoring Procedure of Section 3.3.2 are the following:

- Branch bodies should not contain assignment of two or more different local variables. This precondition is required when the refactoring is applied to Java source code where method parameters are always passed by-value. Thus, it is not possible to return multiple values as method results. In other object-oriented languages, such as C++ where pass by-reference is supported, this precondition can be revoked. On the other hand, if branch bodies include assignments to a single local variable, its value can be returned by the concrete implementations of the polymorphic method as a return value.
- Branch bodies should not contain unstructured control flow statements (break or continue) in case that the refactoring candidate is nested inside iteration statements. If this precondition is violated the refactored code will contain compilation errors.
- Branch bodies should not contain any super method invocations since the move of the branch body outside the Context class will lead to compilation errors.
- The names of the newly introduced classes that belong to the Strategy hierarchy should not conflict with names of existing classes of the system.

These preconditions are also required in the JDeodorant method during refactoring to State/Strategy [15]. As regarding Total Replacement of Conditional Logic, an additional precondition is required that prevents the application of the transformation in cases that the method containing the refactoring candidate overrides a super-class method or implements an interface operation. This precondition prevents compilation errors that result from non-conformance of the Context method signature with the signature of the overridden methods.

3.3.4. Implementation details

The proposed method has been implemented as part of the JDeodorant plug-in for Eclipse [7]. The syntactic analysis of Java source files, performed during execution of the refactoring identification algorithm, is realized through (a) AST parsing capabilities provided by the Eclipse Java Development Tools (JDTs) Core infrastructure and (b) utility classes included in the JDeodorant project. Moreover, the identification of refactoring candidates is based on control and data flow analysis performed on Program Dependence Graph (PDG) representations of class methods that are constructed through relevant infrastructure of JDeodorant [18]. Finally, the transformations required for refactoring to Strategy have been implemented with functionality provided by JDT and the Eclipse Language Toolkit (LTK).

4. Experimental evaluation

The proposed method for automated identification of refactoring opportunities towards the Strategy design pattern has been applied on a set of software projects for evaluation purposes. The objectives of the experimental evaluation are: (a) quality assessment of the refactoring to Strategy suggestions, on the basis of judgments made by two software engineers, (b) estimation of the scalability of the refactoring identification algorithm and its implementation, in terms of execution efficiency.

The selection of the Java code projects, serving as input for the evaluation of our method, is based on the following requirements: (i) the projects should have different sizes and serve various application domains in order to enable, to a certain extent, the generalization of the derived outcomes, (ii) projects should be familiar to the evaluators, ensuring, thus, a high reliability on their judgments, (iii) projects unfamiliar to the evaluators should be of small to medium size in order to facilitate the review and understanding of their functionality. Table 1 summarizes the projects used for

![Fig. 6. Total replacement of conditional logic in Pamvotis.](image-url)
the evaluation of this work. Quality assessment is based on the first three projects, while the scalability of our approach is evaluated on all projects.

Table 2 provides information on the size and structure of the aforementioned projects such as numbers of Source Lines of Code (SLOC), classes and methods. As concerning the last column of this table, it provides the number of candidate SSBS that need to be evaluated by the proposed identification algorithm for each project.

4.1. Quality assessment of refactoring identification algorithm

The evaluation of the refactoring identification algorithm is based on the manual detection of refactoring candidates, in projects FlowScheduler, IceHockeyManager and MSSDesigner (Table 2), by two expert software engineers that will be henceforth referred to as evaluators. Both evaluators are familiar with the Strategy design pattern and its appropriateness for dealing with conditional complexity. For each Java project, the evaluators individually inspect all if\[\backslash{}switch\] conditional statements with at least two branches (i.e., statements that satisfy the minimum requirements for being considered as refactoring candidates) in search of refactoring opportunities. Let S be this set of conditional statements for a given Java project. The approval of a conditional statement as refactoring candidate is contingent on the evaluator’s judgement. The refactoring identification algorithm is executed on the Java projects and quality assessment, from the viewpoint of each evaluator, is performed on its results. Specifically, the automatically identified refactoring opportunities are compared against the refactorings suggested by evaluators and a distinct set of quality metrics is calculated for each one of them.

Depending on the characterization of the elements of S as refactoring opportunities by the evaluator and the identification algorithm, the set S can be divided into the following disjoint subsets: (a) True Positive elements (S\textsubscript{TP}), i.e., correctly identified refactoring opportunities by the algorithm, with respect to the evaluator’s judgement, (b) False Positive elements (S\textsubscript{FP}), i.e., incorrectly identified refactoring opportunities by the algorithm, (c) True Negative elements (S\textsubscript{TN}), i.e., correctly rejected refactoring suggestions by the algorithm, (d) False Negative elements (S\textsubscript{FN}), i.e., valid refactoring opportunities, according to the evaluator, that are falsely rejected by the algorithm.

The quality assessment of the identification algorithm is based on these metrics, namely Precision, Recall and Accuracy. For a given evaluator and Java project, these metrics are calculated on the cardinalities TP, FP, TN, FN of the respective subsets S\textsubscript{TP}, S\textsubscript{FP}, S\textsubscript{TN}, S\textsubscript{FN}:

\[
\text{Precision} = \frac{TP}{TP + FP} \\
\text{Recall} = \frac{TP}{TP + FN} \\
\text{Accuracy} = \frac{TP + TN}{TP + FP + TN + FN}
\]

Table 3 includes the values for precision, recall and accuracy that result from the evaluation of algorithm’s suggested refactorings on projects FlowScheduler, IceHockeyManager and MSSDesigner. The quality metrics are presented separately for each individual evaluator. Their different values are due to the non-identical sets of refactoring candidates identified by the two evaluators, denoting a degree of subjectivity in the assessment of whether branch bodies of a conditional statement represent algorithms that are significant enough to be represented as methods of Concrete Strategy classes. On the basis of the number of identified refactorings (TP + FN), the second evaluator is more conservative in his suggestions, probably due to his higher familiarity with the Java projects. Nevertheless, the quality assessment results show that our refactoring identification algorithm provides, with satisfactory confidence, meaningful refactoring suggestions to both evaluators.

Evaluation results relevant to the recall metric show that the algorithm identifies on average half of the refactoring candidates suggested by evaluators. Note that recall is inversely proportional to the number of False Negatives (FNs), i.e., to “valid” refactorings that were not identified by the algorithm. A study of FNs encountered in Java projects lead to the following categorization of the reasons for being rejected by our method:

1. presence of local variables in branch conditions. These local variables take their values from parameters or fields “controlled” by \textit{Context} clients.
2. presence of attributes in branch conditions that are initialized inside \textit{Context} class. These attributes are assigned a default value during their declaration or take their value from parameters “controlled” by \textit{Context} clients.
3. presence of attributes and local variables in branch conditions that their value is “controlled” by \textit{Context} class.

The majority of FNs fall into the first two cases and, thus, could be possibly eliminated with appropriate handling of local variables and fields (with assignments inside \textit{Context}) participating in branch conditions. Specifically, more extensive data flow analysis is required, embracing these types of variables, in order to conclude whether their values originate from \textit{Context} clients and, thus, take into account the respective conditional statements in the refactoring identification process. In certain cases, local variables or \textit{Context} fields are defined with results of method invocations that retrieve system properties (e.g., \texttt{System.getProperty}) or information from configuration files. Although the source of their assigned values is external to the system, its tracing is a rather challenging task, since it requires semantic analysis of the system’s code.

As concerning the precision metric, it ranges around 50% (33.33–64.29%) for all projects and, thus, almost half of suggested refactorings are meaningful to the evaluators. Precision improves as the number of False Positives (FPs) declines. Analysis of the conditional statements marked as FPs during evaluation revealed that they are, generally, characterized by branch bodies with trivial logic, such as initialization of user interface components, object instantiation and parameterization or output formatting. However, our algorithm suggests these conditional statements as refactoring candidates due to their relatively high average number of statements per branch (\geq 2). The elimination of these FPs requires semantic analysis of branch bodies and will be handled in future versions of this work.

Since refactoring to patterns aims at improving code quality, the evaluation of our method for refactoring to Strategy involves, also, a study of its impact on software quality metrics. The focus is on assessing the improvement to cyclomatic complexity M that results from introducing the Strategy pattern to the automatically identified refactoring opportunities. Moreover, we estimate the
reduction of method size MLOC (Method Lines of Code), in terms of lines of code, for methods declaring the respective conditional statements. We adopt the McCabe’s Cyclomatic Complexity metric [28] as a quantitative indicator of conditional complexity and calculate its value prior and after introducing the Strategy pattern, for each method that includes a refactoring candidate.

Table 4 summarizes these code quality measurements for the three projects that participate in quality assessment. Column M_p includes the average value of cyclomatic complexity, prior to refactoring, over all methods that declare a refactoring candidate identified by our algorithm. Column M_a includes the aforementioned metric after refactoring. The average size in lines of code for methods declaring a refactoring candidate, prior and after refactoring, is presented in columns MLOC_p and MLOC_a respectively. The standard deviation corresponding to each average value is provided inside parentheses.

The code quality measurements show a considerable improvement to McCabe Conditional Complexity (15–30%) and MLOC (20–30%) metrics. Thus, our method succeeds to reduce code size and complexity of several classes on each code base through the introduction of the Strategy design pattern. Note, however, that this process does not eliminate conditional logic (although TRCL, when applicable, achieves a partial elimination). Instead, it manages to distribute it across the Context and Concrete Strategy classes resulting in more maintainable and understandable code units.

4.2. Scalability of the approach

The merits of automated identification of refactoring opportunities escalate with the size of the code project under maintenance. A basic reason is that large projects, in terms of Source Lines of Code (SLOC), number of classes and methods etc., are more difficult to review and maintain, as they are usually implemented by multiple developers. Thus, an essential part of the evaluation of our refactoring identification method relates to its scalability in terms of execution time. The projects that constitute the input data are included in Table 2 and have various size attributes in terms of SLOC, number of classes, methods and candidate SSBs.

Table 5 presents the execution time of the identification algorithm when applied to the projects of Table 2, as well as its decomposition into various time components that correspond to different processing tasks of the algorithm. The measurements have been collected after running the algorithm in a workstation equipped with a 2.2 GHz dual core processor and 4 GB of RAM. The column AST Creation represents the time required for parsing the source files and constructing the AST for each project, while Identification Time corresponds to the actual algorithm execution time. The total execution time for each project is presented in the last column of the table.

Execution time results show a positive correlation between the AST Creation time and SLOC, number of compilation units for each code base (Table 2). On the other hand, the Identification Time generally increases with the number of if/switch statements that are processed by the algorithm in each project. The results justify the scalability of the approach, since total execution time is below 30 s for small to medium size projects and does not exceed 2.5 min for large projects. Note that the actual processing time of the identification algorithm is even lower (<13 s for medium size projects,
<75 s for large ones), since the file parsing and AST creation tasks represent a significant part of the total execution time.

5. Conclusions and future work

We have proposed a method for simplification of conditional logic in object-oriented source code, through the automated introduction of polymorphism with the use of the Strategy design pattern. The method introduces automation to (a) the identification of refactoring opportunities towards the Strategy pattern and (b) the application of the refactoring through a source code transformation. The identification of refactoring opportunities is based on an algorithm that operates on a given class (candidate Context class) and evaluates properties of its conditional statements through syntactic, control and data flow analysis. Suggested refactoring opportunities comprise conditional statements that are characterized by certain analogies to the Strategy pattern, in terms of the purpose and selection mode of strategies: (i) conditional branches include mutually exclusive, non-trivial computations, corresponding to alternative algorithm implementations represented by strategies, (ii) branch selection is not controlled by Context class logic but, instead, by its clients, in proportion to Strategy selection in the Strategy pattern. The presence of these analogies in a conditional statement is detected by ensuring that the values of variables that participate in branch condition expressions are controlled by Context clients. The types of variables that are processed are method parameters and Context fields. On the other hand, the presence of local variables in branch condition expressions results to rejection of the conditional statement from refactoring opportunities, since local variables are always initialized and calculated inside the Context class. As concerning the application of refactoring to Strategy, we provide its specification as a series of primitive refactoring steps. A basic feature of the proposed transformation involves the Total Replacement of Conditional Logic, under certain conditions, that eliminates a conditional statement and replaces it with a polymorphic method invocation on a reference of the abstract Strategy class.

We have implemented and integrated the proposed method in the JDeodorant Eclipse plug-in for refactoring automation support in Java projects. On the basis of this implementation, we have evaluated the proposed identification algorithm in terms of quality of suggested refactorings and runtime efficiency. The evaluation involved execution of the identification algorithm on a set of Java code projects and rating of the refactoring suggestions, in terms of their relevance, by two expert software engineers. Quality assessment results support the effectiveness of our algorithm, since almost half of suggested refactorings were meaningful to the evaluators. Moreover, runtime efficiency of the algorithm is rather satisfactory, as the algorithm processing time on the benchmark Java projects did not exceed 30 s and 2.5 min for code bases of medium and large size respectively.

Our future work will focus on improvements to the refactoring identification method. Quality assessment of this work revealed that discovery of refactoring opportunities to Strategy is a challenging task, even for a software engineer, due to the semantics of the Strategy design pattern that are difficult to be mapped to source code attributes. Towards this direction, we plan to create an extensive data-set of refactoring candidates from various code bases and apply statistical and machine learning methods on attributes of their respective code fragments. Our goal is to automatically infer additional rules for identification of refactorings to Strategy. As part of our future work, we also intend to broaden the scope of applicability of Total Replacement of Conditional Logic for further decreasing the conditional complexity of refactored code fragments.

Acknowledgments

The authors thank Dr. N. Diamantidis and Dr. D. Vassis for their contribution as evaluators in the quality assessment of the proposed method. The authors would also like to thank Dr. Tsantalis, Prof. Chatzigeorgiou, as well as the rest of the JDeodorant development team members for providing us access to the project’s source code. Finally, we would like to thank the anonymous reviewers for their useful comments that improved the quality of this work.

References