The Journal of Systems and Software 83 (2010) 391–404

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Identification of refactoring opportunities introducing polymorphism

Nikolaos Tsantalis, Alexander Chatzigeorgiou *

Department of Applied Informatics, University of Macedonia, 54006 Thessaloniki, Greece

ARTICLE INFO

Article history:
Received 14 November 2008
Received in revised form 3 September 2009
Accepted 3 September 2009
Available online 11 September 2009

Keywords:
Refactoring
Polymorphism
State/Strategy design pattern
Object-oriented design

ABSTRACT

Polymorphism is one of the most important features offered by object-oriented programming languages, since it allows to extend/modify the behavior of a class without altering its source code, in accordance to the *Open/Closed Principle*. However, there is a lack of methods and tools for the identification of places in the code of an existing system that could benefit from the employment of polymorphism. In this paper we propose a technique that extracts refactoring suggestions introducing polymorphism. The approach ensures the behavior preservation of the code and the applicability of the refactoring suggestions based on the examination of a set of preconditions.

1. Introduction

Polymorphism has been widely recognized as one of the most important features of object-oriented programming languages. As polymorphism we refer to subtype polymorphism which according to Day et al. (1995) allows code written in terms of some type *T* to actually work for all subtypes of *T*. The main advantage of polymorphism is that it allows client classes to depend on abstractions (Gamma et al., 1995; Martin, 2003). An abstraction (abstract class or interface) can be extended by adding new subclasses that conform to its interface (i.e. override its abstract methods). However, the client classes that depend on abstractions do not have to change in order to take advantage of the behavior defined in the new subclasses.

Despite the sedulous teaching of polymorphism in object-oriented programming courses and its detailed presentation and discussion in books appealing to professionals, *state-checking* is often employed as an alternative approach to polymorphism in order to simulate *late binding* and *dynamic dispatch*. State-checking manifests itself as conditional statements that select an execution path either by comparing the value of a variable representing the current state of an object with a set of named constants, or by retrieving the actual subclass type of a reference through *RunTime Type Identification* (RTTI) mechanisms. The aforementioned symptoms usually result from either poor quality of the initial design or software aging (Parnas, 1994) caused by requirement changes that were not anticipated in the original design. State-checking introduces additional complexity due to conditional statements consisting of many cases and code duplication due to conditional statements scattered in many different places of the system that perform state-checking on the same cases for different purposes (Fowler et al., 1999). As a result, the maintenance of multiple state-checking code fragments operating on common states may require significant effort and introduce errors.

Although the employment of polymorphism in object-oriented systems is considered as an important design quality indicator, there is a lack of tools that either identify state-checking cases in an existing system or eliminate them by applying the appropriate refactorings on source code. To this end, we propose a technique for the identification and elimination of state-checking problems in Java projects that has been implemented as an Eclipse plug-in. An advantage of the proposed approach over metric-based approaches is the fact that all identified problems are actual cases of state-checking rather than ordinary conditional statements. Moreover, the examination of a set of preconditions ensures that the refactoring suggestions are both applicable and behavior-preserving.

The approach can be considered as semi-automatic, since after the extraction of the refactoring suggestions the designer is responsible for deciding whether a state-checking case should be eliminated or not based on conceptual and design quality criteria. Regarding the automation of the identification process, the main difference of the proposed technique with state-of-the-art Integrated Development Environments (IDEs) offering refactoring support (e.g. Eclipse 3.5, Netbeans 6.7, IntelliJ IDEA 8.1, Visual Studio 2008 along with Refactor! Pro 2.5) is that IDEs determine which refactorings are applicable based on the selection of a code fragment by the developer, while the proposed technique identifies

* Corresponding author. Tel.: +30 2310 891886; fax: +30 2310 891791.
E-mail addresses: nikos@java.uom.gr (N. Tsantalis), achat@uom.gr (A. Chatzigeorgiou).

0164-1212/$ - see front matter © 2009 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2009.09.017

refactoring opportunities without requiring any human intervention. Moreover, the proposed technique assists the designer to determine the effectiveness of the identified refactoring opportunities by grouping them according to their relevance and sorting them according to various quantitative characteristics.

The evaluation of the proposed technique consists of three parts. The first part presents the precision and recall of the approach by comparing the refactoring opportunities identified by an independent expert to the results of the proposed technique on various open-source projects. The second part of the evaluation investigates the impact of three quantitative factors on the decision of the independent expert to accept or reject the refactoring opportunities identified by the proposed technique. The last part refers to the scalability of the technique based on the computation time required for the extraction of refactoring suggestions on various open-source projects which differ in size characteristics.

The rest of the paper is organized as follows: Section 2 provides an overview of the related work. The proposed technique 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 Gamma et al. (1995), polymorphism simplifies the definitions of clients, decouples objects from each other, and lets them vary their relationships to each other at runtime. To this end, polymorphism plays a key role to the structure and behavior of most design patterns. In the literature of object-oriented software engineering, several empirical studies have investigated the impact of polymorphism and design patterns on external quality indicators related with software maintenance.

Brito e Abreu and Melo (1996) have shown that Polymorphism Factor (Brito e Abreu, 1995), which is defined as the number of methods that override inherited methods divided by the maximum number of possible distinct polymorphic situations, has a moderate to high negative correlation with defect and failure densities as well as with rework. In other words, the appropriate use of polymorphism in an object-oriented design should decrease the defect density and rework. However, they have also supported that very high values of Polymorphism Factor (above 10%) are expected to reduce these benefits, since the understanding and debugging of a highly polymorphical hierarchy is much harder than the procedural counterpart.

Prechelt et al. (2001) conducted a controlled experiment to compare design pattern solutions to simpler alternatives in terms of maintenance. The subjects of the experiment were professional software engineers that were asked to perform a variety of maintenance tasks. The independent variables were the programs and change tasks, the program version (there were two different functional equivalent versions of each program, a pattern-based version and an alternative version with simpler solutions) and the amount of pattern knowledge of the participants. The dependent variables were the time taken for each maintenance task and the correctness (i.e. whether the solutions fulfilled the requirements of the task). In most of the cases the experimental results had shown positive effects from the use of design patterns, since maintenance time was reduced compared to the simpler alternative versions.

Ng et al. (2006) performed a controlled experiment on maintaining JHotDraw to study whether the introduction of additional patterns through program refactoring is beneficial regardless of the work experience of the maintainers. For this reason, they used two sets of subjects in their experiment, namely experienced and inexperienced maintainers. They compared two maintenance approaches where in the first approach the subjects performed the maintenance tasks directly on the original program, while in the second approach the subjects performed the maintenance tasks on a refactored version of the original program using additional design patterns to facilitate the required changes. The empirical results have shown that, to complete a maintenance task of perfective nature, the time spent even by the inexperienced maintainers on the refactored version was much shorter than that of the experienced subjects on the original version.

Ng et al. (2007) studied whether maintainers utilize deployed design patterns, and when they do, which tasks they more commonly perform. For this reason, they refined an anticipated change facilitated by the deployment of design patterns into three finer-grained maintenance tasks, namely adding new concrete participants, modifying the existing interfaces of a participant, and introducing a new client. They concluded that regardless of the type of tasks performed by maintainers when utilizing deployed design patterns for anticipated changes, the delivered code is significantly less faulty than the code developed without utilizing patterns.

Other empirical studies have shown that maintenance effort does not only depend on the design quality of a given program (as expressed by the employment of design principles or the existence of design patterns), but also on human factors such as the experience, skills and education of the software developers and maintainers.

Arisholm and Sjøberg (2004) performed a controlled experiment in order to investigate the effect of delegated versus centralized control style on the maintainability of object-oriented software. To this end, two categories of developers (namely experienced and inexperienced) performed several change tasks on two alternative designs that had a centralized and delegated control style, respectively. The results of the experiment have shown that the most experienced developers required less time to maintain the software with delegated control style than with centralized control style, while novice developers had serious problems in understanding the delegated control style and performed much better with the centralized control style. Consequently, they concluded that maintainability of object-oriented software depends, to a large extent, on the skill of the maintainers.

Du Bois (2006) performed a series of controlled experiments to investigate whether the application of two reengineering patterns (Demeyer et al., 2003), namely *Refactor to Understand* and *Split Up God Class*, can improve program comprehension. The experiment involving the decomposition of god classes verified that the particular education of the subject performing the comprehension task affects the way in which a god class is decomposed.

Wendorff (2001) reported on a large commercial project where the uncontrolled use of patterns has contributed to severe maintenance problems. The reasons causing the maintenance problems were that some pattern instances were misused by software developers who had not understood the rationale behind their employment, many software developers overestimated the future volatility of requirements and opted for patterns to build flexibility at the cost of an undesirable increase of complexity, the change of requirements over the lifetime of the project led some pattern instances to become obsolete, and finally some pattern instances were embellished with additional features which were not actually needed. Consequently, the inappropriate application of patterns may have a negative effect on flexibility and maintainability of object-oriented software.

Concerning performance, it is widely believed that the replacement of conditional logic by a polymorphic method call deteriorates performance due to the introduction of an additional indirection through the *virtual function table*. Demeyer (2005) investigated the performance trade-off that is involved when introducing virtual functions by comparing the execution time of four

C++ benchmark programs which contain large conditionals against refactored versions where the conditionals were replaced by virtual function calls. The results of the experiment have shown that the optimized code which was generated by three C++ compilers for the refactored versions performed equally or even better compared to the conditional counterparts.

The catalogue of refactorings by Fowler et al. (1999) refers to state-checking as the *switch statements* bad smell. They argued that the main problem of this smell is code duplication, since the same switch statement is usually scattered in many different places of the code. In such a case, the adaptive maintenance of the code is rather difficult, since the addition of a new clause requires the identification and modification of all these multiple switch statements. The object-oriented paradigm offers the polymorphism mechanism as an elegant way to solve this problem. Fowler et al. proposed two refactorings that eliminate the state-checking code and introduce a new inheritance hierarchy, namely *Replace Type Code with Subclasses* and *Replace Type Code with State/Strategy*. The difference between the two refactorings is that in the first one the inheritance hierarchy is constructed by creating subclasses of the class that originally contained the state-checking code, while in the second one a new *State/Strategy* inheritance hierarchy is created and the class that originally contained the state-checking code becomes the *Context* class in the *State/Strategy* design pattern. It is important to mention that the *Replace Type Code with Subclasses* refactoring is not applicable when the value of the state changes at runtime, since the class type of an object cannot be changed after its creation. Fowler et al. also proposed the *Replace Conditional with Polymorphism* refactoring that eliminates the state-checking code in the case where the inheritance hierarchy already exists.

Demeyer et al. (2003) proposed *reengineering patterns* as a way to codify and record knowledge about modifying legacy software. Reengineering patterns emphasize on the process of moving from an existing legacy solution to a new refactored solution. Their difference with refactorings is that they also include a process for the detection of the symptoms and a discussion of the impact of changes that the refactored solution may introduce. Within the context of state-checking elimination Demeyer et al. proposed several reengineering patterns which are closely related to specific refactorings. For example, the reengineering patterns *Transform Self Type Checks* and *Transform Client Type Checks* are related to *Replace Type Code with Subclasses* and *Replace Conditional with Polymorphism* refactorings, respectively. Moreover, the reengineering patterns *Factor Out State* and *Factor Out Strategy* are related to *Replace Type Code with State/Strategy* refactoring. The detection process of the symptoms for the aforementioned reengineering patterns is given in the form of guidelines that should be followed by the maintainers in order to manually determine whether a specific conditional statement performs state-checking. Therefore, these guidelines do not constitute a concrete technique that could be automated by means of tools.

Kerievsky (2004) proposed a wider set of refactorings as solutions to the design problem of *conditional complexity*. The selection of the appropriate refactoring solution depends on the purpose of the conditional logic behind a complex conditional statement. For example, if conditional logic controls the state transitions of an object, then the *Replace State-Altering Conditionals with State* refactoring should be applied. In the case where conditional logic controls which of several variants of a calculation will be executed, then the *Replace Conditional Logic with Strategy* refactoring should be applied. Kerievsky also introduced two novel refactorings that eliminate conditional structures by introducing polymorphism. The first is *Replace Conditional Dispatcher with Command* that breaks down a conditional structure into a collection of Command (Gamma et al., 1995) objects and replaces conditional logic with code to fetch and execute the Command objects. The second is *Move Accumulation to Visitor* that introduces a Visitor (Gamma et al., 1995) in order to remove a conditional structure that is used to obtain data from instances of classes having different interfaces. Although, the author provides a detailed description on the steps required to apply the proposed refactorings (known as mechanics) along with examples from real-world software, the way to identify cases in the code that could benefit from these refactorings is left up to the designer.

Van Emden and Moonen (2002) proposed an approach for the automatic detection and visualization of *instanceof* and *typecast* code smells. The *instanceof* code smell appears as a sequence of conditional statements that test an object for its type, while the *typecast* code smell appears when an object is explicitly converted from one class type into another. An interesting part of their approach is the visualization of the detected code smells in the form of a graph, where the code smells are presented as additional nodes connected to the code entities that they belong to. In this way it is possible to discern which parts of the system have the largest number of code smells and would benefit the most from refactoring.

Trifu and Reupke (2007) proposed an approach that is based on the idea of combining correlated indicators in order to diagnose certain design flaws, in analogy with the medical world where a disease is diagnosed based on the presence of a specific constellation of symptoms. They distinguish three kinds of indicators, namely aggregating indicators (single metrics or logical expressions combining metrics), structural indicators (patterns in the structure of the code), and semantic indicators (the names of certain program elements, such as variables). Within the context of state-checking they have specified a design flaw named *explicit state checks*. The indicators used for the diagnosis of the specific design flaw are: (a) methods that contain "switch" or "if-else-if" conditional structures, and (b) checks should be performed on an attribute/property/parameter that semantically indicates the state of the current object instance (i.e. a variable that contains the string "state" in its name). The evaluation of the *explicit state checks* design flaws that were diagnosed in three open-source projects has shown that the employed indicators exhibit low precision when they are triggered individually or even simultaneously.

O'Keeffe and Ó Cinnéide (2008) proposed a search-based approach for improving the design of object-oriented programs without altering their behavior. To this end, they formulated the task of design improvement as a search problem in the space of alternative designs. The quality evaluation functions used to rank the alternative designs were based on metrics from the QMOOD hierarchical design quality model. The refactorings used by the search techniques to move through the space of alternative designs were inheritance-related (Push Down Field/Method, Pull Up Field/Method, Extract/Collapse Hierarchy, Replace Inheritance with Delegation, Replace Delegation with Inheritance and many others). Their approach has been validated by two case studies, in which the results of the employed search techniques (Hill Climbing and Simulated Annealing) and evaluation functions have been compared. This work is not directly associated with the elimination of conditional complexity or the introduction of new inheritance hierarchies and polymorphism as a remedy to conditional complexity. However, some of the refactorings used to move through the space of alternative designs may affect the degree of abstraction and polymorphism in a given system. A disadvantage of search-based approaches is that their results rely heavily on the parameterization of the employed search techniques (O'Keeffe and Ó Cinnéide, 2007).

Ó Cinnéide (Ó Cinnéide and Nixon, 1999; Ó Cinnéide, 2000) proposed a method for the automatic introduction of design patterns in terms of refactoring transformations. Based on the observation that design patterns can be decomposed into sequences of

minipatterns (a minipattern is a design motif that occurs frequently but is a lower-level construct compared to a conventional design pattern), he proposed a set of six reusable *minitransformations* that can define most of the design pattern transformations if composed properly. A minitransformation consists of a precondition, an algorithmic description of the transformation, and a postcondition that ensure behavior preservation. Ó Cinnéide also proposed the concept of *precursor* as a starting point for a design pattern transformation. A precursor is a design structure that serves as an indicator of the need for applying a specific design pattern in future maintenance stages. However, the description of most precursors is by nature quite vague (including the precursor of State pattern) and thus their identification cannot be automated.

3. Identification of refactoring opportunities that introduce polymorphism

Let us consider that a state-checking code fragment exists inside the body of method m belonging to class C. We define the following sets that will be used for the description of the proposed technique:

IV_C: the set of Instance Variables (non-static fields) of class C;
M_C: the set of non-static Methods of class C;
P_m: the set of Parameters of method m;
LV_m: the set of Local Variables declared inside the body of method m and before the state-checking code fragment;
NC: the set of Named Constants of all system classes (static final fields of `int`, `short`, `char`, or `byte` type and `enum` constant declarations).

3.1. Identification of refactoring opportunities that introduce the State/Strategy pattern

A code fragment that performs state-checking based on named constants can be either a `switch` statement or an `if/else if` statement (each `if` statement should be the `else` clause of the previous `if` statement, except for the first one).

The set of candidate State Variables SV (variables that can possibly hold a value representing the current state) is the subset of variables from the union of IV_C, P_m and LV_m sets having `int`, `short`, `char`, `byte`, or `enum` type:

$$SV = \{x \in \{IV_C \cup P_m \cup LV_m\} : type_x = primitive \vee type_x = enum\}$$

In the case where the state-checking code fragment under study is a `switch` statement s consisting of n cases the following conditions should be satisfied:

1. The expression of s should be a variable v belonging to SV or an invocation of the getter method of a variable v belonging to SV provided that v is an instance variable.
2. For each `switch case` c of s, the expression of c should be a named constant belonging to NC.
3. n should be greater than one, or equal to one provided that a `default case` exists.

In the case where the state-checking code fragment under study is an `if/else if` statement consisting of n `if` statements the following conditions should be satisfied:

1. The expression of each `if` statement should be (or should contain a conditional sub-expression that is) an infix expression with equality operator. Moreover, one of the operands should be a variable v belonging to SV or an invocation of the getter method of a variable v belonging to SV provided that v is an instance variable, while the other operand should be a named constant belonging to NC.
2. Variable v should be common in all infix expressions of the n `if` statements.
3. n should be greater than one, or equal to one provided that a final `else` clause exists.

If all conditions are satisfied the following information is extracted:

(a) the state variable v
(b) the set of Identified Named Constants (INC) that participate in the specific state-checking code fragment along with the code corresponding to each named constant

Identification of additional named constants related with a state-checking code fragment. It is quite possible that the set of identified named constants INC for a specific state-checking code fragment does not contain all the named constants that the state variable v is actually related with. The identification of all relevant states (represented by named constants) is very important in order to create a State inheritance hierarchy that can be also utilized by other state-checking code fragments which may operate on different but relevant named constants. Otherwise, it could be possible to generate multiple inheritance hierarchies that constitute different concrete implementations of essentially the same state abstraction, causing serious design flaws.

The proposed technique follows two complementary approaches in order to identify the set of Additional Named Constants (ANC) which are conceptually related with the state variable v but do not participate in the specific state-checking code fragment. The set of additional named constants ANC is also added to the information that is extracted for each state-checking code fragment along with the code corresponding to the `default case` or final `else` clause.

1. Approach based on the state variable v
   a. If variable v is an instance variable of class C, then all methods of class C are examined for assignments where the left hand side is variable v.
   b. If variable v is a local variable or parameter of method m, then only method m is examined for assignments where the left hand side is variable v.

If the right hand side of these assignments is a named constant belonging to NC set but not to INC set (NC/INC), then the named constant is added to the ANC set.

2. Approach based on the sets of identified named constants INC

The concept behind this approach is that if two state-checking code fragments operate on at least one common named constant (the intersection of their INC sets is not empty), then their state variables are conceptually related with all the named constants belonging to the union of their INC sets. Thus, a single State inheritance hierarchy should be created for both state-checking code fragments having as many concrete state subclasses as the named constants belonging to the union of their INC sets.

To this end, an algorithm (Fig. 1) is proposed that identifies the maximum number of conceptually relevant named constants by searching for common named constants among the INC sets of all state-checking code fragments. Let us consider that INC_i is the set of identified named constants for state-checking code fragment i. The INC sets are sorted in a list $INCList$ according to their cardinality in descending order. The examination of the INC sets in descending

```
while INCList not empty {
    namedConstants = INC0
    indexSet = {0}
    do {
        previousCardinality = |namedConstants|
        for(i=1; i<size of INCList; i++) {
            if(namedConstants ∩ INCi ≠ Ø) {
                namedConstants = namedConstants ∪ INCi
                indexSet = indexSet ∪ {i}
            }
        }
    } while(previousCardinality < |namedConstants|)

    for each index j in indexSet {
        ANCj = namedConstants \ INCj
        remove INCj from INCList
    }
}
```

Fig. 1. Algorithm for the identification of relevant named constants.

order increases the probability of identifying a larger number of conceptually relevant named constants (in a single iteration) compared to a random order. A set of named constants *namedConstants* is used to temporarily store the conceptually relevant named constants, while a set of indexes *indexSet* is used to temporarily store the indexes of *INC* sets which have common elements.

The proposed approach is unable to identify named constants which conceptually belong to a group of relevant states but do not participate in any conditional structure performing state-checking (i.e. they do not belong to the union of all *INC* sets) or are not assigned to any variable holding the current state (state variable *v*). Usually, such named constants express possible states or types that will become active in a future software release or are already active states whose functionality is covered by the default case or final else clause of the related state-checking code fragments (and thus they do not participate directly in state-checking). Obviously, such cases of named constants require human intervention to be discovered.

3.2. Identification of refactoring opportunities that replace RTTI with polymorphism

A code fragment that performs RunTime Type Identification can be an if/else if statement (each if statement should be the else clause of the previous if statement, except for the first one) consisting of *n* if statements.

The set of candidate Superclass Type Variables (*STV*) is the subset of variables from the union of IV_C, P_m and LV_m sets having the type of a system class which is inherited by other classes of the system.

A valid case of RunTime Type Identification should satisfy the following conditions:

1. The expression of each `if` statement should be (or should contain a conditional sub-expression that is) either:
   a. An `instanceof` expression where the left operand is a variable *v* belonging to *STV*, while the right operand is a class type that inherits the superclass type corresponding to variable *v*.
   b. An infix expression with equality operator where one of the operands is the invocation of method `getClass()` over a variable *v* belonging to *STV* (*v.getClass()*), while the other operand is a type literal (*Type.class*) which is a subclass of the superclass type corresponding to variable *v*.
2. Variable *v* should be common in all expressions of the *n* `if` statements.
3. *n* should be greater than one, or equal to one provided that a final `else` clause exists.

If all conditions are satisfied the following information is extracted:

(a) the variable *v* (reference to superclass type);
(b) the set of Identified Subclass Types (*IST*) that participate in the specific `if/else if` statement along with the code corresponding to each subclass type;
(c) the inheritance hierarchy tree structure corresponding to the identified class types;
(d) the code corresponding to the final `else` clause.

3.3. Handling of compound conditional expressions

In the case where the expression of an `if` statement consists of sub-expressions combined with conditional AND operators (&&), the proposed technique identifies which sub-expression actually performs state-checking and constructs a new conditional expression from the other sub-expressions. The remaining expression will replace the original expression, when the code of the then clause is going to be moved to the appropriate subclass.

To this end, the original expression is represented as a binary expression tree where all the parent nodes are conditional AND operators and the leaf nodes are the actual sub-expressions. Next, all the leaf nodes are examined (according to the first condition of the aforementioned rules) to identify a sub-expression that performs state-checking. If more than one sub-expressions are found to perform state-checking, then the technique selects the sub-expression whose state variable exists in the state-checking expressions of all the other if statements. The remaining expression is constructed by removing the identified leaf node from the binary tree and by replacing its parent node with its sibling node.

In the code example of Fig. 2 the expression of the first if statement consists of three sub-expressions combined with conditional AND operators.

The binary expression tree of the compound expression of Fig. 2 is shown in Fig. 3a. After examining the leaf nodes of the binary expression tree, two out of the three sub-expressions can be considered as valid state-checking expressions. The first one "`dragMode== DRAG_MOVE`" is an equality comparison of variable `dragMode` with the named constant `DRAG_MOVE`, while the third one "`selected instanceof Node`" is an `instanceof` expression used for the purpose of RunTime Type Identification. From the two valid state-checking expressions the first sub-expression is selected "`dragMode == DRAG_MOVE`", since variable `dragMode` appears in the state-checking expression of the second `if` statement "`dragMode == DRAG_LASSO`". Consequently, the remaining expression is constructed by removing leaf node "`dragMode == DRAG_MOVE`" from the tree and by replacing its parent node "&&" with its sibling node "x > 0", as shown in Fig. 3b. Finally, the remaining expression will be the compound conditional expression "`x > 0 && selected instanceof Node`".

In the case where the expression of an `if` statement consists of sub-expressions combined with conditional OR operators (||) and all sub-expressions perform state-checking on the same variable holding the current state, the functionality of the then clause is common for all the named constants that participate in the state-checks. In order to avoid the duplication of the then clause in all the concrete subclasses corresponding to the named constants that participate in the state-checks, an intermediate class is introduced in the created State/Strategy inheritance hierarchy

```
if (dragMode == DRAG_MOVE && x > 0 && selected instanceof Node) {
   ...
}
else if (dragMode == DRAG_LASSO) {
   ...
}
```

Fig. 2. Example of compound conditional expression with AND operators.

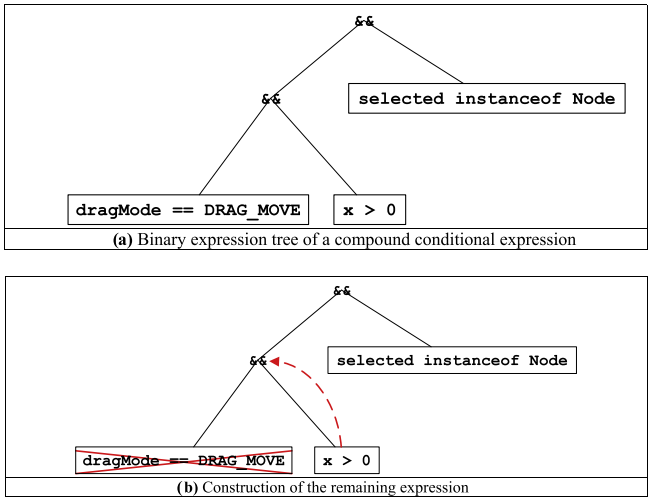


Fig. 3. Handling of compound conditional expressions with AND operators.

between the abstract class playing the role of State/Strategy and the corresponding concrete subclasses. The intermediate class overrides the polymorphic method of the State/Strategy superclass with the common functionality of the then clause, while the concrete subclasses simply inherit the intermediate class without overriding the polymorphic method. The name of the intermediate class is extracted from the corresponding named constants employing a Longest Common Subsequence (LCS) algorithm. In the code example of Fig. 4a the expressions of both if statements consist of two sub-expressions combined with a conditional OR operator. Fig. 4b shows the resulting State inheritance hierarchy which has two intermediate classes (`Destroy`, `Conquer`) containing the common functionality of the corresponding `if` statements.

3.4. Preconditions

According to Opdyke (1992), 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. The proposed technique examines all valid cases of state-checking and disqualifies those cases that do not satisfy the following set of preconditions:

1. The state-checking code fragment should not contain assignments of local variables belonging to the LV_m set (variables of method m declared before the state-checking code fragment) or parameters belonging to P_m. In the case where such a variable is passed as parameter to the polymorphic method, its value would not change after the execution of the polymorphic method, since parameters are passed by-value in Java. This could possibly affect the behavior of the code that followed the state-checking code fragment after the application of the refactoring. An exception to this rule is the case where the state-checking code fragment contains assignments of a single variable belonging to the union of LV_m and P_m sets that is returned inside or after the state-checking code fragment.
2. The state-checking code fragment cannot be extracted if it resides inside the body of an iteration statement (for, while, do-while) and contains unstructured control flow statements (break, continue). The reason is that if a branch of the state-checking code is extracted as a separate method, then the contained unstructured control flow statement will not be surrounded by any iteration thus leading to a compilation error.
3. The state-checking code fragment should not contain any super method invocations, since the move of the code containing such invocations to the corresponding subclass would lead to compilation problems.
4. The names of the created classes belonging to the State/Strategy inheritance hierarchy should not be the same with the names of already existing classes in the same package or even in different packages. In the first case, it is not possible to have two classes with the same name in the same package. In the second case, the classes of the system that do not explicitly import the already existing class will present errors due to the ambiguous type of the conflicting class name. This issue can be resolved by renaming the classes of the State/Strategy hierarchy that lead to conflict.

3.5. Assessing the effect of the identified refactoring opportunities on design quality

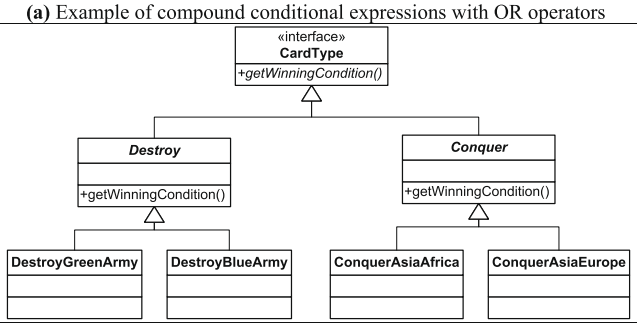
It is reasonable to expect that several cases of state-checking may exist in a software system, especially when it consists of many classes. As a result, it is really important to be able to distinguish the refactoring suggestions which have greater effect on the design of the system. To this end, the proposed technique provides a sorting mechanism for the identified refactoring opportunities.

First of all, the refactoring suggestions are grouped according to their relevance. We can consider that there are two kinds of grouping based on the nature of state-checking. The first one involves the cases that perform state-checking based on named constants and the grouping criterion is the named constants found in common. The second one involves the cases that perform RunTime Type Identification based on subclass types and the grouping criterion is the common inheritance hierarchy that the subclass types may belong to. The philosophy behind this kind of grouping is that the refactoring suggestions belonging to the same group will eventually utilize the same inheritance hierarchy (that either is going to be created or already exists). The groups of refactoring suggestions are sorted according to their size. The higher the number of the refactoring suggestions belonging to a group, the greater the impact of the specific group on design quality, since the degree of polymorphism (i.e. the number of polymorphic methods added to a single inheritance hierarchy) introduced to the system will be higher. In the case where two groups have the same size, they are sorted according to the average number of statements per branch (state) of the state-checking code fragments that they contain. Finally, the refactoring suggestions are also sorted within the group that they belong to according to the number of cases performing state-

```
public boolean getWinningCondition() {

    if(cardType == DESTROY_GREEN_ARMY || cardType == DESTROY_BLUE_ARMY) {
        ...
    }
    else
    if(cardType == CONQUER_ASIA_AFRICA || cardType == CONQUER_ASIA_EUROPE) {
        ...
    }
}
```

(a) Example of compound conditional expressions with OR operators



(b) Resulting State hierarchy with an intermediate inheritance level

Fig. 4. Handling of compound conditional expressions with OR operators.

checking at each class of the group (at first level) and the average number of statements per branch of each state-checking code fragment in the group (at second level).

It should be noted that the refactoring suggestions within a group are independent with each other, in the sense that the application of a refactoring does not affect the other suggestions. In other words, the application of all refactorings belonging to a group leads to the same code regardless of the order in which they are applied.

3.6. Limitations

As already mentioned in the introduction, the proposed approach is semi-automatic in the sense that the decision of whether a refactoring suggestion should be accepted or not is left up to the designer of the examined program. Consequently, a limitation of the approach is that the effectiveness of the refactoring identification technique relies on the expertise of the designer. In general, the designer should consider three factors in order to derive a decision:

1. The number of conditional structures that perform state-checking on specific named constants throughout the code of the program. Obviously, the larger the number of these conditional structures, the more severe the design problem is. The proposed technique assists the designer by grouping the suggestions according to the relevance of the named constants participating in the conditional structures and sorting the resulting groups of suggestions according to their size.
2. The possibility of adding a new state to an already existing group of states due to a future change in requirements. Obviously, if the designer is absolutely sure that the addition of new states in the future is not possible, there is no need to replace the existing solution with one that introduces polymorphism. This factor can be determined based on the requirements of the program under examination.
3. The trade-off between the flexibility that may be introduced by the employment of State pattern and the complexity that may be caused by the number of concrete State subclasses being added. Obviously, the smaller the number of added State subclasses and the larger the size of code being moved to them, the more beneficial the refactoring is.

If we assume that the maintainer of the program under examination has a sufficient knowledge of its design structure and requirements, and exploits effectively the assistance provided by the proposed technique, then the time required for the examination of the refactoring suggestions is significantly reduced.

The proposed technique does not cover all refactoring opportunities that introduce polymorphism. Kerievsky (2004) proposed a catalogue of refactorings that replace simpler solutions to specific design problems with solutions introducing design patterns and consequently polymorphism. However, each design pattern requires a completely different approach for the identification of cases where it can be introduced. As a result, it is impossible to build a common technique that covers all refactoring opportunities introducing design patterns. It should be noted, though, that state-checking conditional logic has been widely recognized as an important design flaw in object-oriented software (Fowler et al., 1999; Demeyer et al., 2003; Kerievsky, 2004), since conditional logic is considered as one of the most common sources of complexity.

It should be noted that conditional structures performing RTTI can be refactored (in order to eliminate the conditionals by employing polymorphism) either by applying "*Replace Conditional with Polymorphism*" (Fowler et al., 1999) or by applying "Move Accumulation to Visitor" (Kerievsky, 2004). In the first case the code contained in the conditional branches is moved to the subclasses of a common hierarchy, while in the second case the code is accumulated in a single Visitor class and double-dispatch is employed. In this particular context, the second approach (i.e. the Visitor design pattern) is required when the designer wants to avoid "polluting" the subclasses with additional operations (Gamma et al., 1995). For both cases the proposed technique would identify the need for introducing polymorphism; however, the selection between the two aforementioned solutions depends on factors that cannot be automatically determined. The tool currently automates only the application of the first solution.

Another issue deals with the existence of additional refactoring opportunities on the conditional structures performing state-checking. Usually, such conflicting refactorings involve the extraction of the code residing in conditional branches as new separate methods (this activity constitutes part of the *Refactor to Understand* reengineering pattern introduced by Demeyer et al., 2003), the extraction of code that is duplicated between

a conditional structure and other parts of the program as a single method, and even the move of a conditional structure (or the method containing it) to another class due to Feature Envy (Fowler et al., 1999) design problem. The aforementioned refactorings can be considered as low-level transformations compared to more sophisticated refactorings introducing polymorphism and design patterns, and thus should be applied first. It should be noted that if the application of low-level refactorings does not affect the branching structure of conditional statements, then the refactoring opportunities introducing polymorphism will be preserved.

Finally, the conditional operator ?: (also known as the *ternary operator*) has not been considered by the proposed technique. The ternary operator is used in the form of conditional expression

$$Condition\ ?\ value\_if\_true : value\_if\_false$$

where *value_if_true* is returned if *condition* is true, and *value_if_false* otherwise. This conditional expression is most commonly used as the right hand side of assignment statements. As a result, its usage potential is limited compared to if and switch statements. Moreover, it is not suitable for the representation of multiple cases or execution branches, since it results in overcomplicated code which is difficult to read and understand. For these reasons, the conditional expression with ternary operator is rarely used compared to if and switch statements.

3.7. Demonstration of the technique on an open-source project

In this section we present the refactoring suggestions extracted by the proposed technique for an open-source project and demonstrate the application of two representative refactorings on source code. The examined open-source project is named Violet (version 0.16) and is a UML editor intended for students, teachers, and authors who need to produce simple UML diagrams quickly (Horstmann, 2005). The extracted suggestions will be presented separately for the two kinds of refactoring solutions which are supported by the proposed technique.

The suggestions corresponding to *Replace Type Code with State/Strategy* refactorings are summarized in Table 1.

The group of refactoring suggestions 1–3 (Table 1) is related to named constants representing different drag modes. The active drag mode affects the way that UML components are painted in the diagram, as well as the handling of various mouse events. We can consider that the State inheritance hierarchy created for this group of refactorings will be sufficiently utilized, since three polymorphic methods will be added (the number of polymorphic methods being added is equal to the size of the group) and a relatively large number of statements (as it is evident from the last column of Table 1) will be moved to the corresponding overriding methods in the concrete State subclasses when the refactorings of the group are applied. The application of *Replace Type Code with State/Strategy* refactoring for the third suggestion is demonstrated in Fig. 5.

As it can be observed from Fig. 5a, method mouseReleased in class GraphPanel contains an if/else if statement that performs state-checking. The state variable is instance variable dragMode having int type, while the set of identified named constants *INC* is {DRAG_RUBBERBAND, DRAG_MOVE} and the set of additional named constants *ANC* that results as described in Section 3.1 is {DRAG_NONE, DRAG_LASSO}. After the application of the refactoring, class GraphPanel plays the role of Context in the *State/Strategy* pattern, as shown in Fig. 5b. The type of the state variable dragMode has been changed to the type of the abstract class DragMode playing the role of State/Strategy. The state-checking code fragment has been replaced with an invocation of the polymorphic method mouseReleased through state variable dragMode. Finally, each concrete State subclass (e.g. class DragMove that represents named constant DRAG_MOVE) overrides the polymorphic method mouseReleased by copying the statements of the corresponding conditional branches.

The suggestions corresponding to *Replace Conditional with Polymorphism* refactorings are summarized in Table 2.

The group of refactoring suggestions 1–4 (Table 2) is related to subclass types that belong to the inheritance hierarchy of interface Node. The classes belonging to the Node inheritance hierarchy represent elements that participate in UML diagrams. The conditional structures corresponding to this group of suggestions perform RunTime Type Identification based on the actual subclass type of the Node reference. The application of *Replace Conditional with Polymorphism* refactoring for the third suggestion is demonstrated in Fig. 6.

As it can be observed from Fig. 6a, method getPoints in class CallEdge contains an if/else if statement that performs RTTI. The reference to superclass type is local variable *n* whose type is Node. The set of identified subclass types *IST* is {CallNode, PointNode} and the conditional statement has also a final else clause (default implementation). The inheritance hierarchy tree structure corresponding to the identified subclass types is shown in Fig. 7. The abstract class RectangularNode has eleven more subclasses which have not been included in the Class Diagram of Fig. 7. After the application of the refactoring, the conditional code performing RTTI has been replaced with an invocation of the polymorphic method getPoints (declared in interface Node) through local variable *n*, as shown in Fig. 6b. Each subclass belonging to *IST* overrides the polymorphic method getPoints by copying the statements of the corresponding conditional branches, while class AbstractNode provides the default implementation by copying the statements of the final else clause.

Table 1
Replace Type Code with State/Strategy refactoring suggestions for Violet 0.16*.

Id	Class	Method	Named constants	Default case	Name of variable holding the state	Kind of variable holding the state	#Cases in a class utilizing the same hierarchy	#Cases in a system utilizing the same hierarchy	Average #statements per branch
1	GraphPanel	mouseDragged	DRAG_MOVE DRAG_LASSO	No	dragMode	Field	3	3	16
2	GraphPanel	paintComponent	DRAG_RUBBERBAND DRAG_LASSO	No	dragMode	Field	3	3	6.5
3	GraphPanel	mouseReleased	DRAG_RUBBERBAND DRAG_MOVE	No	dragMode	Field	3	3	3.5
4	MultiLineString	setLabelText	LEFT CENTER RIGHT	No	justification	Field	1	1	1

* All class names are preceded by package "com.horstmann.violet.framework."

Fig. 5. Application of *Replace Type Code with State/Strategy* refactoring.

Table 2
Replace Conditional with Polymorphism refactoring suggestions for Violet 0.16*.

Id	Class	Method	Subclass types	Default case	Name of superclass type reference	Kind of superclass type reference	#cases in a class utilizing the same hierarchy	#cases in a system utilizing the same hierarchy	Average #statements per branch
1	SequenceDiagramGraph	layout	CallNode ImplicitParameterNode	No	n	Local variable	2	4	1
2	SequenceDiagramGraph	removeEdge	CallNode	Yes	end	Getter invocation	2	4	1
3	CallEdge	getPoints	CallNode PointNode	Yes	n	Local variable	1	4	5.3
4	PackageNode	addNode	ClassNode InterfaceNode PackageNode	Yes	n	Parameter	1	4	1.5

* All class names are preceded by package "com.horstmann.violet."

4. JDeodorant Eclipse plug-in

The proposed technique has been implemented as an Eclipse plug-in (Tsantalis et al., 2008) that not only identifies state-checking problems but also allows the user to apply the refactorings that resolve them on Java source code. Moreover, the tool groups the refactoring suggestions according to their relevance and sorts them within their groups according to the quantitative character-

```
public class CallEdge extends SegmentedLineEdge {
   public ArrayList getPoints() {
      ArrayList a = new ArrayList();
      Node n = getEnd();
      Rectangle2D start = getStart().getBounds();
      Rectangle2D end = n.getBounds();

      if (n instanceof CallNode &&
            ((CallNode)n).getImplicitParameter() ==
            ((CallNode)getStart()).
               getImplicitParameter()) {

         Point2D p = new Point2D.Double(
            start.getMaxX(),
            end.getY() - CallNode.CALL_YGAP / 2);
         Point2D q = new Point2D.Double(
            end.getMaxX(), end.getY());
         Point2D s = new Point2D.Double(
            q.getX() + end.getWidth(), q.getY());
         Point2D r = new Point2D.Double(
            s.getX(), p.getY());
         a.add(p);
         a.add(r);
         a.add(s);
         a.add(q);
      }
      else if (n instanceof PointNode) {
         a.add(new Point2D.Double(
            start.getMaxX(), start.getY()));
         a.add(new Point2D.Double(
            end.getX(), start.getY()));
      }
      else {
         Direction d = new Direction(
            start.getX() - end.getX(), 0);
         Point2D endPoint =
            getEnd().getConnectionPoint(d);
         if (start.getCenterX() < endPoint.getX())
            a.add(new Point2D.Double(
               start.getMaxX(), endPoint.getY()));
         else
            a.add(new Point2D.Double(
               start.getX(), endPoint.getY()));
         a.add(endPoint);
      }
      return a;
   }
}
```

```
public class CallEdge extends SegmentedLineEdge {
   public ArrayList getPoints() {
      ArrayList a = new ArrayList();
      Node n = getEnd();
      Rectangle2D start = getStart().getBounds();
      Rectangle2D end = n.getBounds();

      n.getPoints(a, start, end, this);
      return a;
   }
}
```

Original class after the replacement of conditional code with polymorphic method invocation

```
public abstract class AbstractNode implements Node {
   public void getPoints(ArrayList a,
      Rectangle2D start, Rectangle2D end,
      CallEdge callEdge) {
      Direction d = new Direction(
         start.getX() - end.getX(), 0);
      Point2D endPoint =
         callEdge.getEnd().getConnectionPoint(d);
      if (start.getCenterX() < endPoint.getX())
         a.add(new Point2D.Double(
            start.getMaxX(), endPoint.getY()));
      else
         a.add(new Point2D.Double(
            start.getX(), endPoint.getY()));
      a.add(endPoint);
   }
}
```

Existing class **AbstractNode** providing the default implementation

```
public class CallNode extends RectangularNode {
   public void getPoints(ArrayList a,
      Rectangle2D start, Rectangle2D end,
      CallEdge callEdge) {
      if (getImplicitParameter() ==
         ((CallNode)callEdge.getStart())
            .getImplicitParameter()) {
          Point2D p = new Point2D.Double(
             start.getMaxX(),
             end.getY() - CallNode.CALL_YGAP / 2);
          Point2D q = new Point2D.Double(
             end.getMaxX(), end.getY());
          Point2D s = new Point2D.Double(
             q.getX() + end.getWidth(), q.getY());
          Point2D r = new Point2D.Double(
             s.getX(), p.getY());
          a.add(p);
          a.add(r);
          a.add(s);
          a.add(q);
       }
   }
}
```

Existing class **CallNode** providing the functionality of the first branch

(a) original code (b) refactored code

Fig. 6. Application of *Replace Conditional with Polymorphism* refactoring.

istics described in Section 3.5, assisting the user to determine an appropriate 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. Fig. 8 shows the way that the refactoring suggestions are presented to the user. The first column indicates the type of the extracted refactorings (*Replace Type Code with State/Strategy* or *Replace Conditional with Polymorphism*), while the second column indicates the method that contains the corresponding state-checking code fragment. By double-clicking on a row of the table the corresponding state-checking code fragment is highlighted in the Eclipse editor. The third and fourth columns indicate the number of relevant refactoring suggestions belonging to the same group at a system and class level, respectively. The final column shows the average number of statements per branch of the corresponding state-checking code fragment.

5. Evaluation

The proposed technique has been evaluated in three ways:

(a) To evaluate the precision and recall of proposed technique, we performed an experiment to compare the refactoring opportunities identified by an independent expert to the results of the technique on various open-source projects.
(b) We performed an experiment to investigate the correlation of three quantitative factors (which are used to sort the refactoring suggestions extracted by the technique) with

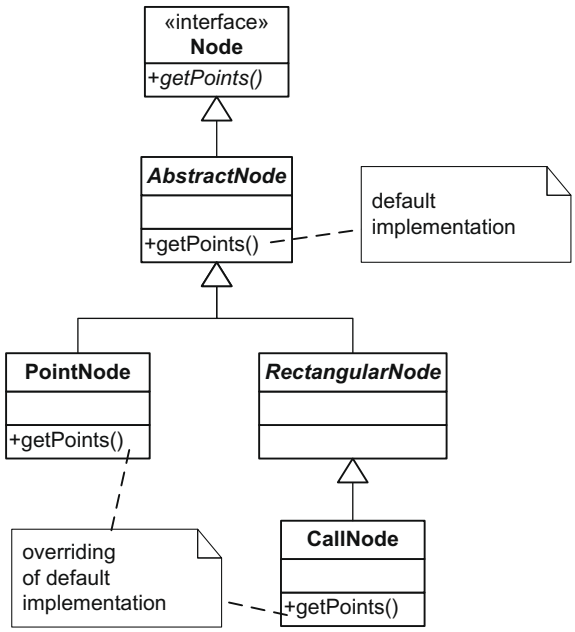


Fig. 7. `Node` inheritance hierarchy tree structure.

the decision of the independent expert to accept or reject the refactoring opportunities identified by the proposed technique.

(c) To evaluate the scalability of the proposed technique, we measured the computation time required for the application of the technique with regard to the size of various open-source projects.

5.1. Evaluation of precision and recall

To evaluate the performance of the proposed technique in terms of exactness and completeness, we performed an experimental study to compare the findings of an independent expert to the results of the proposed technique on various open-source projects.

The expert that participated in the experiment had significant experience in software design (he has been working for more than 12 years as a telecommunications software designer) and deep knowledge of object-oriented design principles and patterns. Moreover, he was unfamiliar with the proposed technique and was able to dedicate a significant amount of time on analyzing the projects under study. The motivation behind his participation in the experiment was the utilization of the results for his PhD research on aspect-oriented design.

The projects which have been selected for the experiment are Violet 0.16 which is a UML editor intended for educational purposes, Ice Hockey Manager 0.1.1 which is a hockey team management game, and Nutch 0.4 which is a web crawler. The reasons for selecting these specific projects were:

- Their source code is open and publicly available allowing the replication of the experiment.
- They have a relatively small size allowing the independent expert to adequately examine them.
- They are implemented in Java programming language enabling the analysis of their source code by the proposed technique.
- The selected software releases correspond to rather immature versions, thus offering potential refactoring opportunities.
- They originate from different application domains allowing, to some extent, the generalization of the conclusions.

Table 3
Size characteristics of the examined open-source projects.

Measure	Violet 0.16	IHM 0.1.1	Nutch 0.4
#Classes	75	86	295
#Methods with body	377	629	1389
Total #conditional structures	231	245	1495
Source lines of code*	6910	8662	23579

* Source lines of code (SLOC) have been measured using SLOCCount.

The size characteristics of the examined projects are shown in Table 3.

The exact question that has been asked to the independent expert was: "Which are the conditional structures that should be replaced with an instance of State design pattern, or employ an RTTI mechanism that should be replaced with a polymorphic call?" We considered as True Occurrences the refactoring opportunities reported by the independent expert. This set of True Occurrences is the baseline against which we compared the proposed technique when calculating precision and recall. The measures required for the classification of the results are defined as follows:

- True Positive: A refactoring opportunity identified by the independent expert, and also by the proposed technique.
- False Positive: A refactoring opportunity identified by the proposed technique, but not by the independent expert.
- False Negative: A refactoring opportunity identified by the independent expert, but not by the proposed technique.
- True Negative: A conditional structure which has not been considered to offer a refactoring opportunity by the independent expert and has not been suggested as a refactoring opportunity by the proposed technique.

The precision and recall for the examined projects are given in Table 4. We have observed that if we exclude from the suggestions of the proposed technique the conditional structures having an

Refactoring Type	Type Checking Method	System...	Class...	Averag...
Replace Conditional with...	com.horstmann.violet.SequenceDiagramGraph::public void layout(java.awt....	4	2	1.0
Replace Conditional with...	com.horstmann.violet.SequenceDiagramGraph::public void removeEdge(co...	4	2	1.0
Replace Conditional with...	com.horstmann.violet.CallEdge::public ArrayList#RAW getPoints()	4	1	5.3333...
Replace Conditional with...	com.horstmann.violet.PackageNode::public boolean addNode(com.horstm...	4	1	1.5
Replace Type Code with ...	com.horstmann.violet.framework.GraphPanel::public void mouseDragged(j...	3	3	16.0
Replace Type Code with ...	com.horstmann.violet.framework.GraphPanel::public void paintComponent...	3	3	6.5
Replace Type Code with ...	com.horstmann.violet.framework.GraphPanel::public void mouseReleased(j...	3	3	3.5
Replace Type Code with ...	com.horstmann.violet.framework.MultiLineString::private void setLabelText()	1	1	1.0

Fig. 8. Grouping and sorting of the refactoring suggestions

Table 4
Precision and recall for the examined open-source projects.

Project	Violet 0.16		IHM 0.1.1		Nutch 0.4	
True Occurrences (TO)	7		9		17	
True Positives (TP)	4		9	(7)	17	
False Positives (FP)	4	(0)	12	(0)	13	(1)
False Negatives (FN)	3		0	(2)	0	
True Negatives (TN)	220	(224)	224	(236)	1465	(1477)
Precision: TP/(TP + FP)	50%	(100%)	43%	(100%)	57%	(94%)
Recall: TP/(TP + FN)	57%		100%	(78%)	100%	
Accuracy: (TP + TN)/(TP + FP + FN + TN)	97%	(99%)	95%	(99%)	99%	(100%)

average number of statements per branch which is lower than two (i.e. conditional structures with a relatively small number of statements), the precision of the technique is significantly improved. The measures resulting from the application of the threshold (i.e. average number of statements per branch >= 2) are given inside parentheses wherever a change was observed.

The false negatives refer to conditional structures using this keyword in place of the variable holding the current state, whereas our identification approach requires the existence of an instance variable, local variable, or method parameter. A conditional code that compares this keyword with a set of named constants (having the type of the class corresponding to this keyword) cannot be considered as a case of state-checking, since the value of this reference cannot change after object creation and as a result the state of the object cannot be modified at runtime. The independent expert supported that such cases could be eliminated by introducing a subclass (of the class corresponding to this reference) for each named constant that participates in the conditional and overriding the method that contains the conditional in each created subclass (i.e. by applying the *Replace Type Code with Subclasses* refactoring).

The independent expert reported a few cases where the variable holding the current state was a field inherited from a superclass. The proposed technique failed to collect these cases, because it requires the fields holding the current state to belong in the same class where the corresponding state-checking code fragment exists. The reason for this requirement is to make feasible the change of the original type of the field to the type of the abstract class playing the role of State. However, the expert supported that the original solutions should not be replaced with an instance of State pattern, and as a result, they were not considered as false negatives.

5.2. Correlation of quantitative factors with expert judgment

The goal of this experiment is to assess the correlation of three factors with the decision of the independent expert to accept or reject the refactoring opportunities identified by the proposed technique. These factors are:

(a) The number of conditional structures performing state-checking on the same set of named constants or equivalently the number of polymorphic methods that can be added in the same inheritance hierarchy of states.
(b) The number of alternative states belonging to a set of named constants or equivalently the number of concrete State subclasses that will be created in an inheritance hierarchy of states.
(c) The average number of statements per branch in a conditional structure performing state-checking or equivalently the average number of statements that will be moved to the concrete State subclasses of an inheritance hierarchy.

The null and alternative hypotheses being tested are the following:

H_{0a}: The decision of accepting or rejecting a refactoring opportunity is not affected by factor a.
H_{1a}: The decision of accepting or rejecting a refactoring opportunity is affected by factor a.
H_{0b}: The decision of accepting or rejecting a refactoring opportunity is not affected by factor b.
H_{1b}: The decision of accepting or rejecting a refactoring opportunity is affected by factor b.
H_{0c}: The decision of accepting or rejecting a refactoring opportunity is not affected by factor c.
H_{1c}: The decision of accepting or rejecting a refactoring opportunity is affected by factor c.

The hypotheses will be tested by univariate logistic regression analyses, one for each factor. The dependent variable is a binary variable representing "agreement" or "disagreement" on the refactoring opportunities identified by the proposed technique, while the independent variable in each case is the corresponding factor. The values for the dependent variable were derived from the True Positives and False Positives of the experiment described in Section 5.1. A True Positive corresponds to an "agreement" of the independent expert with a refactoring opportunity identified by the proposed technique, while a False Positive corresponds to a "disagreement" of the independent expert with a refactoring opportunity identified by the proposed technique. The values for the independent variables (i.e. the three factors being examined in the experiment) were provided by the tool implementing the proposed technique. The data for the analysis have been drawn from project Nutch 0.4, since it provides the largest number of suggestions ($N = 30$ cases). The results from the analysis are shown in Table 5.

Since the p-value is less than the significance level (0.05) for all three factors, we can reject the null hypotheses and claim that the decision for accepting a refactoring opportunity is affected by all factors. In particular, considering the coefficients (B), the decision is affected positively by the number of polymorphic methods to be added to the same hierarchy of states (factor a) and the average number of statements that will be moved to the State subclasses (factor c), while it is affected negatively by the number of State subclasses that will be created (factor b). In other words, the independent expert tends to agree with the refactoring opportunities that sufficiently utilize a newly created inheritance hierarchy of states (by belonging to a relatively large group of opportunities that will utilize the same hierarchy of states), move a relatively large number of statements from the conditional branches to the corresponding State subclasses, and introduce hierarchies of states with a relatively small number of State subclasses. The proposed technique takes into account these quantitative factors when sorting the extracted refactoring suggestions in order to assist the designer in assessing their effect on design quality.

5.3. Threats to validity

Since the two experiments have different goals we list their major threats separately (Wohlin et al., 2000).

Table 5
Logistic regression results for project Nutch 0.4.

Factor	B (S.E.)	Wald X^2	df	Nagelkerke R^2	p
a	1.777 (0.648)	7.533	1	0.783	0.006
b	−0.716 (0.254)	7.930	1	0.657	0.005
c	2.624 (0.920)	8.124	1	0.712	0.004

Threats to internal validity: As threats to internal validity we consider those factors that may cause interferences regarding the relationships being investigated.

For the first experiment (Section 5.1), which is related with the evaluation of precision and recall of the technique, there is a possibility that the human expert has missed a number of refactoring opportunities while examining the code of the projects or misclassified a number of non-valid cases as refactoring opportunities. Obviously, these threats affect the reported precision and recall of the technique. The first threat is mitigated by the fact that the selected projects were relatively small in size and thus could be adequately examined by the independent expert. Moreover, the independent expert was motivated to perform a detailed and infallible analysis of the projects under study by the fact that the results would be utilized for his PhD research. The second threat is mitigated by the expertise of the evaluator and his past experience with design patterns in industrial software development.

For the second experiment (Section 5.2), which is related with the correlation of quantitative factors with expert judgment, there may have been omitted other important factors that affect the decision of the independent expert, such as the possibility of adding a new state to an already existing group of states due to a future change in requirements. Obviously, this threat could affect the accuracy of a multivariate prediction model which involves more than one independent variables as predictors at the same time, and for this reason, we performed univariate regression analysis for each factor separately. In any case, the investigated statistical relationships do not prove a causal relationship between the factors and the expert's decision.

Threats to external validity: Since the experiments have been performed employing a single expert as evaluator and a small number of projects, the study suffers from the usual threats to external validity. In other words, these factors limit the possibility to generalize our findings beyond the selected setting (projects and evaluator). For example, in the experiment regarding the correlation of quantitative factors with expert judgment, the logistic regression results for the other two projects that have been considered in the first experiment (Violet 0.16 and IHM 0.1.1) were not statistically significant.

5.4. Evaluation of scalability

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

(a) Parsing of the system under study using the ASTParser of Eclipse JDT.
(b) Examination of all conditional statements (switch, if/else if statements) in the given system in order to identify valid cases of state-checking. Moreover, the valid cases of state-checking are checked against the set of preconditions defined at Section 3.4.

Table 6 contains various size measures for four open-source projects, namely Nutch 1.0, FreeCol 0.8.3, JMol 11.6.21, and JFreeChart 1.0.13.

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

The CPU time required for the first step depends on the size of the system under examination in terms of lines of code, since all field and method declarations (including the statements inside the body of each method) are parsed and analyzed. The CPU time required for the second step primarily depends on the total number of conditional structures found in the system under examination. The proposed technique requires access to the Abstract Syntax Tree (AST) information both during the identification of refactoring opportunities and the application of refactorings which are eventually selected by the user. A limitation regarding scalability is that the AST information of large projects occupies a large amount of heap memory causing OutOfMemory exceptions. This issue can be resolved either by applying the proposed technique on smaller components of a project (e.g. packages) or by removing AST information for classes that do not exhibit any refactoring opportunities.

Table 6
Various size measures for the examined open-source projects.

Measures	Nutch 1.0	FreeCol 0.8.3	JMol 11.6.21	JFreeChart 1.0.13
#Classes	582	613	548	1037
#Methods with body	2554	5104	6337	9960
#Conditional structures	2391	5598	10730	8042
Source lines of code*	42,955	83,258	106,237	143,062

* Source lines of code (SLOC) have been measured using SLOCCount.

Table 7
CPU times for each step required for the extraction of refactoring suggestions*.

Step	Nutch 1.0	FreeCol 0.8.3	JMol 11.6.21	JFreeChart 1.0.13
a	7984 ms	17,250 ms	13,200 ms	20,890 ms
b	200 ms	578 ms	1780 ms	734 ms

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

6. Conclusions

Despite the wide acknowledgement of the benefits of polymorphism in object-oriented systems, the identification of places in code where polymorphism should be introduced is neither trivial nor supported by tools. In this paper, we have proposed a technique that extracts refactoring suggestions introducing polymorphism as a solution to state-checking problems.

The comparison of the refactoring opportunities identified by an independent expert on three open-source projects to the results of the proposed technique has shown a moderate precision and relatively high recall. The small number of false negatives encourages the use of the proposed technique as a semi-automatic approach, where the designer eventually decides whether a suggested refactoring should be applied or not. A second experiment investigated by means of binary logistic regression the correlation between three quantitative factors and the decision of the expert. The results indicate that the designer agrees in introducing polymorphism when an inheritance hierarchy will be extensively utilized by adding several polymorphic methods, when a large number of statements will be moved to the State subclasses and when the number of State subclasses that will be created is relatively small. Finally, performance analysis has shown that the proposed technique is efficient in terms of computation time but less scalable in terms of memory usage.

Acknowledgements

The authors would like to thank Konstantinos Kouskouras for his contribution to the independent assessment of the proposed technique.

References

Arisholm, E., Sjøberg, D.I.K., 2004. Evaluating the effect of a delegated versus centralized control style on the maintainability of object-oriented software. IEEE Transactions on Software Engineering 30 (8), 521–534.

Brito e Abreu, F., 1995. The MOOD metrics set. In: Proceedings of the Ninth European Conference on Object-Oriented Programming Workshop on Metrics.
Brito e Abreu, F., Melo, W., 1996. Evaluating the impact of object-oriented design on software quality. In: Proceedings of the Third International Software Metrics Symposium, pp. 90–99.
Day, M., Gruber, R., Liskov, B., Myers, A.C., 1995. Subtypes vs. where clauses: constraining parametric polymorphism. In: Proceedings of the 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, pp. 156–168.
Demeyer, S., Ducasse, S., Nierstrasz, O., 2003. Object-Oriented Reengineering Patterns. Morgan Kaufman.
Demeyer, S., 2005. Refactor conditionals into polymorphism: what's the performance cost of introducing virtual calls? In: Proceedings of the 21st IEEE International Conference on Software Maintenance, pp. 627–630.
Du Bois, B., 2006. A Study of Quality Improvements by Refactoring. Ph.D. dissertation, University of Antwerp.
Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D., 1999. Refactoring: Improving the Design of Existing Code. Addison Wesley.
Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley.
Horstmann, C., 2005. Violet. Available from: <http://www.horstmann.com/violet/>.
Kerievsky, J., 2004. Refactoring to Patterns. Addison Wesley.
Martin, R.C., 2003. Agile Software Development: Principles, Patterns and Practices. Prentice Hall.
Ng, T.H., Cheung, S.C., Chan, W.K., Yu, Y.T., 2006. Work experience versus refactoring to design patterns: a controlled experiment. In: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 12–22.
Ng, T.H., Cheung, S.C., Chan, W.K., Yu, Y.T., 2007. Do maintainers utilize deployed design patterns effectively? In: Proceedings of the 29th International Conference on Software Engineering, pp. 168–177.
Ó Cinnéide, M., Nixon, P., 1999. A methodology for the automated introduction of design patterns. In: Proceedings of the IEEE International Conference on Software Maintenance, pp. 463–472.
Ó Cinnéide, M., 2000. Automated Application of Design Patterns: A Refactoring Approach. Ph.D. dissertation, University of Dublin, Trinity College.
O'Keeffe, M., Ó Cinnéide, M., 2007. Getting the most from search-based refactoring. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 1114–1120.
O'Keeffe, M., Ó Cinnéide, M., 2008. Search-based refactoring for software maintenance. The Journal of Systems and Software 81 (4), 502–516.
Opdyke, W.F., 1992. Refactoring Object-Oriented Frameworks. Ph.D. dissertation, University of Illinois at Urbana-Champaign.
Parnas, D.L., 1994. Software aging. In: Proceedings of the 16th International Conference on Software Engineering, pp. 279–287.
Prechelt, L., Unger, B., Tichy, W.F., Brössler, P., Votta, L.G., 2001. A controlled experiment in maintenance comparing design patterns to simpler solutions. IEEE Transactions on Software Engineering 27 (12), 1134–1144.
Trifu, A., Reupke, U., 2007. Towards automated restructuring of object oriented systems. In: Proceedings of the 11th European Conference on Software Maintenance and Reengineering, pp. 39–48.
Tsantalis, N., Chaikalis, T., Chatzigeorgiou, A., 2008. JDeodorant: identification and removal of type-checking bad smells. In: Proceedings of the 12th European Conference on Software Maintenance and Reengineering, pp. 329–331.
Van Emden, E., Moonen, L., 2002. Java quality assurance by detecting code smells. In: Proceedings of the Ninth Working Conference on Reverse Engineering, pp. 97–106.
Wendorff, P., 2001. Assessment of design patterns during software reengineering: lessons learned from a large commercial project. In: Proceedings of the Fifth European Conference on Software Maintenance and Reengineering, pp. 77–84.
Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A., 2000. Experimentation in Software Engineering: An Introduction. Kluwer Academic Publishers.

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 is an assistant professor of software engineering in the Department of Applied Informatics at the University of Macedonia, Thessaloniki, Greece. He 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. His research interests include object-oriented design, software maintenance and metrics. He is a member of the IEEE and the IEEE Computer Society.