Searching for Refactoring Opportunities to apply the Strategy Pattern

Guinther de B. Pauli[1], Eduardo K. Piveta[1]

[1] Programa de Pós Graduação em Informática
Universidade Federal de Santa Maria - Santa Maria, RS - Brazil

piveta@inf.ufsm.br, guintherpauli@gmail.com

Abstract. *It is difficult to maintain and to adapt poorly written code presenting shortcomings in its structure. Refactoring techniques are used to improve the code and the structure of applications, making them better and easier to modify. Design patterns are reusable solutions used in similar problems in object-oriented systems, so there is no need to recreate the solutions. Applying design patterns in the context of refactoring in a corrective way becomes a desired activity in the life cycle of a specific software system. However, in large-scale projects, the manual examination of artefacts to find problems and opportunities to apply a design pattern is a hard task. In this context, we present a metric-based heuristic function to detect where the Strategy design pattern can be applied in a given project. To evaluate the heuristic function and its results we have also built a tool to show the results. This tool can examine source code using ASTs (Abstract Syntax Trees), searching for opportunities to apply the Strategy pattern, indicating the exact location in the source code where the pattern is suggested, also showing some evidences used in the detection.*

1. Introduction

Software systems evolve and are constantly modified over time usually because of requirements changes, bug fixes, performance improvements and migration to new platforms. Software should be designed to be flexible, scalable, and easy to maintain. To deal with these factors, design patterns [1] can be used to improve the quality of a software system in order to make it easier to modify over time.

A problem related to the use of design patterns is that the system designer must have a deep knowledge to determine which pattern can be applied to solve a specific problem. Based on this, there is a lack of tools that enable the application of patterns by a semi-automatic approach, even if the developer makes the final decision about applying or not the pattern. The use of patterns without this kind of support is a hard task that can introduce code errors and unnecessary complexity.

Refactoring [2, 3, 4] is the process of improving the design of software systems without changing its external observable behavior. Refactoring can improve the quality attributes [5, 6] of a software system by the application of transformations that preserve its behavior.

An important method to improve the quality of a software system is the use of refactoring to apply design patterns to obtain benefits, such as code reuse, low coupling between classes, promote best practices in object-oriented development, ease of maintenance, and evolution of the software system. It also allows the addition of new features in a simple way without introducing bugs. Based on these factors, searching the source code for refactoring opportunities aiming the application of design patterns is a relevant practice, because there is a gain in developer productivity if this process is supported by semi-automatic tools.

Therefore, we propose an approach to detect refactoring opportunities applied to the Strategy pattern [1], and use metrics and a heuristic function to search and find them. We also have implemented a tool called AROS (Automatic Refactoring Opportunity Search) [23] to evaluate and discuss the results. The main contributions of this paper are: (i) a refined and formal definition for the "Switch Statements" [3] bad smell, indicate precisely which sentences have switch problems with conditional complexity that can be refactored, (ii) a heuristic function capable to detect if a switch statement can be refactored by the application of the Strategy pattern, reduce the cyclomatic complexity [26] of the source and improve its maintenance index [27].

The remainder of this paper is organized as follows. Section 2.1 discusses about the search for refactoring opportunities. Section 2.2 presents the Strategy design pattern, while Section 2.3 describes how the *Replace Conditional Logic with Strategy* [18] refactoring can minimize problems related to conditional complexity by the application of polymorphism. Section 3 presents our approach and how we search source code to find opportunities to apply the Strategy pattern. Section 4 presents the evaluation and the results. Section 5 presents related works. Finally, Section 6 presents conclusions and future works.

2. Background

2.1. Refactoring Opportunities

A refactoring opportunity can be described as a potential improvement in a given source code in relation to a quality attribute, or a specific location where a refactoring can be applied [4]. Discover w*here* the appropriate location is on the source code and *which* refactoring to apply in a software system is not a simple task, and it relies on the developer experience. Thus, the use of refactoring as a simple approach has stimulated efforts to develop semi-automatic approaches [7, 8, 9, 10, 11, 12, 13, 14, 15] to detect design flaws. The correct application of appropriate refactorings in a given context enhances the project quality without changing its behavior. However, the identification of inconsistences on source code is not a simple task, such as methods, methods fragments, and attributes that should be moved to other classes.

A motivation to improve the design of a software system is to locate shortcomings in source code and refactor as a possible solution. Bad smells [3] describe problems and a list of related refactorings that can help to improve the code. Refactoring focuses primarily in the treatment of these problems, but the implementation of improvements depends on the developer skills, who performs software maintenances. Bad smells are a set of design problems and refactoring is a solution, but it is not in accordance with a structure of a design pattern. Finding bad smells involve inspecting all the source code, which may become impractical for medium and large-scale systems. In this scenario, semi-automatic support to detect shortcomings is essential.

Using some of these techniques, Refactoring Browser [16] was one of the first tools to provide semi-automatic support to apply refactorings. Nowadays most development environments provide some support for refactoring, although such tools reduce the effort involved in the refactoring process.

Some semi-automatic approaches [7, 11] try to suggest improvements using metrics to identify places that a refactoring may be necessary, so developers are responsible for determining precisely what changes should be made. The refactoring searching opportunities applied in design patterns brings an interesting technique to be used in life cycle of object-oriented software systems.

2.2. The Strategy Design Pattern

The Strategy design pattern [1] defines a family of algorithms, encapsulate each of them and make them interchangeable. Strategy lets the algorithm vary independently from client classes that use it. This pattern can be applied when many classes have similar purposes and differ only in their behavior (implementation). Strategy can be used when it is necessary to encapsulate a solution to a problem which client classes should not be aware of.

This pattern brings some benefits, such as the elimination of conditional statements to select desired behavior (algorithm) to solve a specific problem. Using inheritance and abstraction, Strategy encapsulates these behaviors into subclasses and uses polymorphism to replace the static conditional, opening the architecture for easy inclusion of new strategies to solve the same problem.

Figure 1 shows the structure of the Strategy pattern. It defines an abstract superclass called Strategy with an algorithm interface, generally an abstract or virtual method. Descendent classes can inherit this strategy base class to provide a specific implementation to the algorithm interface and it is known as concrete strategies. Client classes use a special class called *Context* that uses delegation to invoke the polymorphic method through the strategy interface.

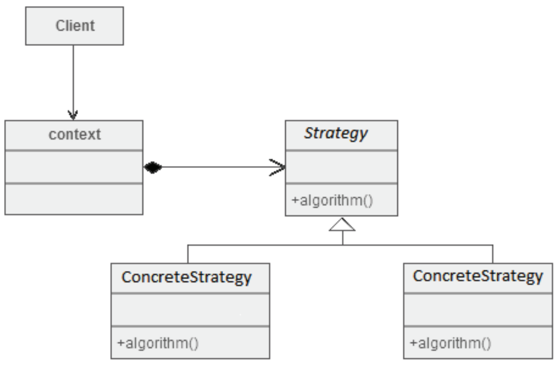


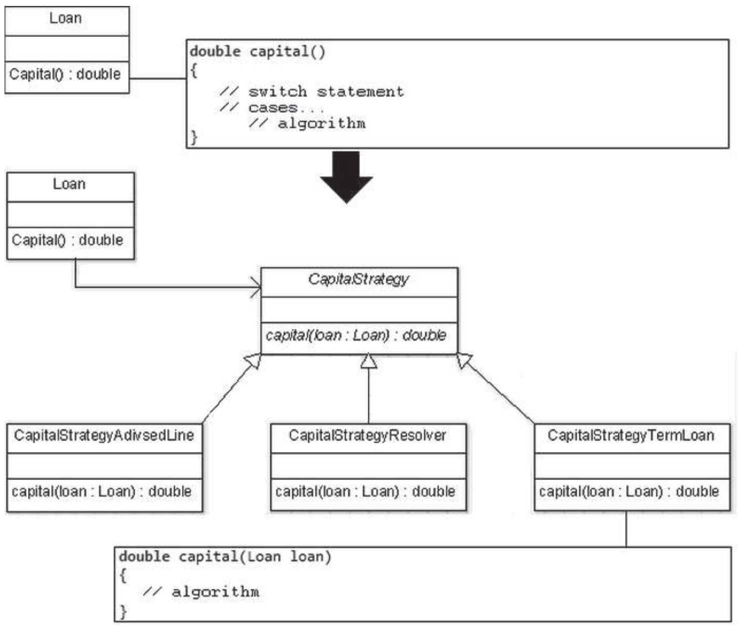
Figure 1. The Strategy design pattern [1], adapted from [18]

2.3. Replace Conditional Logic with Strategy

The *Replace Conditional Logic with Strategy* [18] refactoring intends to apply the Strategy design pattern to remove a conditional test that decides the choice of a particular algorithm. Its application consists in encapsulating each one of the algorithms within a family of classes, called *strategies*, and each class represents a variation of the algorithm. The client class, which previously had a direct connection with the algorithm, should use a context class that receives a concrete implementation of a strategy, so we can eliminate conditional logic using polymorphism.

Sometimes, conditional expressions make the code complex, especially if several conditional tests are nested. The use of refactoring in this scenario brings many advantages, such as the ease to modify a source code at runtime, since the client class is linked to an abstraction. In addition, it makes the source code clearer because static conditionals are not used, and the choice is made dynamically through polymorphism.

Figure 2 shows the *Replace Conditional Logic with Strategy* refactoring. At the top of Figure 2 we can see a class named *Loan*, which has specific logic to calculate loan amounts according to various conditions tested in a switch block. Each code related to a *case* statement on the *switch* could be transformed in a method within a class (strategy). By this way, the client class can communicate with any strategy transparently using an abstraction, as we can see at the middle of Figure 2. Programming to an interface or abstraction is one of the main foundations of object-oriented programming, and it is clear at the center of Figure 2, where the Strategy pattern is applied.


Loan
Capital() : double
Loan
Capital() : double
CapitalStrategy
capital(loan : Loan) : double
CapitalStrategyAdivsedLine
capital(loan : Loan) : double
CapitalStrategyResolver
capital(loan : Loan) : double
CapitalStrategyTermLoan
capital(loan : Loan) : double

```
double capital()
{
   // switch statement
   // cases...
      // algorithm
}
```

```
double capital(Loan loan)
{
   // algorithm
}
```

Figure 2. The Replace Conditional Logic with Strategy [18] refactoring, adapted from [18]

3. Searching for Opportunities to apply the Strategy Pattern

This section describes our approach to automatically detect the "Switch Statements" bad smell [3] using a set of metrics. We have considered the template framework proposed by Munro [7]. This template helps to give a more precise definition of a bad smell, compared to the informal descriptions from the original author [3]. The characteristics of the bad smells are used to define a set of measurements and interpretation rules for a subset of bad smells.

The motivation to use the Strategy pattern is due to the fact that it simplifies conditional statements, such as *if* and *switch / cases*. This kind of source code generates more costs and effort in the maintenance process. Using polymorphism and "Programming to an Interface/Abstraction" [1], the Strategy pattern is able to eliminate redundant tests and conditional complexity. Another motivation is that we have found many approaches that search for refactoring opportunities [7, 8, 9, 10, 11, 12, 13, 14, 15], however, few of them present a design pattern as a final result of the source code transformation. This is one of the main contributions of our approach compared to the existing ones. Most approaches are focused on searching and applying primitives refactoring, such as Move Method, Move Attribute and Extract Class [3] and are not focused on the application of design patterns.

The main problem is that the definition of the Switch Statement bad smell is informal, and only intended to guide a developer to manually locate them within a software system. We say informal because this definition does not provides an exact criterion, using well-defined metrics, to indicate what really is not adequate. Switch statements are considered a bad smell by the original authors, but not all switches need to be eliminated and some criteria should be used to locate the bad ones.

This informality does not have sufficient information to allow semi-automatic identification using a tool. Switch statements can be considered as a problem because it is not specified exactly how many tests are necessary, and the exact size of the code block that could characterize the need for refactoring. On the other hand, some switch statements are quite simple and does not require refactoring. It is needed a precise definition about the problem related to the conditional statements bad smell, so we can describe more precisely the Switch Statements bad smell as follows, using Munro's framework:

Bad smell name: Switch Statements - This kind of construction tends to duplicate code, and similar statements may be distributed throughout a program. When a clause is added or removed in a switch block, it is necessary to change other parts of the source code.

Characteristics: One of the most common fields about complexity in a program lies in conditional logic, because it tends to grow and becomes more sophisticated over time. Many changes are necessary if a new conditional test is needed on the statement and this code is already used in other places. When we have to test these conditional statements using unit tests [17], it is necessary to include some logic to each conditional sentence, which can be a problem to analyze the code test coverage.

Design heuristic: A switch statement may contain source code to evaluate *type codes*, such as enumerations. These type codes should be overridden by subclasses of an abstract base class, and their respective codes fragments should be encapsulated in methods in descendant subclasses, invoked through polymorphic calls.

Measurement process: The problem lies in the interpretation whether a switch statement is really complex. It is done based on the number of conditional tests, and analyzing if the same sentences can be extracted to polymorphic methods in subclasses. It is necessary because some sentences may have simple logic and it is not necessary to apply the Strategy pattern. We have done it by counting the number of lines involved in the switch block and it helps to improve the measurement process. If the switch block uses a type code such as an enumeration, it indicates that a specialized class is necessary. The number of conditional tests in the switch block can also justify the application of Strategy pattern.

Interpretation:

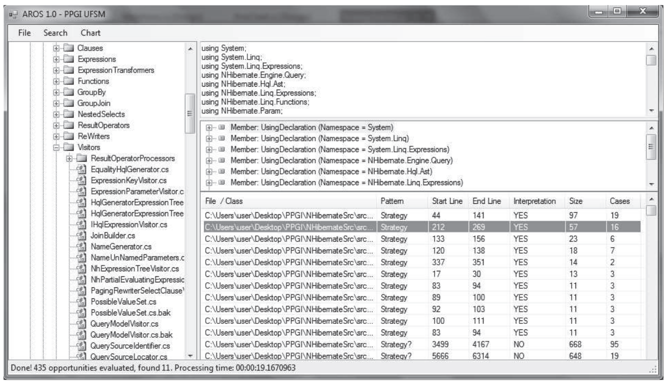
```
1: public bool isStrategyOpportunity(
2:   int numberOfCases,
3:   int sizeOfSwitch,
4:   bool isTypeCode)
5: {
6:   return (numberOfCases > cNumberOfCases) &&
7:          (sizeOfSwitch > cSizeOfSwitch) &&
8:          (isTypeCode);
9: }
```

In this function, *numberOfCases* (line 2) represents the number of cases found in a switch statement, *sizeOfSwitch* (line 3) represents the total size in lines of code (LOC) of the parsed switch statement block and *isTypeCode* (line 4) indicates if the switch uses an enumeration type in its definition. In order to find opportunities to apply the pattern, the function compares the size of a switch block and the number of tests using constants defined by the developer, called *cNumberOfCases* and *cSizeOfSwitch* (lines 6 and 7). The function also searches for *type codes* (line 8) and the evaluation is also based on them. This comparison is necessary to identity if a switch block is considered a real bad smell or a simple conditional statement that not necessarily need to be refactored. Using this approach, developers can define different values for these variables allowing the decision of which are the best opportunities to refactor.

The proposed approach was implemented by developing a C# tool we called AROS, which identifies the *Switch Statements* bad smell and whether it can be eliminated by applying the *Replace Conditional Logic with Strategy* refactoring. The tool uses an open source library called NRefactory [19], which includes support for code manipulation through ASTs.

AROS does not need integration with an IDE and it can run as a "stand alone" application and it is possible to analyze any C# project from the root directory. The tool reads and processes each file on the project, processing its AST. When a node is identified in the tree that has a switch statement, the tool provides an interpretation (YES or NO) to the conditional statement and indicates if the application of a Strategy pattern would be a good solution. After the detection of a switch block, an analysis is done, and each child node is evaluated, including case expressions. The number of lines (size) is evaluated computing the cases statements. These values are compared with predefined variables in code, which can be changed dynamically to obtain more accurate results. The algorithm evaluates if the tests are made on type codes, as enumerations, which is a strong indicator that a subclass should override this primitive type. The code in each case is evaluated to determine whether the set of all tests and results justify the opportunity to apply a Strategy. Thus, the code in each "type code" test could be considered as a polymorphic method in subclasses of an abstraction, eliminating conditional logic.

Figure 3 shows a snapshot of the tool user interface. The main screen allows the developer to open a project or a single class file. After this process, the ASTs are created, which is graphically presented to the user to allow the navigation on the source code. When a node is selected, its associated code is also selected, in order to help the developer to analyze the source code. If the tool accuses an interpretation YES to the assessment, it indicates *where* the file / class has a bad smell, and which lines are with the switch statement that can be eliminated by applying the Strategy pattern.



```
using System;
using System.Linq;
using System.Linq.Expressions;
using NHibernate.Engine.Query;
using NHibernate.Hql.Ast;
using NHibernate.Linq.Expressions;
using NHibernate.Linq.Functions;
using NHibernate.Param;
```

```
Member: UsingDeclaration (Namespace = System)
Member: UsingDeclaration (Namespace = System.Linq)
Member: UsingDeclaration (Namespace = System.Linq.Expressions)
Member: UsingDeclaration (Namespace = NHibernate.Engine.Query)
Member: UsingDeclaration (Namespace = NHibernate.Hql.Ast)
Member: UsingDeclaration (Namespace = NHibernate.Linq.Expressions)
```

File / Class	Pattern	Start Line	End Line	Interpretation	Size	Cases
C:\Users\user\Desktop\PPGI\NHibernateSrc\src...	Strategy	44	141	YES	97	19
C:\Users\user\Desktop\PPGI\NHibernateSrc\src...	Strategy	212	269	YES	57	16
C:\Users\user\Desktop\PPGI\NHibernateSrc\src...	Strategy	133	156	YES	23	6
C:\Users\user\Desktop\PPGI\NHibernateSrc\src...	Strategy	120	138	YES	18	7
C:\Users\user\Desktop\PPGI\NHibernateSrc\src...	Strategy	337	351	YES	14	2
C:\Users\user\Desktop\PPGI\NHibernateSrc\src...	Strategy	17	30	YES	13	3
C:\Users\user\Desktop\PPGI\NHibernateSrc\src...	Strategy	83	94	YES	11	3
C:\Users\user\Desktop\PPGI\NHibernateSrc\src...	Strategy	89	100	YES	11	3
C:\Users\user\Desktop\PPGI\NHibernateSrc\src...	Strategy	92	103	YES	11	3
C:\Users\user\Desktop\PPGI\NHibernateSrc\src...	Strategy	100	111	YES	11	3
C:\Users\user\Desktop\PPGI\NHibernateSrc\src...	Strategy	83	94	YES	11	3
C:\Users\user\Desktop\PPGI\NHibernateSrc\src...	Strategy?	3499	4167	NO	668	95
C:\Users\user\Desktop\PPGI\NHibernateSrc\src...	Strategy?	5666	6314	NO	648	19

Done! 435 opportunities evaluated, found 11. Processing time: 00:00:19.1670963

Figure 3. AROS: a tool to search for opportunities to apply the Strategy design pattern

4. Evaluation

This section presents a case study to evaluate our approach. The experiment was conducted using an open source software system used in unit tests called NUnit [22] framework (version 2.6.2, with 27k lines of code).

It is impractical to show all the results due to the large volume of classes involved in the experiment, so we present only the most relevant results. Table 1 indicates the interpretation result, that indicates if the heuristic function evaluated a switch statement as a possible opportunity to apply the pattern. Furthermore, we present two metrics that help the cyclomatic complexity (CC) measure and maintenance index (MI) of each class. In the CC metric, lower values are better while in the MI index, higher values are better. The results of the evaluation are calculated based on the *numberOfCases* and *sizeOfSwitch* constants. In this experiment, the values 2 and 10 were considered, respectively.

Table 2 presents the main classes evaluated by the tool. A true-positive indicates the evaluation rule show the presence of a switch statement and a real opportunity to apply the Strategy pattern, according the prior defined heuristic rule. A true-negative indicates a switch was evaluated but it was decided not to apply Strategy to solve the bad small. A false-positive indicates the approach failed to identify a possible application of Strategy pattern, which indicates the criteria should be improved. A false-negative indicates the approach suggests a “No” interpretation incorrectly, also needing improvement.

Table 1 also shows the final results of the evaluation and a number included in each cell to indicate how the results can be interpreted after a manual inspection of source code. The tool found 69 switch statements, and our approach indicated 13 possible candidates to apply the Strategy pattern. The accuracy of our approach criteria was 78.26% after manual inspection of the source code. It is possible to see a high index of True-Negative (46), which indicates a switch statement with bad smell but not capable of applying a Strategy. As an example, it makes no sense to create dozen of classes to eliminate a switch statement that presents few lines of code in each test.

Table 1. Possible interpretations and results

Interpretation	Yes	No
True	True-Positive (8)	True-Negative (46)
False	False-Positive (15)	False-Negative (0)

In most cases where the interpretation indicates YES, we applied the *Replace Conditional Logic with Strategy* refactoring on the source code indicated by the tool. After running the unit tests provided by NUnit, we have a certain degree of confidence that the external observable behavior was not affected and all tests had been successfully executed.

Furthermore, we could observe that after applying the refactoring, the cyclomatic complexity index of the classes had considerably reduced its value, which show our approach accurately identified places in the code that had complex conditional logic. Consequently, the index of maintenance for each class had their value increased, which also indicates that our approach has helped improve the way software systems can be evolved, eliminating unnecessary conditional statements that can be replaced by the Strategy pattern.

We conclude that several code snippets use a type code apply conditional logic on these types excessively, which would be more appropriate the addition of a specific class for each tested type, and each test can then be moved to a polymorphic method in concrete classes by applying the Strategy pattern. Finally, we conclude that some switch statements has many conditional tests but it contain small blocks of code associated to them, such as a single method call, or on the other hand, the logical structure is very simple to be replaced by the Strategy pattern. In these cases, creating a class for each test would generate unnecessarily complex architectures, which are difficult to maintain.

For example, examining the process to evaluate the first class shown in Table 2 using our approach, the *ProviderReference* class, according indicated by the tool, has a bad smell with an opportunity to apply the Strategy pattern between the lines 71 and 92, as seen in Listing 1 (original code from NUnit). This class has the value 17 for the cyclomatic complexity (CC) and 67 for the maintenance index (MI). After applying the *Replace Conditional Logic with Strategy* refactoring in the location indicated by the tool, the CC was reduced to 11, while the MI was increased to 73, which indicates that the tool has successfully detected the opportunity to apply the pattern, reducing the maintenance cost and complexity. Listing 2 shows the refactored code (summarized) for this example. Basically, each *case* used in the switch statement of Listing 1 was transformed in a concrete strategy class. Similarly, the MI index for the *EventCollector* class (the second class in Table 1) was increased from 69 to 73, while the CC was reduced from 37 to 28. Listing 3 shows an example of a correct NO interpretation (*PlatformHelper,* third class of Table 2), a set of small tests that not represents a real opportunity to apply the strategy pattern.

Table 2. Main evaluation results for NUnit

Class	Lines	LOC	Tests	CC	MI	Interpretation
NUnit.Core.Builders.ProviderReference	71-92	26	3	17	67	YES
NUnit.ConsoleRunner.EventCollector	79-121	42	3	37	69	YES
NUnit.Core.PlatformHelper	135-217	82	23	53	59	NO
NUnit.Framework.Constraints.MsgUtils	74-119	45	13	43	59	NO

Listing 1. Switch Statement bad smell in NUnit, a real opportunity to apply a Strategy

```
67: private object GetProviderObjectFromMember(MemberInfo member)
68: {
69:   object providerObject = null;
70:   object instance = null;
71:   switch (member.MemberType)
72:   {
73:     case MemberTypes.Property:
74:       PropertyInfo providerProperty = member as PropertyInfo;
75:       MethodInfo getMethod = providerProperty.GetGetMethod(true);
76:       if (!getMethod.IsStatic)
77:         instance = Reflect.Construct(providerType, providerArgs);
78:       providerObject = providerProperty.GetValue(instance, null);
79:       break;
80:     case MemberTypes.Method:
```

```
81:       MethodInfo providerMethod = member as MethodInfo;
82:       if (!providerMethod.IsStatic)
83:         instance = Reflect.Construct(providerType, providerArgs)
84:         providerObject = providerMethod.Invoke(instance, null);
85:       break;
86:     case MemberTypes.Field:
87:       FieldInfo providerField = member as FieldInfo;
88:       if (!providerField.IsStatic)
89:         instance = Reflect.Construct(providerType, providerArgs);
90:       providerObject = providerField.GetValue(instance);
91:       break;
92:   }
93:   return providerObject;
94: }
```

Listing 2. Code after applying the Replace Conditional Logic with Strategy refactoring

```
private object GetProviderObjectFromMember(MemberInfo member)
{
  object providerObject = null;
  object instance = null;
  var ctx = new Context(Factory.GetInstance(member.MemberType));
  ctx.AlgorithmInterface(member, ref providerObject, ref instance);
  return providerObject;
}
...
public class Context // context implementation according Strategy Pattern
...
public abstract class Strategy
{
  public abstract void AlgorithmInterface(
       MemberInfo member, ref object providerObject, ref object instance);
}
public class StrategyField : Strategy
{
  public override void AlgorithmInterface(…)
  {
    // here is the code of lines 87-90 of Listing 1
  }
}
public class StrategyMethod : Strategy
{
  public override void AlgorithmInterface(…)
  {
    // here is the code of lines 81-84 of Listing 1
  }
}
public class StrategyProperty : Strategy
{
  public override void AlgorithmInterface(…)
  {
    // here is the code of lines 74-78 of Listing 1
  }
}
```

Listing 3. Switch Statement bad smell in NUnit, but not a real opportunity to a Strategy

```
switch (platformName.ToUpper())
{
  case "WIN32":
    isSupported = os.IsWindows;
    break;
  case "WIN32S":
    isSupported = os.IsWin32S;
    break;
  case "WIN32WINDOWS":
    isSupported = os.IsWin32Windows;
    break;
  // more 20 similar small tests
```

5. Related Work

There are many researches and efforts [7, 8, 9, 10, 11, 12, 13, 14, 15] to optimize the semi-automatic search for refactoring opportunities. An important research was done by Mens and Tourwe [4], they present an extensive overview of existing research in the field of software refactoring.

Tsantalis and Chatzigeorgiou [8] propose a method to identify opportunities to the application of the Move Method [3] refactoring; aiming to minimize the Feature Envy bad smell [3]. It is based on algorithm that applies the distance between entities, such as attributes, methods, and classes. Simon et al. [11] define a metric that measures the cohesion between the attributes and methods based on its distance. The main goal is to identify methods that use characteristics from other classes in the system. The results of the calculated distances are displayed in a three-dimensional perspective, which helps the developer to manually identify refactoring opportunities, such as Move Attribute, Move Method, Extract Class and Inline Class [3].

El-Sharqwi et al. [20] propose an approach to refactor a software model using design patterns. The authors suggest a XML structure that contains a flawed design, transformation rules (refactorings), and a structure model that consists of application patterns. Similarly, Kim [21] presents an approach to refactor software models using patterns to improve the quality using three components: a problem, a transformation and a solution. Balazinska et al. [9] propose a method to refactor object-oriented software systems through the identification of clones in source code as duplicated. Mens and Tourwe [10] show how automated support can be used to identify refactoring opportunities to detect bad smells [3], like an obsolete parameter or an inappropriate interface, but they do not clearly indicate which design pattern could be applied.

Seng et al. [12] propose a method based on surveys which can be helpful in assisting a software engineer to improve the structure of a software system. It is done by suggesting a list of refactoring using evolutionary algorithms that simulates refactoring. Their work can help in the task of defining refactoring for improving the structure of the class in object-oriented systems. It also detects the "God Class" bad smell [3]. Tekin and Erdemir [13] approach is based on graph mining used to detect similar structures in object-oriented systems, which can provide useful information about the project, such as design patterns commonly used, frequent design defects, and clones.

Piveta [14] provides a detailed process for refactoring, including mechanism for the selection and creation of quality models, the selection of refactoring patterns, and the creation and use of heuristic rules, the search for refactoring opportunities and prioritization, the assessment of the effects of refactoring on software quality, and the trade-off analysis and the application of refactoring patterns. Our approach extends that research in order to identify refactoring opportunities to apply the Strategy design pattern.

O'Keeffe and Cinneide present two works [24, 25] related to search-based refactoring. The authors propose a semi-automatic refactoring opportunities search, such as Make Superclass Abstract, and Replace Inheritance with Delegation [3].

Based on the observation of the existing reviews, we have found that all papers define approaches to detect some opportunity to apply some primitive refactorings, like Move Method or Extract Class [3], but not a refactoring to a given design pattern, as *Replace Conditional Logic with Strategy* [18]. For example, if some found opportunities were combined, this could indicate the application of a design pattern, like Adapter, Decorator, Template Method or Strategy [1]. If a method is moved to a class, this being an abstract base class, and this method can be overridden in descendant classes to define different behaviors for the same result, a Strategy pattern could be suggested as final result of the refactoring. Furthermore, many works propose to increase software quality based on the application of some design patterns in source code, but they not provide an automatic tool to support the search. Based on this, we extend these approaches to i) propose an automatic tool to support the efforts to locate where a refactoring can be applied, ii) suggest a refactoring to a pattern (Strategy) instead of a primitive refactoring (like Move Method), iii) evaluate medium and large-scale projects instead of small projects.

6. Conclusion and Future Work

Our approach shows how to identify opportunities to apply the Strategy design pattern, because the exhaustive use of conditional statements makes software system complex to maintain and evolve. As future work, we plan to extend the approach and the implementation of the tool, to search for opportunities to apply other important and useful patterns, including Adapter, State, and Template Method [1]. We are studying the possibility to analyze different versions of the same source code, in order to detect the increasing complexity of switch statements along the evolution of the software system.

7. References

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

[2] Opdyke, W.F., “Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks”, PhD thesis, Univ. of Illinois at Urbana-Champaign, 1992.

[3] Fowler, M., “Refactoring: Improving the Design of Existing Programs”, Addison-Wesley, 1999.

[4] Mens, T., Tourwé, T., “A survey of software refactoring”. IEEE Trans. Softw. Eng., 30(2):126–139, Feb. 2004.

[5] ISO. Iso/iec 9126-1:2001 - product quality - part 1: Quality model. Technical report, Intl. Org. for Standardization, 2001.

[6] Boehm, B., In, H., “Identifying quality-requirement conflicts”, Software, IEEE, vol.13, no.2, pp.25,35, Mar 1996.

[7] Munro, M.J., “Product Metrics for Automatic Identification of "Bad Smell" Design Problems in Java Source-Code”, Software Metrics, 2005. 11th IEEE International Symposium, vol., no., pp.15,15, 19-22 Sept. 2005.

[8] Tsantalis, N., Chatzigeorgiou, A., “Identification of Move Method Refactoring Opportunities”, Software Engineering, IEEE Transactions on, vol.35, no.3, pp.347,367, May-June 2009.

[9] Balazinska, M., Merlo, E., Dagenais, M., Lague, B., Kontogiannis, K., “Advanced clone-analysis to support object-oriented system refactoring”, Reverse Engineering, 2000. Proceedings. Seventh Working Conference on , vol., no., pp.98,107, 2000.

[10] Tourwé, T., Mens, T. “Identifying Refactoring Opportunities Using Logic Meta Programming”, Seventh European Conference on Software Maintenance and Reengineering Proceedings, March 2003.

[11] Simon, F., Steinbruckner, F., Lewerentz, C., “Metrics based refactoring”, Software Maintenance and Reengineering, 2001. Fifth European Conference on, vol., no., pp.30,38, 2001.

[12] Seng, O., Stammel, J., Burkhart, D., “Search-based determination of refactorings for improving the class structure of object-oriented systems”, Proceedings of the 8th annual conference on Genetic and evolutionary computation (pp. 1909-1916). ACM, 2006.

[13] Tekin, U., Erdemir, U., Buzluca, F., “Mining object-oriented design models for detecting identical design structures”, Software Clones (IWSC), 2012 6th International Workshop on , vol., no., pp.43,49, 4-4 June 2012.

[14] Piveta, E. K., “Improving the search for refactoring opportunities on object-oriented and aspect-oriented software”, PhD thesis, Universidade Federal do Rio Grande do Sul, 2009.

[15] Bavota, G., De Lucia, A., Oliveto, R., “Identifying Extract Class refactoring opportunities using structural and semantic cohesion measures”, Journal of Systems and Software, Volume 84, Issue 3, March 2011, Pages 397-414, ISSN 0164-1212, 10.1016/j.jss.2010.11.918.

[16] Robert’s Refactoring Browser (Roberts, 1999).

[17] Beck, K., “Test Driven Development: By Example”, Addison-Wesley, 2002.

[18] Kerievsky, J., “Refactoring to Patterns”, Addison-Wesley, 2008.

[19] NRefactory. https://github.com/icsharpcode/NRefactory.

[20] El-Sharqwi, M., Mahdi, H., El-Madah, I., “Pattern-based model refactoring”, Computer Engineering and Systems (ICCES), 2010 International Conference on, vol., no., pp.301,306, Nov. 30 2010-Dec. 2010.

[21] Kim, D. “Software Quality Improvement via Pattern-Based Model Refactoring”, High Assurance Systems Engineering Symposium, 2008. HASE 2008. 11th IEEE , vol., no., pp.293,302, 3-5 Dec. 2008.

[22] NUnit. http://www.nunit.org/

[23] AROS. http://www.sourceforge.net/projects/aros2dp

[24] O'Keeffe, M., Cinnéide, M. Ó., “Search-based refactoring: an empirical study”, J. Softw. Maint. Evol.: Res. Pract., 20: 345–364. 2008.

[25] O'Keeffe, M., Cinnéide, M. Ó., “Search-based refactoring for software maintenance”, Journal of Systems and Software, Volume 81, Issue 4, Pages 502–516. 2008.

[26] McCabe, T.J., “A Complexity Measure”, Software Engineering, IEEE Transactions on , vol.SE-2, no.4, pp.308,320, Dec. 1976.

[27] ISO/IEC 9126 – Software and System Engineering – Product quality – Part 1: Quality model. 1999-2002.