Refactoring generics in JAVA: a case study on *Extract Method*

Raúl Marticorena, Carlos López
Computer Languages and Systems Area
University of Burgos
Burgos, Spain
Email: {rmartico,clopezno}@ubu.es

Yania Crespo, F. Javier Pérez
Computer Science Department
University of Valladolid
Valladolid, Spain
Email: {yania, jperez}@infor.uva.es

***Abstract*—The addition of support for genericity to mainstream programming languages has a notable influence in refactoring tools. This also applies to the JAVA programming language. Those versions of the language specification prior to JAVA 5 did not include support for generics. Therefore, refactoring tools had to evolve to modify their refactoring implementations according to the new language characteristics in order to assure the correct effects when transforming code containing generic definitions or using generic instantiations.**

This paper presents an evaluation of the behaviour of refactoring tools on source code that defines or uses generics. We compare the behaviour of five refactoring tools on a well known refactoring, *Extract Method*, and its implementation for the JAVA language. We distill the lessons learned from our evaluation into requirements that have to be taken into account by refactoring tools in order to fully conform to this new language feature.

***Keywords*-generics; generic programming; refactoring; extract method; test cases**

I. INTRODUCTION

Software Maintenance is an area of special interest in Software Engineering. Most of the cost and time invested in a software project is devoted to the maintenance phase [1]. As software evolves as part of the maintenance activities, it is in continuous decay. One of the most recommended techniques, with increasing success, to improve software structure during evolution is known as *software refactoring*.

Refactoring is a transformation of the software's *internal structure*, the main motivation of which is to enhance software comprehensibility and reduce maintenance costs, keeping *observable behavior* unchanged [2]–[4]. Ideally, this transformation may be achieved automatically or semi-automatically, provided with appropriate tools. The relevance of automating refactoring in recent years can be seen in workshops focused on refactoring tools development [5]–[7].

Refactoring tools must deal with source code. Source code is written in programming languages. Hence, as programming languages evolve, refactoring tools must evolve too, in order to adapt to new language features. In recent years, some programming languages have updated their specifications to provide support for new characteristics such as generics and annotations (metadata support). This is the case of the JAVA language. First versions of the JAVA language kept the language syntax almost unchanged. Version 1.1 introduced inner and anonymous classes, version 1.4 added the `assert` statement, but version 1.5, also known as JAVA 5, is an important evolution of the language. JAVA 5 incorporated generic (parametric) types, automatic conversion (autoboxing), static imports, typed enumerations, functions with variable number of arguments, and enhanced for-loop statements allowing smooth implementations of the iterator design pattern.

Generics in JAVA supposed an important step towards programming type-safeness. The use of parametric types avoids downcasting to some extent. This allows more legible code to be written and avoids from including additional code for type checking purposes. Generic methods and type inference jointly provide a simple reuse technique. Constrained genericity, multiple bounds, unknown type and `extends` or `super` bounds [8] help to rewrite existing code in a simpler and safer manner. Then, the opportunity of refactoring legacy non-generic code to generic code [9], [10] arises as a challenge for refactoring tools. Moreover, source code exploiting generic capabilities proliferates. Therefore, refactoring tools, when targeting code using generics, must be able to guarantee that refactoring operations previously available do obtain the correct results, according to the new language specification.

In order to analyze the current situation, this work presents a case study of the *Extract Method* refactoring on a set of well-known tools: ECLIPSE [11], NETBEANS [12], REFACTORIT [13], INTELLIJ IDEA [14] and CODEGUIDE [15]. The study is generics-centered, exploring issues such as type parameter definition in classes and methods, bounding, unknown type, and type parameter instantiation. A suite of test cases is defined and described. They are used to evaluate how refactoring tools perform the *Extract method* in the absence and presence of generics. Results of the study can be used to state some helpful properties we recommend for refactoring tools.

The rest of the paper is organized as follows: Sec. II looks inside the definition of the *Extract Method* refactoring. Sec. III presents a suite of test cases to analyze and evaluate refactoring tools behaviour for the *Extract Method* in the

absence of generics. This suite comes with a frame for comparing refactoring tools. The results of the comparison of five refactoring tools is presented. Sec. IV completes the study with the definition of another test suite in the presence of generics. It also shows the results of comparing the same tools as in the previous Sec. Sec. V analyzes works related with the development of integrated or standalone refactoring tools, as well as their architectonic and design solutions. Finally, the conclusions and future work are presented in Sec. VI.

II. The *Extract Method* refactoring

The *Extract Method* refactoring was originally defined in Fowler's Refactoring Catalog [2][1]. It can be considered as one of the most relevant refactorings since an important number of other refactorings in this Catalog refer to it and also because it is a solution for several code defects (*Bad Smells*) [2]. From the perspective of developing refactoring tools, Fowler explained that the automation degree for this refactoring reveals the situation he called "Crossing Refactoring's Rubicon" [2]. It therefore seems to be the ideal case study for conducting this work.

There is no commonly agreed refactoring definition. No consensus exists on either concepts (refactoring arguments, preconditions, postconditions) or on transformations (actions). The definition presented in Table I is inspired by the original description in Fowler's book [2], but intends to clarify the preconditions and effects of the refactoring (postconditions) as well as the actions to be performed. Each element is described in a textual manner. It is also possible to tackle more formal descriptions based on predicate logic [16] or OCL [17]. Nevertheless, the purpose here is to present the refactoring in a simple but more precise way, rather than introducing more complex descriptions.

The execution of the *Extract Method* refactoring depends, to a great extent, on the variables used in the method's scope. Fields need no special treatment since their scope is the entire class, hence values do not need to be packaged in parameters or as return values in order to "survive" the invocation of the extracted method. Formal arguments must be treated as local variables that may not be assigned (Fowler recommends in his book the use of the *Remove Assignments to Parameters* refactoring). Formal arguments and local variables are jointly the center of attention here. Different cases can be described separately:

- **No variables**. No local variable is used in the selected code fragment. In this case, the extracted method will not receive any arguments nor return any value.
- **Read**. The selected code fragment reads local variables. These local variables must be passed as arguments when calling the extracted method. They can be named as *inputs*.
- **Read/Write**. Local variables inside the selected code fragment are read and also assigned. It can be considered that there are *inputs* and *outputs*. Preconditions enable just one *output* value.
- **Write**. Variables are declared in the fragment and modified on it. This can be considered as *outputs*.

Depending on how instructions following the selected code fragment use variables, the cases of Read/Write as well as Write variables can be split to:

- The modified variable is not used in instructions subsequent to the code fragment. If the variable is used in instructions before the extracted code fragment, it should be passed as value argument, in any other case, its declaration can be moved into the extracted method.
- The modified variable is used in instructions subsequent to the code fragment. This forces the tool to make the extracted method return the value and the calling code assigned to it.

Table I
EXTRACT METHOD AT A GLANCE

Description Extract a code fragment into a new method of the same class, replacing the extracted code fragment by a call to the extracted method.
Motivation A code fragment can be invoked as a new different method. This enables existing duplication to be removed and reuse in other places without creating redundant copies.
Inputs The code fragment (*f*) and the name of the new method (*N*)
Preconditions
- The code fragment must be valid: compilation succeeds even if the selected code fragment is removed from the method. - There is no other method in the class with the same signature as the method to be created. - If more than one local variable or parameter is assigned in the code fragment, at most one of them is read in the control flow of the original method after the execution of the extracted code fragment. - The code fragment does not contain conditional returns: it always returns (contains unconditional return) or always flows from the beginning to end. - The code fragment does not jump outside itself.
Actions
- Add method skeleton with name *N*. - Add formal arguments to the new method *N*. Formal arguments are inferred form parameters read in the extracted fragment and local variables read in the fragment and assigned in the original method's control flow before the fragment. - Move the extracted fragment code. - if necessary: Add a return type to the new method's signature and a return instruction to the method body. - if necessary: Add `throws` clauses to the new method's *N* signature. - Replace the extracted code fragment with the appropriate method call, passing the proper parameters.
Postconditions
- The class, where the code fragment was contained, has a new method named *N* and signature with inferred formal arguments and return type. - The original method contains a call to the new method *N* instead of the extracted code.

[1]Available at http://www.refactoring.com/catalog/extractMethod.html

[2]Crossing Refactoring's Rubicon, http://www.martinfowler.com/articles/refactoringRubicon.html

Generics introduce a new problem concerning the presence of formal type parameters declarations in methods. In JAVA 5, it is possible to declare type parameters at the class level, but also at the method level. With generic methods, the extracted method must include the necessary type parameters declarations and their bounds in order to guarantee the semantic correction of the refactored code. Nevertheless, when type parameters are declared at the class level, they have a broader scope. In this case, no special treatment is necessary. This is similar to the case of class attributes in terms of variables.

At the base of the provided *Extract method* refactoring definition, a study of how it is implemented in five of the most extended refactoring tools is presented in the next two Sections. First, the study deals with refactoring source code in the absence of generics, and later on, with refactoring source code in the presence of generics.

III. STUDYING THE *Extract Method* REFACTORING IN THE ABSENCE OF GENERICS

The case study has been divided into two parts detailed in this and the following sections. The first suite of test cases, those defined in the current section, have been designed for use in a comparative study of refactoring tools when the *Extract Method* refactoring is run over source code without generics. The test cases in the current section has also been divided in two groups with different aims. The first group of tests fulfills the refactoring preconditions. On the other hand, the second group is comprised of tests that do not fulfill the preconditions. The whole code set is available at http://pisuerga.inf.ubu.es/rmartico/extractmethod.

The test cases of the first subgroup, those that fulfill the preconditions, try to cover all the significant cases, in order to see the effects of applying the *Extract Method* refactoring:

- **Without local variables**. This is the simplest case, since local variables are not used in the extracted code fragment. Neither is it necessary to propagate values (or references), nor to collect return values.
- **With input variables**. We need to propagate the values of the local variables inside the extracted code fragment to use them in the method. They are passed as formal arguments.
- **With input variables and one output variable**. We need to propagate the values of the variables inside the extracted code fragment to use them in the extracted method. In addition, we need to collect the new value from one of the variables. This value is obtained in the next steps. Besides, if the variable is declared inside the code fragment, it may be necessary to add the declaration of the returned variable.
- **Loop reentrance**. The extracted code fragment is inside a loop, and uses auto-increments or auto-decrements on one variable (*i.e.* i++;). The test case is repeated using nested loops. As a result of the iteration, however, the incremented variable is not apparently accessed in the control flow of the original method after the extracted fragment is left. It is necessary to return and to collect its value between iterations.
- **Add exceptions to method signature**. The test checks that the extracted method throws the correct exception set. A first test extracts a code fragment with a set of the checked exceptions to be added in the new method signature. In the second case, only one exception must be thrown, since the code fragment contains the instruction `try-catch` for the second exception.

The second suite contains the test cases that do not fulfill the preconditions. The tests are detailed in the following cases:

- **Return of several variables**. Several variables are modified inside the extracted code fragment, and are accessed later. This problem also appears in loop reentrance with several variables, using a simple or nested loop.
- **Code fragment is not complete**. The selected code fragment does not compile once it is extracted to the new method, although formal argument or variable declarations are added. Its extraction generates a syntactically incorrect code.
- **Conditional return**. The code fragment may not return, depending on the method's input values. The code fragment must not contain conditional returns: it always returns, or it runs from the beginning to the end.
- **No jumps out of the fragment**. The code fragment does not contain jumps out of the fragment. This problem appears when labels are used with `continue` or `break` instructions. This problem also appears if there are `break` or `continue` instructions inside a loop, and the loop is not included in the code fragment.
- **Method extracted with the same signature**. The extracted method has the same signature as other methods in the current class.

The results collected with this test suite can be seen in Table II. For each tool, it shows the results for inputs that fulfill refactoring preconditions. In a general way, they obtain the same refactored code in most of the cases (fulfilling preconditions).

The loop reentrance is one of the most complicated cases to detect. It seldom fails in ECLIPSE but more frequently in CODEGUIDE. Regarding the exception handling, most of the tools refactor the code correctly. However, there are variants in how they detect the inheritance relationship between exception classes, and thus, different numbers of exceptions are added in the `throws` clause of the extracted method.

The behaviour of each one of the tools, taking into account the non-fulfillment of preconditions that disallow running the refactoring, is shown in Table III.

Table II
TEST CASES BASED ON FULFILLED PRECONDITIONS

Test Cases	ECLIPSE 3.5.0	NETBEANS 6.5.1	REFACTORIT 2.7.beta	INTELLIJ IDEA 8.1.3	CODEGUIDE 8.0
Without variables	Ok	Ok	Ok	Ok	Ok
With input variables	Ok	Ok	Ok	Ok	Ok
With input variables and one output variable	Bug: it always returns a value but is never accessed in the next instructions	Ok	Ok	Ok	Error
With input variables and one output variable with type declaration	Ok	Ok	Ok	Ok	Ok
Several variables are modified but never accessed in the next instructions	Ok	Ok	Ok	Ok	Ok
Loop reentrance	Ok	Ok	Ok	Ok	Error
Loop reentrance with nested loop	Error	Ok	Ok	Ok	Error
Add exception in method signature	Ok (2 except.)	Ok (2 except.)	Ok (only IOException)	Ok (only IOException)	Ok (only IOException)
Add exception with nested `try`	Ok	Ok	Ok	Ok	Ok

Table III
TEST CASES BASED ON NON-FULFILLMENT OF PRECONDITIONS

Test Cases	ECLIPSE 3.5.0	NETBEANS 6.5.1	REFACTORIT 2.7.beta	INTELLIJ IDEA 8.1.3	CODEGUIDE 8.0
Return of several variables	Ok	Ok	Ok	Ok	Error
Return of several variables with loop (loop reentrance)	Ok	Error	Ok	Ok	Error
Return of several variables with nested loop (loop reentrance)	Error	Error	Ok	Ok	Error
Code fragment is not complete	Ok	Ok	Ok	Ok	Ok
Conditional return	Ok	Ok (additional generated code)	Ok	Ok (additional generated code)	Ok
No jumps out of the fragment	Ok	Ok (additional generated code)	Ok	Ok (additional generated code)	Ok
Method extracted with the same signature	Ok	Error	Ok	Ok	Bug: if method m() exists

We can also observe that each tool has a particular behaviour. Either they detect the non-fulfilment of the preconditions and avoid running the refactoring (this is the usual behaviour when more than one return value exists) or they warn the user about the non preservation of the behaviour. The final decision, about continuing or cancelling the operation, is delegated to the user (*e.g.,* a method with the same signature).

In cases such as "conditional return" or "no jumps out of the fragment", NETBEANS and INTELLIJ IDEA add some additional code to the extracted method, which returns a boolean value. The method invocation is included in an `if` instruction conditioned by the returned value to run the `return` or labelled `break` instruction . In CODEGUIDE, the user cannot introduce the new method name.

These problems get worse if we combine the multiple return problem with the loop reentrance. In this case, ECLIPSE, NETBEANS and CODEGUIDE show problems in detecting these situations. Even in some cases, bugs and errors have been shown in these tools.

Basically, although the *Extract Method* refactoring was defined a decade ago, its implementation and behaviour is very different in each tool. As we shall see in the next section, the errors increase when new generics-related features are included in the code.

IV. STUDYING THE *Extract Method* REFACTORING IN THE PRESENCE OF GENERICS

When Fowler defined this refactoring, all examples were coded with the JAVA syntax previous to the 1.5 version [18], [19]. Generics were included in subsequent versions, and hence the problems about generics had not been taken into account.

In this section, generics are considered in refactoring as well as their implementation in different tools. The code examples are extracted from the JAVA API source code (package `java.util`), and other code examples are taken from [8]. In each case, we point out the extracted code fragment and the expected result. After applying these test cases, the observed results are collected on the selected tool set.

All previous preconditions and postconditions, explained

in Sec. III, are fulfilled. Formal type parameters or type variables must be taken into account in the classes and the methods where they are declared:

- **Class formal parameter**: its scope is constrained to the class where it is declared. It can be used as any other type. For example: `public class Vector<E>` where `E` is the formal parameter in the class, and can be used in the type declarations in the class code.
- **Method formal parameter**: its scope is constrained to the method where it is declared. For example: `<E> void add(List<E> list, E element)` where `E` is the method formal parameter. It can only be used in the `add` method scope. When *Extract Method* refactoring is applied, the formal parameter declarations and their bounds must be propagated to the new method if the formal parameter is used within declarations contained in the extracted code.

Although this work does not seek to give an exhaustive list, Table. IV shows a summary of the test cases and the collected results. The suite of test cases with generics are enumerated next:

- **With class formal parameter:** If we apply the refactoring on a class `java.util.ArrayList<E>` with a method `remove` using lines 4-11 of Listing. 1, the extracted method can access the type `E` without additional changes.

Listing 1. Class formal parameter

```
1 public E remove(int index) {
2     RangeCheck(index);
3     modCount++;
4     // beginning
5     E oldValue = (E) elementData[index];
6     int numMoved = size - index - 1;
7     if (numMoved > 0)
8         System.arraycopy(elementData,
9         index+1, elementData, index,
10                numMoved);
11    // end
12    elementData[--size] = null;
13    return oldValue;
14 }
```

This behaviour is fulfilled in ECLIPSE, NETBEANS, INTELLIJ IDEA and CODEGUIDE. In the particular case of REFACTORIT, a new `import` clause is added (`import java.util.ArrayList.E;`) that generates compilation errors.

- **Using unknown type:** We need to propagate unknown type declarations (`?`), and their bounds.

Listing 2. Unknown type declarations

```
1 public boolean addAll(
2         Collection<? extends E> c) {
3     // beginning
4     Object[] a = c.toArray();
5     int numNew = a.length;
6     ensureCapacity(size + numNew);
7     System.arraycopy(a, 0, elementData,
8         size, numNew);
9     // end
10    size += numNew;
11    return numNew != 0;
12 }
```

In Listing. 2, we apply the refactoring on lines 3-9. The type of formal argument `c` is propagated correctly, with the exception of REFACTORIT that declares a new formal argument with type `Collection`.

- **Method formal parameter inferred from generic array type:** When we apply the refactoring on a method with formal parameters, as the method `toArray` in Listing. 3, extracting lines 6-11, the new method must contain a formal parameter `T` inferred from the generic array type.

Listing 3. Method formal parameter inferred from generic array type

```
1 public <T> T[] toArray(T[] a) {
2     if (a.length < size)
3         return (T[]) Arrays.copyOf(
4             elementData, size,
5             a.getClass());
6     // beginning
7     System.arraycopy(elementData, 0, a,
8         0, size);
9     if (a.length > size)
10        a[size] = null;
11    // end
12    return a;
13 }
```

This test case does not pass in ECLIPSE, NETBEANS, CODEGUIDE and REFACTORIT. This last refactoring tool, REFACTORIT, tries to add a new import clause `java.util.ArrayList.T`. INTELLIJ IDEA is the only tool that completes the refactoring in a correct way.

- **Type inference from declarations:** If we extract the code between lines 3-7 in Listing. 4, it must be declared a formal parameter `<T>` in the new method.

Listing 4. Type inference from declarations

```
1 public static <T> List<T> toList(T[] arr){
2     List<T> list = new ArrayList<T>();
3     // beginning
4     for (T elt : arr){
5         list.add(elt);
6     }
7     // end
8     return list;
9 }
```

In ECLIPSE and INTELLIJ IDEA, type `T` is inferred from the non-generic array declarations (e.g. `T elt`). With NETBEANS, CODEGUIDE and REFACTORIT, the formal parameter is not included, so generated code is not correct.

- **Bounded unknown type with formal parameter** In the cases of bounded unknown type with formal parameter, the bounds must be propagated to the extracted method.

Listing 5. Bounded unknown type with formal parameter

```
1public static <T> void
2     copy(List<? super T> dst,
3          List<? extends T> src){
4     // beginning
5     for (int i = 0; i < src.size(); i++){
6         dst.set(i, src.get(i));
7     }
8     // end
9}
```

In Listing. 5, if we extract the code between lines 4-8, with ECLIPSE, NETBEANS and CODEGUIDE, the formal parameter and bound `T` are not added. REFACTORIT does not add the formal parameter but declares the variables `dst` and `src` as type `List`. INTELLIJ IDEA completes the refactoring with success.

- **Simple bound in method formal parameter** Simple bound of the formal parameter must be propagated to the extracted method. In Listing. 6, if we extract the lines 9-16, ECLIPSE, NETBEANS, REFACTORIT and CODEGUIDE add formal parameters `<S, T>` but simple bounds are lost.

Listing 6. Simple bound in method formal parameter

```
1public static <S extends Readable,
2               T extends Appendable>
3     void copy(S src, T trg, int size,
4               boolean flag)
5     throws IOException{
6     CharBuffer buf = CharBuffer.
7                         allocate(size);
8     int i = src.read(buf);
9     // beginning
10    while(i>0){
11        buf.flip();
12        trg.append(buf);
13        buf.clear();
14        i = src.read(buf);
15    }
16    // end
17}
```

REFACTORIT includes `import` clauses for type `java.util.List.T` and `java.util.List.S`. Only INTELLIJ IDEA correctly completes the execution of this refactoring.

- **Multiple bound in method formal parameter** (see Listing. 7). The last test studies the propagation of multiple bounds. As before, if we extract lines [10 - 17] in Listing. 7, we obtain the same behaviour as in the previous case with simple bound on the selected refactoring tool set.

Listing 7. Multiple bound in method formal parameter

```
1public static <S extends
2            Readable & Closeable,
3               T extends
4            Appendable & Closeable>
5    void copy(S src, T trg, int size)
6    throws IOException{
7    CharBuffer buf = CharBuffer.
8                       allocate(size);
9    int i = src.read(buf);
10   // beginning
11   while(i>0){
12       buf.flip();
13       trg.append(buf);
14       buf.clear();
15       i = src.read(buf);
16   }
17   // end
18   src.close();
19   trg.close();
20}
```

In conclusion, we can observe that generics are not completely included in current implementations of *Extract Method* refactoring. INTELLIJ IDEA is the only exception that provides a complete support with these test cases. The remaining tools do not include full support. Generics do not need additional preconditions or postconditions, but require a complete and correct type system support. This is a basic condition to guarantee behaviour preservation. These problems prove that refactoring tools need to evolve when new features appear in mainstream programming languages such as JAVA.

V. RELATED WORK

Tool support for automatic refactoring is widespread nowadays. Either standalone refactoring tools or *plug-ins* or *add-ins* for Integrated Development Environments are available. Implementing refactoring operations from Fowler's catalog [2] has been the predominant trend. The refactoring operations implemented by each tool are listed as options in an additional menu to attract software developers with these special automatic software transformations. The refactoring use is studied [20] [21] in order to understand how programmers refactor.

These refactoring operations were originally implemented as independent algorithms. Implementations did not fulfill some relevant quality aspects such as scalability, reusability and language independence. Excessive coupling to the execution environment, as well as to the source language, did not favor internal reuse when the source language was evolved, or the refactoring support. The great problems of software maintenance and evolution emerged.

The historical precedent of refactoring tools was the *Refactoring Browser* [22]. It was one of the first tools, if not the very first, integrating refactoring into a development

Table IV
TEST CASES IN THE PRESENCE OF GENERICS

Test cases	ECLIPSE 3.5.0	NETBEANS 6.5.1	REFACTORIT 2.7.beta	INTELLIJ IDEA 8.1.3	CODEGUIDE 8.0
Class formal parameter	Ok	Ok	Error	Ok	Ok
Unknown type declarations	Ok	Ok	Error	Ok	Ok
Method formal parameter of generic array type	Error	Error	Error	Ok	Error
Type inference from declarations	Ok	Error	Error	Ok	Error
Unknown type bound with formal parameter	Error	Error	Error	Ok	Error
Simple bound in method formal parameter	Error	Error	Error	Ok	Error
Multiple bound in method formal parameter	Error	Error	Error	Ok	Error

environment. It works for target sources written in the SMALLTALK programming language. As with other tools in the same line, it is an example of the tendency to be closely tied to the environment and the target source programming language. REFACTORIT [13] is an example of a refactoring *plug-in* of a tool which is not tied to a specific environment but is tied to the programming language JAVA. On the contrary, the refactoring *plug-in* ECLIPSE [23] is tied to the environment, but the environment evolution has revealed the importance of decoupling refactoring support from the source programming language and execution environment. Some refactoring efforts have also been done on generics in [24] [9], [10] [25].

In [26] we can find an in depth analysis of refactoring and how refactoring support must be included in development environments. It proposes a solution which provides with a language independent layer that can be used as an independent application program interface (API). This API received the name LTK (Language Toolkit) and was used for JAVA and C++ as proof of concept.

This new form of organizing the internal structure of the refactoring tool also takes into account the fact that refactoring effects can go beyond the target source code itself. At present, a software project contains a great number of additional files in very different formats. The evolution of the ECLIPSE IDE in recent years reveals the need (and tendency) of *toolkits* fulfilling the above mentioned scalability, reusability and language independence criteria. The ECLIPSE internal design has undergone a remarkable evolution. Design solutions in recent versions of the product coincide to some extent with the works presented at [27], [28]. These works state the need of framework based solutions and describe the design solution for a refactoring tool that allows the components required to define and execute a refactoring operation to be reused.

The same tendency can be observed in the .NET language family. The inherent language independent nature of this platform impels development and assistance tools to provide multi-language capabilities. Rational solutions must be reuse based solutions, for instance when including refactoring support for the language family.

RESHARPER [29] is a good example of this. The experience the contributor obtained from the development of similar tools such as INTELLIJ IDEA [14] (for JAVA) probably eases the design and implementation of new refactoring tools. Some other contributors such as REFACTORPRO [30] and *Visual Assist X for Visual Studio* [31] are in the same line, and provide tool support for refactoring for Visual Basic .NET, ASP.NET, C# y C++.

The benefits of solutions with multi-language support can be observed in the case of C++. The development of refactoring tools for C++, such as xrefactory [32], was very limited. Nevertheless, the inclusion of C++ in the .NET family supposes that the refactoring *plug-ins* already developed for the .NET family, could provide refactoring support for C++; and this in spite of the great difficulties of implementing refactoring for C++, as was stated in the very beginnings of refactoring [3], [33].

Despite all this, the very limited efforts in Visual Studio, the *de facto* standard IDE for .NET languages, to include refactorings as part of the IDE itself is surprising. Refactoring support in Visual Studio is limited to third party products.

Another clear example of reusing internal support for different languages is CODEGUIDE [34]. It offers an IDE to work in JAVA, C#, JSP, J# and Visual Basic .NET. It is a distinguishing feature of this tool to bring support for languages from very different platforms. The IDE offers refactoring support but the available refactoring operation set is still small.

Language independence in refactoring has been a research trend for the last decade. The first works appeared in 2000. One of the most known examples is described in [35], which defines the FAMIX meta-model. FAMIX allows information to be shared between different CASE tools for reengineering purposes. One of the tools was a refactoring tool named MOOSE [36]–[38]. The FAMIX meta-model includes support for the main object-oriented features such as: classes, methods, attributes or inheritance. Invocation and property access are supported too. However, it does not support either some complex aspects of inheritance in

programming languages or the generics concepts tackled in this paper.

In the same line, another meta-model based solution was presented in [39], [40]. Extending the UML 1.4 meta-model with eight language independent resources, the new meta-model named GrammyUML was created. Semantic actions from UML were not considered in this work.

Also, a similar proposal can be found in MOON (Minimal Object-Oriented Notation) [41]. MOON represents minimal abstractions to define and analyze refactorings. MOON is centered in describing a family of languages: object-oriented, statically typed, with or without generics languages. Some strengths are the support of the complex aspect of inheritance, and full support for generics with different variants for bounding generic parameters. MOON was originally defined as a model language which can be represented as a meta-model. The main goal is to define the basis for language independence and reuse in refactoring tools.

The MOON meta-model, as a framework core, and its extension, as a framework instance, for the JAVA language were defined in [42]. This work was revised and completed in several iterations which lead to the development of the architecture presented in [43]. The refactoring tool developed on top of this architecture includes full support for the Extract Method refactoring, the case of this paper, but does not include the identification of extract method refactoring opportunities [44]. The tool is successful in executing all the tests in the suite described in this work.

VI. CONCLUSIONS AND FUTURE WORK

As pointed out in [11], changes in programming languages can also force changes in development tools. This applies to refactoring tools in particular, and this happened to JAVA 5.0 and ECLIPSE. The addition of generics to this language had an impact on the type system and produced ubiquitous changes everywhere in the source code. The amount and complexity of test cases increased, for dealing with genericity, covariance, contravariance, unknown types, etc.

The proposal presented in this work shows a new way to address the study and comparison of refactoring tools. We suggest approaching these studies through the definition of test cases, although we are aware of the difficulty in defining a complete test set for a big number of refactorings and their variants for different programming languages. However, the definition of these sets allows an objective product comparison. Moreover, this proposal can be extended to the whole set of possible combinations between refactorings and programming languages.

In addition, genericity is a feature that is becoming more widely used in software development because of its inherent advantages, which are, among others, a more type-safe code and an improved legibility. Therefore, we can expect that more source codes will be making use of generics to benefit from these advantages. As a consequence, generics have to be considered when implementing refactoring tools.

Hence, it is obvious that there is a need for building architectures which could allow for a greater degree of reuse in the development of refactoring tools. In particular, these architectures should be ready to accommodate the new features that could be added to mainstream programming languages –*e.g.,* JAVA, .NET, etc.– in the future.

With regard to other aspects, generics-related refactorings are not usually supported in refactoring tools and development environments. Some exceptions to this exists, such as CODEGUIDE [15] and INTELLIJ IDEA [14], which include the *Generify* refactoring and ECLIPSE, which supports the *Infer Generic Type Arguments* refactoring. We could not find any other refactoring product supporting generics. The lack of generics-related refactorings in refactoring catalogs reveals there is a need to keep working more deeply in this area. Future development and support of generics-related refactorings must fulfill some particular requirements [45], and should be built on top of reusable refactoring engines, frameworks and libraries. Nevertheless, the limited support for reuse and generics we have found, in most of the examples shown in this paper, means there is ample scope for work in this field.

ACKNOWLEDGEMENTS

Raúl Marticorena, Carlos López, Yania Crespo and Javier Pérez are partially funded by the Spanish government (Ministerio de Ciencia e Innovación, project TIN2008-05675).

REFERENCES

[1] L. Erlikh, "Leveraging legacy system dollars for e-business," *IT Professional*, vol. 2, no. 3, pp. 17–23, 2000.

[2] M. Fowler, *Refactoring. Improving the Design of Existing Code*. Addison-Wesley, 2000.

[3] W. F. Opdyke, "Refactoring object-oriented frameworks," Ph.D. dissertation, University of Illinois at Urbana-Champaign, IL, USA, 1992. [Online]. Available: citeseer.nj.nec.com/opdyke92refactoring.html

[4] D. B. Roberts, "Practical analysis for refactoring," Ph.D. dissertation, University of Illinois at Urbana-Champaign, IL, USA, 1999.

[5] D. Dig, in *WRT'07 1st Worskhop on Refactoring Tools*, ser. ISSN 1436-9915, In ECOOP'07, 21th European Conference Object-Oriented Programming, Berlin, Germany. Danny Dig and Michael Cebulla, 2007. [Online]. Available: https://netfiles.uiuc.edu/dig/RefactoringWorkshop/

[6] ——, in *WRT'08 2nd Worskhop on Refactoring Tools.*, ser. ISBN 978-1-60558-220-7, In OOPSLA'08, 23th International Conference on Object Oriented Programming, Systems, Languages and Applications, Nashville, Tennessee, USA. Dig, Danny and Fuhrer, Robert. M and Johnson, Ralph, 2008. [Online]. Available: http://refactoring.info/WRT08

[7] R. Fuhrer and W. F. Opdyke, in *WRT'09 3rd Worskhop on Refactoring Tools*, In OOPSLA'09, 24th International Conference on Object Oriented Programming, Systems, Languages and Applications, Orlando, Florida, USA. Fuhrer, Robert and Opdyke, William F., 2009. [Online]. Available: https://netfiles.uiuc.edu/dig/RefactoringInfo/WRT09/

[8] M. Naftalin and P. Wadler, *Java Generics and Collections*, 1st ed., O'Reilly, Ed. O'Reilly, 2007.

[9] J. D. Frank Tip, Robert Fuhrer and A. Kiezun, "Refactoring techniques for migrating applications to generic Java container classes," Tech. Rep., 2004.

[10] R. Fuhrer, F. Tip, A. Kieżun, J. Dolby, and M. Keller, "Efficiently refactoring Java applications to use generic libraries," in *ECOOP'05, 19th European Conference Object-Oriented Programming*, Glasgow, Scotland, July 27–29, 2005.

[11] R. M. Fuhrer, M. Keller, and A. Kiezun, "Advanced refactoring in the eclipse jdt: Past, present, and future," in *WRT'07, 1st Workshop on Refactoring Tools, Berlin, Germany.* ECOOP'07, june 2007, pp. 31 – 32.

[12] NetBeans.org, "refactoring: Generify refactoring specification," http://refactoring.netbeans.org/, 2006, java IDE.

[13] RefactorIt, "Refactorit - aqris software," 2007. [Online]. Available: http://www.aqris.com/display/ap/RefactorIt/

[14] JetBrains, "IntelliJ IDEA :: The Most Intelligent Java IDE," http://www.jetbrains.com/idea/, 2006, java IDE. [Online]. Available: http://www.jetbrains.com/idea/

[15] Omnicore, "CodeGuide," 2007. [Online]. Available: http://www.omnicore.com/en/codeguide.htm

[16] B. Beckert, U. Keller, and P. H. Schmitt, "Translating the object constraint language into first-order predicate logic," in *In Proceedings, VERIFY, Workshop at Federated Logic Conferences (FLoC*, 2002, pp. 113–123.

[17] L. Heaton, "Object constraint language, v 2.0," Tech. Rep., May 2006. [Online]. Available: http://www.omg.org/docs/formal/06-05-01.pdf

[18] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification Third Edition*. Boston, Mass.: Addison-Wesley, 2005.

[19] B. McLaughlin and D. Flanagan, *Java 1.5 Tiger. A Developer's Notebook*. O'Reilly.

[20] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," pp. 287–297, 2009.

[21] Z. Xing and E. Stroulia, "Refactoring practice: How it is and how it should be supported - an eclipse case study," pp. 458–468, 2006.

[22] D. Roberts, "Refactoring Browser," 1999. [Online]. Available: http://st-www.cs.uiuc.edu/users/brant/Refactory/

[23] B. Daum, *Eclipse 3 para desarrolladores Java*, 1st ed. Anaya Multimedia, 2005.

[24] A. Donovan, A. Kiezun, M. S. Tschantz, and M. D. Ernst, "Converting Java Programs to Use Generic Libraries." in *OOPSLA'04, 19th International Conference on Object Oriented Programming, Systems, Languages and Applications, Vancouver, British Columbia, Canada*, J. M. Vlissides and D. C. Schmidt, Eds. ACM, 2004, pp. 15–34. [Online]. Available: http://dblp.uni-trier.de/db/conf/oopsla/oopsla2004p.html#DonovanKTE04

[25] D. von Dincklage and A. Diwan, "Converting Java classes to use generics," in *OOPSLA'04, 19th International Conference on Object Oriented Programming, Systems, Languages and Applications, Vancouver, British Columbia, Canada*, 2004. [Online]. Available: citeseer.ist.psu.edu/vondincklage04converting.html

[26] L. Frenzel, "The Language Toolkit: An API for Automated Refactorings in Eclipse-based IDEs," Eclipse Magazine, January 2006. [Online]. Available: http://www.eclipse.org/articles/Article-LTK/ltk.html

[27] Y. Crespo, C. López, and R. Marticorena, "Un Framework para la reutilización de la definición de refactorizaciones," in *Actas JISBD'04, IX Jornadas de Ingeniería del Software y Bases de Datos, Málaga, Spain, ISBN 84-688-89830*, November 2004.

[28] C. López, R. Marticorena, Y. Crespo, and J. Pérez, "Towards a language independent refactoring framework," in *1st ICSOFT 06 International Conference on Software and Data Technologies. Setubal, Portugal. ISBN: 972-8865-69-4*, vol. 1, sep 2006, pp. 165–170. [Online]. Available: http://giro.infor.uva.es/Publications/2006/LMCP06

[29] JetBrains, "ReSharper:: The Most Intelligent Add-In to Visual Studio," JetBrains, 2007. [Online]. Available: http://www.jetbrains.com/resharper/

[30] D. E. Inc., "Refactor Pro for Visual Studio .NET," 2007. [Online]. Available: http://www.devexpress.com/Products/NET/IDETools/Refactor/

[31] I. Whole Tomato Software, "Visual Assist X for Visual Studio," 2007. [Online]. Available: http://www.wholetomato.com/

[32] Xref-Tech, "Xrefactory - A C/C++ Development Tool with Refactoring Browser," July 2007. [Online]. Available: http://xref-tech.com/xrefactory/main.html

[33] R. E. Johnson and W. F. Opdyke, "Refactoring and aggregation," in *Object Technologies for Advanced Software, First JSSST International Symposium*. Springer-Verlag, 1993, vol. 742, pp. 264–278. [Online]. Available: citeseer.ist.psu.edu/johnson93refactoring.html

[34] Omnicore, "X-develop - multi-language cross-platform IDE," 2007. [Online]. Available: http://www.omnicore.com/en/xdevelop.htm

[35] S. Tichelaar, S. Ducasse, S. Demeyer, and O. Nierstrasz, "A Meta-Model for Language-Independent Refactoring," in *Proc. International Workshop on Principles of Software Evolution (IWPSE)*. IEEE Computer Society Press, 2000, pp. 157–169. [Online]. Available: http://citeseer.nj.nec.com/tichelaar00metamodel.htm

[36] S. Ducasse, M. Lanza, and S. Tichelaar, "Moose: an Extensible Language-Independent Environment for Reengineering Object-Oriented Systems," in *Proc. Int'l Symp. Constructing Software Engineering Tools (CoSET)*, June 2000.

[37] ——, "The Moose Reengineering Environment," *Smalltalk Chronicles*, 2001. [Online]. Available: citeseer.ist.psu.edu/ducasse01moose.html

[38] O. Nierstrasz, S. Ducasse, and T. Gîrba, "The Story of Moose: An Agile Reengineering Environment." in *ESEC/SIGSOFT FSE*, M. Wermelinger and H. Gall, Eds. ACM, 2005, pp. 1–10. [Online]. Available: http://dblp.uni-trier.de/db/conf/sigsoft/fse2005.html#NierstraszDG05

[39] P. Van Gorp, H. Stenten, T. Mens, and S. Demeyer, "Towards Automating Source-Consistent UML refactorings," in *UML*, 2003, pp. 144–158.

[40] P. Van Gorp, N. Van Eetvelde, and D. Janssens, "Implementing Refactorings as Graph Rewrite Rules on a Platform Independent Meta model," in *Proceedings of 1st Fujaba Days*, october 2003. [Online]. Available: citeseer.ist.psu.edu/vangorp03implementing.html

[41] Y. Crespo, "Incremento del Potencial de Reutilización del Software mediante Refactorizaciones," Ph.D. dissertation, Universidad de Valladolid, 2000, available at http://giro.infor.uva.es/docpub/crespo-phd.ps.

[42] C. López and Y. Crespo, "Definición de un Soporte Estructural para Abordar el Problema de la Independencia del Lenguaje en la Definición de Refactorizaciones," Departamento de Informática. Universidad de Valladolid, Tech. Rep. DI-2003-03, septiembre 2003, available at http://giro.infor.uva.es/docpub/lopeznozal-tr2003-03.pdf.

[43] R. Marticorena and Y. Crespo, "Dynamism in Refactoring Construction and Evolution. A Solution Based on XML and Reflection," in *3rd International Conference on Software and Data Technologies (ICSOFT)*, July 2008, pp. 214 – 219.

[44] N. Tsantalis and A. Chatzigeorgiou, "Identification of extract method refactoring opportunities," in *CSMR*, A. Winter, R. Ferenc, and J. Knodel, Eds. IEEE, 2009, pp. 119–128.

[45] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Trans. Softw. Eng.*, vol. 30, no. 2, pp. 126–139, 2004.