

ASSISTING REFACTORIZING TOOL DEVELOPMENT THROUGH REFACTORIZING CHARACTERIZATION

Raúl Marticorena, Carlos López

Language and Informatic Systems, University of Burgos, EPS Campus Vena Edificio C, Burgos, Spain
rmartico@ubu.es, clopezno@ubu.es

Javier Pérez, Yania Crespo

Department of Computing, University of Valladolid, ETSII Campus Miguel Delibes, Valladolid, Spain
jperez@infor.uva.es, yania@infor.uva.es

Keywords: Refactoring tools, characterization, IDE, object-oriented programming.

Abstract: Tool support for refactoring is widespread nowadays. The most widely known IDEs include refactoring support, and many refactoring-specific tools are also available. Developers are aware of refactoring activities and they do refactor their applications even manually or in an assisted way. For the users of refactoring tools, the current state of the art is well documented in refactoring catalogs, where objectives, motivation, mechanisms, etc. are defined. There are also available collections of criteria to detect refactoring opportunities; compilations of guidelines to help decide when and how to apply refactorings. However, refactoring tool implementors can not only base their work on the documentation available in refactoring catalogs because they do not offer clear guidelines on how to build the tools to automate these refactorings. Implementing refactoring tools without any guidance, involves reasoning about which will be the better plan to implement refactoring operations in the tool, their complexity or their graphical interface design, etc. This paper introduces a refactoring characterization, and proposes how it can help refactoring tool implementors on making decisions.

1 Introduction

Refactoring has become a well known and useful technique for software developers in a short time. Since 1992, where W. Opdyke (Opdyke, 1992) stamped the term and defined which can be considered the first refactoring catalog, advances in refactoring have been increased exponentially.

Fowler *et al.* published the first book on refactoring (Fowler, 2000) which spreads the seed all around. This book contains the definition of one of the most known refactoring catalogs¹. Other well known catalogs can be found in (Kerievsky, 2004), (Ambler and Sadalage, 2006), (Lippert and Roock, 2006) and (Meszaros, 2007).

The impact of refactoring is so that developers are aware of refactoring activities and they do refactor their applications even manually or in an assisted way. Nevertheless, authors agree that the activities concerning refactorings must be automated (Mens and

Tourwé, 2004) in order to succeed. The most widely known IDEs include refactoring support, and many refactoring-specific tools are also available. Implementing refactoring tools involves reasoning about which will be the better schedule to implement refactoring operations in the tool in a rational and effective way. This includes issues on reusability, complexity and graphical user interface design.

In this work, we will present a characterization of refactoring operations. It will provide some objective criteria, that should drive decisions in the refactoring tools development process, to developers of refactoring-specific tools or IDEs' refactoring modules.

In the remainder of this paper, some related works on refactoring implementation and refactoring characterization approaches are detailed in Section 2. Section 3 describes the proposed characterization based on features. Section 4 provides the characterization of the Fowler's catalog and shows how the characterization was used to help making decisions in the

¹See also <http://www.refactoring.com>

development of a refactoring tool. Section 5 concludes and describes some future work.

2 Related Works

In (Crespo and Marqués, 2001), we find an analysis of refactorings. This analysis is based on a classification, in order to characterize and to compare refactorings from several works. The characterization is made from seven features:

motivation: reuse, requirement changes or maintenance.

direction: vertical (inheritance), horizontal (client).

effects: preserve clients, preserve objects, aggressive.

consequences: global, localized.

initiated: by inference, by demand.

human interaction: automatic, semiautomatic, manual.

target: analysis, design, implementation artifacts.

Some conclusions can be extracted from this work, that helps to infer refactoring properties. However, this work is performed from a high-level point of view that is not useful to solve refactoring implementation problems at a lower level of abstraction.

In (Zannier and Maurer, 2003), a new refactoring characterization is presented. This characterization is organised around three complexity levels: atomic, sequential and complex. Basic object-oriented concepts—package, class, method, field, etc.—define a feature named scope. Refactorings are then classified using the scope of the input and output elements of the refactoring. This approach has some advantages: it is easy to apply and it does not depend on implementation details, therefore it can be applied with several goals. Nevertheless, it defines just three large groups of refactorings. Some additional implementation decisions are difficult to make, due to the lack of a more fine-grained classification, which could allow refining these big groups.

On the basis of (Crespo and Marqués, 2001), (López et al., 2006) proposed a characterization aimed at helping in refactoring definition. The main features of this characterization are:

knowledge requirement: required design patterns.

target: design models, code implementation.

inputs: system, class, attribute, method, instruction, entity.

preconditions: system, class, attribute, method, instruction, entity.

actions: create, update, delete.

postconditions: system, class, attribute, method, instruction and entity.

Additionally, it takes into account low level details, such as the number or type of the elements used in preconditions, actions or postconditions. This approach depends on a particular solution and programming language and, therefore, it is difficult to reuse in other contexts. Gathering all that information is useful but, as a negative consequence, some decisions should be made before knowing certain details. As a consequence, it could be possible that some implementation changes could have effects on the refactoring characterization. This situation must be avoided.

3 Refactoring Characterization

The main goal which drives this characterization is to define simple features which can be extracted from refactoring descriptions (from recipes or semiformal definitions). These features should also be objective, in the sense that different people should obtain the same values from their descriptions. Therefore, we try to avoid a subjective issue.

Hence, instances of characterization features must be extracted from catalogs with a basic observation, hiding language and implementation details. Next subsections detail each feature.

3.1 Design and Language Issues

Design and language issues, required to apply certain refactoring, points out some clues about its complexity. Programmers can have different knowledge and experience on the programming language, making it easier or more difficult to apply, and to implement certain refactoring. We define three levels:

Basic: a refactoring uses common concepts of Object-Oriented (OO) languages (modules, classes, attributes, methods and inheritance). It talks about concepts of an OO programming language, which can be managed by any developers with basic knowledge of OO programming.

Advanced: advanced concepts of OO programming, such as contracts, exceptions, genericity, delegates and annotations. These concepts require a good knowledge of the programming language and skills in well known “good practices” of OO programming and design.

Design Pattern: the refactoring involves some details regarding design patterns, such as participants, roles, etc. defined in the design pattern specification. The designer or the programmer must be acquainted with design pattern catalogs and the different idioms for each language. In (Kerievsky, 2004), we can find a complete catalog of these refactorings.

Some examples of the application of this feature are: *Inline Method* (basic), *Replace Exception with Test* (advanced) and *Form Template Method* (design pattern) (Fowler, 2000). We assume that each level subsumes the previous ones. Those programmers which apply design patterns, are also knowledgeable on basic and advanced OO concepts.

This feature is interesting in education because it allows gradating contents.

3.2 Scope

This feature tries to observe the refactoring effects as seen by the users, without implementation details.

When we apply a refactoring, we can log three different change levels:

Intraclass (I): changes that do not affect other classes. Changes are contained within the class' scope. They modify the methods' body or those entities not referenced from other classes (e.g. private or non-exported methods). Contracts between classes remain without changes.

Client (C): client classes need to be updated (with changes in the contracts). Changes affect public or exported properties which are used by client classes belonging to the same or different packages, namespaces or workspaces.

Inheritance (H): classes with inheritance relationships are modified. The class hierarchy is affected by changes to the inherited properties. The basic effect is that changes must be usually propagated in both ways: ancestors and descendants.

A refactoring can have effects at the three levels. For example, *Add Parameter* (Fowler, 2000), apparently a simple refactoring, has effects on its own class (Intraclass), clients that use the method (Client), and ancestor or descendants that define or override the method (inHheritance).

3.3 Inputs

Inputs are the cornerstone of refactorings, because the users want to refactor "something" (refactoring target) and additional information is needed to drive the

refactoring execution. If the refactoring offers different alternative paths, user interaction is mandatory.

The refactoring inputs are important factors for characterizing refactorings. Basically, the programmer needs to identify:

Root input: (or target element) the software element that users select when invoking the refactoring, from visual inspection or aided detection. For example: *Self Encapsulate Field* (Fowler, 2000), needs only one target input, the field to encapsulate. The remaining refactoring actions do not need user interaction.

Additional inputs: it is very common that users provide some extra information about how to run the refactoring. They are defined as additional inputs. Wizards and other graphical components are built to assist the users to introduce these values. For example: *Move Field* (Fowler, 2000) needs a field as the root input, but the user has also to select the target class (additional input) where the field must be moved to. User interaction is mandatory in order to choose the target class.

We always need a root input that identifies which kind of refactoring can be applied. This point is used intensively in currently available tools. All of them give some sort of inference, showing the closed set of refactorings that can be applied to the element the user has selected. Therefore, we need to classify refactorings by their root element because this will be used to define the specific set of refactorings that should be enabled in our refactoring tool or IDE when some kind of element is selected for refactor.

One of the current problems that users face when applying refactorings, appears when the tool does not offer any assistance about the applicable set of refactorings. Some works such as (Emerson Murphy-Hill, 2008) point out this problem. As a consequence of this, people are reluctant to apply refactorings because they do not have a clear idea of what refactorings can be applied.

3.4 Actions

We define actions as operations that change the source code state. Refactorings usually involve many smaller actions. From reviewing refactoring mechanisms (Fowler, 2000) or even low-level refactorings (Opdyke, 1992), we can deduce that many actions are performed by executing one simple refactoring. These actions are linked to the implementation solution (metamodels, logic predicates, grammarware, XML, etc.), but for the sake of simplicity one abstract action must characterize the refactoring. In this work, refacto-

ring snapshots are considered, from the initial to final state, selecting one action that defines the refactoring.

Enlarging these catalogs by adding new refactorings, would make it very easy to add new actions. Nevertheless the characterization purpose is to give a simple and easy method. Therefore, we must generalize this feature to a closed set of actions, with a narrow number of options, where any refactoring could be classified.

Regarding source code edition, we can think about basic text operations as copy, paste, delete, find, etc. In concern of the refactoring process, we can think in similar terms, and reduce the number of applicable actions to (ordered from lower to greater complexity):

Add: it links a previously created element to its scope where it will be contained.

Rename it changes the name of an element identified in the system by name.

Remove it cuts the relationship between an element and the scope where it was contained.

Replace as the sequential run of **Remove** and **Add** in the same scope.

Move as the sequential run of **Remove** and **Add** in different scopes.

4 The Refactoring Characterization in Practice

In this section, some refactorings from the classical Fowler's catalog (Fowler, 2000) are characterized. Most of currently available tools and IDEs choose this catalog as the first reference to follow. It is an advantage because it provides a common point of view, with a well known vocabulary.

4.1 Applications on Building Graphical User Interface

A common problem with refactorings tools is to define a graphical library or API that allows to build refactoring user screens that guide refactoring execution. Inputs give an approximation of the refactoring complexity. Big number of inputs are more difficult to manage, with more complicated graphical user interfaces. This is a basic question, and it is relatively easy to give an answer using the **inputs** feature of the refactoring characterization. If we have a refactoring with only one input (root input), it can be classified as **automatic** and it will be initiated automatically from the environment selection by the programmer. On the other hand, if the refactoring needs some additional

parameters, we need user interaction and the refactoring is **semiautomatic**. In the first case, you do not need any dialog window or wizard, but with semiautomatic refactorings, some kind of graphical support must be available.

Next step is to use the root input type. Depending on the tool, certain software elements could be available or not. Since it is important to mark the root input type. If your tool only works at design level, refactorings with a block of instructions as input, do not have sense and could not be included.

From a reuse point of view, we need to know what refactorings have a similar GUI. These questions can also be answered with this characterization. For example, refactorings as *Replace Error Code with Exception*, *Replace Exception with Test*, *Replace Method with Method Object*, *Replace Parameter with Explicit Methods*, *Replace Parameter with Method* and *Separate Query from Modifier*, have the same inputs: one method (as root input) and one code fragment. All these refactorings will have a common GUI, since the refactoring GUI API could be reused in a great number of refactorings.

Other different case is the *Add Parameter* refactoring. We need one method (as root input), one type and one name to build the new parameter. There is no other refactoring that shares these inputs, as this concrete GUI could not be reused. From our previous experience (Marticorena et al., 2007) (Marticorena and Crespo, 2008), we used our characterization. Refactoring classification using the root input, also allows to define clearly the set of available refactorings. If the tool includes one concept (i.e. class, method, etc.), it is relatively easy to help users showing just refactorings with its suitable type input. Our Eclipse plug-in points current available refactorings depending on the current selected element. As can be seen in Fig.1, it is selected the `check` method in the text editor and *Available Refactorings* view shows the suitable refactoring list.

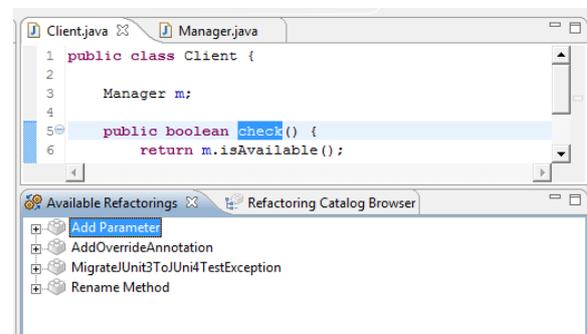


Figure 1: Running available refactorings.

4.2 Applications on Scheduling

A different and more complicated problem is to determine a schedule of refactoring implementation. If we have a set of refactorings, extracted from some catalogs, what refactorings should be implemented at first steps? Some criteria should be applied to maximize reuse and to minimize efforts.

In (Fowler, 2000), we find the “use” relationship, but these relationships are established by the author, from his own experience. Refactorings are grouped by functional criteria, but these groups do not define any implementation order.

We could take the “use” dependency graph extracted from (Fowler, 2000) and begin to build the refactorings using this graph following a topological sorting. This method is also used in bottom-up testing. Nevertheless, our proposal tries to give a simple method, with similar results, but without previous experience applying the refactoring set (usually in a manual way).

In order to divide the problem, we will face the refactoring implementation focused on functional groups (Fowler, 2000), instead of focusing on the complete set of refactorings. For example, in the group *Moving Features Between Objects*, we find seven refactorings. In Figure. 2, the graph of “use” relationships inferred from the catalog in (Fowler, 2000) is presented.

The process uses the previously explained criteria: *Design and Language Issues*, *Scope*, *Inputs (root and additional inputs)* and *Action* type. These criteria are used to schedule the refactoring implementations, from the first criterion to the last in a similar way to the clause `order by` in a SQL query (the order in the clause marks the precedence in the results). When two elements have same values in the i^{th} feature, it is applied the $i^{th} + 1$ to solve the tie.

An example of applying the proposed characterization to the refactorings is shown in Table 1 on the Category *Moving Features Between Objects* from (Fowler, 2000). The elements in the table are obtained in descending complexity. If we take the ordered refactorings in Table 1, it is very similar to one of the possible topological sorting extracted from the dependency graph of (Fowler, 2000) (see Fig. 2). Some refactorings, as *Remove Middle Man* or *Introduce Foreign Method*, without dependencies in the graph, can be scheduled following the proposed order.

About refactoring implementation schedule, our solution is based on repositories. These repositories contain functions and predicates that allow to ensemble the refactoring pieces (pre/postconditions and actions). Applying previous features explained in Sec.

3, we obtain those refactoring with lowest complexity as the first candidates to be implemented.

5 Conclusions and Future Work

The characterization presented in this position paper uses only static information, that can be extracted from an abstract refactoring definition, with some degree of language independence. This level of abstraction has some advantages. It offers a starting point to implement a refactoring, therefore the tool builder can be more confident on the initial decisions taken during the first steps of the refactoring tool development.

Nevertheless, using the proposed characterization can also trigger some questions about the refactoring definition itself. It can warn us to check whether the refactoring has been correctly defined, or even to reason about if two different characterization refers to the same refactoring. Our approach can be used as a tool to compare how different people understand the refactoring definitions included in refactoring catalogs.

Future work could include to apply this method to very different catalogs. For example, in (Ambler and Sadalage, 2006), refactorings are applied to databases. Concepts such as design and language issues, scope, inputs and actions, should be adapted to new domains. Although, we think that a similar characterization could be defined and applied with low effort, partially reusing the results of this work.

ACKNOWLEDGEMENTS

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

REFERENCES

- Ambler, S. W. and Sadalage, P. J. (2006). *Refactoring Databases: Evolutionary Database Design*. Addison-Wesley Professional.
- Crespo, Y. and Marqués, J. M. (2001). Definición de un Marco de Trabajo para el Análisis de Refactorizaciones de software. *Actas JISBD'01, VI Jornadas de Ingeniería del Software y Bases de Datos*, pages 297–310.
- Emerson Murphy-Hill, A. P. B. (2008). Refactoring tools: Fitness for purpose. *IEEE Software*, 25(5):38 – 44.
- Fowler, M. (2000). *Refactoring. Improving the Design of Existing Code*. Addison-Wesley.

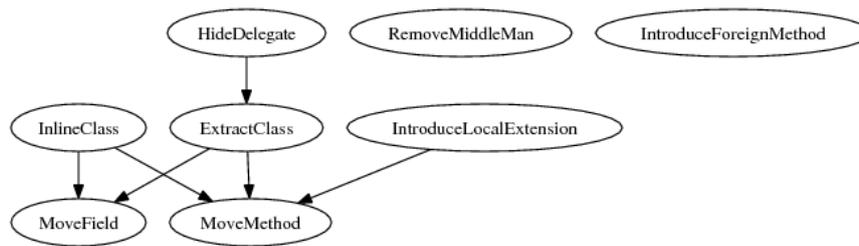


Figure 2: Moving Features Between Objects.

Table 1: *Moving Features Between Objects* Characterization.

Refactoring	Design and Language Issues	Scope	Root Input	Additional Inputs	Action
Hide Delegate	Design Pattern	ICH	1 Class	N Classes	Move
Remove Middle Man	Design Pattern	ICH	1 Class	N Classes	Remove
Introduce Local Extension	Basic	ICH	1 Class	1 Class, N methods	Add
Extract Class	Basic	ICH	1 Class	N Attributes	Move
Inline Class	Basic	ICH	1 Class	1 Class	Move
Move Method	Basic	ICH	1 Method	1 Class	Move
Move Field	Basic	ICH	1 Attribute	1 Class	Move
Introduce Foreign Method	Basic	IC	1 Class	N Instructions	Add

Kerievsky, J. (2004). *Refactoring to Patterns*. Addison-Wesley.

Lippert, M. and Rook, S. (2006). *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully*. Wiley, 1st edition. 0470858923 ISBN-13: 978-0470858929.

López, C., Marticorena, R., and Crespo, Y. (2006). Caracterización de Refactorizaciones para la Implementación en Herramientas. In *Actas JISBD'06, XI Jornadas de Ingeniería del Software y Bases de Datos, Sitges, Barcelona, 2006*. ISBN:84-95999-99-4, pages 538–543.

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

Marticorena, R., López, C., Crespo, Y., and Pérez, J. (2007). Reuse Based Refactoring Tools. In *WRT'07, 1st Workshop on Refactoring Tools, Berlin, Germany.*, pages 21–23. ECOOP'07.

Mens, T. and Tourwé, T. (2004). A survey of software refactoring. *IEEE Trans. Softw. Eng.*, 30(2):126–139.

Meszaros, G. (2007). *xUnit Test Patterns: Refactoring Test Code (Addison Wesley Signature Series)*. Addison-Wesley Longman, Amsterdam.

Opdyke, W. F. (1992). *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, IL, USA.

Zannier, C. and Maurer, F. (2003). Tool support for complex refactoring to design patterns. In Marchesi, M. and Succi, G., editors, *XP*, volume 2675 of *Lecture Notes in Computer Science*, pages 123–130. Springer.