Refactoring and its application to ontologies

Jordi Conesa, Universitat Oberta de Catalunya, Spain

Antoni Olive, Universitat Politècnica de Catalunya, Spain

Santi Caballé, Universitat Oberta de Catalunya, Spain

ABSTRACT

Over the last years a great deal of ontologies of many different kinds and describing different domains has been created, and new methods and prototypes have been developed to search them easily. These ontologies may be reused for several tasks, such as for increasing semantic interoperability, improving searching, supporting information systems or the creation of their conceptual schemas. Searching an ontology that is relevant to the users' purpose is a big challenge, and when the user is able to find it a new challenge arises: how to adapt the ontology in order to be applied effectively into the problem domain. It is nearly impossible to find an ontology that can be applied as is to a particular problem. Usually they need to be adapted to make it more usable (e.g., delete irrelevant terms, adding additional constraint, etc.) and to adapt it to the conceptualization the user has in mind (i.e., restructuring it to represent the same semantics but with a better structure). Is in that context where the ontology refactoring takes special interest. Ontology refactoring is the process of modifying the structure of ontologies but preserving their semantics. Refactoring has successfully applied to several domains, but has not applied effectively to ontologies yet. This chapter tries to clarify what ontology refactoring is, what ontology refactoring is not, the interrelation among ontology refactoring and other previous techniques that have the same or similar semantics, and the similarities and differences of ontology refactoring with refactoring applied in other contexts. The chapter also presents a possible catalog of ontology refactoring operations that try to unify the operations defined from previous research with other names or refactoring operations from other contexts.

KEYWORDS

Ontology, ontology refactoring, Refactoring, ontology refactoring definition, ontology refactoring catalog

1 INTRODUCTION

Lately, lots of ontologies of many different kinds and describing different domains have been created and new methods and tools have been developed to search them easily. Maybe the most prominent of these tools is Swoogle[1], a web-based ontology search engine that searches over ten thousand available ontologies. Even though the facilities these new search technologies provide, finding an ontology that is relevant to a given purpose is a big challenge. And even, when we are able to find an ontology that is good enough to be reused, a new challenge arises: how to adapt the ontology in order to be applied effectively into the problem domain. It is nearly impossible to find an ontology that can be applied as is to a particular problem. In fact, once found an ontology it needs to be restructured to make it more usable (by deleting irrelevant terms, adding additional constraints…) and to adapt it to the conceptualization the user has in mind (by restructuring it to represent the same semantics but with a better structure) (Conesa J, 2008).

Evolution operations have played an important role in the software engineering field. Some of these operations aim to modify a program or a conceptual schema and can be used to add, modify or delete the functionalities of a system or to improve the quality of the system without modifying its semantics. These operations are called refactoring operations when they are applied to object-oriented artifacts or restructuring operations when applied to other kind of artifacts. Over the last decade, several researchers have attempted to use refactoring operations not only in software but at higher degrees of abstraction (Sunye, Pollet et al., 2001): databases, Unified Modeling Language (UML) models (OMG, 2003), Object Constraint Language (OCL) rules (OMG, 2003), and more shyly in ontologies. But not only refactoring operations allow for modifying the structure of something while keeping its semantics, other technologies propose similar evolution operations that can be considered as refactoring operations, such as some schema evolution operations of, program slicing operations…

Ontology refactoring is the process of modifying the structure of ontologies but preserving their semantics (Conesa J, 2008). Refactoring has successfully applied to several domains (Mens and Tourwe, 2004), but has not applied effectively to ontologies yet. The two main lacks of the application of refactoring to ontologies is the lack of an unambiguous and agreed definition of ontology refactoring and a catalog of the refactoring operations that makes sense for refactoring ontologies. Such agreed catalogs are present within other fields such as software refactoring[2] and database refactoring[3].

We believe that the refactoring work of other fields, in combination with other techniques such as program restructuring (Griswold and Notkin, 1993) or schema transformation (Batini, Ceri et al., 1992), may be used to successfully define ontology refactoring and identify a catalog of its potential operations within the context of ontologies. Therefore, the goals of this chapter are: 1) to define clearly what a refactoring is, 2) to analyze various techniques for restructuring programs and schemas in order to identify in what cases these techniques or their operations can be useful for ontology refactoring, and 3) to present an exhaustive catalog of ontology refactoring operations that includes, after being adapted, operations from refactoring in other fields (software, databases and UML models mostly) and from other schema transformation techniques.

---

[1] http://swoogle.umbc.edu/

[2] http://www.refactoring.com/catalog/index.html

[3] http://www.agiledata.org/essays/databaseRefactoringCatalog.html

In the next section, the reader will find a brief history of existing restructuring techniques and how they have evolved to become the refactoring techniques we have today. Section 3 defines ontology refactoring taking into account the definitions of refactoring in other fields. Later, a catalog of ontology refactoring operations is presented. Such a catalog has been created by adapting refactoring operations from other techniques or from the refactoring of other fields. Finally, section 5 concludes the chapter presenting our conclusions and lessons learnt.

2. FROM PROGRAM REESTRUCTURING TO REFACTORING PASSING THROUGH SCHEMA TRANSFORMATION

In the last decades, several operations with common purposes but different uses have been defined using different names: refactoring operations (Opdyke, 1992; Moore, 1996; Roberts, Brant et al., 1997; Fowler, 1999; Ambler, 2003), restructuring operations (Eick, 1991; Batini, Ceri et al, 1992; Assenova and Johannesson, 1996; Halpin, 2001), program transformation operations (Griswold and Notkin, 1993; Griswold, Chen et al., 1998), model transformation operations (Sunye, Pollet et al., 2001; Roberts, 1999; Correa and Werner, 2004), etc. This section examines chronologically the techniques behind these names in order to show that some of they pursue the same goal: modifying the structure of a given artifact without changing its semantics, while improving some of its quality factors. We will also see that most of these techniques were conceived to restructure programs and therefore they should be adapted if we want to use them to restructure ontologies.

2.1 Program restructuring

Program restructuring (Griswold and Notkin, 1993) is a technique developed in the, 1980s. Its purpose is to restructure programs without modifying their behavior. This technique differs from refactoring in that it only deals with non-object-oriented programs. The following paragraphs discuss the main restructuring techniques that appeared in the 1990s.

Griswold and Notkin (1993) presented a set of restructuring operations that improve the quality of a program while maintaining its behavior. These operations make it possible to change the name of variables, replace expressions with variables or variables with expressions, replace a sequence of commands with a function that executes them and a call to such a function, etc. The same paper showed how to automate the execution of these operations. The same authors also defined a framework that supports the user in the detection and execution of restructuring operations for large programs (Griswold, Chen et al., 1998).

There are other program restructuring approaches, but they are not applicable to ontologies due to the limitations they impose such as in (Bergstein, 1991), where a set of five operations for restructuring an object-oriented schema (or program) while maintaining its possible instances. This kind of preservation is called object-preserving, and means that all non-abstract classes of the two schemas have the same name and attributes, including the inherited ones.

2.2 Schema transformation

Schema transformation (Batini, Ceri et al., 1992) has been widely studied in the last two decades. The following lines describe schema transformation and summarize the most relevant work in this area that should be taken into account in defining ontology refactoring.

Schema transformations are applied to an initial schema and produce a resulting schema. These transformations can be classified based on how they change the information contained in the transformed schema (Batini, Ceri et al., 1992):

- *Information-preserving transformations*: The execution of the transformation preserves the information contained in the schema.
- *Information-changing transformations*: These may be further classified as 1) *augmenting transformations*: The resulting schema contains more information than the original one; 2) *Reducing transformations*: The resulting schema contains less information than the original one; and 3) *Non-comparable transformations*: Neither of the previous categories is true.

Schema transformation operations that belong to the first group may be regarded as refactoring operations because they maintain the semantics of the modified schema, i.e. the information it contains. We are interested in this kind of operation. Hereinafter, when we use the term schema transformation operations we are referring to information-preserving transformation operations.

Identifying whether two conceptual schemas define the same information (i.e. they are equivalent) is very difficult. Several authors have studied this topic and presented different equivalence proposals (Hofstede, Proper et al.; Batini, Ceri et al., 1992; Proper and Halpin, 2004) based on: sets, logics, contextual equivalence, substitution equivalence, etc. We believe that the most semantically coherent definition is:

> *Two conceptual schemas are equivalent if and only if whatever universe of discourse, state or transition can be modeled in one can also be modeled in the other. (Halpin, 2001)*

Unfortunately, it is very difficult, or even impossible, to demonstrate that two schemas are equivalent using this definition. Therefore, most schema transformation approaches use the following definition of equivalence for substitution:

> *two schemas are equivalent if and only if any of them may be derived from the other* (Halpin and Proper, 1995)

From the last definition we can infer that schema transformation operations are bidirectional. Therefore, a schema transformation operation can transform schema S1 to another schema S2 and undo the change by transforming schema S2 to schema S1.

Batini et al. (1992) defined a set of information-preserving schema transformation operations that improve the readability and conciseness of conceptual schemas (Batini, Ceri et al., 1992). These operations are:

1. *Deletion of redundant cycles of relationships*: This operation deletes paths made up of relationship types already defined by other relationship types.
2. *Addition/deletion of derived attributes/subsets*: This operation adds or deletes derived attributes or classes.
3. *Elimination of dangling subentities in generalization hierarchies*: This operation merges an entity with its subtypes. The application is only valid when the subtypes are empty and therefore they do not contain any field.
4. *Elimination of dangling entities*: An entity absorbs another entity related to it by relationship type.
5. *Creation of a generalization*: This operation creates generalization and specialization relationships between entities with shared attributes.

6. *Creation of a new subset*: This operation is executed when an entity E_1 participates in a relationship type with a minimum cardinality of zero. In such a case, a new subtype of E_1 is created that only contains the registers of E_1 that participate in the relationship type. The relationship type is also moved to the new subtype, and its minimum cardinality is changed to one.

Batini et al. (1992) also defined schema transformation operations that preserve the semantics of a transformed schema but which convert it to its third normal form. We consider these operations technology-dependent and therefore not relevant for this chapter.

(Eick, 1991) presented six schema transformation operations that make it possible to:

- Move attributes between the entity types of the conceptual schema related by relationship types and generalization/specialization relationships.
- Split a class in two and relate the two resulting classes using a relationship type.
- Create a new subtype or supertype of an existing entity type.
- Move the ends of the relationship types through the taxonomy of entity types.

(Assenova and Johannesson, 1996) compiled the most relevant information-preserving schema transformation operations and added a new operation to the list. They also analyzed how to apply these operations to improve the quality of a conceptual schema by studying the quality factors improved by the execution of each operation. Eight of the nine operations presented focus on restructuring relationship types, and only one focuses on restructuring entity types. As the authors mention in the paper, the presented list is incomplete and it would be useful to extend it with new operations.

Halpin (2001) defined a set of information-preserving schema transformation operations for ORM (Halpin, 1996) conceptual schemas. We believe that two of his operations are especially relevant to our work:

- Specialize or generalize a predicate: This operation replaces an n-ary relationship type with m relationship types with an arity of n-1. As mentioned above, this operation is bidirectional. Therefore, it also makes it possible to merge m n-ary relationship types into one n+1-ary relationship type.

- Absorb a relationship type: By applying this operation, a relationship type that restricts the possible values of another relationship type is absorbed by the latter.

All relevant operations presented in this section are included in the catalog of ontology refactoring operations of Section 3.

2.3 Software refactoring and its evolution

The concept of refactoring was born in the early, 1980s when Opdyke (1992) defined refactoring as "*a program transformation that reorganizes a program without changing its behavior"*.

Before Opdyke (1992), several authors presented the idea of creating a set of modification operations for object-oriented programs that preserve their behavior. There are some algorithms and heuristics that produce good[4] class organizations. Examples include (Bergstein, 1991), which proposed a set of

[4] A good class organization in this context means an organization that promotes code reusability, contains a minimum number of multiple and redundant inheritances, and other criteria, depending on the point of view of the author.

class-transformation operations that preserve the behavior of an object-oriented schema with certain structural limitations (discussed in the previous subsection); and Casais (1989), which presented a set of techniques for restructuring the inheritance of a class diagram without losing functionality.

In his thesis, Opdyke (1992) defined 23 primitive refactorings and showed how to compose them in order to create more complex refactoring operations. For each refactoring operation, this author defined a precondition that guarantees that the behavior of the program will not change after the operation is executed. Opdyke defined the software refactoring operations in terms of C++.

Later, Roberts (1994) weakened Opdyke's refactoring definition in order to be able to define new refactorings that do not preserve program behavior but may be useful for designers. Roberts also extended the refactoring definition with postconditions in order to simplify the creation of complex refactoring operations by sequencing simpler ones and decrease the amount of analysis needed to identify what complex refactoring chains can be applied.

However, we believe that the best software refactoring definition is the one presented by Fowler (1999):

- **Refactoring (noun):** A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.
- **To refactor (verb):** To restructure software by applying a series of refactorings without changing its observable behavior.

According to Philipps and Rumpe (2001), the definition of external or observable behavior of a given program depends of its functional requirements. For example, in real-time information systems, execution time is considered a functional requirement. Therefore, in that case many typical refactoring operations may be inapplicable because their changes may negatively affect the execution time and therefore violate the functional requirements and therefore change its observable behaviour.

Fowler (1999), in his book, defines 72 refactorings and briefly summarizes each of them, explaining, for each operation, under what conditions it is useful, its execution steps, and the validations that should be carried out after its execution in order to guarantee that the program's behavior is preserved. Fowler also defined the code smells, which are conditions that denote certain structures in the code that suggest (and sometimes cry out for) the possibility of refactoring. With each code smell, Fowler associates the set of refactoring operations that can be applied to improve the program.

Software refactoring became very important with the release of Extreme Programming (Beck, 1999), a programming technique characterized by repetitive and aggressive application of refactoring operations to a program.

Although refactoring was designed to be applied manually, some tools have been developed to perform it automatically, such as Refactoring Browser (Roberts, Brant et al., 1997), RefactorIT[5], Together ControlCenter[6], etc.

Despite the contribution of the code smells, knowing when and where to apply refactoring operations is still a big challenge. Finding what refactoring operations can be applied effectively and where is

[5] http://www.refactorit.com

[6] http://www.togethersoft.com/products/controlcenter/index.jsp

what is known as detecting refactoring opportunities. When detecting refactoring opportunities the following questions should be answered:

- What refactoring operations can be applied effectively: the refactoring operations whose execution will improve the program should be obtained.
- Where they can be applied: in what parts of the program the previous refactoring operations can be applied effectively.
- In which order they should be applied: refactoring operations should be sequenced when several refactoring operations can be applied effectively at the same moment. Inferring in what order they should be applied is not easy since it will depend on the quality factor we want to improve and the application of one operation may imply that another operation cannot be applied anymore and therefore should be discarded.
- What refactoring operations should not be applied: since refactoring operations are unidirectional there are operations that undo the modifications done by other operations. This should be taken into account in order to avoid cycles in the execution of refactoring operations.

In order to completely automate the refactoring process, we must:

1. Automatically determine what operations can be applied and to what part of the schema, and
2. Execute them automatically.

As aforesaid, the first step is known as identifying the refactoring opportunities. The second step has been studied intensely in recent years (Roberts, Brant et al., 1997; Roberts, 1999), and is now close to being solved.

The detection of refactoring opportunities is non-deterministic and a lot of semantic information is required to identify the best refactoring operation for each case. Hence, this detection is only possible in some cases, and even in such cases its automation is not trivial.

For all these reasons, the refactoring process cannot be totally automated and require designer/programmer intervention in order to be applied successfully. The most sophisticated tools detect some refactoring opportunities and use them to propose refactorings, but it is usually the programmer who in the last term has to hand-pick what refactoring operations to apply and the order in which they are applied.

Several approaches to automatically selecting and applying certain refactoring operations in certain contexts and cases have appeared (Ducasse, Rieger et al., 1999; Kataoka, Ernst et al., 2001; Simon, Steinbrukner et al., 2001; Gorp, Stenten et al., 2003; Koru, Ma et al., 2003; Tourwe and Mens, 2003; Xing and Stroulia, 2003; Mens and Tourwe, 2004; Rysselberghe and Demeyer, 2004; Grant and Cordy, 2003; Simon, Steinbrukner et al., 2001; Kataoka, Ernst et al., 2001; Tourwe and Mens, 2003; Koru, Ma et al., 2003).

2.4 Conceptual schema refactoring

Most of current integrated development environments support the refactoring of programs. These tools work well for small granularity refactorings such as the extraction of a fragment of source code to a new method (*extract method*) or the one that renames the member of a class (*rename member*), but

they are neither intuitive nor easy to work with for big refactorings such as when a class is split into several classes and its original attributes and methods distributed between them (*extract hierarchy*).

On the other hand, inconsistencies can appear between the source code and its specification. To solve these inconsistencies, refactoring operations can be applied on a higher abstraction level (the level of specification, analysis or design) and such changes automatically propagated to the lower abstraction levels. Several authors have tried to adapt software refactoring to conceptual schemas, such as Sunye et al. (2001), the people of the Refactoring Project group (Gorp, Stenten et al., 2003; Tourwe and Mens, 2003; Mens, 2004 and Porres, 2003).

Some authors use a tool to perform refactoring operations on UML models (Astels, 2002; Marko Boger and Fragemann, 2002). This tool maintains consistency between the UML diagrams and the associated source code. Other authors, such as Bottoni, Parisi-Presicce et al. (2002), have proposed an inverse technique – that is, applying refactoring operations directly to the source code and defining techniques to propagate the changes to the associated conceptual schemas to maintain consistency.

Even though these proposals take conceptual schemas into account in the refactoring process, their vision of a conceptual schema is too simple. They see it as a very low-abstraction-level schema, usually represented by a set of UML models that directly represent a program. Using the MDA approach (Kleppe, Warmer et al., 2003) we can define these models as platform-specific models (PSM).

Table 1 summarizes the main techniques of conceptual schema refactoring. This table describes the purpose of such approaches, the artifacts they refactor, the abstraction level in which they work, the refactoring operations they define and their drawbacks in our context: the need for a refactoring catalog for ontologies.

Table 1: Summary of the main approaches to conceptual schema refactoring

Approach	Purpose	Artifact / Abstraction level	Operations defined	Drawbacks
(Sunye, Pollet et al., 2001)	To define the refactoring of UML models	UML class diagrams and state machine diagrams / Design level	*Insert generalization element (GE), Delete GE, Move method, Generalize element* and *Specialize element*	Preconditions and postconditions not fully specified Too simple to be applicable to analysis levels
(Marko Boger and Fragemann, 2002)	To define a tool to refactor UML models	UML state machines, activity and class diagrams / Design level	*Rename method, Move up method, Rename class* and *Merge states*	Not much formalization The available document has little and informal information, and the supposed full document is written in German

Approach	Purpose	Artifact / Abstraction level	Operations defined	Drawbacks
(Zhang, Lin et al., 2005)	To define transformation operations at the metamodel level	Any model that can be expressed as an instance of a metamodel: Petri nets, quality models, UML models, etc. / Design level	Subset of Fowler's catalog: *Add class, Extract superclass, Extract class, Remove class, Move class, Rename class, Collapse hierarchy, Add attribute, Remove attribute, Rename attribute, Pull up attribute* and *Push down attribute*	Not focused on defining refactoring operations The defined operations are too simple
(Judson, Carver et al., 2003)	To use patterns to define transformation operations at the metamodel level	Any UML diagram / Any abstraction level	None	Not focused on defining refactoring operations
(Porres, 2003)	To define refactoring operations as transformation rules and show how to automate and define them	Any UML model / Any abstraction level	*Encapsulate attributes, AddSetter and AddGetter.*	It defines how to formalize refactoring operations, but not their catalog or opportunities The user needs to be fully familiar with the UML metamodel
(Gorp, Stenten et al., 2003)	To extend the UML metamodel to link the source code with UML models. This link is used to define and automate two refactoring operations	UML models / Design level	*Extract method* and *Pull up method*	The metamodel extension became useless after the addition of Action Semantics in UML 1.5, which subsumed the proposed metamodel

Approach	Purpose	Artifact / Abstraction level	Operations defined	Drawbacks
(Correa and Werner, 2004)	To adapt the concept of refactoring to OCL integrity constraints	UML integrity constraints / Any abstraction level	*Add variable from expression, Split AND expression, Add operation definition from expression* and *Replace expression by operation call*	It does not study how the modification of UML models affects OCL integrity constraints, define an exhaustive list of OCL refactoring operations, or create any new operations (they are all adapted from Fowler's operations)

Roberts (1999) offered a definition for model refactoring:

> *A model refactoring is a pair R = (pre;T) where pre is the precondition that the model must satisfy, and T is the model transformation.*

This definition takes the operational perspective on refactoring. The definition allows certain operations that do not preserve the semantics of a refactored model to be considered refactoring operations.

Sunye et al. (2001) were the first to define the refactoring of UML models by defining a set of refactoring operations applicable to class diagrams and statecharts. However, they only explicitly formalized and defined the refactoring operations that deal with state machines. Their paper was the first to show refactoring as an analysis and design technique rather than a technique that was only applicable to source code. However, the authors did not study how the execution of refactoring operations affects the integrity constraints of the refactored schema, which in the case of the UML are expressed using OCL (OMG, 2003). Finally, the authors classified refactoring operations in five groups, based on the kind of action they perform: insertion, deletion, movement, generalization and specialization. For each group, they discuss the following:

- The refactoring operation *Insert generalizable element* adds a new element between two adjacent elements of a taxonomy (parent and child). Their operations apply the *generalizable element* concept to both classes and associations. It is not clear how this can be done and under what conditions it is allowed for associations.
- The operation *Remove generalizable element* is the inverse of the previous operation. It deletes an element without changing the behavior of the taxonomy. The operation also links the subtypes of the deleted element to its supertypes. This operation cannot be executed when the element is directly or indirectly referred to in other diagrams.
- The operation *Move* is used to transfer a method from one class C_1 to another class C_2. After moving the method to C_2, a new method that calls the moved method should be created for C_1. The following conditions must be guaranteed before this operation is applied:
    - A navigable binary association with a multiplicity of one must exist between C_1 and C_2, and

- The code of the operation to be moved must not refer to the attributes of C_1.

- *Generalization* refactoring may be applied to elements contained in classes, such as attributes, methods, operations and association ends. Since private characteristics are not accessible to the subclasses, they cannot be moved. Before moving an element, all subtypes of the destination class must have defined the characteristic to be moved.
- *Specialization* refactoring is the inverse of the previous operation. It consists in sending an element to the subtypes of the class to which it belongs.

In presenting the above operations, the authors do not formally define their preconditions and the textual conditions they present are incomplete. Furthermore, their paper does not take into account how to propagate refactoring changes to other UML models or to integrity constraints (even graphical ones such as cardinalities, disjointness and completeness), the refactoring opportunities that allow us to determine what refactoring operations may be applied, or the bad smells that indicate when a model should be refactored.

Hence, obtaining a catalog of refactoring operations from the aforementioned paper is difficult. Furthermore, it is unclear whether all the presented operations are refactoring operations, because we believe that some of them are closer to evolution schema operations: *Add attribute, Add method, Delete attribute, Delete class*, and so forth. Other operations can be seen as adaptations of Fowler's refactoring operations, such as the operation *move* which can be seen as an adaptation of the *Move method* operation or the operation *Generalization* which can be seen an adaptation of Fowler's *Pull up method/attribute* operations.

Refactoring presented by Sunye et al. (2001) works at design level, which means that it assumes the UML diagram is a direct representation of a program and therefore avoid using high level abstraction constructs such as ternary associations, multiple inheritance, multiple classification, association classes, etc. Applying refactoring operations on design level has its advantages, but applying them to a higher abstraction level provides even more benefits because an improvement on a higher abstraction level tends to generate *n* improvements at design level, with *n* greater than 2.

Ever though the validity of conceptual schema refactoring was demonstrated, efforts have focused on refactoring automation and in particular on how to represent refactoring operations to support a CASE tool to automate refactoring opportunities and execution. For example, Marko Boger and Fragemann (2002) defined various refactoring operations for static and dynamic UML models and presented a browser that supports the execution of the presented refactoring operations.

In the same direction, Zhang, Lin et al. (2005) presented a framework for automatically executing refactoring operations on models. This framework works with metamodels and can therefore refactor other kinds of models, such as Petri nets, service quality models and so forth. With UML, this framework can execute a subset of the operations defined by Fowler (1999): *Add class, Extract superclass, Extract class, Remove class, Move class, Rename class, Collapse hierarchy, Add attribute, Remove attribute, Rename attribute, Pull up attribute* and *Push down attribute*. Moreover, the tool allows to define new refactoring operations using a language called *Embedded Constraint Language* (ECL) (Gray, 2002), which is an extension of OCL that allows actions on conceptual schemas to be defined. This tool directly adapts software refactoring operations without taking certain model characteristics into account. It therefore has some drawbacks, such as the lack of refactoring operations to deal with association ends (moving up and down associations in a taxonomy for example). In consequence, basic operations such as *Collapse hierarchy* (which merge a class with its subtypes) cannot be applied to a class that participates in associations.

Judson, Carver et al. (2003) defined model transformation operations using metamodels. They created patterns at the metamodel level that graphically define various model transformation families that can be systematized. Although this approach allows us to write refactoring operations for all kinds of

UML models, it focuses on representing transformation operations through patterns rather than defining a catalog of refactoring operations.

Porres (2003) proposed implementing conceptual schema refactorings as a collection of transformation rules. Each transformation rule is defined by five elements: its name, a comment, a sequence of parameters, a guard condition that defines when the rule can be applied, and the body of the rule, which implements its effect. Porres' paper also defined a simple algorithm that simulates the intuition of the designer in the application of some refactorings. The algorithm detects the refactoring opportunities and sequences them automatically using some heuristics. The SNW tool (Porres, 2002) implements Porres' approach and allows editing UML models and creating, deleting and modifying refactoring operations.

There is a very well-known research group at the University of Antwerp called *Refactoring Project*[7] very prominent in the area of model refactoring. Its main goal is to design tools that make refactoring easier, more usable and more automatic. The group uses graph-rewriting algorithms to automate refactorings. Specifically, models are rewritten as graphs and then tools of graph management are used to detect refactoring operations that can be applied. One problem of such an approach is that the graphs of models are too large and therefore it is quite difficult for designers/programmers to work with them interactively. To solve this problem, Eetvelde and Janssens (2004) presented a method to restructure graphs hierarchically, making them more usable and readable for the user. In Gorp, Stenten et al. (2003) they defined a way to specify refactorings of UML models. To do so, an ad-hoc extension of the UML metamodel was performed to link the source code of an application with the UML diagrams that represent it. This extension became unnecessary after UML version 1.5 because the addition of Action Semantics allowed such links to be represented. Refactorings are defined by operations composed of a precondition, a postcondition and a bad smell condition, which is specified as an OCL operation that returns a value according to a certain metric. This value determines how close a given element (the one used as a parameter of the operation) is to being refactored. With this information, the system can find and execute certain refactoring operations semi-automatically, using the following algorithm:

1. The user specifies a threshold value for each source code smell of the refactoring operations
2. A tool evaluates all source code smell operations for all parameters.
3. A refactoring may be executed on a given set of model elements when the value of the source code smell operation is greater than the threshold value and its precondition is true.
4. After each refactoring operation is executed, its postcondition is evaluated to determine whether the refactoring operation has been executed successfully and maintains program behavior.

This approach can detect some refactoring opportunities for conceptual schemas and their associated source code. However, it is not clear whether all bad smells can efficiently be represented as OCL operations.

2.5 Integrity Constraints Refactoring

One relevant part of a conceptual schema is its integrity constraints. The integrity constraints define the conditions that the instances of the schema have to satisfy in order to be correct. Integrity constraints have been the most forgotten element in the refactoring of conceptual schemas. Correa and Werner are the first who proposed refactor integrity constraints (Correa and Werner, 2004). In particular, their work deals with integrity constraints written in the OCL language. The authors adapted a few classical software refactoring operations (Fowler, 1999) to OCL and presented a set of

[7] http://www.win.ua.ac.be/~lore/refactoringProject/index.php

OCL smells for these operations. The OCL smells are equivalent to Fowler's code smells: rules that imply that some OCL constraint can probably be improved through refactoring.

The OCL smells presented in this paper are:

1. *Magic literal*. It detects when an integrity constraint uses a literal within the constraint source.
2. *And chain.* It detects when a constraint consists of two or more subconstraints linked by the operator *and*.
3. *Long journey*. It detects when an OCL expression traverses a large number of associations.
4. *Rules exposure*. It detects when the business rules are specified in the postconditions or preconditions of the system operations.
5. *Duplicated code*. It detects when OCL expressions are duplicated throughout the conceptual schema.

Correa and Werner also defined five operations that deal with the abovementioned OCL smells:

- *Add variable from expression*: It adds a variable declaration to an OCL expression.
- *Replace expression by variable*: It replaces part of an OCL expression with a variable that explains its content.
- *Split AND expression*: It splits an integrity constraint made up of two or more constraints connected by *ands*.
- *Add operation definition from expression* and *Replace expression by operation call expression*: These operations deal with operations and have the same function as the first and second operations listed above.

As mentioned above, this paper only adapts part of software refactoring to OCL in order to improve the quality of OCL constraints. The list of OCL smells and operations is incomplete because there are other software refactoring operations whose adaptation to OCL might be very useful, such as the operations *Substitute algorithm* and *Remove double negative*. The former replaces a piece of the source code with another code with the same meaning; the latter eliminates double negatives used in a source fragment, thereby greatly improving its readability.

2.6 Refactoring and quality

Software refactoring operations restructure programs in order to improve their quality. Software quality is relative and depends on the point of view from which the program is examined. For example, for one person a quality program is one that performs its tasks as quickly as possible, but for another a quality program is easy to understand and modify.

Refactoring operations do not affect all quality factors equally. A refactoring operation that improves the sharing and readability of code may worsen time performance; or an operation that improves execution time may worsen readability. In consequence, there is significant dependency between the quality factors we want to improve, the refactoring operations we need to apply and the operations we need to avoid, in order to improve the desired quality factors.

The relationship between refactoring operations and quality factors was not studied until refactoring had reached a certain maturity. Nevertheless, today it is one of the most prominent refactoring topics. The next paragraphs discuss the major works that deal with the relationship between the quality of software and refactoring operations. Other works about refactoring and quality not addressed herein are (Tahvildari, 2003; Zimmer, 2003).

Simon et al. were among the first researchers to take quality metrics into account in the refactoring process (Simon, Steinbrukner et al., 2001). They defined a framework that uses object-oriented metrics to detect certain refactoring opportunities for the operations *Move attribute*, *Move method*, *Extract class* and *Inline class*. In particular, they use distance-based cohesion metrics to determine when refactoring may be applied.

Du Bois proposed a technique for rewriting the postconditions of refactoring operations in order to infer automatically how the execution of a refactoring operation will affect software quality in terms of the desired metrics (Bois, 2004). Bois, Demeyer et al. (2004) adapted this technique to create a set of refactoring policies to take into account in the process of detecting refactoring opportunities. These policies were created to improve the cohesion and coupling of the system to be refactored. Du Bois' idea is promising, but it deals with few refactoring operations (five in particular) and only defines six policies that do not cover all cases.

Other works focus on how to detect refactoring opportunities and execute their associated operations taking into account the non-functional requirements of the system being refactored. In this context, Yu, Mylopoulos et al. (2003) presented a framework in which the designer used soft-goal interdependence graphs (SIG) (Chung, Nixon et al., 2000) to identify the refactoring operations that could be applied to a program in order to satisfy the non-functional requirements expressed in the SIG. SIGs are graphs that show the dependencies between goals, subgoals, resources, etc. The authors also presented a four-step algorithm for identifying and executing refactoring operations. This technique is totally manual. Moreover, it is difficult to apply due to two problems: 1) it is not clear which refactoring operations can improve a given aspect of the software, and 2) it is not clear how many operations are necessary to satisfy a given soft goal. Moreover, in the general case, the convergence of the process is not guaranteed because there may be operations that improve and worsen inverse concepts. In such cases, the application of a refactoring operation may satisfy a soft goal g_1 but violate another one g_2 and the application of another refactoring operation to satisfy the soft goal g_2 may worsen and therefore violate g_1 (which had previously been satisfied). In such a case, the algorithm will not converge.

2.7 Refactoring in other contexts

After software refactoring has become mature, refactoring has been applied to other contexts, such as requirements gathering (Yu, Li et al., 2004), databases, product lines (Critchlow, Dodd et al., 2003), aspects (Deursen, Marin et al., 2003) and even improving the binary code of executable programs to increase their performance.

Due to the close relationship between database refactoring and ontology refactoring we give special attention to the refactoring work in such a context. One of the first problems with database refactoring is the high degree of coupling that databases have with other sources, such as the source code of its information system, the source code of other information systems, the source code of the processes of loading and extracting data, backup processes, persistence layers, database schemas, scripts for migrating data, documentation and so on. This high degree of coupling makes it very difficult to:

1. Quantify the extent to which the application of a refactoring operation improves the quality of a database and the applications that use it, and

2. Apply refactoring, because it may be too expensive to identify, modify and rebuild all of the sources that use the refactored elements.

Furthermore, as Fowler said, refactoring a database implies restructuring not only the database schema but also the code of the applications that use the database (Fowler and Sadalage, 2003).

Fowler and the company Pramod Sadalage carried out a preliminary study on database refactoring. The study examined the application of agile techniques in the creation of the database of an information system at Pramod. The study concluded that database refactoring affects more people than software refactoring and therefore good communication is required between all stakeholders involved in the project (programmers, analysts, database administrators, etc.) in order to refactor the database successfully. This time, the definition that Fowler presents of database refactoring is too ambiguous and informal. In particular, they defined a database refactoring operation as any operation that can be executed on a database. This definition includes additive, subtractive and neutral operations. This seems to contradict our conceptual view of database refactoring: a modification of a database that maintains its behavior (the intension of the database) and the semantics of the data (the extension of the database). Obviously, Fowler's (1999) definition of database refactoring does not satisfy this statement because not all additive and subtractive operations maintain the behavior and data semantics of a database.

Ambler studied database refactoring in greater detail (Ambler, 2003) and defined database refactoring more coherently[8]:

> ***Refactoring*** *is the process of changing the database schema that improves its design while retaining both its behavioral and informational semantics.*

Ambler also defined a catalog of database refactoring operations divided into five categories:

1. *Structural refactorings*: Operations that modify the database structure.
2. *Architectural refactorings*: Operations that modify the database architecture.
3. *Data quality refactorings*: Operations that improve the quality of the data stored in the database. This improvement is made using codes, standard types or other means.
4. *Referential integrity refactorings*: Operations that modify how the referential integrity constraints of the database are validated.
5. *Performance refactorings*: Operations that improve the performance of the database.

Most of Ambler's refactoring operations are adaptations of the software operations, but he presents a very interesting set of new operations. However, we believe that some operations of the catalog should be revised because it is not clear whether they are information-preserving, as they may imply a change in the functionality or semantics of the data stored in the database. For example, the operation *Remove application specific constraint* deletes integrity constraints and is applied when a business rule associated with an integrity constraint contradicts any of the other applications that share the database.

2.8 Discussion

In this section we have seen the most prominent research in refactoring operations during last years. From program restructuring to database refactoring passing thru schema evolution operations, refactoring operations and conceptual schema operations. As aforesaid, most of the operations presented in these technologies are the same but with different names and few variations. Even though this disparity of names, the creation of a software refactoring catalog has establishes a kind of de facto standard naming of operations that posterior refactoring techniques such as database and conceptual schema refactoring tried to follow. The only difference we can find in the operations of each of the presented technologies are the kind of artifact to be refactored and whether the operation is bidirectional or unidirectional. For example schema transformation operations are bidirectional and therefore a schema transformation operation can transform schema S1 to another schema S2 and undo the change by transforming schema S2 to schema S1. Refactoring operations, however, are unidirectional. Therefore, if an operation transforms schema S1 to S2 (*inline class* for example), then

[8] http://www.agiledata.org

in order to transform schema S2 to S1 we need to execute the inverse operation, which is the operation (*extract class*). We also have seen that the big challenge in refactoring is not to execute the refactoring operations but to identify their opportunities, that is what refactorings to apply and where.

Even though there have been some attempts to use refactoring in the context of ontologies (Ostrowski., 2008; Ondrej et al, 2008; Conesa, 2004), its application to ontologies is quite immature. The main problems we find in the application of the refactoring to onotologies are two: 1) it is not defined what refactoring means in the contexts of ontologies, and 2) it lacks of a catalog of refactoring operations similar to the ones create for software or databases. For example, in Protegé we can find several functionalities for refactor ontologies. For example, version 4 of Protégé allows execute 13 refactoring operations. Even though some of them can be seen as an adaptation of software refactoring the name conventions they have used are quite different, making very difficult to understand what is the exact meaning of a given refactoring operation even when the user has experience in the refactoring field.

Table 2 shows a brief comparison between the techniques presented in this section and shows, for each technique, the artifact to be refactored, whether there exists an agreed definition of its purpose, whether the operations are bidirectional and if there exists an agreed catalog of operations.

Table 2: Summary of the main refactoring techniques dealt in the paper

	Refactored artifact	Agreed definition	Directionality of application	Existence of a catalog of operations
Program restructuring	Programs	Yes	Unidirectional	
Schema evolution	Conceptual schemas (usually database and OO schemas)	Yes	Bidirectional	Several small catalogs of operations
Software refactoring	Programs	Yes	Unidirectional	Yes
Conceptual schema refactoring	Conceptual Schemas (usually UML schemas)	No	Unidirectional	No
Database refactoring	Databases and related programs	Yes	Unidirectional	Yes
Ontology refactoring	Ontologies	No	Unidirectional	No

In conclusion, in order to easily apply refactoring to ontologies, a catalog that contains the refactoring operations applicable to conceptual schemas must be created. This catalog should be similar to the catalog of software refactoring[9]. Such a catalog should identify for each refactoring operation its preconditions (when refactoring can be applied to a given set of ontology concepts), the steps for executing the refactoring on ontologies (its effect) and its expected result (postconditions). Furthermore, we think a deeper study of conceptual schema refactoring is necessary to detect specific new smells of conceptual schema refactoring.

[9] http://www.refactoring.com/catalog/index.html

3 ONTOLOGY REFACTORING

Up to now, refactoring has been applied to a wide range of sources and artifacts: software, UML models, use cases, etc. Wikipedia[10] offers the refactoring definition that we feel best identifies its meaning regardless of the artifact to be refactored:

> ***Refactoring*** *is the process of rewriting written material to improve its readability or structure, with the explicit purpose of keeping its meaning or behavior.*

All artifacts that have been refactored up to now follow a grammar and can therefore be expressed as a textual representation, making the above definition applicable in all cases. However, each artifact tends to have its own definition of refactoring that clearly specifies the meaning of *meaning* and *behavior* in its context. For example, software refactoring is defined as *the process of restructuring software to make it easier to understand and cheaper to modify without changing its observable behavior.* Since the purpose of all software systems is to offer a predetermined behavior – that is, always returning the same outputs for a given set of inputs – the correspondence between the two refactoring definitions is clear. In fact, this definition is a specialization of Wikipedia's definition, because it has the same meaning but clearly specifies the quality factors that a software refactoring operation should improve and the definition of *meaning* and *behavior* of the software.

In the database field, Ambler (2003) redefined refactoring as *the process of changing the database schema that improves its design while retaining both its behavioral and informational semantics.* This definition is not very precise, but like the previous one, it specifies the quality factors to be improved (the design) and the meaning and behavior of the database, defined as the informational semantics (the same information base is expressed) and the behavioral semantics (triggers, stored procedures, etc.).

Ontologies are neither programs nor databases. Therefore, their refactoring definition should be refined, as in the aforementioned cases. The abstraction level of an ontology is higher than that of a database schema or a program. Therefore, all quality factors related to a particular implementation of a conceptualization or the representation of a domain population are irrelevant to the quality of an ontology, because its goals are essentially semantic. Hence, we can define ontology refactoring as: *the process of restructuring ontologies to improve their readability or structure, while preserving their relevant knowledge.*

Ontologies are designed to satisfy a set of requirements. A change to an ontology preserves its relevant knowledge if the knowledge necessary to satisfy its requirements is not deleted in the process. In other words, a refactored ontology should be capable of inferring the same knowledge as the original ontology. Therefore, the purpose of ontology refactoring is not to change the associated conceptualization (*conceptual change*) (Thagard.P, 1992) but rather the way in which the associated conceptualization is represented (*representational change*).

One of the most difficult tasks in the execution of refactoring operations is modifying the source code that refers to the refactored elements. For example, if we execute the refactoring operation *Change method* (Fowler, 1999) to change the name of the method *GetPersons()* to *GetPeople()*, all references to the method *GetPersons()* in the source code must be identified and replaced with references to the method *GetPeople()*. This problem also occurs in ontology refactoring, even when the ontology does not define behavioral information, because ontologies may contain general integrity constraints for example some SWRL rules. These constraints may refer to any concept of the ontology. Therefore, after a concept is changed, all integrity constraints that refer to that concept must be modified to maintain the syntactic consistency of the ontology.

[10] http://en.wikipedia.org/wiki/Refactor

Another problem in ontology refactoring is the fact that tests cannot be performed to check that the ontology's behavior has not been modified. Tests of this kind are necessary in software and database refactoring, but cannot be applied to ontologies because, as mentioned above, ontologies cannot be executed. However, a validation of some sort may be carried out on various ontology instantiations to check, after each execution of a refactoring operation, whether any instances have been lost as a consequence of the change. To perform this test, the refactoring operation must not change only the structure of the ontology but also its information base (instances of the ontology).

Figure 1: The catalog of ontology refactoring operations

4 A CATALOG OF ONTOLOGY REFACTORING OPERATIONS

As mentioned above, some works have dealt with the application of refactoring operations to conceptual schemas and ontologies. However, an exhaustive catalog of refactoring operations applicable to ontologies has not yet been defined.

This section presents a catalog of refactoring operations applicable to ontologies adapted from the fields of software refactoring, database refactoring, OCL refactoring, schema transformation and program restructuring. This catalog presents the operations defined in these areas that can improve certain aspects of an ontology or conceptual schema while maintaining its semantics. In the particular case of software refactoring, some operations have been rejected because they focus on implementation and therefore do not make sense on the abstract level of an ontology.

At first glance, the proposed operations may seem too simple and specialized to cause any improvement. However, as Fowler (1999) mentioned in his book, refactoring steps tend to be small, but they can be combined in sequences to construct advanced refactoring operations that can satisfy more ambitious goals and use higher-level tactics.

Since Fowler's classification of software refactoring operations has become the standard de facto in software refactoring, we used the categories of Fowler's classification in the catalog of ontology refactoring operations with classification purpose. We have extended fowler classification by adding a new level in order to deal with the refactoring operations that deal with general integrity constraints.
Figure 1 presents the taxonomy we use to classify refactoring operations. In particular, we group ontology refactoring operations in two main categories:

- Structural refactoring operations: These operations can refactor the structural elements of an ontology (concepts, relationship types, generalization relationships and instances and constraints embedded in the ontology language).

- Integrity constraint refactoring operations: These operations can refactor the dynamic elements of an ontology such as derivation rules written in SWRL or OCL and its general integrity constraints.

The first group of refactoring operations can be applied to any ontology, and the second group is only relevant in ontologies that contain general integrity constraints or operations. However, we have not considered the refactoring of operations because it has little practical application in today's ontologies.

We have kept certain refactoring operations even though the elements they deal with are theoretically not very relevant to ontologies because they belong to a lower abstraction level. Examples of such elements include directed relationship types and iterations. These operations have been kept because such elements exist in some current ontologies.

The catalog of software refactoring is currently the largest and best-documented refactoring catalog even created. Therefore, most of the refactoring operations in our catalog have been adapted from it. This catalog is the origin of all refactoring operations unless otherwise specified. When a single operation has been defined for more than one artifact, we use the definition that is closest to the operation's meaning in the context of ontologies.

4.1 Structural refactoring operations

Structural refactoring operations are those which modify the structural concepts of the ontology – that is, its classes, its relationship types, its instances, and the integrity constraints that are embedded in the ontology language, such as the inheritance relationships that determine that a concept is a subtype of another concept, or the cardinality constraints of a relationship type.

The structural relationship refactoring operations are classified within three main categories according to their purpose:

1. Moving features between concepts: operations that change the context in which the properties of an ontology (relationship types and integrity constraints) are defined.
2. Organizing data: operations that change the ontology structure in order to make it easier to work with its data.
3. Dealing with generalization: operations used to change the taxonomy or move ontology properties (relationship types and integrity constraints) through the taxonomy of the ontology. Note that both classes and relationship types may have taxonomical structures in ontologies.

The operations of each category are explained in the following lines.

Moving features between concepts

Move Integrity Constraint: This operation moves an integrity constraint from one entity type to another. This is an adaptation of Fowler's *Move method* operation.

Change relationship context: This operation changes the type of one of the participants in a given relationship type. The old and new participants in the relationship type cannot be related with a generalization/specialization relationship. This operation is based on Fowler's *Move field* operation. In our approach, we see an attribute or field as a special case of a binary relationship type.

Extract entity type: This operation splits an entity type in two and distributes its content between the two new entity types. It is based on Fowler's *Extract class* operation.

Inline entity type: This is the inverse of the operation *Extract entity type*. It merges two entity types into one new entity type. It is based on Fowler's *Inline class* operation.

Extract datatype: This operation changes an entity type into a datatype. Although datatypes are a particular case of entity types, we created this operation because they have a prominent role in ontologies. In the ontologies that incorporate the concept of datatypes, such as in OWL, this operation converts a class in a datatype and changes all the relationship types that dealt with the class to attributes, or data properties in the case of OWL.

Organizing data

Delete PartOf constraint: This operation deletes a *PartOf* integrity constraint related to a given binary relationship type. In the particular case of UML ontologies, this operation replaces an aggregation with an association. It is based on Fowler's *Change value to reference* operation.

Add PartOf constraint: This is the inverse of the operation *Delete PartOf constraint*. It is based on Fowler's *Change reference to value* operation.

Change unidirectional relationship type to bidirectional: This operation changes the navigability of a relationship type from unidirectional to bidirectional. In the particular case of ontology languages that do not support bidirectional relationship types, like OWL, this operation creates a new relationship type between the same participants but with inverse direction and define it as the inverse of the original relationship type. It is based on Fowler's *Change unidirectional relationship type to bidirectional* operation.

Change bidirectional association to unidirectional: it is the inverse of the operation *Change unidirectional association to bidirectional* and based on Fowler's *Change bidirectional relationship type to unidirectional* operation.

Replace enumeration with subclasses: This operation replaces an enumeration attribute used to determine the type of the instances of a given entity type *E* with a set of subentity types of *E*. Each of the entity types created represents one of the possible values of the deleted attribute. This operation was adapted from the software refactoring operation called *replace type code with subclasses*.

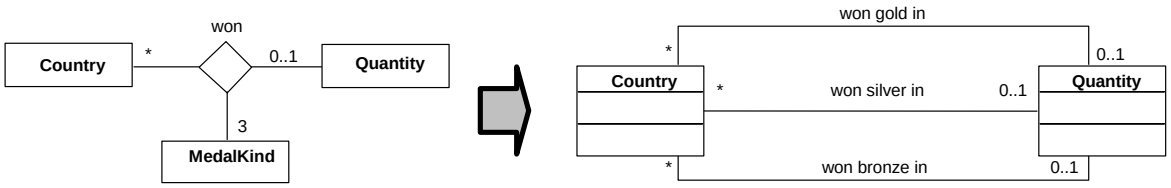


Figure 2: Example of the application of the refactoring operation Relationship type specialization.

Replace subclass with an enumerated type: This is the inverse of the previous operation. This operation replaces a set of subtypes of a common entity type *E* with a relationship type in *E* that represents the same information. An integrity constraint is created to restrict the number of possible values of the new relationship type; this number is equal to the number of deleted subtypes. This operation was adapted from the software refactoring operation *Replace subclass with fields*.

Extract ontology: Some ontology languages allow an ontology's knowledge to be grouped in small blocks based on its meaning, context or domain. For example, the OWL languages allow for defining small ontologies that can be reused in other context. Therefore, this operation splits a given ontology into two subontologies.

Move concept: This operation moves a concept that belongs to one ontology to another ontology.

Combine attributes: This operation merges several attributes into one. It was adapted from the database refactoring operation *Combine columns representing a single concept*. This operation is very useful when a designer creates relationship types too specific for the objectives of the ontology. For example, a designer may define phone numbers using three relationship types: one to identify the country, another to identify the city and finally the number without these two prefixes. Before the refactoring operation, three relationship type instances were needed to identify a phone number. For example, a phone number in Atlanta (USA) would be represented by 1, 404 and 1234567. After the refactoring operation, this phone number would be represented using only one relationship: 1404123456.

Split attribute: This is the inverse of the operation *Combine attributes*. It splits 1 attribute into *n*, with $n>1$. The types of the new attributes should be the same as the original or any of its subtypes. This operation is based on the database refactoring operation *Split column.*

Rename concept: This operation changes the name of a concept (an entity or relationship type) and it has been adapted from the database refactoring operations *Rename attribute* and *Rename table* (Ambler, 2003).

Apply standard types: Sometimes, different attributes that represent the same (or similar) concepts have different types. For example, a phone number may be represented by a *String* and a cell phone number by an *Integer*. Obviously, the quality of the ontology is improved if represent all phone numbers in the same way. This refactoring operation solves this problem, changing the type of different attributes that represent similar things until all of them have the same types or share a common supertype. In this example, this refactoring operation would change the end of one relationship type from *Integer* to *String* and therefore all phone numbers of the ontology would be represented in the same way. This operation has been adapted from database refactoring.

Relationship type specialization: This operation change one *n*-ary relationship type for *m* (*n*-1)-ary more specific relationship types. This operation is particularly useful when one participant in a relationship type can only have a predefined set of values known prior to the conceptualization phase. In such a case, the *n*-ary relationship may be replaced with *m* relationship types, where *m* is the number of possible values of the participant. Figure 2 shows a fragment of an ontology that represents the number and kinds of medals each country has won in the Olympic Games. The entity type *Medal Kind* can only have three instances: *Gold*, *Silver* and *Bronze*. Hence, we can apply this refactoring operation to replace the ternary relationship type *Won* with three binary relationship types: one for each possible instance of the *Medal kind* entity type. The new relationship types are called *Won gold in*, *Won silver in* and *Won bronze in* and represent the number of medals each country won of a each type. This operation was adapted from Halpin's schema transformation operations (Halpin, 2001).

Relationship type generalization: This is the inverse of the previous operation (*Predicate specialization*). It replaces *m n*-ary relationship types with similar semantics with one relationship type with an arity of *n+1*. The new participant in the relationship type is used to identify the semantics (previous relationship type) used in each instance of the relationship type. This operation was adapted from Halpin's operation *Predicate specialization* (Halpin, 2001).

External relationship specialization: This operation makes more specific a given relationship type based on the values of another relationship type. To do this, the first relationship type absorbs the second one. This operation was adapted from the schema transformation field (Halpin, 2001).

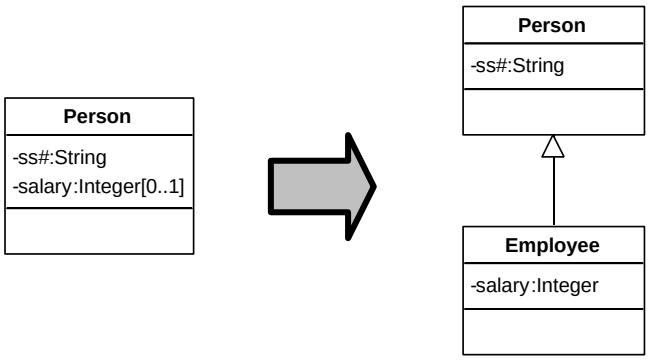
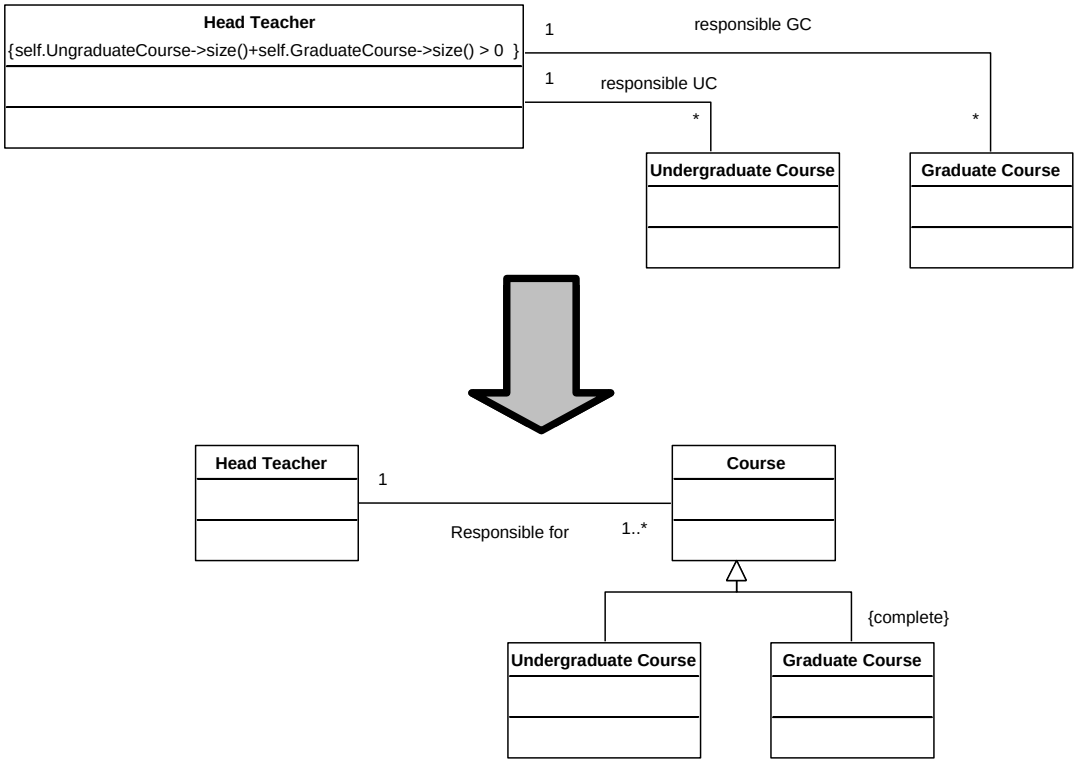


Figure 3: Example of Transforming Partial Relationship Type refactoring operation.

Transforming partial relationship type: This operation changes a partial relationship type (with a minimum cardinality of 0) into a total one (with a minimum cardinality of 1). To do so, a new subtype that contains the instances that used to participate in the partial relationship type is created. Thereafter, the relationship type is redefined by pointing the new created subtype and changing the minimal cardinality to one. Figure 3 shows an example of application of this refactoring operation to the partial attribute *salary*. This operation was adapted from the schema transformation area, specifically from the operation *Transforming partial attributes* defined in Assenova and Johannesson (1996).

Transforming partial relationship types that are total in union: Sometimes, an integrity constraint can represent that the union of two partial relationship types is total, which means that for each instance of a partial participant there is at least one instance of these relationship types that relates it. In this case, this operation makes sense and replaces the two partial relationship types with one total relationship type. It also creates a new entity type defined as the union of the entity types that participated in the previous relationship types. This refactoring operation also deletes the integrity constraint that indicated the totality of the union of the two relationship types. For example, the relationship types *ResponsibleGC* and *ResponsibleUC* shown in Figure 4 are partial, but the integrity constraint defined in the figure states that their union is total because a *Head teacher* is responsible for a course. By applying this operation the entity type *Course* is created as the union of *UndergraduateCourse* and *GraduateCourse*, and the two partial relationship types are replaced with the total relationship types



ResponsibleFor. This operation was adapted from Assenova and Johannesson's list of schema transformation operations (Assenova and Johannesson, 1996).

Figure 4: Example of Transforming partial relationship types that are total in union refactoring operation.

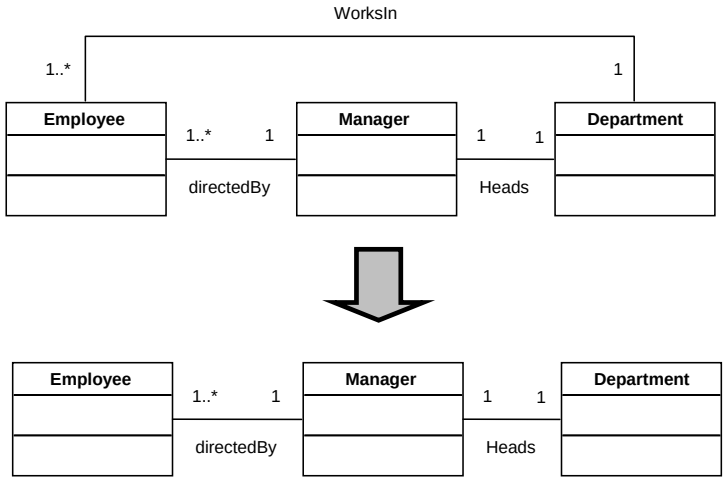


Figure 5: Example of refactoring operation for deleting relationship types redundancy.

Deleting relationship types redundancy: This operation, based on a schema evolution (Batini, Ceri et al., 1992), deletes redundant relationship types. To be redundant, two relationship types must share both the same participants and semantics. Therefore, this operation always requires designer intervention to determine whether the *n* relationship types, which are supposedly redundant, have the same semantics. For example, there are two paths between the entity types *Employee* and *Department* of Figure 5 with the following meanings: 1) the department where the employee works (*WorksIn*), and 2) the department the employee's manager directs (*DirectedBy->Heads*). If the employee's manager

works in the same department as all of his or her subordinates, then the two paths represent the same information and are therefore redundant. In such a case, we can apply this refactoring operation to delete the redundant relationship type *WorksIn*.

Defining derived concepts: This operation defines a new entity type or relationship type whose population is derived from the information base of the ontology. It can be applied, for example, to the relationship type *age* of an entity type *Animal* in order to state that the age of an animal is a derived relationship type obtained by subtracting the animal's birth date from today's date. This operation is based on the schema transformation operation *defining derived concepts* (Batini, Ceri et al., 1992).

Deleting derived concepts: This operation deletes a derived concept (entity type or relationship type). This is possible when derived concept is redundant in a conceptual schema. If it is not relevant, then its elimination does not imply any loss of ontology semantics.

Implicit subsets: This operation deletes redundant generalization paths. It was adapted from the schema transformation operations of Batini, Ceri et al. (1992).

Dealing with generalization

Pull up relationship type: Conceptually speaking, this operation replaces one participant in a relationship type with its supertype. This operation is based on the software refactoring operation *Pull up property end*, but it needs to be modified completely in order to adapt it to ontologies and take into account the possible integrity constraints that can affect it: disjointness and completeness of classes.

Push down relationship type: This operation replaces one participant in a relationship type with the subtypes of that participant. It is the inverse of the operation *Generalize relationship* and is based on the software refactoring operation *Push down property end*.

Pull up IC: Conceptually speaking, this operation replaces the context of a relationship type with the supertype of that context.

Push down IC: This operation replaces the context of a relationship type with the subtype of that context. This operation, which we created, is the inverse of the integrity constraint *Pull up IC*.

Extract subconcept: This operation creates a new concept as a subtype of an existing one and moves some of the existing concept's properties to the new one. This operation is based on the software refactoring operation of the same name.

Extract superconcept: The operation creates a new concept as a supertype of an existing one and moves some of the existing concept's properties to the new one. It is the inverse of the operation *Extract subconcept* and has its origins in the software refactoring operation *Extract superconcept*.

Collapse hierarchy: This operation collapses one concept (entity type or relationship type) with its subtypes. The new concept contains the semantics of the two collapsed elements. There are two versions of this operation: *Collapse hierarchy up* and *Collapse hierarchy down*. In the "up" version, the supertype absorbs the subtypes and therefore the new concept takes the name of the supertype. In the "down" version is the subtype the one that keeps its name. This operation is based on the software refactoring operation *Collapse hierarchy*.

Extract hierarchy: This operation splits a concept into several concepts related by generalization relationships and spreads the properties of the original concept to the new concepts. It is the inverse of the operation *Collapse hierarchy* and has its origin in the software refactoring operation of the same name.

Promote refinement to relationship type: The relationship types defined in an ontology tend to be used, with some restrictions, by the subtypes of the entity types where they are defined. In such cases, a relationship type may be redefined using either a predefined construction of the ontology or a general integrity constraint. When a relationship type is only used by the elements where it is redefined or their subtypes, then it and its redefinition can be replaced with another relationship type that uses as participants the entity types where the relationship type was redefined and takes into account the integrity constraints added in the redefinition. We created this operation from scratch because it is highly applicable in large and general ontologies.

4.2 Integrity constraint refactoring operations

The operations in this category can be used to restructure the contents of general integrity constraints. We call general integrity constraints to integrity constraints which cannot be represented directly with the ontology language and needs to use a new language to express them, such as the OCL language in the case of UML ontologies. These operations are useless when the ontology being refactored does not allow general integrity constraints, as in the case of an ontology that only uses OWL language.

The integrity constraint refactoring operations are classified within two main categories according to their purpose:

1. Composing integrity constraints: operations that improve the quality of integrity constraints.
2. Simplifying integrity constraints: operations that simplify the conditions and expressions used within integrity constraints.

The operations presented in this section are generic enough refactor general integrity constraints written in either imperative or declarative languages. Next two subsections enumerate and describe briefly the refactoring operations.

Composing integrity constraints

Extract IC: This operation splits one integrity constraint in two and spreads its code to the two created constraints. This operation is based on the characteristics of the software operation *Extract method.*

Inline IC: This operation merges two integrity constraints into one. It is the inverse of the operation *Extract IC.* It is based in the software refactoring operation *Inline method.*

Inline temp: This operation replaces a temporal variable used in an integrity constraint with the expression that defines its value. This operation is very useful when the temporal variable is used infrequently. It was adapted from the software refactoring operation of the same name.

Introduce explaining variable: This operation assigns a variable the result of an expression used in an integrity constraint and replaces the expression with a reference to the new variable wherever it occurs. This operation improves maintainability when the same expression is used several times in an integrity constraint. It is the inverse of the operation *Inline temp* and is based on the software refactoring operation *Introduce explaining variable*.

Replace array with tuple: This operation replaces an array used in the body of an integrity constraint with a tuple. The new tuple will have one field for each row of the array. We included this operation in our catalog because some of the languages for representing general integrity constraints may support tuples and arrays. It is an adaptation of the software operation *Replace array with object*.

Replace magic number with symbolic constant: This operation replaces a number used in an integrity constraint with a constant of the same value. It was adapted from the software operation of the same name.

Substitute expression: This operation replaces an expression of an integrity constraint with another expression with the same meaning. It is an adaptation of the software refactoring operation *Substitute algorithm*.

Remove double negative: Using the *DeMorgan* rule, this operation replaces a double negation with an affirmation, for example replacing *if(Not NotFound)* for *if(Found)*. This operation was adapted from the software refactoring operation of the same name.

Split iterator: This operation separates one iteration in n (n>1), where each of the n new iterators performs a different activity. We decided to include in the catalog some operations that deal with iterators because they are often used in the languages that represent general integrity constraints. This operation is based on the software refactoring operation *Split Loop*.

Replace iteration with recursion: This operation replaces an iteration structure with its equivalent recursive call. It was adapted from the software refactoring operation of the same name.

Replace recursion with iteration: This operation replaces a recursive call structure with its equivalent iteration structure. It is the inverse of the operation *Replace iteration with recursion*.

Consolidate identifier: This operation can be applied when two or more expressions use different identifiers to refer to the same concept. It forces all expressions to access the instances of a concept in the same way, which means using the same identifier. This operation is based on the database refactoring operation *Consolidate key strategy* (Ambler, 2003).

Simplifying conditional expressions

Consolidate conditional expression: This operation combines n conditionals defined within an integrity constraint in a single conditional. It was adapted from the software refactoring operation of the same name.

Consolidate duplicate conditional fragments: This operation extracts the parts of a conditional structure that are repeated in all of its branches. It was adapted from the software refactoring operation of the same name.

Replace conditional with polymorphism: In some cases, a conditional states restrictions that depend on the type of the individual. This operation distributes these restrictions through the taxonomy using the inheritance as the template design pattern (Gamma, Helm et al., 1995). In particular, this operation distributes the conditional through the subtypes of the context entity type of the integrity constraint where it is defined. Thus, for each subtype, the conditions that its instances should satisfy are defined. This operation was adapted from the software refactoring operation of the same name.

Introduce null object: This operation creates a new subtype of a given entity type E. This subtype represents the instances of E with undefined information, that is, the instances that have no value for a given relationship type. This operation is very useful for deleting conditional constraints such as "*if (something=null) then* ..." It was adapted from the software refactoring operation of the same name.

Reverse conditional: This operation modifies a conditional to make it more comprehensible. It negates the entire conditional and therefore the condition and the *then* and *else* branches.

5 CONCLUSIONS

Over the last two decades refactoring operations have been applied effectively to software. Ontology has also applied successfully in some cases to other artifacts such as databases and UML models. There have been some attempts to apply refactoring to ontologies, but with little success. The main problems in applying refactoring to ontologies are that it is not clear what refactor an ontology means, and we need a catalog of refactoring operations similar to the ones create for software or databases.

In this chapter we studied the refactoring work of other fields in combination with other techniques such as program slicing or schema transformation. The lessons learnt of this study has been used to define what ontology refactoring is and to present a catalog of ontology refactoring operations, which reuse the operations presented in other fields (software refactoring, database refactoring, model refactoring and schema transformation) that are suitable to refactor ontologies. Therefore, the main contributions of this chapter are clarification of the meaning of ontology refactoring and the creation of a catalog that contains ontology refactoring operations. The catalog has been created following the classification and naming conventions of previous refactoring catalogs to make the catalog easier to use and understand the catalog.

The presented work provides a firs step in the formalization of ontology refactoring, but more work needs to be done in that direction. In particular, we believe that the coherent next step would be to define formally the operations of the presented catalog and implement them in the most prominent ontology tools, such as Protégé. In addition, the identification of ontology refactoring opportunities needs also to be addressed in order to identify in what cases some refactoring operations can be applied to automatically improve ontologies.

ACKNOWLEDGEMENT

This work has been partly supported by the Spanish Ministry MICINN (TIN2008-00444) and the European Union (FP7-ICT-2009-5-257639).

REFERENCES

Ambler, S. (2003). Agile Database Techniques : Effective Strategies for the Agile Software Developer, Wiley.

Ambler, S. W. (2002). Agile Modelling: Effective Practices for EXtreme Programming and the Unified Process, John Wiley & Sons Inc.

Assenova, P. and P. Johannesson (1996). Improving quality in Conceptual modelling by the use of schema transformations. 15th International Conference on Conceptual Modeling.

Astels, D. (2002). Refactoring With UML. Conference on eXtreme Programming and Agile Processes in Software Engineering XP2002**:** 67--70.

Batini, C., S. Ceri, et al. (1992). Conceptual Database Design: An Entity-Relationship Approach, Benjamin/Cummings.

Beck, K. (1999). Extreme Programming Explained, Addison-Wesley Pub Co.

Bergstein, P. L. (1991). Object-preserving class transformations. Conference proceedings on Object-oriented programming systems, languages, and applications. Phoenix, Arizona, United States, ACM Press**:** 299-313.

Bois, B. D. (2004). Opportunities and challenges in deriving metric impacts from refactoring postconditions. Proceedings WOOR'04 (ECOOP'04 Workshop on Object-Oriented Re-engineering). S. Demeyer, S. Ducasse and K. Mens. Antwerpen, Belgium.

Bois, B. D., S. Demeyer, et al. (2004). Refactoring - improving coupling and cohesion of existing code. In Proceedings WCRE'04 (Working Conference on Reverse Engineering). Victoria, Canada, IEEE Press**:** 144-151.

Bottoni, P., F. Parisi-Presicce, et al. (2002). Coordinated Distributed Diagram Transformation for Software Evolution. Proc. of the Workshop on 'Software Evolution Through Transformations'(SET'02). R. Heckel, T. Mens and M. Wermelinger, Electronic Notes in Theoretical Computer Science (ENTCS). **72**.

Casais, E. (1989). Reorganizaing an object system. Object Oriented Development. D. Tsichritzis. Geneve**:** 161-189.

Conesa J. (2004): Ontology-Driven Information Systems: Pruning and Refactoring of Ontologies. Doctoral Symposium of the seventh international conference on the Unified Modeling language. http://www-ctp.di.fct.unl.pt/UML2004/DocSym/JordiConesaUML2004DocSym.pdf

Conesa J. (2008): Pruning and refactoring ontologies in the development of conceptual schemas of information systems. PhD Thesis. Technical University of Catalonia.

Correa, A. and C. Werner (2004). Applying Refactoring Techniques to UML/OCL Models. Proceedings of International Conference of UML. Lisbon, Portugal**:** 173-187.

Critchlow, M., K. Dodd, et al. (2003). Refactoring Product Line Architectures. Proceedings of the First International Workshop on REFactoring, Achievements, Challenges, Effects (REFACE). Victoria, Canada.

Chung, L., B. A. Nixon, et al. (2000). Non-functional requirements in software engineering, Kluwer Academic Publishers.

Deursen, A. v., M. Marin, et al. (2003). Aspect mining and refactoring. Proceedings of the First International Workshop on REFactoring: Achievements, Challenges, Effects (REFACE03). Victoria, Canada.

Ducasse, S. e., M. Rieger, et al. (1999). A Language Independent Approach for Detecting Duplicated Code. Proceedings {ICSM}'99 (International Conference on Software Maintenance). H. Yang and L. White, IEEE**:** 109--118.

Ducasse, S. e., M. Rieger, et al. (1999). Tool Support for Refactoring Duplicated OO Code. ECOOP'99, Springer. **1743:** 177 - 178.

Eetvelde, N. V. and D. Janssens (2004). A Hierarchical Program Representation for Refactoring. Electronic Notes in Theoretical Computer Science. **82**.

Eick, C. F. (1991). A Methodology for the Design and Transformation of Conceptual Schemas. Very Large Data Bases, Barcelona.

Ernst, M. D., J. Cockrell, et al. (2001). "Dynamically discovering likely program invariants to support program evolution." IEEE Transactions on Software Engineering **27**(2): 1-25.

Fowler, M. (1999). Refactoring: Improving the Design of Existing Code, Addison-Wesley.

Fowler, M. and P. Sadalage. (2003, January, 2003). "Evolutionary Database Design." from http://www.martinfowler.com/articles/evodb.html

Gamma, E., R. Helm, et al. (1995). Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley.

Gorp, P. V., H. Stenten, et al. (2003). Towards Automating Source-Consistent UML Refactorings. UML, 2003, Springer. **2863:** 144 - 158.

Grant, S. and J. R. Cordy (2003). An Interactive Interface for Refactoring Using Source Transformation. Proceedings of the Fist International Workshop on Refactoring: Achievements, Challenges and Effects (REFACE'03). Victoria, Canada**:** 30-33.

Gray, J. G. (2002). Aspect-Oriented Domain-Specific Modeling: A Generative Approach Using a Meta-weaver Framework. Department of Electrical Engineering and Computer Science. Nashville, Vanderbilt University.

Griswold, W. G., M. I. Chen, et al. (1998). "Tool Support for Planning the Restructuring of Data Abstractions in Large Systems." IEEE Transactions on Software Engineering **24**(7): 534-558.

Griswold, W. G. and D. Notkin (1993). "Automated assistance for program restructuring." ACM Transactions on Software Engineering and Methodology **2**(3): 228-269.

Halpin, T. (1996). Conceptual schema and relational database design, Prentice-Hall, Inc.

Halpin, T. (2001). Information Modeling and Relational: From conceptual analisys to logical design, Morgan Kaufman.

Halpin, T. A. and H. A. Proper (1995). Database Schema Transformation & Optimization. International Conference on Conceptual Modeling, LNCS.

Hofstede, A. H. M. t., H. A. Proper, et al. "A note on Schema equivalence."

Judson, S. R., D. L. Carver, et al. (2003). "A Metamodeling Approach to Model Refactoring."

Kataoka, Y., M. D. Ernst, et al. (2001). Automated Support for Program Refactoring Using Invariants. ICSM**:** 736-743.

Kleppe, A., J. Warmer, et al. (2003). MDA Explained. The Model Driven Architecture: Practice and Promise, Addison-Wesley.

Koru, A. G., L. Ma, et al. (2003). Utilizing Operational Profile in Refactoring Large Scale Legacy Systems. Proceedings of the First International Workshop on REFactoring: Achievements, Challenges, Effects (REFACE). Victoria, Canada.

Marko Boger, T. S. and P. Fragemann (2002). Refactoring Browser for UML. Objects, Components, Architectures, Services, and Applications for a NetworkedWorld: International Conference NetObjectDays, NODe, 2002**:** 77--81.

Mens, K. and T. Tourwe (2004). "Delving source code with formal concept analysis." Computer Languages, systems & structures.

Mens, T. (2004). "A Survey of Software Refactoring." IEEE Transactions on Software Engineering **30**(2): 126-139.

Moore, I. (1996). Automatic Inheritance Hierarchy Restructuring and Method Refactoring. Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. San Jose, California, United States**:** 235 - 250.

OMG. (2003). "OMG Revised Submission, UML 2.0 OCL."

Opdyke, W. F. (1992). Refactoring Object-Oriented Frameworks. Ph.D. Thesis., University of Illinois.

Ostrowski D. A. (2008): Ontology Refactoring. Second IEEE International Conference on Semantic Computing (ICSC, 2008): 476-479.

Ondrej Sváb-Zamazal, Vojtech Svátek, Christian Meilicke, Heiner Stuckenschmidt (2008): Testing the Impact of Pattern-based Ontology Refactoring on Ontology Matching Results. 3rd International Workshop on Ontology Matching.

Philipps, J. and B. Rumpe (2001). Roots of Refactoring. Tenth OOPSLA Workshop on Behavioral Semantics. Tampa Bay, Florida, USA.

Porres, I. (2002). A toolkit for manipulating UML models, TUCS Turku Centre for Computer Science.

Porres, I. (2003). Model Refactorings as Rule-Based Update Transformations. UML, 2003, Springer**:** 159 - 174.

Proper, H. A. and T. A. Halpin (2004). Conceptual Schema Optimisation - Database Optimisation before sliding down the Waterfall, Department of Computer Science - University of Queensland**:** 34.

Roberts, D., J. Brant, et al. (1997). A refactoring tool for Smalltalk. Theory and Practice of Object Systems. **3 Number 4**.

Roberts, D. B. (1999). Practical Analysis For Refactoring Ph.D. thesis., University of Illinois.

Rysselberghe, F. V. and S. Demeyer (2004). Evaluating Clone Detection Techniques from a Refactoring Perspective. Proceedings of the 19th International Conference on Automated Software Engineering (ASE'04). Linz, Austria**:** 336-339.

Sacco, G. (2000). "Dynamic Taxonomies: A Model for Large Information Bases." IEEE Transactions on Data and Knowledge Engineering **12**(3): 468-479.

Simmonds, J. and T. Mens (2002). A Comparison of Software Refactoring Tools, Programing Tecnology Lab.

Simon, F., F. Steinbrukner, et al. (2001). Metrics Based Refactoring. Proceedings of the 5th European Conference on Software Maintenance and Reengineering, IEEE Computer Society Press.

Sunye, G., D. Pollet, et al. (2001). Refactoring UML Models. UML, 2001. M. Gogolla and C. Kobryn, Springer. **2185:** 134 -148.

Tahvildari, L. (2003). Quality-Driven Object-Oriented Code Restructuring. In the IEEE Software Technology and Engineering Practice (STEP) - Workshop on Software Analysis and Maintenance: Practices, Tools, Interoperability (SAM). Amsterdam, The Netherlands.

Thagard.P (1992). Conceptual Revolutions, Princeton University Press.

Tourwe, T. and T. Mens (2003). Identifying Refactoring Opportunities Using Logic Meta Programming. Proceedings of the 7th European Conference on Software Maintenance and Reengineering, IEEE Computer Society Press.

Xing, Z. and E. Stroulia (2003). Recognizing Refactoring from Change Tree. Proceedings of the First International Workshop on REFactoring: Achievements, Challenges, Effects (REFACE). Victoria, Canada.

Yu, W., J. Li, et al. (2004). Refactoring Use Case Models on Episodes. Proceedings of the 19th International Conference on Automated Software Engineering (ASE'04). Linz, Austria**:** 328-331.

Yu, Y., J. Mylopoulos, et al. (2003). Software refactoring guided by multiple soft-goals. Proceedings of the First International Workshop on REFactoring, Achievements, Challenges, Effects (REFACE). Victoria, Canada.

Zhang, J., Y. Lin, et al. (2005). Generic and Domain-Specific Model Refactoring using a Model Transformation Engine. Model-driven Software Development - Research and Practice in Software Engineering. Springer, Springer.

Zimmer, J. A. (2003). Graph Theoretical Indicators and Refactoring. Prooceedings of the Third XP and Segond Agile Universe Conference. F. M. a. D. Wells. New Orleans, USA, Springer. **2753**.