An Analytical Method for Refactoring Object-Oriented Code

Sergio Pissanetzky [1]

Private Consultant

Summary

We present a new method based on software analysis for refactoring object-oriented programs. The segment to be refactored is decomposed by a parser into structural elements and described by five sets of relations organized in sparse matrix format. Rows correspond to tuples, columns to variables in the segment, column partitions to classes, and row partitions to methods. Row and column permutations and rearrangement of partitions are refactorings. A heuristic search algorithm is proposed, which follows prescribed rules or menu selections. The output is a design for the given segment.

Our work draws heavily from mathematical notation by establishing a parallel between the "refactoring" of equations and computer code. The method divides the refactoring problem into parts suitable for separate analysis, makes critical refactoring information readily available, and helps to appreciate the nature of refactoring with a visual presentation.

Examples illustrate the algorithm. The first two examples demonstrate how rules are applied, class and method definition, variable classification into attributes and arguments, temporaries or return values for methods, how different search paths yield different refactorings, and hierarchical model nesting. The Fowler refactoring Move Method is illustrated. The third example starts from a poor UML design and produces two different good designs. It illustrates how designs can be refactored, the use of relational operations to refactor, and the conversion between conditional logic and polymorphism. The technology can automate refactoring, or be combined with development and enhance coherence and integration in program evolution.

Keywords: refactoring, restructuring, behavior preservation, coherence, coding tools, development environments.

[1] Sergio@SciControls.com

1 Introduction

Refactoring is a technology that improves the quality of existing code by applying a series of elementary behavior-preserving transformations called *refactorings*. Research in refactoring officially started in 1992 with the publication of a PhD thesis [1] by William Opdyke. More recently, Martin Fowler published a catalog of refactorings in book form [2], listing more than 70 refactorings. An extensive survey of software refactoring was published by Tom Mens et al. [3]. Much of the work loosely clusters along two main directions, methods based on graph representations of the code and intended for automation, and methods that support manual operations such as finding segments of code that need improvement, or deciding which refactorings to apply.

In the first group, we should mention the detection of "bad smells" [2, 4, 5, 6], applying design patterns to automate refactoring in Java, although the authors admit to possible errors [7], feature decomposition of programs, where a feature is an increment in program functionality [8], detection of duplicated code [9, 10, 11], detection of program invariants [12], and using dynamic data obtained during runtime [13]. The use of code metrics is commanding attention, including using distance cohesion metrics to associate methods and variables of a class [14], using a combination of code metrics and visualization [15], and using object-oriented metrics for bad smell detection [14].

Efforts in the second group include [16, 17, 18, 19]. Graph transformation and rewriting was formalized and used to analyze dependencies in refactoring [20, 21]. M. Verbaere et al. [22] note that existing tools are language-specific and contain significant bugs, even in sophisticated development environments, and propose a language for refactoring that manipulates a graph representation of the program.

There is also a third, incipient direction. T. Mens et al. [3] suggest that refactorings can be classified according to the code quality attributes they affect, such as robustness, extensibility, reusability, or performance, and then used to enhance the attributes of interest. They propose to estimate the effect by considering the internal quality attributes affected, such as code size, complexity, coupling and cohesion, which can be measured if access to program structure information is available. Code metrics can be used to determine the effect of refactorings on maintainability [23], or quality [24]. Performance can be improved by replacing conditional logic with polymorphism [25].

There has been work on specialized refactoring systems such as architectural refactoring of inherited software, particularly corporate software [26] and clustering techniques for architecture recovery [27]. And in aspect-oriented programming, using program slicing to improve method and aspect extraction [28], role-based refactoring [29], role mapping [30], and aspect refactoring [31]. Refactoring of non-object-oriented code has also been considered. [32]. Code refactoring has been linked to algebraic factoring [8], and, in the particular case of user interface design, to matrix algebra [33]. The need to teach refactoring in Computer Science curricula has now been recognized [34]. In spite of all the efforts, refactoring remains a complex and risky procedure.

We propose a method for refactoring object-oriented software where the segment of code of interest is separated into elements. The method is behavior-preserving and static. It is based on code analysis [2], intended for automation, language-independent, applicable to objet-oriented and non-object-oriented code, and inspired on mathematical notation.

Mathematical notation is mature and precise. For centuries, mathematicians have been writing equations, condensing properties and behavior into objects, "refactoring" their equations to make them more meaningful and easier to manipulate, using patterns, and perfecting a notation that would allow them to express all that. For example, a symmetric positive definite matrix A is an

[2] From Merriam-Webster: *analysis* 1: separation of a whole into its component parts, 2: an examination of a complex, its elements, and their relations, 3: a method of resolving complex expressions into simpler ones.

object, and its factorization $\mathsf{A} = \mathsf{LU}$ into the product of a lower triangular matrix L and an upper triangular matrix U, is a pattern, frequently used when solving large systems of linear equations. Mathematical notation consists of equations interspersed with textual descriptions of conditional logic that specify the cases where the equations apply and the *sequences* in which they should be "executed". The notation can be viewed as a language intended for mathematicians, but not for machines. Powerful "refactorings", such as the symbolic rationalization of expressions, factorization, symmetrization, or simplification, are common in computer algebra systems [35, 36], and automated bug-free tools are available, including an open framework for symbolic computation within C++ [37]. Links between code refactoring and algebraic refactoring have been established in the literature [8, 33]. It is only fit that we continue to draw from all that experience.

Fortran, the first major computer language, was created as a *formula translating* system, with statements that formalize conditional logic, cases and sequences, or deal with the subscripts used in matrices and tensors, and variables and operators that represent the equations themselves. Concepts and notations from Fortran were adopted by C, Basic and Pascal, and later by modern object-oriented languages such as C++ and Java. Refactoring has been practiced even in the early days of Fortran, but, not surprisingly, it intensified and was recognized as a separate discipline only after the introduction of objects. The structural elements from mathematical notation, combined in ingenious and meaningful ways, can still be found in object-oriented languages.

In Section 2, we introduce a model where the structural elements of a program segment are described in terms of 5 sets of relations, and relational operations are used to realize the refactorings. We formalize some aspects of the model, and introduce a sparse matrix representation for sets of relations where partitions of the matrix correspond to classes and methods, and which also helps to present and explain the algorithm and visualize the refactorings. The model can be created by a specialized parser from existing code.

In Section 3, we describe a heuristic algorithm for effecting the refactorings. A set of rules for the operation of the algorithm is proposed, based on traditional object-oriented concepts. The output from the algorithm is a design, from which code can be developed.

The algorithm is demonstrated by way of several examples. Section 4 presents example E1, which starts with a program that has already been parsed into relations, presents the corresponding model in sparse matrix format, and examines the transformation of one of the matrices as the rules are applied. The example assumes that no object-oriented information at all is available, and attempts to create a partition that corresponds to a reasonable class and method structure. In the course of the example, the Fowler refactoring Move Method is illustrated. The example concludes with the corresponding object-oriented design produced by the algorithm.

Section 5 presents example E2, which starts the same as example E1 but results in a *different* matrix partition, corresponding to a different design. Since the two designs correspond to the *same* program, a comparison between them serves as an illustration of a major refactoring. The example also suggests a possible link with development.

Example E3 presented in Section 6 is very different. It starts from a given UML design that needs refactoring, creates the relational model directly from the design, and refactors it using relational operations such as join and normalization. *Two* different refactored UML designs are obtained. Both could have been obtained manually by applying a well-known pattern, although the algorithm had no knowledge of the pattern. A similar example was presented in the literature [3], but only one of the two refactored designs was reported. Expectations and conclusions are the subject of section 7.

2 The Relational Model

The relational model describes the structure of a program in terms of relations. The C expression

```
a = b - c;
```
(1)

is a description of the *structure* of a function, the table of subtraction, not the function itself. In words: "a is the *codomain*, b and c are the *arguments*, in that order, '-' is the *operator*, and '1' is the *name*." The quoted statement describes a relation of degree 5. The C statement

```
a = a + 1;
```
(2)

describes the structure of a *mutator*, where "a is the codomain and argument 1, '1' is a literal, '+' is the operator, and '2' is the name." This quoted statement describes a relation of degree 4. In order to formalize the description of the structure of functions such as expressions (1) or (2) by means of relations, define the following domains:

- For each variable in the expression, define a domain with the same name and with the set {c, m, 1, 2, ...}.
- For each literal in the expression, define a domain with the same name and the set {1, 2, ...}. Literals can only be arguments, not codomains or mutators.
- Define an additional domain OP for the operators, and another domain PK for the names, which act as primary keys.

In the domain definitions, c stands for *codomain*, m for *mutator*, and 1, 2, ... designate arguments in order. This notation is sufficient for our purposes, but more complex cases require numbered codomains or mutators, as in c1, c2, ... and m1, m2, ..., and arguments designated as a1, a2, ... for consistency. A domain is also known as a *type*, and we use the two terms interchangeably. The structure of expression (1) can now be described by a single tuple in a relation of degree 5:

PK	OP	a	b	c
1	-	c	1	2

(3)

Expression (2) can also be described by a relation, but this time the degree is 4. In general, when dealing with a program, many relations of various degrees are involved. But sets of relations of different degrees are difficult to comprehend for humans. To circumvent the problem, there is, in fact, a simple solution: a *sparse matrix* [38] representation of the set of relations. To create the representation, let D be the set of all domains used by the relations of the given set, and let T be the set of all tuples, irrespective of degree. Associate a column of the matrix to each domain $d \in D$, and a row to each tuple $t \in T$, then fill-in the elements from the tuples into the corresponding elements of the matrix, leaving the remaining elements blank. The following matrix describes the structure of expressions (1) and (2) at the same time, and is acceptable for presentation:

PK	OP	a	b	c	1
1	-	c	1	2	
2	+	m			2

(4)

The relations we used so far are all normalized. The expression

```
a = b + c * d;
```
(5)

would result in an unnormalized relation which references two functions. It means

```
h = c * d;
a = b + h;                                                    (6)
```

which consists of two different relations. A join of the two on h would yield (5), but the separation is needed to preserve the sequence of execution. A conditional statement such as

```
if(b)i = 1;
else i = 2;                                                   (7)
```

is also a relation:

b	i
true	1
false	2

(8)

This case is different, however, because the relation is not a reference to a function but the function itself. Relation (8) affects the sequence of execution, which is part of the structure of the program. The examples expand on these issues.

To describe the model, we assume that the program of interest can be converted to a *normal form* that contains only normalized function references and conditional logic. The model is obtained by representing the structure of the expressions in the program with normalized relations. When an entire program is converted, the model is said to be in its *expanded form*. The expanded form contains no object-oriented features and no language-specific features. Object-oriented programs and programs written in imperative languages usually can be converted this way. If a program can be converted, then it can be refactored with relational support. The assumption that the entire program has to be converted is too strong for refactoring applications, which typically require conversion of only a small part of the program. Other applications such as reverse engineering or translation between languages may require full conversion, but they are not discussed here. Here, we use the expanded form as a conceptual artifice to explain the model. The expanded form of the relational model consists of the following 5 sets of relations:

1. The first set of relations is the **Matrix of Calculations**, which contains the relations corresponding to function references in the program in sparse matrix format. Partitions of the matrix are used to encapsulate user types and create classes and methods. Repartitioning results in refactoring transformations.

2. The **Matrix of Sequences** with the sequences of execution and flow of control switches. This set describes the conditional logic in the program, including polymorphic structures, which are considered conditional logic. Permutations and relational operations in the sequences produce refactoring transformations. Examples can be found in example E3.

3. The **Actors** that initiate the sequences. Multiple actors can initiate multiple sequences.

4. The predecessor-successor **Constraints**, which establish the degrees of freedom for partitioning and sequence permutations. It is very important to avoid overconstraining the system. Additional constraints can be enforced, for example to prevent certain critical parts of the code from being affected in applications such as real-time software, embedded software or safety software.

5. The set of **Scopes** describes the point of declaration and scope range for the variables in the program. In the expanded model, the range is from the declaration to the end of the normal program. If a partition cuts through a range, the range has to be adjusted. If the scope needs to be extended, a method invocation has to be created as a *scope connector*.

The matrix of calculations can also be designated the *matrix of services*, containing all calculations and services available in the system, in addition to those provided by the classes defined in the program. The matrix of sequences can be designated as the *matrix of selections*, because it selects services from the matrix of services. Its implementation would require a further level of indirection to refer the variables to their associated domains. For our purposes here, we will be content with the simpler and more restrictive definition above.

Additional structures may be needed. For example, the partitions of the matrix of calculations are not part of the matrix and must be separately stored. Assignments of methods to classes also must be stored separately. The matrices are conceptual devices. For actual storage and processing, a *bipartite graph* [39] could be used, which is invariant under row and column permutations if unlabeled. Bipartite graph partitioning has been applied to text mining [40]. Constrained sparse matrix permutation and partitioning has been applied to the problem of multiprocessor load balance [41]. Replacing conditional logic with polymorphism does not degrade performance, it actually *enhances* it [25].

The model is nested. Processing in the expanded model creates partitions that represent classes and methods. These partitions can be described by separate submodels and eliminated from the main model, leaving only their declarations and making the main model smaller and easier to process. The submodels, in turn, can be partitioned in the same way, creating a hierarchy of nests. Example E1 expands on this matter. Conversely, known classes and methods can be described with submodels, many of which may never be needed in expanded form. A typical process of refactoring starts with one expanded class, and follows references leading to other classes, which should only then be expanded. The extension of the expanded model is determined by the depth of the refactoring. The range of responsibilities for the model must be specified by defining realistic input and output interfaces. Here are some ideas:

- A parser provides expanded input for the model, one class at a time, along with class information and references to associated classes, and upon requests initially received from the user and later from the model. It seems reasonable to consider converting existing parsers to this type of operation.

- A language module receives output from the model in the form of expanded classes and methods, and references to previously existing classes. The information should be sufficient for the module to be able to reuse the existing classes and generate code for the expanded classes in accordance with the rules of the corresponding language.

We anticipate two different modes of operation. In the first mode, one class or a few classes are parsed, the refactoring is performed, and output code is regenerated. In the second mode, the model represents the entire program. It is created when development starts, and remains attached to the program and updated when development or refactoring happens. The model is a permanent component of the development environment.

If a language provides a parser and a conversion module for the interfaces, we say that the language is *subscribed* to the model. No languages are currently subscribed, but several manual examples are included.

3 The Encapsulation Algorithm

The purpose of the encapsulation algorithm is to define classes and methods by partitioning the matrix of calculations. A *partition* of a matrix is a partition of the set of rows and a partition of the set of columns into subsets. A subset of rows defines a method. A subset of columns defines a class, where some of the variables in the subset become class members, and others become arguments, return values, or temporaries in the methods assigned to that class.

Row and column permutations become necessary to fit the partitions. Columns can be freely permuted because their order is irrelevant. The rows, however, are linked by the sequences of execution. Permuting rows implies rearranging the sequences, and must comply with the constraints. We suggest using trial and error to guarantee compliance: if a partition does not comply, it is discarded and another one is tried. Neither rows or columns have to be physically permuted, of course, but in the examples we do permute them physically to enhance the quality of presentation.

The model in its expanded form contains no object-oriented information. Creating rules for an algorithm that can partition the matrix of calculations and define meaningful object-oriented structures, with no outside help, is of course a major challenge. However, we have used some well-established notions of object-oriented programming to prepare 14 preliminary rules, which are sufficient for our examples. The rules appear to be simple if judged in relation with the number and significance of the features they accommodate. The merit of the algorithm is in its ability to make all the information needed to apply the rules readily available. We expect the rules to refine and expand as experience is gained.

R1 Domain constructors. If `F` is the matrix of calculations, and $\mathtt{F}_{ij}$ `= c`, a codomain, then row `i` is the *domain constructor* for type `j`.

R2 Subset refinement. Subsets of columns or rows can be iteratively refined. They can be initially defined, and then expanded or collapsed as needed as the algorithm progresses.

R3 Uninitialized variables. Every variable column must contain a codomain reference "c", otherwise we are dealing with an uninitialized variable.

R4 Class constructors. Given a class, a *class constructor* is obtained as a subset of the domain constructors for the domains of the class. The class constructor is said to be *direct* if no class members appear as arguments in the domain constructors. Otherwise, the class constructor is said to be *indirect*. An indirect class constructor can be converted into a class method.

R5 Row symmetries. If a subset of rows has the right structural symmetries, it should be further partitioned into two of more invocations of a simpler method. "Right" symmetries means that the simpler methods must all have the same number of rows and identical structure.

R6 Model nesting. In the expanded model, methods and classes are expanded to full detail. During processing, they can be extracted and represented by a separate model.

R7 Methods. A set of rows in the matrix of calculations defines a method. Only one return value allowed, but *mutators* can change the values of arguments. Non-blank values in the rows indicate references to the corresponding classes or arguments for the method if the method belong to that class.

R8 Method variables. Once a variable has been specialized as a class variable, but not as a class attribute, then it is a *method variable*. Method variables belong to one of three categories:

argument, return value, or temporary. The category is determined by the external access as follows:

external access	method variable
m	argument
c	return value
1, 2, 3, ...	argument
none	temporary

R9 Two or more columns with identical structures in the matrix of calculations are candidates for encapsulation.

R10 Classes. A subset of columns encapsulated in a partition is an instance of a class. The corresponding subset of variables must be further partitioned into a sub-subset of class attributes and a sub-subset of method variables. Either one can be empty.

R11 Classes. Non-blank values in the columns of a class indicate methods that reference the class.

R12 Permutations. Columns can be permuted freely. Rows can be permuted subject to the constraints.

R13 Column symmetries. If a subset of columns has the right structural symmetries, it should be further partitioned into two or more instances of a class. "Right" symmetries means that the instances must all have the same number of columns and identical structure.

R14 Private methods. A method with all its references in a class can be assigned as a private method of that class.

The operation of the algorithm is a search in a graph, with edges corresponding to rules and vertices to intermediate partition states. The graph is not a tree, because it is possible for rules to be applied in a different order and still lead to the same state. At each vertex, the algorithm applies a rule and travels to the next vertex. If a constraint is violated, the algorithm backtracks to the last visited vertex, and tries again, until it reaches a leave or a vertex with no legal outgoing edges. Different refactorings can be obtained depending on the path followed.

A formal prove that the algorithm preserves behavior would depend on the definition of behavior. The scope of the proof would be restricted to programs that can be expressed in the form described above. The first step would be to use relational operations such as join and union to formally express the entire program as one single relation, which establishes a mapping from all possible sets of input values to the corresponding output values. At this point, behavior could be defined as that mapping. The final step would be to consider the relational operations used in the algorithm and prove that they leave the behavior invariant. Informally, if we think of behavior as the mapping defined by the relations in the matrix of calculations, considered in the order defined by the sequences, it should be apparent that behavior will be preserved even if the sequences are changed, provided the constraints are not violated.

4 Example E1

The examples are non-trivial, rich in features, and very different. The examples illustrate how the rules work and intend to demonstrate that critical decisions can be made using information readily

available in the model. They are presented with sufficient detail that they may serve as a user's requirements document for a tool builder.

The matrix of calculations can display a set of many different relations in a single table. In this simple case, it can also show the sequence of execution, the partitions used to encapsulate user types and define classes and methods, and even the refactorings themselves. It is an ideal tool for presentation.

Example E1 is taken from mathematical notation. Example E1 begins with Program A, which is in normal form, contains no conditional logic, and is designed specifically to help visualize important features of the relational model. Example E1 has some similarity with an example used in our early work [42]. The example also demonstrates the Fowler refactoring Move Method.

The exercise results in two different object-oriented designs, D1 and D2, both equivalent to Program A. Since Program A can be obtained from either D1 or D2, conversion between D1 and D2 is also possible, and the example is a non-trivial exercise in refactoring. Here is Program A:

Program A

```
1.    d    =   1;
2.    a    =   2;
3.    b    =   3;
4.    Rx   =   4;
5.    Ry   =   5;
6.    Vx   =   6;
7.    Vy   =   7;
8.    Fx   =   8;
9.    Fy   =   9;
10.   ta   =   a * Fx;
11.   tb   =   a * Fy;
12.   tc   =   d * Vx;
13.   td   =   d * Vy;
14.   te   =   ta + tc;
15.   tf   =   tb + td;
16.   tg   =   b * Fx;
17.   th   =   b * Fy;
18.   Rx   =   Rx + te;
19.   Ry   =   Ry + tf;
20.   Vx   =   Vx + tg;
21.   Vy   =   Vy + th;
```

There is only one actor and one sequence of execution, and there is no need to write that explicitely because we can assume that the order of execution is the order the rows are written, with execution starting at the first row. Also, there is no need to worry about the scope of the variables. The predecessor-successor constraints are as follows:

Successor	Predecessors
10	2, 8
11	2, 9
12	1, 6
13	1, 7
14	10, 12
15	11, 13

16	3, 8
17	3, 9
18	4, 14
19	5, 15
20	12, 16
21	13, 17

Note that 12, 13, are predecessors of 20, 21, respectively, because the values of Vx, Vy must be used in 12, 13 before they can be changed in 20, 21. The refactorings and other relational operations we are about to perform, must be tested for constraint conformance. Matrix A^0, shown in Figure 1, is the matrix of calculations for Program A.

PK	OP	d	a	b	Rx	Ry	Vx	Vy	Fx	Fy	ta	tb	tc	td	te	tf	tg	th	1	2	3	4	5	6	7	8	9
1	=	*c*																	1								
2	=		*c*																	1							
3	=			*c*																	1						
4	=				*c*																	1					
5	=					*c*																	1				
6	=						*c*																	1			
7	=							*c*																	1		
8	=								*c*																	1	
9	=									*c*																	1
10	*		1						2		*c*																
11	*		1							2		*c*															
12	*	1					2						*c*														
13	*	1						2						*c*													
14	+										1		2		*c*												
15	+											1		2		*c*											
16	*			1					2								*c*										
17	*			1						2								*c*									
18	+				*m*										2												
19	+					*m*										2											
20	+						*m*										2										
21	+							*m*										2									

Figure 1: The Matrix of Calculations A^0 of size 21×26 for Program A. Two-letter column captions have been stacked to make them fit.

We begin by partitioning away all columns associated with literals and with the domains PK and OP, all of which are of no interest for the refactoring. If program A had been obtained from existing object-oriented code, there would usually be additional information available suggesting how to select variables for encapsulation. However, for the sake of this example, we assume that there is no such information at all. The algorithm proceeds in steps by applying the rules of Section 3, as indicated below:

- (R9). The process starts with a seed. To find one, we compare the columns of matrix A^0 and

select those with the most structural similarities. At the top of the list are {d, a, b}, {Vx, Vy}, and {Fx, Fy}, with 3 appearances each and identical structures in all cases. We choose {d, a, b}. The consequences of seed selection are examined in example E2, where a different seed is used.

- (R10). Form block column {d, a, b}. This defines an object, say g, of a new class, say class G. Since all three columns have the same structure, tentatively declare all three as attributes of class G, meaning that there are no method variables in this partition.
- (R1). The constructors for the subdomains of g appear in rows 1, 2, 3.
- (R4). Form block row {1, 2, 3}, the constructor for class G, say $\bar{g}$.
- (R11). We notice that block column {d, a, b} has non-empty intersections with rows 10, 11, 12, 13, 16, 17. This is a new method.
- (R5). We notice a structural symmetry that indicates a split of the new method in two invocations of a simpler method, say method m: block rows {12, 10, 16} and {13, 11, 17}. Let m1 and m2 be the two invocations of method m. Method m has to be assigned to a class, but we will do that later. When forming the block rows, we must change the sequence of execution, which in this simple example is accomplished by permuting the rows.
- (R12). We must verify constraint compliance. If the constraints were violated, we would be forced to abandon our attempt, but they are not. At this point, matrix A^1 is obtained, as shown in Figure 2.

To continue partitioning matrix A^1, apply the following rules:

- (R7). The intersections of a method with columns are arguments for the invocations of that method. In this case, the two invocations of m intersect column subsets {Vx, Fx, ta, tc, tg} and {Vy, Fy, tb, td, th}, respectively, which require new partitions. They also intersect column subset {d, a, b}, which is already in a partition.
- (R12). Column permutations are allowed with no restrictions, because columns are in no particular order. They are necessary for presentation, not for the algorithm.
- (R10). The encapsulated columns define two instances, say h1, h2, of a new class, say H.

Figure 3 shows matrix A^2, obtained from A^1 after rearranging the columns as explained and defining the new class.

The next step is to find constructors for the two instances of the new class H.

- (R4) The domain constructors for the 5 subdomains encapsulated in h1 are in rows 6, 8, 12, 10 and 16. Rows 6, 8 are a *direct constructor*, while rows 12, 10, 16 are an *indirect constructor*, in the sense they are not self-sufficient, they depend on rows 6, 8 being executed first. Therefore, we leave rows 12, 10, 16 out of the constructor for now, we can always move them in later by applying rule R2. This avoids constraining the algorithm more than is necessary. Therefore, the class constructor for h1, say $\bar{h1}$, is in rows 6, 8.
- (R4) The same considerations apply to h2, the second instance of class H. The constructor for this instance is in rows 7, 9. The two class constructors have identical structures.

			g																literals								
	PK	OP	d	a	b	Rx	Ry	Vx	Vy	Fx	Fy	ta	tb	tc	td	te	tf	tg	th	1	2	3	4	5	6	7	8	9
	1	=	c																	1								
$\bar{g}$	2	=		c																	1							
	3	=			c																	1						
	4	=				c																	1					
	5	=					c																	1				
	6	=						c																	1			
	7	=							c																	1		
	8	=								c																	1	
	9	=									c																	1
	12	$*$	1					2						c														
$m1$	10	$*$		1						2		c																
	16	$*$			1					2								c										
	13	$*$	1						2						c													
$m2$	11	$*$		1							2		c															
	17	$*$			1						2								c									
	14	$+$										1		2		c												
	15	$+$											1		2		c											
	18	$+$				m										2												
	19	$+$					m										2											
	20	$+$						m										2										
	21	$+$							m										2									

Figure 2: Matrix A^1. A partially partitioned matrix of calculations, showing an object `g` and a constructor $\bar{\texttt{g}}$ of a new class `G`, and two occurrences of a new method `m`.

- (R4). It appears, therefore, that method `m1`, being an indirect class constructor for object `h1`, needs to be assigned to class `H`, passing an object of class `G` as an argument. The same consideration applies to `m2`.
- (R8). We notice that domains `ta, tc` are used in row 14 of A^2 to calculate `te`, which is then used in row 18 to recalculate `Rx`. Since neither `ta, tc, te, Rx` have any other connections, we can encapsulate `Rx` and `te` into `h1` and make them temporaries in some method.
- (R8). We also notice that `tg` is connected only inside class `H`, in row 20, other than its constructor in row 16. This makes `tg` also qualify as a temporary.
- (R14). At this point, rows 14, 18, 20 have all their references in `h1`. They qualify as a private method of class `H`. Alternatively, they can be incorporated into an existing method. We have chosen to incorporate them into method `m1`. The constrains do not prevent this.
- (R8). The same considerations apply to `h2`, the second instance of class `H`, with respect to domains `Ry, tb, td, tf, th` and rows 15, 19, 21. The result is matrix A^3, shown in Figure 4.

			g			$h1$					$h2$								literals								
	PK	OP	d	a	b	Vx	Fx	ta	tc	tg	Vy	Fy	tb	td	th	Rx	Ry	te	tf	1	2	3	4	5	6	7	8	9
	1	=	c																	1								
$\bar{g}$	2	=		c																	1							
	3	=			c																	1						
	4	=														c							1					
	5	=															c							1				
	6	=				c																			1			
	7	=									c															1		
	8	=					c																				1	
	9	=										c																1
	12	$*$	1			2			c																			
$m1$	10	$*$		1			2	c																				
	16	$*$			1		2			c																		
	13	$*$	1								2			c														
$m2$	11	$*$		1								2	c															
	17	$*$			1							2			c													
	14	$+$						1	2								c											
	15	$+$											1	2					c									
	18	$+$														m		2										
	19	$+$															m		2									
	20	$+$				m				2																		
	21	$+$									m				2													

Figure 3: Matrix A^2, illustrating two instances `h1, h2` of a new class `H`, in addition to the previously existing object `g`, constructor $\bar{g}$ and methods `m1, m2`.

Matrix A^3 can be used to illustrate an application of the Fowler refactoring Move Method [2]. Suppose a request is made (quite improperly) to move method `m1` to class `G`. One way to achieve that would be to move temporaries `ta, tc, tg, te` as temporaries to class `G`, and give class `G` access privileges to modify class `H` members `Rx, Vx`. That completes the refactoring.

The partitioning is now finished, but there is one more point to make: application of rule R6. The various classes and methods in matrix A^3 can be extracted into separate models, leaving only one row and one column each in matrix A^3. Define the *class matrices*:

$\bar{g} =$

OP	d	a	b	$i1$	$i2$	$i3$
$=$	c			1		
$=$		c			1	
$=$			c			1

			g			$h1$							$h2$							literals								
	PK	OP	d	a	b	Rx	Vx	Fx	ta	tc	tg	te	Ry	Vy	Fy	tb	td	th	tf	1	2	3	4	5	6	7	8	9
	1	=	c																	1								
$\bar{g}$	2	=		c																	1							
	3	=			c																	1						
	4	=				c																	1					
$\bar{h1}$	6	=					c																	1				
	8	=						c																			1	
	5	=											c											1				
$\bar{h2}$	7	=												c												1		
	9	=													c													1
	12	$*$	1				2			c																		
	10	$*$		1				2	c																			
$m1$	16	$*$			1			2			c																	
	14	$+$							1	2		c																
	18	$+$				m						2																
	20	$+$					m				2																	
	13	$*$	1											2			c											
	11	$*$		1											2	c												
$m2$	17	$*$			1										2			c										
	15	$+$														1	2		c									
	19	$+$											m						2									
	21	$+$												m				2										

Figure 4: Matrix A^3, illustrating two new constructors $\bar{h1}$ and $\bar{h2}$ for class `H`, and the expanded methods `m1` and `m2`.

$\bar{h} =$

OP	R	V	F	$i4$	$i5$	$i6$
$=$	c			1		
$=$		c			1	
$=$			c			1

$m =$

OP	d	a	b	R	V	F	$t1$	$t2$	$t3$	$t4$
$*$	1				2			c		
		1				2	c			
			1			2			c	
$+$							1	2		c
$+$					m					2
$+$				m		m			2	

With these definitions, matrix A^3, the main model, can be converted into the much smaller class

matrix A^4:

$$\mathsf{A}^4 =$$

OP	g	$h1$	$h2$	1	2	3	4	5	6	7	8	9
$\bar{g}$	c			1	2	3						
$\bar{h1}$		c					1		2		3	
$\bar{h2}$			c					1		2		3
$m1$	1	m										
$m2$	1		m									

The final design for example E1 can be obtained from matrix A^4 and the definitions above, and is shown in Figure 5. If a language module were available, it would be expected to produce code, for example in C++ or Java, directly from the design. Alternatively, developers can write the code, and then parse it back and compare with the class matrices for debugging.

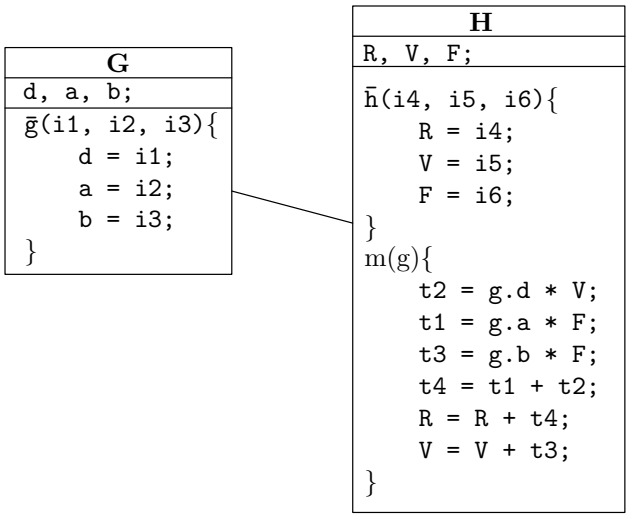

Figure 5: Design D1, the final design for example E1, corresponding to seed selection {`d, a, b`}, and obtained from matrix A^4 and the definitions for class `G`, class `H`, their class constructors, and method `m`.

5 Example E2

This example uses the same simple program Program A of Section 4 used for example E1. Starting with matrix A^0, shown in Figure 1, we explore the effects of seed selection by seeding the algorithm with the second-best seed, {`Vx, Vy`}. Following a procedure similar to the one followed in example E1, a very different result is obtained: matrix A^5, shown in Figure 6. Matrix A^5 describes two classes, say `G` and `V`. Object `g` is an instance of class `G`, and $\bar{\mathtt{g}}$ is the class constructor. Objects `v1, v2, v3, v4, v5, v6` and `v7` are all instances of class `V`. Class `V` has one constructor with 3 invocations, $\bar{\mathtt{v1}}$, $\bar{\mathtt{v2}}$, and $\bar{\mathtt{v3}}$. Class `V` also has three methods, method `w` with 3 invocations, method `p`

		g	$v1$	$v2$	$v3$	$v4$	$v5$	$v6$	$v7$	literals
	PK OP	d a b	R R x y	V V x y	F F x y	t t a b	t t c d	t t e f	t t g h	1 2 3 4 5 6 7 8 9
	1 =	c								1
$\bar{g}$	2 =	c								1
	3 =	c								1
$\bar{v1}$	4 =		c							1
	5 =		c							1
$\bar{v2}$	6 =			c						1
	7 =			c						1
$\bar{v3}$	8 =				c					1
	9 =				c					1
$w1$	10 *	1			2	c				
	11 *	1			2	c				
$w2$	12 *	1		2			c			
	13 *	1		2			c			
$w3$	16 *	1			2				c	
	17 *	1			2				c	
p	14 +					1	2	c		
	15 +					1	2	c		
$q1$	18 +		m					2		
	19 +		m					2		
$q2$	20 +			m				2		
	21 +			m				2		

Figure 6: Matrix A^5, obtained from $matrixA^0$ of Figure 1 by seeding the algorithm with {`Vx, Vy`}. This matrix corresponds to design D2 (not shown).

with 1 invocation, and method `q`, with 2 invocations. Class `V` will immediately be recognized as the very well-known class *vector* in two dimensions. Method `w` is an overload of operator '*', which takes a double and returns a vector. Method `p` is an overload of operator '+', which takes a vector and returns another. And method `w` is an overload of operator '+=', which takes a vector and returns void or a reference to *this*. Class *vector* is indeed the accepted standard for program A. Matrix A^5 corresponds to design D2 (not shown).

We recall that matrix A^5 contains a set of relations that describe the structure of the functions in program A. Although the subject of this paper is not development, it is important to motivate interest by establishing a connection. Suppose the developers were asked to convert program A from two dimensions to three. The conversion is accomplished by adding one attribute to class `V`. That would cause all 7 corresponding block columns, and all 9 corresponding block rows, to expand by 1 line, and 3 more literals will be required for initialization. The matrix, currently of size 21×26, will grow to size 30×36, and will be identical to matrix B as reported in our early work [42]. The change to three dimensions will *not* break the partition, meaning that refactoring will not be necessary. However, assume now that a different request is made for matrix A^5, that developers add (obviously inappropriate) code to calculate something like

```
Fx = Rx + Vy
```

This would invalidate the partition of matrix A^5, and require a major refactoring. Problems that ensue when developers don't realize that the structure is damaged and continue working are well known.

6 Example E3

The problem is: calculate the cost to equip an aircraft model A, B or C to carry mail or spray crops. Figure 7 shows the initial design. The required services are offered by a Document class, which has three subclasses, AModel, BModel and CModel, in correspondence with models A, B and C. The code fragments in the subclasses are used to prepare lists of equipment and call methods in the associated Mail or Spray classes to calculate the actual cost. For example, AM determines what mail equipment is needed for a model A aircraft, instantiates a Mail object, and invokes its MCost method, passing the list, and AS creates a list of spray equipment for model A, instantiates a Spray object and invokes its SCost method. The problem with this design is that it contains mail and spray functionality in all Document subclasses, making it difficult to understand or upgrade. An upgrade for passenger or cargo transportation would require changes in all the subclasses, and would make it even more complex and difficult to understand. The design needs refactoring.

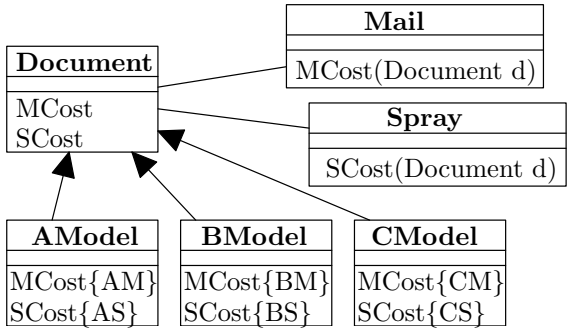

Figure 7: Document class hierarchy and helper classes for Design D3a.

The first step is to separate the object-oriented features and store them in a separate structure, such as the class matrices explained in Section 4, or a graph representation such as the one defined in [20], conveniently augmented to include any language-specific features that need to be preserved. We prefer the class matrix representation because it interfaces better with the rest of the model. There is no need to do that here explicitly.

The following C code remains after the object-oriented features have been separated and the class hierarchy has been replaced with conditional logic:

Program 1

```
1.    b = b0; // A, B or C
2.    a = a0; // mail or spray
      switch(a){
3.    case mail:
```

```
                switch(b){
4.              case A: AM; break;
5.              case B: BM; break;
6.              case C: CM; break;
                }break;
7.      case spray:
                switch(b){
8.              case A: AS; break;
9.              case B: BS; break;
10.             case C: CS; break;
                }break;
        } //end switch(a)
11.     stop;
```

The C code is not part of the model, it is meant only for presentation. Execution begins when a document describing an aircraft model A, B or C is made available to the program. This is symbolically indicated by statement 1. Statement 2 symbolizes that a second document describing either mail or spray equipment has been provided, and initiates a request to calculate the corresponding installation cost. In pursuit of the request, the program considers two cases, mail or spray, as indicated by statement switch(a). In each case, and depending on the model of the aircraft, helper code is executed to calculate the desired cost, for example AM calculates the cost for the "A model-mail" combination, and so on.

The sequences of execution for Program 1 are given in the following table. The constraints or functions will not be given explicitly because they are very simple in this example.

actor	stat	a	b	next
1				1
	1			11
2				2
	2	mail		3
	2	spray		7
	3		A	4
	3		B	5
	3		C	6
	4			11
	5			11
	6			11
	7		A	8
	7		B	9
	7		C	10
	8			11
	9			11
	10			11

The table actually contains several relations of various degrees. We separate explicitly the two relations that contain the conditional logic:

R1

stat	a	next
2	mail	3
2	spray	7

R2

stat	b	next
3	A	4
3	B	5
3	C	6
7	A	8
7	B	9
7	C	10

Relations R1, R2 define the *refactoring region* for this problem. The region has several *input links* in column R1.stat, and several *output links* in column R2.next. It also has a number of *internal links* in column R2.stat, which act as foreign keys, and the corresponding primary keys R1.next. Refactoring implies changing the sequence of execution in the refactoring region, but only the internal links can be changed. The external links, as well the correspondence between foreign keys and primary keys, must be preserved. For example, we must make sure that a path still exists after refactoring between 2 in column R1.stat and 4 in column R2.next passing through "mail" and "A". To ensure consistency of the internal links, we create relation R3 below, as a join of R1 and R2 on R1.next = R2.stat. This step ensures behavior preservation.

R3

R1.stat	a	b	R1.next = R2.stat	R2.next
2	mail	A	3	4
2	mail	B	3	5
2	mail	C	3	6
2	spray	A	7	8
2	spray	B	7	9
2	spray	C	7	10

Relation R3 contains the same information as relations R1 and R2, but is not normalized. In fact, R1 and R2 are the normalized versions of R3, and can be obtained from R3 by normalizing it. A *projection* [43] of R3 on {R1.stat, a, R1.next} yields R1, and a projection on {R2.stat, b, R2.next} yields R2. To effect the refactoring, we change the internal links as indicated in relation S3 below:

S3

R1.stat	a	b	R1.next = R2.stat	R2.next
2	mail	A	12	4
2	mail	B	13	5
2	mail	C	14	6
2	spray	A	12	8
2	spray	B	13	9
2	spray	C	14	10

where 12, 13, 14 are names for the new internal cases. To normalize relation S3, we perform the projections on {R1.stat, b , R1.next} and {R2.stat, a, R2.next}, and we obtain the new relations S1 and S2, respectively:

S1

stat	b	next
2	A	12
2	B	13
2	C	14

S2

stat	a	next
12	mail	4
13	mail	5
14	mail	6
12	spray	8
13	spray	9
14	spray	10

Integrating S1 and S2 with the remaining sequences, the new refactored sequences are obtained as follows:

actor	stat	b	a	next
1				1
	1			11
2				2
	2	A		12
	2	B		13
	2	C		14
	12		mail	4
	12		spray	8
	6			11
	8			11
	13		mail	5
	13		spray	9
	5			11
	9			11
	14		mail	6
	14		spray	10
	6			11
	10			11

and the refactored C code is:

Program 2

```
1.      b = b0; // A, B or C
2.      a = a0; // mail or spray
        switch(b){
12.     case A:
                switch(a){
4.              case mail: AM; break;
8.              case spray: AS; break;
                }break;
13.     case B:
                switch(a){
5.              case mail: BM; break;
9.              case spray: BS; break;
                }break;
14.     case C:
```

```
                switch(a){
6.              case mail: CM; break;
10.             case spray: CS; break;
                }break;
        } //end switch(b)
11.     stop;
```

The last task remaining before the refactoring is complete, is to obtain an object-oriented design from Program 2. We will convert all conditional logic in Program 2 into class hierarchies. As explained before, making the conversion is optional, depending on the metrics. One can convert all conditionals, some of them, or none. For the sake of this example, we have chosen to convert all. The conditional logic in Program 2 is controlled by two parameters, a and b. There is 1 appearance of switch(b) with three cases, and 3 appearances of switch(a) with 2 cases each. The 3 appearances of switch(a) have the same structure. This results in two class hierarchies, one with 2 subclasses associated with variable a, and the other with 3 subclasses associated with variable b, all represented in Figure 8, where we have reused the same names for the b hierarchy and we have named the a hierarchy Visitor. In the refactored design, the required services are still offered by class Document,

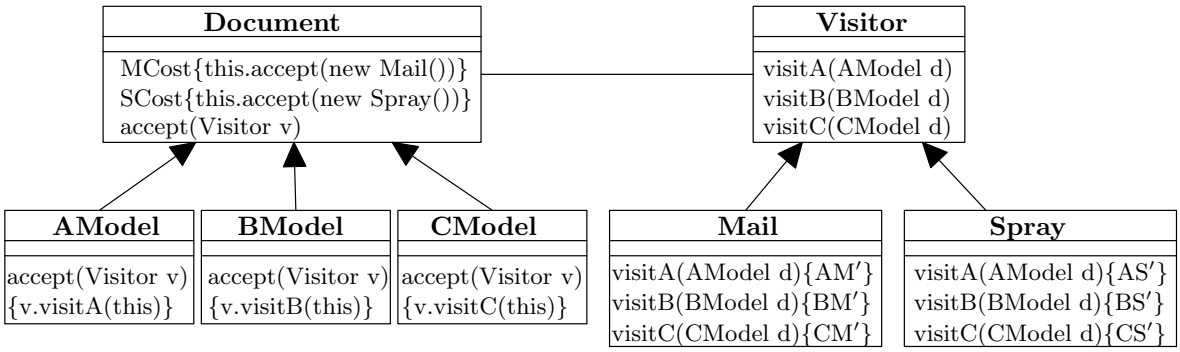

Figure 8: Design D3b. Refactored design corresponding to relation S3.

but MCost and SCost work differently: they resolve between aircraft models and between mail and spray immediately, and use a separate class hierarchy for detailed processing. For example, MCost discriminates between mail and spray by instantiating a Mail object, and invokes accept, which in turn discriminates between models by invoking separate methods in Visitor. The code fragments in the Visitor subclasses perform the actual calculations, For example, AM′ determines a list of mail equipment for a model A aircraft and calculates its cost. All cost calculations are now grouped and class Visitor offers a common interface for them. An upgrade to cargo transportation would only require a new subclass in the Visitor hierarchy, with the appropriate visit methods. Of course, upgrading to a new aircraft model D would now require changes in all subclasses of Visitor, but the new design is still much more understandable.

It is important to note that in [3], a *Visitor* design pattern [44] was used to define the refactoring, and the transformation took 20 elementary Fowler refactorings, all at the level of the object-oriented code and under the control of a refactoring tool. Our method is very different. There is no concept of Fowler refactorings. The only refactoring that took place is the conversion of relation R3 into relation S3, obtained by replacing set {3, 3, 3, 7, 7, 7} with set {12, 13, 14, 12, 13, 14}. It may

be argued that selecting the replacement set is equivalent to using a design pattern. A very simple one, indeed. The rest of the procedure involves using well-established relational operations to deal with relations, and language-specific tools such as a parser and a conversion module to deal with the object-oriented code. The responsibilities are well separated, the refactoring tool deals with refactoring in a language-independent manner, and the language tools deal with languages in a refactoring-independent manner.

A final point is worth making. We have followed certain rules to obtain the design of Figure 8 from relation S3, which we had obtained in turn by refactoring relation R3. What would happen if we applied the same rules directly to relation R3, *without* refactoring? It is not difficult to show that the design of Fig. 9 would be obtained. Design D3b is an authentic refactoring of D3a, because

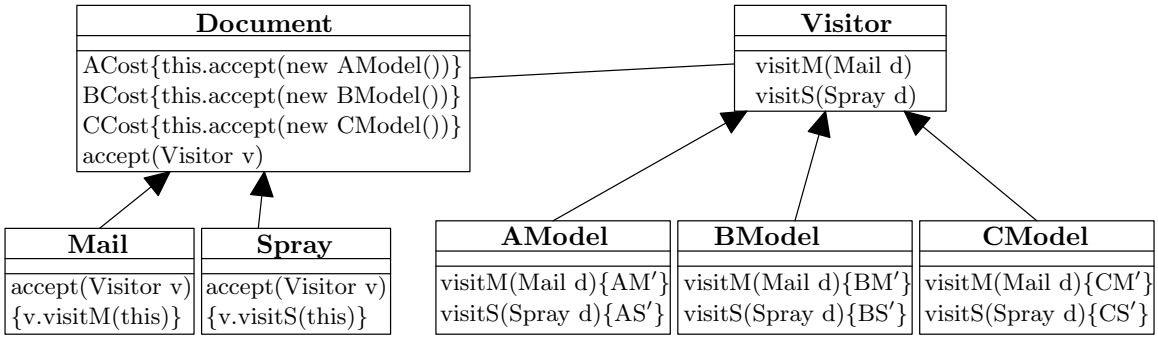

Figure 9: Design D3c, obtained directly from relation R3.

it has been obtained by means of a structural change, the refactoring of relation R3 into relation S3. Design D3c, however, is separated from D3a only by the choice of whether a condition is described by a conditional statement or by a class hierarchy. A comparison between D3b and D3c shows that the roles of aircraft models and tasks are reversed. Only designs equivalent to D3a and D3b are mentioned in [3], but there is no equivalent to D3c, suggesting that the model can discover new possibilities that may be missed otherwise. This is not our only experience in this regard.

7 Summary and Conclusions

From an operational standpoint, the technology presented here consists of three main components with well separated responsibilities, because refactoring is performed by a language-independent tool while language issues are handled by language-specific tools that understand their respective languages. The first component is the parser, which should accept the source code. Optionally, parsers can also accept UML designs, Java bytes code, CodeDOM or MSIL. The output from the parser is the database described in Section 2, including the sparse matrices, their partitions, and the necessary class matrices. This operation is not very different from what a regular parser does, including handling preprocessor directives and constructing compilation units, although modifications are necessary to produce the right type of output. The parser should be incremental because the code segments that need parsing depend on the refactoring. No serious obstacles should arise for developing the parser.

The second component is the algorithm of Section 3, the heart of the system, which understands

and implements the rules for refactoring. The component should offer a menu of recommendations for the analyst to determine a target for refactoring, such as a method or a class. A text search in the source code can find other places where references are made to the target. These can be left expressed as references in the main model, or may have to be fully parsed, depending on the refactoring. An important advantage of the algorithm is that it is very visual and simple enough to be understood by anyone with an object-oriented background. It could even be presented visually on the screen. The analyst can control the process via menu selections, or let it run on its own. An *undo* feature could be included to encourage users to try their ideas. If references to other parts of the code are found, the other parts will have to be fully parsed as well. The output from this component is a design, with complete methods with their arguments and classes described by class matrices.

The responsibility for the third component, the language module, is to reconstruct the code. The unit would reuse unchanged original code, and generate code for the rest, based on the class structure supplied by the algorithm and following the syntax and semantics of the corresponding language. This includes tasks such as adding visibility qualifiers or the `const` qualifier, replace conditional logic found in the matrix of sequences with polymorphism, or, in the case of C++, deciding where to use references or pointers. Or perhaps not to use pointers at all. Alternatively, reconstruction could be done manually by developers. They would reuse the sections of code that were not changed, and then, working from the new design, they would add new code for the refactored segments. The final step would be to parse it all back into relations, convert into a design, and compare with the design supplied by the algorithm for verification.

One can consider a second mode of operation, where a *current* model of the entire program is kept as a permanent component of a development system, always ready to support refactoring operations. The model would be updated every time that development or refactoring happens.

In conclusion, the proposed technology appears to be feasible and amenable for implementation. It can automate refactoring. If combined with development, it can help to improve coherence in program evolution. The algorithm is simple, visual, easy to understand, and can be controlled by menu selections. It is a good example of divide and conquer, because it spreads out the various ingredients that intervene in refactoring and makes them more manageable. It also offers ample opportunities for collaboration with other important methods. For example, several types of code metrics could be used as part of the algorithm to recommend which refactoring rules are best to apply at each step.

The algorithm is also analytical. Unlike graph representations which concentrate in modeling the object-oriented features of the program and tend to generate additional complexity, relational representations do not add complexity because they model the program in a natural way, uncovering its structural elements and making them readily available for processing. The combination of all these features offers an attractive choice for automatic refactoring.

Biography

Sergio Pissanetzky retired after a rewarding career as a Research Scientist, Professor, Entrepreneur, and Consultant. He earned his Ph.D. degree in Physics in Argentina, from the National University of Cuyo, in 1965.

Dr. Pissanetzky has served as a Member of the Editorial Board of the International Journal for Computation in Electrical and Electronic Engineering, as a Member of the Advisory Committee of the International Journal Métodos Numéricos para Cálculo y Diseño en Ingeniería, and as a member of the International Committee for Nuclear Resonance Spectroscopy, Tokyo, Japan. He has held professorships at Texas A&M University and the Universities of Buenos Aires, Córdoba and Cuyo, Argentina. He has also held positions as a Research Scientist with the Houston Advanced Research Center, where he worked with the Superconducting Supercollider federal project, as Chairman of the Computer Center of the Atomic Energy Commission, Bariloche, Argentina, and as a Scientific Consultant at Brookhaven National Laboratory and Los Alamos National Laboratory. He was the founder of Magnus Software Corporation, where he focused on development of specialized applications for the Magnetic Resonance Imaging (MRI) and the High Energy Particle Accelerator industries.

Dr. Pissanetzky holds several US and European patents and is the author of three books and many peer reviewed scientific and technical papers. He now lives in a quite suburban neighborhood in Texas, where he spends much of his time doing what he loves: research.

References

[1] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, 1992.

[2] Martin Fowler. *Improving the Design of Existing Code*. Addison-Wesley Professional, Boston, Massachusetts, 1999.

[3] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, February 2004.

[4] E. van Emden and L. Moonen. Java quality assurance by detecting code smells. *Proc. Working Conf. Reverse Eng.*, pages 97–108, 2002.

[5] T. Dudziak and J. Wloka. Tool-supported discovery and refactoring of structural weaknesses in code, February 2002.

[6] T. Tourwé and T. Mens. Identifying refactoring opportunities using logic metaprogramming. *Proc. European Conf. Software Maintenance and Reeng.*, pages 91–100, 2003.

[7] J. Rajesh and D. Janakiram. Jiad: a tool to infer design patterns in refactoring. *Proc 6th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming. Verona, Italy*, pages 227–237, 2004.

[8] Jia Liu, Don Batory, and Christian Lengauer. Feature oriented refactoring of legacy applications. *Proc 28th international conference on Software engineering. Shanghai, China*, pages 112–121, 2006.

[9] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. *Proc. Int. Conf. Software Maintenance*, pages 109–118, 1999.

[10] M. Balazinska, E. Merlo, N. Dagenais, B. Lagüe, and K. Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. *Proc. Working Conf. Reverse Eng.*, pages 98–107, 2000.

[11] Stephane Ducasse, Oscar Nierstrasz, and Matthias Rieger. Lightweight detection of duplicated code. a language-independent approach. *IAM-04-02*, pages 1–30, February 2004.

[12] Y. Kataoka, M. D. Ernst, W. G. Griswold, and D. Notkin. Automated support for program refactoring using invariants. *Proc. Int. Conf. Software Maintenance*, pages 736–746, 2001.

[13] Robert W. Bowdidge. Refactoring gcc using structure field access traces and concept analysis. *ACM SIGSOFT Software Engineering Notes, Proc. 3rd International Workshop on Dynamic Analysis WODA '05*, 30(4), May 2005.

[14] F. Simon, F. Steinbrückner, and C. Lewerentz. Metrics based refactoring. *Proc. European Conf. Software Maintenance and Reeng.*, pages 30–38, 2001.

[15] M. Lanza and S. Ducasse. Understanding software evolution using a combination of software visualization and software metrics. *Proc. Langages et Modèles à Objects*, pages 135–149, August 2002.

[16] W. L. Hürsch and L. M. Seiter. Automating the evolution of object-oriented systems. *Proc. Symp. Object Technology for Advanced Software*, pages 2–21, 1996.

[17] L. Tahvildari and K. Kontogiannis. A metholdology for developing transformations using the maintainablity soft-goal graph. *Proc. Working Conf. Reverse Eng.*, pages 77–86, October 2002.

[18] P. Bottoni, F. Parisi-Presicce, and G. Taentzer. Coordinated distributed diagram transformation for software evolution. *Electronic Notes in Theoretical Computer Science*, (4):72, 2002.

[19] N. Van Eetvelde and D. Janssens. A hierarchical program representation for refactoring. *Proc. UniGra Workshop*, 2003.

[20] Tom Mens, Niels Van Eetvelde, Serge Demeyer, and Dirk Janssens. Formalizing refactorings with graph transformations. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(4):247–276, July 2005.

[21] Tom Mens, Gabi Taentzer, and Olga Runge. Analysis refactoring dependencies using graph transformation. *Software Systems Modeling (SoSyM)*, 2006.

[22] Mathieu Verbaere, Ran Ettinger, and Oege de Moor. Jungl: a scripting language for refactoring. *Proc 28th International Conference on Software Engineering. Shanghai, China*, pages 172–181, 2006.

[23] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya. A quantitative evaluation of maintainability enhancement by refactoring. *Proc. Int. Conf. Software Maintenance*, pages 576–585, October 2002.

[24] L. Tahvildari and K. Kontogiannis. A metric-based approach to enhance design quality through meta-pattern transformations. *Proc. European Conf. Software Maintenance and Reeng.*, pages 183–192, 2003.

[25] Serge Demeyer. Maintainability versus performance: What's the effect of introducing polymorphism? Technical report, Lab. on Reeng., Universiteit Antwerpen, Belgium, 2002.

[26] M. V. Ksenzov. Architectural refactoring of corporate program systems. *Programming and Computing Software.*, 32(1):31–43, January 2006.

[27] M. Bauer and M. Trifu. Architecture-aware adaptive clustering of oo systems. *Proceedings of the Eighth European Conference on Software Maintenance and Reengineering, 2004. (CSMR 2004)*, pages 3–14, March 2004.

[28] Ran Ettinger and Mathieu Verbaere. Untangling: a slice extraction refactoring. *Proc 3rd International Conference on Aspect-oriented Software Development. Lancaster, UK*, pages 93–101, 2004.

[29] Jan Hannemann, Gail C. Murphy, and Gregor Kiczales. Role-based refactoring of crosscutting concerns. *Proc 4th international Conference on Aspect-oriented Software Development. Chicago, Illinois*, pages 135–146, 2005.

[30] Mirko Streckenbach and Gregor Snelting. Refactoring class hierarchies with kaba. *Proc 19th annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications. Vancouver, BC, Canada*, pages 315–330, 2004.

[31] Marius Marin, Leon Moonen, and Arie van Deursen. An approach to aspect refactoring based on crosscutting concern types. *Proc 2005 Workshop on Modeling and Analysis of Concerns in Software. St. Louis, Missouri*, pages 1–5, 2005.

[32] Alejandra Garrido and Ralph Johnson. Challenges of refactoring c programs. *Proc International Workshop on Principles of Software Evolution. Orlando, Florida*, pages 6–14, 2002.

[33] Harold Thimbleby. User interface design with matrix algebra. *ACM Transactions on Computer-Human Interaction (TOCHI)*, pages 181–236, 2004.

[34] Suzanne Smith, Sara Stoecklin, and Catharina Serino. An innovative approach to teaching refactoring. *Proc. 37th SIGCSE Technical Symposium on Computer Science Education SIGCSE '06*, 38(1), March 2006.

[35] Michael J. Wester (editor), editor. *Computer Algebra Systems - A Practical Guide*. Wiley, Chichester, 1999.

[36] Donald E. Knuth. *The Art of Computer Programming*, volume Vol. 2: Seminumerical Algorithms. Addison-Wesley, 1998.

[37] Christian Bauer, Alexander Frink, and Richard Kreckel. Introduction to the ginac framework for symbolic computation within the c++ programming language. *J. Symbolic Computation*, (33):1–12, 2002.

[38] Sergio Pissanetzky. *Sparse Matrix Technology*. Academic Press, London, 1984. Russian translation, Mir, Moscow, 1988.

[39] A. L. Dulmage and N. S. Mendelsohn. A structure theory of bipartite graphs of finite exterior dimension. *Trans. Roy. Soc. Canada*, 53:1–13, 1959.

[40] Bin Gao, Tie-Yan Liu, Xin Zheng, Qian-Sheng Cheng, and Wei-Ying Ma. Consistent bipartite graph co-partitioning for star-structured high-order heterogeneous data co-clustering. *Proceeding of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining. Chicago, Illinois, USA*, pages 41–50, 2005.

[41] E. M. Garz and I. Garca. Approaches based on permutations for partitioning sparse matrices on multiprocessors. *The Journal of Supercomputing*, 34(1):41–61, October 2005.

[42] Sergio Pissanetzky. *Refactoring with Relations. A new Method for Refactoring Object-Oriented Software*. SciControls.com, Texas, USA, July 2006.

[43] E. F. Codd. A Relational Model of Data for large shared Data Banks. *Comm. ACM*, 13(6):377–387, 1970.

[44] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Languages and Systems*. Addison Wesley, 1994.