Refactoring With Relations

A New Method for Refactoring Object-Oriented Software

Sergio Pissanetzky

Professionally typeset by LaTeX. This work is in compliance with the mathematical typesetting conventions established by the International Organization for Standardization (ISO).

Dr. Pissanetzky retired after a rewarding career as an Entrepreneur, Professor, Research Scientist and Consultant. 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. He has served as Member of the International Editorial Board of the "International Journal for Computation in Electrical and Electronic Engineering", as a Member of the International 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. Dr. Pissanetzky has held professorships in Physics 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, as Chairman of the Computer Center of the Atomic Energy Commission, San Carlos de Bariloche, Argentina, and as a Scientific Consultant at Brookhaven National Laboratory. Dr. Pissanetzky holds several US and European patents and is the author of three books and numerous peer reviewed technical papers. Dr. Pissanetzky earned his Ph.D. in Physics at the Balseiro Institute, University of Cuyo, in 1965. Dr. Pissanetzky has 35 years of teaching experience and 30 years of programming experience in languages such as Fortran, Basic, C and C++. Dr. Pissanetzky now lives in a quite suburban neighborhood in Texas.

Website: http://www.SciControls.com

Disclosure
The material presented in this work is the subject of ongoing research. The ideas and algorithms, though theoretically sound, have not been tested other than with the examples in the book. No representation is made regarding their practical value.

Trademark Notices
Product and company names mentioned herein may be the trademarks of their respective owners.

ISBN 0-9762775-4-9

Refactoring With Relations

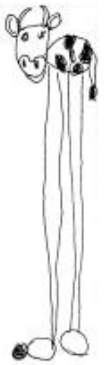
A New Method for Refactoring Object-Oriented Sofware

First edition

Sergio Pissanetzky

July 2006

A cow's legs are very long, they reach all the way to the ground.
Anonymous. My best attempt at explaining that programs, too, have a foundation.



Preface

This book is an introduction to the Relational Model of Computer Programs and one of its many applications: the automatic polyglot refactoring of computer software. Refactoring is frequently defined as a behavior-preserving source-to-source transformation of a computer program, performed with the purpose of improving the quality of code. There are several problems with this definition. We can "develop" a program, "compile" a program, "buy" or "sell" a program, "install" a program, "execute" a program. Clearly, refactoring does not transform the program, only its source code. The program is not the same as the code.

There are no precise definitions of "behavior" or "quality". Without them, we don't know exactly what it is that we are trying to preserve, or to improve, and we can't automate refactoring.

As a consequence, refactoring is still perceived as a mostly manual, last resort operation, dangerous, expensive and error-prone. The fear of change originates from insufficient control. Tools have been slow to appear. The few existing tools are language-specific, cover only light refactorings, and leave the critical determinations of quality and behavior to the user. Several are not dependable because of the bugs they contain [17].

There is no generally accepted theory of refactoring. Some theories model the source code, not the program, and try to preserve as much of the source code as possible. And there is a literature explosion. Several books teach developers how to refactor or when to refactor, or how to write programs that need fewer refactorings in the first place. Refactoring is recognized as a major obstacle in software development. Even new languages have been created, motivated in part by a desire to eliminate some known sources of refactorings. Why is it becoming so difficult to be a developer? The sheer abundance of this material is telling us that something is wrong.

Clearly, the *program* is what needs to remain unchanged. In this work, we create a description of the program in terms of relations, called the *relational model*. The model defines precisely what is meant by the term "behavior." It provides a point of reference, a canonical representation of the program, a container that preserves the structure and function of the program in a pristine state. It contains no language-specific features, and no object-oriented features, and it stands unique on both counts. It can be generated from existing code by a specialized parser. The model is, effectively, a relational database. The model can be directly compiled or caused to run in an interpreter. The

model effectively separates the linguistical features from the structure of the code.

Next, we identify those transformations that leave the program invariant. We call them *invariant transformations*. These, and no others, are the refactoring transformations. The set of all invariant transformations solves a fundamental problem, because it defines the term "behavior-preserving". The ability to identify the invariant transformations is an essential feature of the model. Yet, they are very easy to identify, and we think we know why.

Mathematics offers three tools for dealing with the complexity of information: the set, the graph, and the relation. Every organized body of information is either a set, a graph, or a relation. A computer program is a relation. The relational model describes the program mathematically by means of a set of relations. In the book, we give a more rigorous definition of relation, but for now, let us think of a relation as a database table. Each column of the table is a *type* or *domain*. A type is a set, a collection of "things", with a name. Any things, even objects or other types. Each row is a *value* or *tuple* of the relation.

If one of the types is identified as a *cotype* or *codomain*, and the remaining types are the *arguments*, then the relation maps the arguments over to the codomain. If the mapping is one-one or many-one, the relation is a *function*. We use the term *relational function* to refer to this definition in particular and differentiate it from other definitions used in programming.

The program is described by four sets of relations, each containing one or more relations. The first set contains all the relational functions in the program, in no particular order. The second set contains relations describing the execution sequences and flow of control statements, some of them dependent on values found in the codomains. The third set contains the predecessor-successor constraints for the sequences, and the last set, the actors that initiate the sequences.

Relations are not dynamic, like programs are. Execution of a program "creates" objects and fills memory with data. Relations, instead, define types as sets of *all possible values* of that type. The values do not have to be explicitly represented in memory, just defined. Thus, execution in the relational model is equivalent to a search operation, where initial input data, intermediate values and final results are successively found in the corresponding sets. Of course, the values found must be explicitly represented, but conceptually, execution is still a search operation. The power of relations stems from these two features: the complete freedom for defining the sets, and the closure property or inclusion of all possible values in the sets.

For presentation purposes, we expand the relations to form a matrix, the rows of which correspond to the relational functions, and the columns to all the variables or literals in the program, with their fully-qualified names. Two more columns are added, one with the primary keys that identify the rows, and the other with operator identifiers. The remaining elements of the matrix identify the arguments, codomains, and mutators. The matrix is a *sparse matrix*, one where only a few elements in each row are meaningful. Sparse matrices are used internally in databases to store relations, but only a full representation is human-readable, and is what we use for all the examples. The remaining components of the model, actors, sequences and constraints, are presented as tables.

The purpose of the matrix presentation of the model is to illustrate the refactoring transformations. The basic transformations consist of row and column permutations. There is really no need

to permute anything, because we keep separate descriptions of the permutations, but we do physically permute rows and columns in order to demonstrate the algorithm. Columns can be permuted freely, but row permutations are legal only when they do not violate the predecessor-successor constraints. Enforcement of the predecessor-successor constraints guarantees the preservation of behavior.

A *sparse matrix partitioning* algorithm is used to create new user types. Partitioning is an NP-complete problem, at least in some cases, which explains why it is so difficult to properly refactor a program by hand. In the relational model, we effectively take the complexity away from the development environment and let the machine deal with it. The algorithm has control parameters and can be guided from business experience or UML models either directly or via menu selections. To illustrate the algorithm visually, we select a subset of variables to be encapsulated, permute the columns to cause the selected columns to be adjacent, and define them as a partition. This partition is a new class with the selected variables as member attributes. We then propagate the partitions to the rows, creating the constructors and member methods for the new class. The algorithm demonstrates that very few type combinations are legal, because most violate the constraints, but this can be determined beforehand and the menu selections can be based on that determination. As new types are created, they are separated into new structures, also relational, that become definitions of the new classes and the implementations of their methods. The new structures become part of the relational model, the partitions in the matrix are replaced with their new types, and the process continues recursively until there is nothing more left to encapsulate. The new structures, in turn, may have to be partitioned into more new types.

The visual display of the matrix offers a striking demonstration of the workings of the algorithm, the creation of new user types, and the pervasive nature of encapsulation. It also helps to visualize why development frequently breaks the structure of a program and makes refactoring necessary. Refactoring certainly appears very different in this new light.

There remains the issue of the source code and its quality. After encapsulation, the relational model contains a description of the object-oriented features of the program. Generating the code itself from the model is a matter of applying the semantical and syntactic rules of the language. For now, this has to be done manually. We expect, however, that language modules will be developed to automate this task. Our concept is that languages should *subscribe* to the relational model by providing the specialized parser and the conversion module for that language. This will allow conversion from the relational model to any subscribed language, as well as automatic translation of code between languages. The fact that the resulting code is machine-generated defines the concept of quality.

Several examples are presented in the book. One of them demonstrates the automatic conversion of a 30-line C program into C++ code. Two different encapsulations are used to create two different refactorings of the same program. The set of all possible refactorings corresponds to the set of all possible legal partitionings of the sparse matrix. Structure-breaking development and the ability of the algorithm to fix development errors are both illustrated by other examples.

The relational model can promote tool integration and improve connectivity between various

tools used to manipulate a program during its lifetime, such as UML models. Some of the tools use alternative models that interface only manually with the program. But the models describe functional features of the program, the *same* features that are described by relations in the relational model. They are in fact the same relations. There must be a one-one correspondence and a seamless integration.

The scope of our proposition turns out to be much wider than refactoring alone, even than computation alone. Basically, anything that functions or does something is a relation and is amenable to a similar model and similar processing. We learn in terms of relations, but we reason and communicate in terms of objects. How do those relations become objects in our minds? Can a relational model help to understand intelligence? Can it help to understand the process that goes on in an analyst's mind when he or she decides to define a new class? A relational model of genetic information stored in DNA might one day shed light on the logic of DNA. How is the information that tells a cell how to multiply or what to become actually stored? Will natural languages also fall within reach of the relational model? These questions are fascinating, but we are not going to say much more about them. For now, we concentrate on refactoring alone.

There is a great deal of work to be done, and we hope to encourage some research in the area. A critical issue is algorithms for the recognition of object-oriented features in the partitioned matrix. For example, we have covered user types and member methods, and some of polymorphism, but we have no material yet on inheritance. Let's stop trying to strengthen the refactoring tree by pulling from its leaves. Let's work on the roots for a change.

The basic concepts that support the Relational Model for computer programs are discussed in Chapter 1. This chapter also includes a brief tutorial on relations. Chapter 2 introduces the model by example, and presents an elementary discussion of the mechanics of the type encapsulation algorithm used in the model. Chapter 3 introduces a sparse matrix representation of the relational model, and demonstrates how the block-partitioning of the sparse matrix is equivalent to type encapsulation and the creation of new user types and other object-oriented features. Several explicit examples for the cases discussed in Chapter 2 are presented. The sparse matrix representation is a means for visualizing the model and the relational transformations performed by the encapsulation algorithm.

Chapter 4 discusses some of the experimental evidence that supports our concepts. Presented is a case study on the refactoring of a 40,000 line C++ program. The definition of deep refactoring is introduced, and the true nature and complexity of proper refactoring is discussed. A programming style called Strong Ownership is introduced as a temporary solution for developers, until new automated tools based on the relational model are developed.

Chapter 5 is an outlook to the future. We discuss the possible application of the relational model to other areas of computer programming, such as development, reverse engineering, maintenance, code reuse, and translation of code between languages. There is also an outlook to the future, with applications far beyond the reach of computer programming.

Who should read this book

This book is of interest to computer scientists and tool developers. Anyone working on methods for software development should read it. The theory described here is also of interest for reverse-engineering and code reuse and maintenance. Analysts will find the material interesting and suggestive, but not directly applicable to their work. There is only one section of direct interest to developers, and that is the section on Strong Ownership. We recommend that every developer should read this material and start using the SO programming style now. However, since it is not worth buying the book just to read two pages, the section on SO has been included in the page sample, which is available from our web site. There is no other material useful for developers, unless they are very curious. Developers and analysts need to wait until new tools are developed.

Long-legged cow by Isabella Pissanetzky

Contents

1 Description of the Relational Model **1**
- 1.1 Motivation . . . 1
- 1.2 Relation Basics . . . 4
- 1.3 The Relational Model . . . 5
- 1.4 Execution in the Relational Model . . . 7

2 Relational Model Examples **9**
- 2.1 The Simple Program Example . . . 9
- 2.2 The Sequence of Execution . . . 10
- 2.3 The Mechanics of Conversion . . . 12

3 Refactoring the Sparse Matrix **17**
- 3.1 Sparse Matrices . . . 17
- 3.2 Partitioning P1 . . . 21
- 3.3 Further Processing of Matrix A^3 . . . 26
- 3.4 Partitioning P2 . . . 28
- 3.5 Partitioning P3 . . . 30
- 3.6 Class elements . . . 33
- 3.7 Storage Schemes for the Relational Model . . . 33

4 Deep Refactoring and Strong Ownership **43**
- 4.1 The Nature of Refactoring . . . 44
- 4.2 The Experimental Study . . . 45
- 4.3 A Case Study in Deep Refactoring . . . 47
- 4.4 Detours and Method Misplacement . . . 49
- 4.5 Conclusions from the Experiments . . . 50
- 4.6 Strong Ownership . . . 52

5 Applications and Outlook **55**
5.1 What is Next 55
5.1.1 Relation-assisted Development 56
5.1.2 Refactoring 57
5.1.3 Translation 58
5.1.4 Reverse Engineering 58
5.1.5 Automatic Code Generators 58
5.1.6 Legacy Code 59
5.1.7 Other possible applications 59
5.2 Concluding Remarks 61

phatical view of the pervasive nature of encapsulation. Once a user type forms by permuting columns and encapsulating a set of them as the new type, encapsulation propagates through the entire matrix, forcing functions and other variables to be encapsulated as well. Which in turn forces still other functions and other variables to also be encapsulated. Encapsulation is a global feature of the program, definitely not a local feature. The global nature of encapsulation is at the root of the need for refactoring code.

Using sparse matrices implies a higher level of abstraction. For readers who are not familiar with the subject, we include a short tutorial. Readers are encouraged to read section 3.1 carefully. It is also the most difficult to read but it is the one that contains the important material. Sparse matrices offer the best possible view of the process, but are hard to fit in a page, even for the simplest of examples, so readers are advised to make enlarged paper copies of them.

We said that sparse matrices are effective for visualizing a small example, but they may not be appropriate for dealing with the complex searches and relational operations involved in the conversion. The container should be based on the relational model, which has the required capabilities. This example intends to illustrate the following points:

- Show how a program can be expressed as a set of relations and is itself a relation.
- Demonstrate how relational operations can be used to convert code from the relational model, or from a C program, directly into object-oriented code, such as C++.
- Show two different refactorings of the same C++ program obtained with different sequences of relational operations.
- Present the automation of the classical refactoring Move Method [6].
- Point out some limitations of current refactoring methods.
- Discuss the use of the relational model as a container for the structure of code.
- Illustrate the malleability or capacity of the model for adaptive change under the control of outside influences.
- Discuss the application of a sparse matrix to describe an entire program, using the same format for object-oriented and non-object-oriented code.
- Illustrate how various object-oriented structures are recognized in the sparse matrix.
- Show that the sparse matrix can be compiled or caused to run in an interpreter.

- Demonstrate that the block-partitioning of the sparse matrix is equivalent to encapsulation and creation of new user types.
- Propose that programming languages, and even machine code, are different ways of expressing the same relations.
- Propose that programming languages, traditionally used one-way for man-machine communication, can also be used by programs to communicate with humans in a dialogue.
- Propose that code models and object models such as UML are different views of the same relational model.
- Propose that refactoring, reverse engineering, compilation and translation between languages are all relational operations based on the relational model.
- Explain why development frequently breaks the structure of code and results in the need for refactoring.
- Discuss some ideas that may lead to new methods for software development.
- Conjecture that learning is the process of acquiring relations, and that the conversion of relations into objects is intelligent thinking.
- Propose that meaning comes from objects, not objects from meaning.
- Propose that the block-partitioning algorithm connects detail with meaning because it creates meaningful objects from detailed relations.

1.2 Relation Basics

This work is for readers who understand software but may be less familiar with relations. For them, and in this section, we cover the basics of relations. When possible, we use informal database language such as *table* for relation, *row* or *record* for tuple, and *column* or *field* for attribute, but we also use the formal terms on occasion.

Domain. In the theory of relations [3], a *domain*, or *type*, is a set with a name. A domain is characterized by two features, the set and the name. This means that two domains are different if their names are different, even if their sets are the same. A set is a collection of "things" considered as a whole, but the things can be anything. They can be other domains,

Chapter 2

Relational Model Examples

2.1 The Simple Program Example

The first step is to enter the data and structure of the code. There is more than one way of doing this, and we will briefly revisit the issue below, in section 5.1.1. For the sake of this example, we start with a simple program written in C and parsed in such a way that each line of code contains one single operation. The program describes one step in the time simulation of the motion of the center of mass of a body, relative to a coordinate system x, y, z and under the action of an applied force, as established by Newton's second law, but this is immaterial for our purposes. All variables and literals in the program are of the same primitive type, say double, and there are no user types. Here is the program:

Program A

```
 1.   d   =  0.2;
2.    a   =  0.1;
3.    b   =  0.5;
4.    Rx  =  1;
5.    Ry  =  2;
6.    Rz  =  3;
7.    Vx  =  4;
8.    Vy  =  5;
9.    Vz  =  6;
10.   Fx  =  7;
11.   Fy  =  8;
12.   Fz  =  9;
```

```
13.   ta   =   a * Fx;
14.   tb   =   a * Fy;
15.   tc   =   a * Fz;
16.   td   =   d * Vx;
17.   te   =   d * Vy;
18.   tf   =   d * Vz;
19.   tg   =   ta + td;
20.   th   =   tb + te;
21.   ti   =   tc + tf;
22.   tj   =   b * Fx;
23.   tk   =   b * Fy;
24.   tl   =   b * Fz;
25.   Rx   =   Rx + tg;
26.   Ry   =   Ry + th;
27.   Rz   =   Rz + ti;
28.   Vx   =   Vx + tj;
29.   Vy   =   Vy + tk;
30.   Vz   =   Vz + tl;
```

All statements in program A are expressions. The program is very simple, of course, but a suitable parser can convert any OO program into similar C code. This already illustrates one of the ways for entering the code data into the model: by parsing existing OO code. The existing code could be legacy code or new code under development, and it is obviously not restricted to any particular language.

As explained in Section 1.2, a *domain* is a set with a name, and a *function* is a relation that describes a one-to-one or many-to-one mapping from values in the argument domains to a value in the result domain or *codomain* [3]. A domain is also known as a *type*, and a codomain as a *cotype*, and we use all the terms interchangeably. For example, consider expression 13 above, which describes a function with the arguments a and Fx and the cotype ta. It can be described by a relation similar to (7) in Section 1.3. A program consists entirely of relations. In fact, the entire program can be described by a single relation.

2.2 The Sequence of Execution

The sequence of execution describes the order that the executable statements in a program must execute. *Actors* initiate a sequence of execution by interacting with the program, for example via the mouse of keyboard. Flow of control statements can alter the sequences,

Matrix A^3 can be converted directly into code in an object-oriented language such as VB, C#, Java, C++ or Pascal. Part of the process of conversion is the recognition of iso-structural blocks in the matrix, which act as patterns for refactoring. As we explained below, block columns correspond to classes, and block rows to member methods in those classes. If the program contains multiple instances of a class, then the same class structure will appear repeatedly, and must be recognized to avoid duplications. In the particular case of matrix A^3, an examination immediately shows a class structure repeated three times, indicating three objects of that class. In an automated version, the ability to compare class structures must be part of the language module. For now, however, since language modules are not yet available, the conversion has to be done manually. Here is the C++ code:

Program D

```
 class G{
        double d, a, b;
        G(double d0, double a0, double b0){d = d0; a = a0; b = b0;}
};
class H{
        double R, V, F;
        H(double R0, double V0, double F0){R = R0; V = V0; F = F0;}
        void M(const G & g){
                double td = g.d * V;
                double ta = g.a * F;
                double tj = g.b * F;
                double tg = ta + td;
                R += tg;
                V += tj;
        }
};
void main(){
        G g(0.2, 0.1, 0.5);
        H x(1, 4, 7);
        H y(2, 5, 8);
        H z(3, 6, 9);
        x.M(g);
        Rx = x.R;
        Vx = x.V;
        y.M(g);
        Ry = y.R;
```

```
        Vy = y.V;
        z.M(g);
        Rz = z.R;
        Vz = z.V;
}
```

And here is the Pascal code [11] obtained directly from matrix A^3:

Program D1

```
 program D1;
type
    G = class
        d , a , b : double;
        constructor Create( d0 , a0 , b0 : double );
    end;
    H = class
        R , V , F : double;
        constructor Create( R0 , V0 , F0 : double );
        procedure M( aG : G );
    end;
constructor G.Create( d0 , a0 , b0 : double );
begin
    d := d0;
    a := a0;
    b := b0;
end;
constructor H.Create( R0 , V0 , F0 : double );
begin
    R := R0;
    V := V0;
    F := F0;
end;
procedure H.M( aG : G );
var
    td , ta , tj , tg : double;
begin
    td := aG.d * V;
    ta := aG.a * F;
    tj := aG.b * F;
```

At this point, method M can be defined by matrix M as follows:

M =

	OP	d	a	b	R	V	F	t1	t2	t3	t4
1	*	1				2		C			
2	*		1				2		C		
3	*			1			2			C	
4	+							2	1		C
5	+				m						1
6	+					m				1	

and method H, intentionally named after the domain because it will be the constructor of class H, is defined by matrix H:

H =

	OP	R	V	F	R0	V0	F0
1	=	C			1		
2	=		C			1	
3	=			C			1

Using the new definitions, partitioned sparse matrix A^4 is now obtained:

A^4 =

	OP	d	a	b	.2	.1	.5	H1	1	4	7	H2	2	5	8	H3	3	6	9
1	=	C			1														
2	=		C			1													
3	=			C			1												
4	H							C	1	2	3								
5	H											C	1	2	3				
6	H															C	1	2	3
7	M	1	2	3				m4											
8	M	1	2	3								m4							
9	M	1	2	3												m4			

where we have used m4 to indicate that the codomain is also argument 4 of the function. Matrix A^4 is equivalent to A^3. Matrix A^4 can be further processed. Define the new variable:

G = {d, a, b}

with its corresponding domain G, and the new internal domain:

G = {d0, a0, b0}

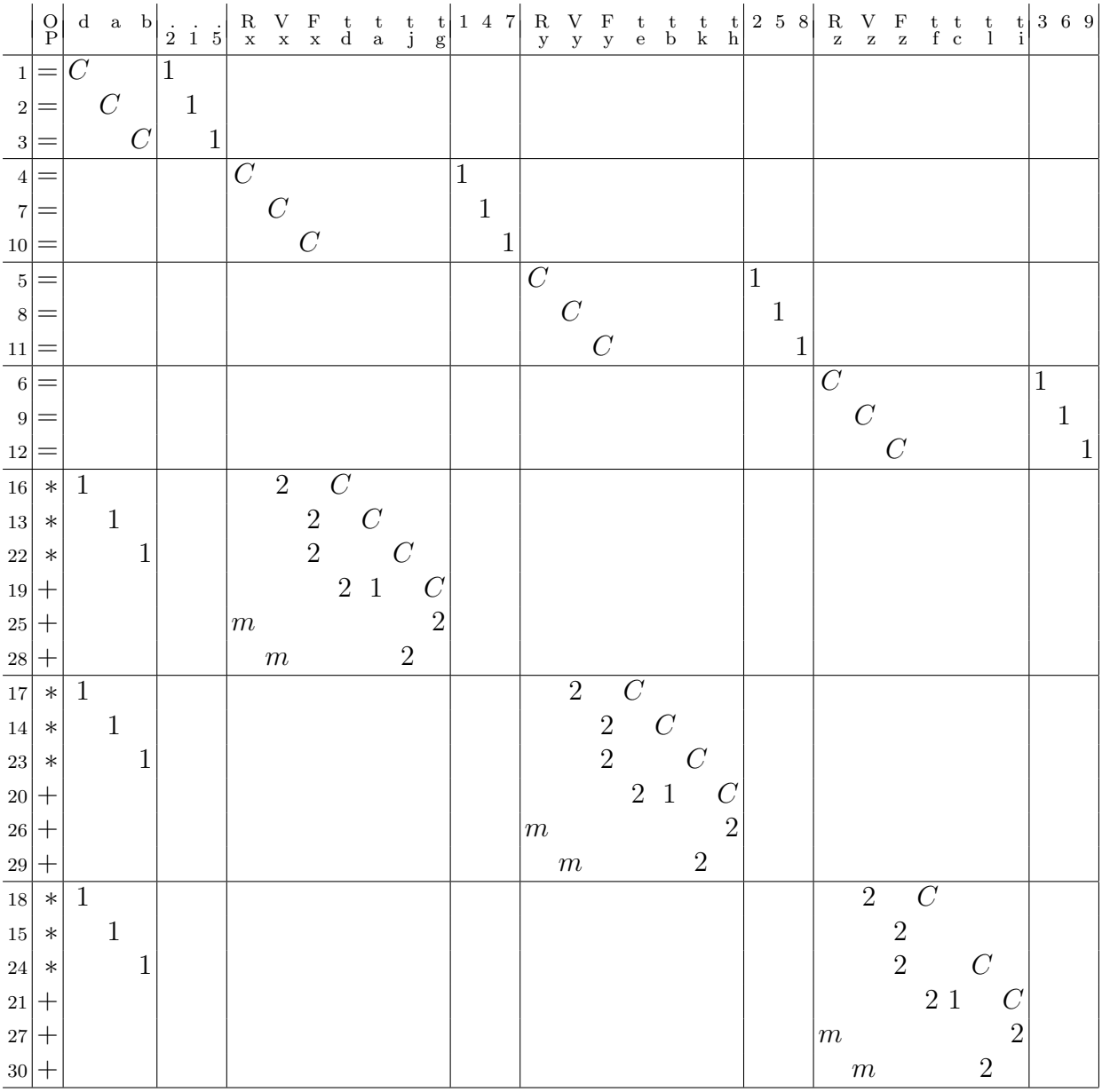


Figure 3.4: Final partition. Matrix A^3, the relational model for Program D.

- If there are no preconditions, perform the refactoring.
- Repeat until all refactorings are complete.

The obvious observation is that the refactoring algorithm itself is iterative, or at least cascading. One refactoring requires other refactorings, which in turn may require still more refactorings.

Prior to the experiment, the code was thoroughly cleaned and debugged. Many light refactorings were performed, where *light refactoring* was defined in section 3.2. Among them, there was one where we protected all public attributes still remaining in the classes, and installed public Get and Set methods where needed. Several of the Get methods were used to supply objects to other classes that needed to call those objects' Set methods. Since the Set methods are non-const, the corresponding Get had to return non-const objects by pointer or reference. Yet, these Get methods were declared with the const qualifier, because they had to be invoked from const objects. Specifically, the following steps were performed prior to the start of the experiment:

R1 Preparation of test cases.

R2 Cleaning and preliminary refactoring. All light refactorings with simple or no preconditions were performed prior to the experiment. Public attributes were eliminated, methods and classes were extracted or moved, hierarchies were rearranged. The initial code looked clean and concise.

R3 Diagnosis. Three types of targets were determined: failure to containerize data structures, encapsulation violations, typically by public const Get methods returning a reference to a non-const object, and casts. A few cases of each type were found, all subject to heavy preconditions.

R4 Decomposition. For each major target, determine the preconditions that need to be satisfied, recursively find the refactorings needed to satisfy each precondition, and decompose each of the sub-refactorings in turn. This resulted in a multi-level tree of refactorings, the *refactoring tree*, associated with each major target.

R5 Backtracking. The leaves of each tree are refactorings with no preconditions. These were performed, and the leaves were removed, creating a new set of leaves. The procedure was repeated until the tree was empty. More rigorously, the tree was not completely static. Some portions of it were affected by the refactorings, and had to be updated.

imposed by the underlying logical structure of the code. They tell us that our programming happened at the leaf level of the tree, and must now rise to the root level. The role of SO is to bring regular development closer to the root level.

The refactoring tree can also be viewed as a path for the iterative solution of a coupled system with many equations and many unknowns. The equations are the logical constraints expressed by the preconditions and the flow of control, and the unknowns are structural features such as the composition of classes, the arguments of methods, or the inheritance hierarchies. To solve the system of equations we must find a structure that satisfies all the equations, but this is not an easy task. Possible approaches are:

A1 Manual refactoring. Solves the system in the hardest possible way: by direct substitution in place. This is the traditional approach.

A2 SO coding. The effect of SO is to decouple the system and make it simpler and easier to solve by hand.

A3 Computerized refactoring. The first step to automate is to expose the equations and unknowns hidden in the code, and make them explicit. The relational model discussed in this book can do this.

We have postulated that the container where the structure of code is to be encapsulated should be based on the relational model. As we have seen, these conclusions are justified in theory in the development of the relational model. Relations can describe data, function, code, relationships, flow of control, actors, scenarios, states, even architecture and relationships between the various models.

4.6 Strong Ownership

SO is simply a characterization of well-known rules of object-orientation. The prescriptions of SO follow below. They are not necessarily disjoint.

SO1 Ownership. Every object has one owner, and one only. The owner constructs the object and destroys it when no longer needed, or prior to its own destruction. The owner, and only the owner, has authority to modify the object after its construction.

SO2 User independence. A class is user-independent, it offers services to its users but makes no assumptions about them.

SO3 Containerization. The structure of data must be encapsulated into the appropriate container. Containers are non-intrusive and do not own their content.

SO4 Casting. Down casts, away from the base class, are not allowed.

SO5 Object-orientation. The rules of encapsulation, inheritance and polymorphism are part of SO.

A modification of SO1 is possible, where ownership of an object can be transferred to another owner by a *sale* procedure. A modification to SO4 can be considered, where downcasts would be allowed in containers. SO1 is parser-enforceable, and so is SO4. SO1 specifically proscribes public const Get methods that return a pointer or reference to a non-`const` object. We believe, but have not tested, that SO should include other features from the relational model discussed below. Paramount among them is a ban on pointers, which are banned from database relations [3].

SO creates a tree, the *ownership tree*, where the vertices are individual objects, not classes, edges represent the ownership relationship, and the parent of an object is its owner. The tree establishes a chain of command and channels for messages and data to travel, and thus effectively encapsulates functionality. SO prevents memory leaks, forces misplaced methods to their right places, and fixes detours. Concepts similar to SO have been described by [9].

Coding in SO style is more difficult that regular coding, but it is a great deal easier than the equivalent regular coding followed by refactoring. It puts more load on the developer's shoulders, but it alleviates the total load for the development team.

Bibliography

[1] A. Chang. Application of sparse matrix methods in electric power system analysis. *Proc. Symposium on Sparse Matrices and their Applications. Yorktown Heights, NY*, pages 113–122, 1969.

[2] A. R. Curtis and J. K. Reid. The solution of large sparse unsymmetric systems of linear equations. *J. Inst. Math. Appl.*, 8:344–353, 1971.

[3] C. J. Date and H. Darwen. *Databases, Types, and the Relational Model. The Third Manifesto.* Addison-Wesley, Reading, Massachusetts, third edition, 2006.

[4] 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.

[5] A. L. Dulmage and N. S. Mendelsohn. *Graphs and Matrices, in Graph Theory and Theoretical Physics.* Academic Press, New York, 1967.

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

[7] F. G. Gustavson. Some basic techniques for solving sparse systems of linear equations. *Sparse Matrices and their Applications. Proc. Symp. at IBM Research Center, New York.*, pages 41–52, 1972.

[8] Joshua Kerievsky. *Refactoring to Patterns.* Addison Wesley Professional - Signature Series, August 2004.

[9] Bartosz Milewski. *C++ In Action: Industrial Strength Programming Techniques.* Addison-Wesley, 2001.

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

[11] Pablo I. Pissanetzky. Pascal program. Private communication, 2006.

[12] Sergio Pissanetzky. *Sparse Matrix Technology.* Academic Press, London, 1984.

[13] Sergio Pissanetzky. *Sparse Matrix Technology - Russian translation.* Mir, Moscow, 1988.

[14] Sergio Pissanetzky. *Vectors, Matrices, and C++ Code.* SciControls.com, Texas, USA, October 2004.

[15] Sergio Pissanetzky. *Rigid Body Kinematics and C++ Code.* SciControls.com, Texas, USA, July 2005.

[16] Stefan Roock and Martin Lippert. *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully.* John Wiley & Sons, June 2006.

[17] 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.

[18] William C. Wake. *Refactoring Workbook.* Addison-Wesley Professional, August 2003.

Index

Actor, 10
Actors
table of, 6
Applications, 59
Argument, 5
Arguments, vi
Automatic code generator, 58
Automation
of Move Mehtod refactoring, 3

Behavior
defined, v
preservation, vii
Bipartite graph, 34
invariance under permutations, 34
maximum assignment, 35
Block-partitioning
of a sparse matrix, 2, 17
of matrix A^0, 21
Block column, 32
as an encapsulated user type, 20
Block row, 32
as a constructor, 20
as a member method, 20
Browsers, 59
Business rules
entering to relational model, 55
Butterfly effect, 48, 50

Calculations
table of, 6
Canonical representation
of a program, 6
Class
defining, 14
identifying, 22
Class element, 33
Code
as a model of the program, 51
good or bad, 45
legacy, 59
Code generation
automatic, 58
Code structure
separating from language, 2
Codomain, vi, 5
Complexity, 48
Concluding remarks, 61
Conslusions
from the experiment, 50
Constraints
predecessor-successor, vi, 2
table of, 6
Constructor
identifying, 22
Container
render as sparse matrix, 2
Control
flow, 19
Conversion
C to C++, 2

mechanics, 1, 12
non-OO to OO, 2
to OO code, 2, 3
Cotype, vi, 5

Database table, vi
Data entry
at the OO side, 6
Declarative programming, 6
Deep refactoring, 26, 47
Defining
a class, 14
a method, 15
an object, 14
Definition
of behavior-preserving, 35
of deep refactoring, 49
of refactoring, 35
Detorus, 49
Development
automating invariant steps, 56
breaking the structure, 25, 30
breaks the structure, vii
fixing errors, vii
fixing errors automatically, 30
iterative nature, 44
new ideas, 4
relation-assisted, 56
teaching the program new skills, 56
Dialog
with program, 4
DNA, viii
Domain, vi, 4
of objects, 4

Element
class, 33
Elementary class, 7
Encapsulation
algorithm control parameters, 32
and block-partitioning, 4
and user types, 11
details of the algorithm, 21
example of problems, 48
global nature, 3
of user types, 3
pervasive nature, vii
promoting, 11
propagation, 3
recursive, 2
seeding the algorithm, 13, 20
selecting candidates, 12
Engineering
reverse, 58
Execution
as a search operation, vi, 7
in the relational model, 7
of a program, 7
sequences, 10, 19
Execution sequence, vi, 2
Experimental conclusions, 50
Experimental studies
recommendations, 55
Experimental study, 45

Flow of control, 19
Function
defined, 5
relational, vi
Further processing
of matrix A^3, 26

Graph, vi, 1
bipartite, 34

html, 59

Identifying
a class, 22
a constructor, 22
a method, 22
a temporary, 22
Incremental refactoring, 57
Inheritance, 60
Intelligence
as a conversion, 4
as recursive encapsulation, 60
understanding, viii
Invariant transformation, vi
Iso-structural blocks, 23
Iso-structural subgraphs, 12

Language
expressing relations, 1
natural, viii
subscribing, vii
Language module, 23
Learning
as relations, 4
Leaves
removing from refactoring tree, 46
Legacy code, 59
converting to a relational model, 59
extracting components, 59
Light refactoring, 26
Literature explosion, v

Malleability
of relational model, 3
Matrices
sparse, 17
Matrix
A^0, 18, 37
A^1, 21, 38
A^2, 21, 39
A^3, 22, 40
A^3, further processing, 26
A^4, 27
A^5, 28
B, 28, 41
C, 42
H, defining a member method, 27
M, defining a member method, 27
sparse, vi
Meaning
adding to relations, 5
from objects, 4
Mechanics of conversion, 12
Messages, 49
Method
defining, 15
identifying, 22
Method misplacement, 49
Minor refactoring, 26
Mistakes
cascade of, 48
Motivation, 1
Move Method refactoring
automation, 3
Mutator, 7

Newton's law, 9

Object
defining, 14
Operations
defining new, 13
Operator
algebraic, 7
logical, 7
overloaded, 7
Other applications, 59
Overloading, 60

Overriding, 60

Parsers
business-specific, 55
natural language, 55
Partitioning
and encapsulation, 4
of a sparse matrix, vii
P1, 21
P2, 28
P3, 30
Partitions
as user types, 3
Pascal
conversion to, 24, 25
Pattern-matching tests, 12
Patterns, 58
relational, 59
Polymorphism, 7, 60
Pool of languages, 58
Preconditions
example of analysis, 47
for refactoring, 47
Predecessor-successor
constraints, 2
Predecessor-successor constraints, vi, 11
Primary key, 5
Program
A, 9
A, developed, 56
as a relation, vi, 1, 10
as a set of relations, 6
B, 14
C, 15
canonical representation, v, 6, 56
container, v, 1
D, 23
D1, 24
data describing, 2
data processed, 2
decomposition with a parser, 6
description, v
different refactorings, 29
E, 28
expressed as relations, 3
F, 30
G, 33
mathematical foundation, 20
mathematically described, vi
removing OO information, 32
simple example, 9
structure, v
translation between languages, 58
Programming
declarative, 56
object-oriented, 56
receiving feedback from the program, 57
Programming language
relational pool, 58
subscribing to relational model, 58
using for machine-man communication, 57
Program model, 1
Program structure
separating from language, 2
Propagation
of encapsulation, 3

Quality of code
defined, vii

Research outlook, 55
Refactoring
as a relational operation, 4
Refactoring, 44, 57
a dangerous procedure, 44
as a coupled system of logical equations, 52

as a nested operation, 51
Backtracking, 46
code under development, 12
complexity, 50
computerized, 52
current limitations, v, 30
current state, 44
decomposition, 46
deep, 26, 47, 49
definition, v, 35
deterministic logic, 50
diagnosis, 46
example of decomposition, 48
Get and Set methods, 46
incremental, 57
iterative nature, 46
light, 26
logical constraints, 52
manual, 45
minor, 26
Move Method, 30
nature, 44
nested, 48
of recently developed code, 56
patterns, 23
preconditions, 45, 47
rebuilding damaged code, 57
size of beast, 49
targeting casts, 46
targets, 46
theories, v
tools, v
tree, 46
Refactoring algorithm
control parameters, 29
Refactoring logic, 30
Registers
as relations, 58
Relation, vi, 1, 2
as a table, 5
basics, 4
closure property, vi
definition, 5
degree, 5
sparse format, vi
Relation-assisted development, 56
Relational function, 5
Relational model, v, 5
as a container, 3
as support for developers, 57
creating, 7
creating from executable code, 58
creating from legacy code, 59
definition, 6
eagle's eye view, 19
entering information, 1
execution, 7
features, 55
for areas of human activity, 55
for systems other than software, 60
in a development environment, 57
of DNA, 59
populating, 6, 9
populating from OO code, 10
recognizing OO structures, 3
storage schemes, 33
with a wider scope, 59
Relational model interfaces
language modules, 55
parsers, 55
Relational operations, 3
in refactoring, 4
Relational patterns, 59
Relational sources
developers, 59
natural language, 60

programs, 59
textbooks, 59
UML models, 59
Relation Basics, 4
Realtion
normalization, 5
Relations
visualizing, 17
working with, 5
Relational model
for architecture, 60
Research
encouragement, viii
Reverse engineering, 58

Sequence
of execution, vi, 2
Sequences
of execution, 19
table of, 6
Sequences of execution, 10
Set, vi, 1
Simple program, 2
Simple program example, 9
Software development, 56
Source code
C to C++ conversion, vii
generating from model, vii
quality, vii
reconstruction, 32
Sparse matrices, 17
uses, 17
using in databases, 34
Sparse matrix, vi
Block-partitioning, 17
classes show vertically, 20
compiling, 3
converting into OO code, 20, 23
converting to OO code, 56
definition, 17
executing, 4
methods show horizontally, 20
partitioning, vii
row-wise representation, 33
tutorial, 3
un-partitioning, 32
used to describe a program, 3
Statement
conditional, 7
executable, 6, 10
flow of control, 7, 10
iterative, 7
Storage schemes
for the relational model, 33
Strong ownership, 52

Table
of actors, 6
of calculations, 6
of constraints, 6
of sequences, 6
Temporaries
identifying, 22
Tool integration, 1, 57
promoting, vii
Tools
semi-automatic, 45
Transformation
behavior-preserving, vi
invariant, vi
Translation, 58
automatic, vii
between languages, 58
Tree
of refactorings, 46
Type, vi, 4

Type condensation, 13

UML, vii, 4
    conversion into OO code, 56
    models, viii
UML models, 19
User type
    as a coupled logical system, 25
User types
    as partitions, 2
    creating, vii, 20
    generating, 2
    mechanics of creation, 12
    predefined, 12
    recursively creating, 26

Value, 5

Web sites, 59
What is next, 55