Available online at www.sciencedirect.com
SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science

ELSEVIER Electronic Notes in Theoretical Computer Science 130 (2005) 281–300

www.elsevier.com/locate/entcs

Refactoring Towards a Layered Architecture

Márcio Cornélio[1]

Departamento de Sistemas Computacionais
Escola Politécnica — Universidade de Pernambuco
Recife, PE, Brasil

Ana Cavalcanti[2]

Department of Computer Science
University of York
York, England

Augusto Sampaio[3]

Centro de Informática
Universidade Federal de Pernambuco
Recife, PE, Brasil

Abstract

In this paper we present how refactoring of object-oriented programs can be accomplished by using formal refinement. Our approach is based on the use of refactoring rules designed for a sequential object-oriented language of refinement (ROOL) similar to Java. We define a strategy that aims at structuring programs according to a layered architecture that involves the application of refactoring rules, object-oriented programming laws, and data and algorithm refinement. As the laws are proved in a weakest precondition semantics of ROOL, correctness of refactoring is ensured by construction.

Keywords: Refactoring, Formal Refinement, Refinement Calculus

[1] Email:mlc@upe.poli.br
[2] Email:Ana.Cavalcanti@cs.york.ac.uk
[3] Email:acas@cin.ufpe.br

doi:10.1016/j.entcs.2005.03.015

1 Introduction

Object-oriented programming has been acclaimed as a means to obtain software that is easier to modify than conventional software [20]. However, changing an object-oriented program often requires structural changes such as moving attributes and methods between classes, and partitioning one complex class into several ones. Such modifications should change just the internal software structure, without affecting the software behaviour as perceived by users. This activity is called *refactoring* [16]. Work on refactoring usually describes the steps used for program modification in a rather informal way [16,23,25].

In our approach, formal refactoring is achieved by the application of programming laws that deal with commands as well as with object-oriented features like methods and classes [3,4]. These laws were proposed for ROOL [8,7], an acronym for Refinement Object-Oriented Language, which is a subset of sequential Java with classes, inheritance, visibility control for attributes, dynamic binding, and recursion.

Programming laws are the basis for the derivation of refactoring rules, along with laws that lead to data refinement of classes [12]. These laws precisely indicate the modifications that can be done to a program, with corresponding proof obligations. Using laws, program development is justified and documented. Program transformations accomplished by the use of refactoring rules and programming laws preserve program behaviour [12]. Our language has a weakest precondition semantics, which supports the formal justification of the laws we use and, consequently, of our strategy. The proof that of soundness of all laws proposed for ROOL [3,12,4] with respect to a weakest precondition semantics [8,7] is presented in [12].

A system structured according to an architecture composed of independent layers of software that deal, in an orthogonal way, with database access, GUI, distribution and functional requirements, has classes with purposes clearly separated [6]. Well-structured programs are essential to improve reuse and extensibility. Using a layered architecture, we can, for instance, integrate Object-Oriented Programming Languages and Relational Databases without compromising software quality factors like reusability and extensibility [26].

In this paper we show how refactoring of object-oriented programs can be accomplished by using refactoring rules [12] and programming laws [3,4]. We present a refactoring strategy, exemplifying its application with the use of template classes. Using this strategy, we refactor a program that is representative of a number of real applications.

Our case study was first reported, and informally developed, in [26] and concerns the integration of object-oriented programming languages with re-

lational databases. We transform the original program, which initially does separate architectural concerns, into one whose architecture achieves software quality factors such as reusability and extensibility. The formal development of this case study has served to identify new refactoring rules for ROOL and to improve our refactoring strategy. It was initially presented in [12].

This paper is organised as follows. In Section 2, we present an overview of ROOL with some basic laws of commands and classes. In Section 3, we present two refactoring rules we use in the derivation we present here. After that, in Section 4, we present a strategy for program refactoring in ROOL that aims at structuring programs according to a layered architecture, along with a sketch of our case study. In Section 5, we discuss some related work. Finally, in Section 6, we summarise the results achieved and point some directions for future research.

2 ROOL and Laws

ROOL [8,7] is an object-oriented language based on sequential Java. It allows reasoning about object-oriented programs and specifications, as both kinds of constructs are mixed in the style of Morgan's refinement calculus [21,22]. The semantics of ROOL, as usual for refinement calculi, is based on weakest preconditions. The imperative constructs of ROOL are based on the language of Morgan's refinement calculus [21], which is an extension of Dijkstra's language of guarded commands [13]. In a refinement calculus, specifications are regarded as commands. In fact, we use the term command to refer to commands, in its usual sense, and programming constructs in which specifications and commands are mixed.

A program $cds \bullet c$ in ROOL is a sequence of classes cds followed by a main command c. Classes are declared as in the following example, where we define a class *Account*.

class *Account* **extends object**
  pri *balance* : **int**
  ...
  meth *getBalance* $\widehat{=}$ (**res** r : **int** $\bullet$ r := **self**.*balance*)
  meth *setBalance* $\widehat{=}$ (**val** s : **int** $\bullet$ **self**.*balance* := s)
  new $\widehat{=}$ **self**.*balance* := 0
end

Classes are related by single inheritance, which is indicated by the clause **extends**. The class **object** is the superclass of all classes. So, the **extends** clause could have been omitted in declaration of *Account*. The class *Account*

includes a private attribute named *balance*; this is indicated by the use of the **pri** qualifier. Attributes can also be protected (**prot**) or public (**pub**). ROOL allows only redefinition of methods which are public and can be recursive; they are defined using procedure abstraction in the form of Back's parameterized commands [1,10]. A parameterised command can have the form **val** $x : T \bullet c$ or **res** $x : T \bullet c$, which correspond to the call-by-value and call-by-result parameter passing mechanisms, respectively. For instance, the method *getBalance* has a result parameter r, whereas *setBalance* has a value parameter s. Initialisers are declared by the **new** clause.

Commands in ROOL are similar to those of Morgan's refinement calculus [21]. In particular, in a specification statement $x : [\psi_1, \psi_2]$, we call x the frame, and the predicates ψ_1 and ψ_2 are the precondition and the postcondition, respectively. When executed in a state that satisfies ψ_1, this program terminates in a state that satisfies ψ_2 modifying only the variables in x. In an initial state that does not satisfy ψ_1, the command $x : [\psi_1, \psi_2]$ aborts: all possible behaviours and nontermination are to be expected.

A set of algebraic laws for ROOL has already been defined in [3,4]. Laws for commands deal with the small grain constructs, whereas laws for classes consider the medium grain constructs. Many laws of commands are similar to the laws of imperative programming presented, for example, in [18], but ROOL has laws that support object-oriented features such as method calls, classes, and type cast and test. These laws were proved to be sound [12] with respect to a weakest precondition semantics of ROOL

The laws of ROOL, mainly those related to object-oriented features, address context issues. We use $cds_1 =_{cds,c} cds_2$, where cds is a context of class declarations for cds_1 and cds_2, and c is the main command to denote the equivalence of sets of class declarations cds_1 and cds_2. This is notation is just an abbreviation for the program equivalence $cds_1\, cds \bullet c = cds_2\, cds \bullet c$, which is formalised in [8,7]. Below we present some examples of laws. We write '$(\rightarrow)$' when some conditions must be satisfied for the application of the law from left to right. We also use '$(\leftarrow)$' to indicate the conditions that are necessary for applying a law from right to left. We use '$(\leftrightarrow)$' to indicate conditions necessary in both directions. Conditions are described in the **provided** clause of laws.

Using Law 2.1 $\langle$*method elimination*$\rangle$, we can remove a method from a class if it is not called by any class in cds, in the main command c, nor inside class C. For applying this law from right to left, the method m cannot be already declared in C nor in any of its superclasses or subclasses, so that we can introduce a new method in a class. The notation $B.m$ refers to calls to a method m via expressions whose static type is exactly B. The subclass relation is denoted by $\leq$. We write $B \leq A$ to denote that a class B is a

subclass of a class A.

Law 2.1 ⟨*method elimination*⟩

class C **extends** D ads **meth** $m \mathrel{\widehat{=}} pc$ **end**; ops **end**	$=_{cds,c}$	**class** C **extends** D ads ops **end**

provided

(→) $B.m$ does not appear in cds, c nor in ops, for any B such that $B \leq C$.

(←) m is not declared in ops nor in any superclass or subclass of C in cds.

□

Law 2.2 ⟨*class elimination*⟩ when applied from left to right, allows the elimination of a class that is not referred to in the whole program. The application in the reverse direction introduces a new class in the program.

Law 2.2 ⟨*class elimination*⟩

$cds\ cd_1 \bullet c = cds \bullet c$

provided

(→) The class declared in cd_1 is not referred to in cds or c;

(←) (1) The name of the class declared in cd_1 is distinct from those of all classes declared in cds; (2) the superclass appearing in cd_1 is either **object** or declared in cds; (3) and the attribute and method names declared by cd_1 are not declared by its superclasses in cds, except in the case of method redefinitions.

□

To apply this law from left to right, the name of the class declared in cd_1 must not be referred to in the whole program. In order to apply this law from right to left, the name of the class declared in cd_1 must be distinct from the name of all existing classes; the superclass that appears in the declaration cd_1 is **object** or is declared in cds. Finally, only method redefinition is allowed for the class declared in cd_1.

Law 2.3 ⟨*private attribute-coupling invariant*⟩ allows us to change private attributes in a class, relating them with new attributes. The application of this law changes the bodies of the methods declared in the class. The changes follow the traditional laws for data refinement [21]. By convention, the attributes denoted by x are abstract, whereas those denoted by y are concrete. The coupling invariant CI relates abstract and concrete attributes. The notation $CI(mts)$ indicates the application of CI to each of the methods in mts:

applying *CI* changes the methods according to the laws of data refinement [21], that is, guards are augmented in order to assume the coupling invariant and every command is extended by modifications to the concrete variables so that they maintain the coupling invariant. We write **pri** $a : T;\ ads$ to denote the attribute declaration **pri** $a : T$ and all declarations in *ads*, whereas *mts* stands for declarations of methods and initialisers.

Law 2.3 ⟨*private attribute-coupling invariant*⟩

class A **extends** C **pri** $x : T;\ ads$ mts **end** $cds \bullet c$	$\preceq_{CI}$	**class** A **extends** C **pri** $y : T';\ ads$ $CI(mts)$ **end** $cds \bullet c$

□

The symbol $\preceq$ indicates that this law involves a simulation between attributes that are related by the coupling invariant *CI*. Simulation for data refinement of classes and its proof of soundness was presented in [9].

There also laws to deal with moving attributes and methods to superclasses, changing types of attributes and parameters, for instance, and other features. They can be found in [3,4].

Presently, ROOL has a copy semantics rather than a reference semantics. Of course, pointers are ubiquitous in practice. We decided, however, to concentrate initially on other aspects of object-orientation and Java like inheritance, dynamic binding, visibility, and type tests and casts. In general, the results we obtain are also valid in the presence of pointers, but they would need to be revised in the presence of sharing.

3 Refactoring Rules

In [12] we present a comprehensive set of refactoring rules which capture and formalises most of the refactorings informally introduced in [16]. Here we present two of the rules used in the derivation of the layered architectural pattern.

Refactoring rules are described by means of two boxes written side by side, along with **where** and **provided** clauses. We use the **where** clause, when necessary, to write abbreviations. The provisos for applying a refactoring rule are listed in the **provided** clause of the rules. The left-hand side of the rule

Rule 3.1 ⟨*Delegation Elimination*⟩

class A **extends** C
  pri $b : B;\ ads_a$
  meth $m \mathrel{\widehat{=}}$
    $(pds \bullet \mathbf{self}.b.n(\alpha(pds)))$
  mts_a
  new $\mathrel{\widehat{=}}$ **self**$.b :=$ **new**$.B$
end
class B **extends** D
  pri $x : T;\ ads_b$
  meth $n \mathrel{\widehat{=}} (pds \bullet c)$
  mts_b
end

$=_{cds,c}$

class A **extends** C
  pri $x : T;\ ads_a$
  meth $m \mathrel{\widehat{=}} (pds \bullet c)$
  mts_a
end
class B **extends** D
  pri $x : T;\ ads_b$
  meth $n \mathrel{\widehat{=}} (pds \bullet c)$
  mts_b
end

provided

(↔) (1) **super** does not appear in n; (2) $b \neq$ **null** $\wedge\ b \neq$ **error** is an invariant of A; (3) **self**$.y$ does not appear in n, for any attribute y in ads_b;

(→) (1) **self**$.a$ does not appear in n, for any public or protected attribute a that is declared by D or by any of its superclasses; (2) **self**$.p$ does not appear in n, for any method p declared in mts_b or in any superclasses of B; (3) **self**$.b$ does not appear in mts_a;(4) x is not declared in ads_a nor in any superclass or subclass of A;

(←) (1) b is not declared in ads_a nor in any superclass or subclass of A; (2) **self**$.x$ does not appear in mts_a;

presents the class or classes before the rule application; the right-hand side presents the classes after the rule application: the transformed classes. We must note, however, that many of the refactoring rules are equalities and can be applied in both directions.

3.1 Delegation Elimination

Rule ⟨*Delegation Elimination*⟩ (Rule 3.1) allows the elimination of delegation between two classes, when applied from left to right. The application from right to left allows the introduction of delegation between classes.

On the left-hand side of this rule, any call to the method m of class A is forwarded to the class B. The class A declares the attribute b of type B and initialises it with an object of B. The class B declares the attribute x of type T and attributes ads_b. In the method n of B there might be occurrences of the expression **self**.x. On the right-hand side, the class A does not declare an attribute of type B, but the attribute x that is also declared in B. The method m of A is defined by the same parameterised command that defines the method n of B.

Proof

We can prove the soundness of this refactoring rule in the following way. From left to right, using Law ⟨*change visibility: from private to public*⟩ [12], from left to right, we change the visibility of the attribute x of class B to public. We eliminate any calls to the method n of class B that appear inside method m of class A. Then, we proceed with data refinement of class A. We use law Law 2.3 ⟨*private attribute-coupling invariant*⟩ and other laws for data refinement. Finally, we remove the attribute b of class A by using law ⟨*attribute elimination*⟩ [12], from left to right. The proof from right to left is similar.

3.2 Interface Clientship

Rule ⟨*Interface Clientship*⟩ (Rule 3.2) introduces clientship between a class B and a class D, which models an interface by leaving the body of method m defined by using **abort**. The class D is adequately extended (representing interface implementation) in order to introduce a concept initially described in class B. By applying this rule, we can later provide different implementations of this concept.

On the left-hand side of this rule, class B declares an attribute x, and a method m (among other methods in mts_b). On the right-hand side, we introduce class D whose method m is defined by a parameterised command with body **abort**, modelling the effect of a Java interface. A Java interface contains a set of signatures of abstract methods; by defining the method bodies to be **abort**, we give them the most abstract definition.

On the right-hand side of this rule, class E extends D, declares an attribute x and redefines m. Class B is a client of D, and initialises its at-

Rule 3.2 ⟨*Interface Clientship*⟩

class B **extends** A
  pri $x : T;\ ads_b$
  meth $m \mathrel{\widehat{=}} (pds_m \bullet c_m[c'_m])$
  mts_b
end

$=_{cds,c}$

class B **extends** A
  pri $d : D;\ ads_b$
  meth $m \mathrel{\widehat{=}}$
    $(pds_m \bullet c_m[\mathbf{self}.d.m])$
  new $\mathrel{\widehat{=}}$ **self**$.d :=$ **new** $E()$
  mts_b
end
class D
  meth $m \mathrel{\widehat{=}} (pds_m \bullet \mathbf{abort})$
end
class E **extends** D
  pri $x : T;$
  meth $m \mathrel{\widehat{=}} (pds_m \bullet c'_m)$
end

provided
(→) (1) D and E are not declared in cds; (2) **self**$.x$ does not appear in mts_b;
(←) (1) Classes D and E are not referred to in cds or c.

tribute d with an object of class E, avoiding in this way program abortion. Command c_m is defined in terms of the call **self**$.d.m$.

For the application of rule ⟨*Interface Clientship*⟩, from left to right, we require that classes D and E are not declared in cds. Also, attribute x should not be accessed in methods of B other than m. In order to apply this rule from right to left, there must be no references to classes D and E, except in class B.

Proof

We can prove the soundness of this refactoring rule in the following way. We introduce classes D and E using Law 2.2 ⟨*class elimination*⟩ from right to left. However, attribute x of class E must be public, initially. The next step is

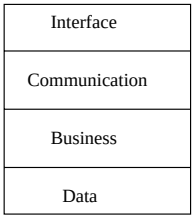

Fig. 1. The four-layer architecture

the data refinement of class B. Instead of using the attribute x of class B, we use x of class E, by applying Law 2.3 $\langle$*private attribute-coupling invariant*$\rangle$. For this reason, we introduce class E with a public attribute x. Finally, by using Law $\langle$*change visibility: from private to public*$\rangle$ [12], from right to left, we change the visibility of x to private. Accesses to x are now realised by calling method m of E. □

4 A Layered Architecture

Programs structured in accordance with a layered architecture support enhancements and reuse [24]. Therefore, refactoring of object-oriented programs should be conducted, whenever necessary, to obtain a final program with a layered architecture [2]. Here, we aim at a layered architecture originally designed for the integration of object-oriented programming languages with relational databases [26].

The main purpose of the architecture is to avoid, as much as possible, data storage and retrieval to be mixed with code that implements the functional requirements of a system. To achieve this purpose, classes are separated into two groups: classes that describe the objects required by the modelling of the systems' (functional) requirements; and classes for data storage and manipulation. The connection between classes of these groups is defined by interfaces. Classes of the first group are independent of the effective implementation of the data storage and manipulation operations, because these classes do not rely on knowledge of the data structure used for storage, but only on the methods defined by an interface. Classes of the first group contain what we call *business code*, which implements the functional requirements of a system. Classes of the second group, however, know how the persistence operations are implemented and depend on the data structure used for storage. They contain *data code* for manipulating data structures.

More generally, this architecture is viewed as being composed by four independent layers (Figure 1). Classes that model the functional requirements constitute the *business* layer, and classes for data storage and manipulation

constitute the *data* layer. The classes that contain code for communication among subsystems compose the *communication layer*; and classes that implement the user interface compose the *interface layer*. Here we concentrate on structuring the application into the business and data layers.

The classes of the *business layer* are divided into three groups: *basic* classes, representing basic entities; *collection* classes, representing groups of basic objects; and *control classes*, which define the control flow of functional requirements. The *collection* classes include methods for adding, searching, and removing items of a collection, and for invoking typical operations of business objects. If the facade pattern [17] is adopted, a single (control) class synthesises the functionality of the application.

From a poorly structured system we intend, by means of data refinement and application of refactoring rules, to reach a well-structured system adherent to the layered architecture described here.

4.1 The Architectural Pattern Derivation

Our refactoring strategy consists of three stages. Each stage involves the introduction of new classes, and data and algorithmic refinement of an already existing class. Data refinement typically involves the introduction of new attributes to restructure a class in order to improve reuse. From the first stage (Stage 1) to the last one (Stage 3), the program changes from a poorly structured one to a well-structured program according to the layered architecture described previously. In practice, it might not be necessary to follow all the steps proposed here: the developer should identify in which stage of development its program is, and apply refactoring from this stage to the last one. The main reason for dividing the development in stages is the simplification of data refinement.

Stage 1

In the first stage, we deal with a class that is monolithic. Data and business code are mixed. The purpose of this stage is to identify basic entities in such a monolithic description, and model each entity as a separate class, with its relevant attributes and methods.

A general form of a monolithic class is given in Figure 2. The attribute *aTable* is used to model a database. The type of *aTable* is given by a partial injective function ($\nrightarrow$) from a type T_1 to a type T_2. The method *update* is used to update a record of the table. It takes the record identifier n and its new value m as arguments; it also has a result parameter *rp*: a string that reports whether the update was successful or not. First, the method *update* checks if

```
class Application
  pri aTable : T1 ⇸ T2;  ...
  meth update ≙ (val n, m : T1, T2; res rp : string •
    if (n ∈ dom aTable) → self.aTable := self.aTable ⊕ {n ↦ m};
                          rp := "Updated"
    [] (n ∉ dom aTable) → rp := "Not_Updated"
    fi)
  ...
end
```

Fig. 2. The class *Application* in the beginning of Stage 1

n belongs to the domain of *aTable*, which is a business rule. If it does, then *aTable* is updated at position n with the expression *exp* in which there may be occurrences of the expression m, a data operation. The symbol '$\oplus$' in the body of method *update* stands for function overriding. The class *Application* also presents methods for inserting new elements in *aTable*, deleting already existing elements, and a method for inspecting the value associated with a given element in the domain of the table. At the end of this stage, we want to have the concept (class) which characterises the elements stored in *aTable* separated. The class *Application* is transformed as shown in Figure 3.

We introduce the class *BasicEntity* by using Law 2.2 ⟨*class elimination*⟩, from right to left. This class captures the concept introduced by the domain and range of the attribute *aTable*. This class provides get and set methods for the attribute at_2, because it is changed along the lifetime of objects of the class *BasicEntity*. The value of attribute at_1 is usually established at object creation and not modified along the lifetime of an object of *BasicEntity*. This reflects the fact that attribute at_1 acts like a key, used to identify which attribute at_2 is associated with it, in the class *Application*. The class *BasicEntity* represents basic objects necessary to implement the functional requirements of the system.

```
class BasicEntity
  pri at1, at2 : T1, T2;
  meth setAt2 ≙ (val m : T2 • self.at2 := m)
  meth getAt2 ≙ (res m : T2 • m := self.at2)
  new ≙ (val n, m : T1, T2 • self.at1 := n; self.at2 := m)
end
```

The next step is to prepare the class *Application* for data refinement. This preparation consists of applications of law ⟨*simple specification*⟩ [21,12] to assignments to the attribute *aTable*. The application of this law changes assign-

```
class Application
  pri data : seq BasicEntity;
  meth update ≙ (val n, m : T1, T2; res rp : string •
    var p, i : BasicEntity, int • self.search(n, p, i);
        if (p is BasicEntity) → p.setAt2(m); self.data(i) := p;
                                rp := "Updated"
        [] (p = null) → rp := "Not_Updated"
        fi
    end)
  meth search ≙ (val j : int; res obj, pos : Pair, int • ... )
  ...
end
```

Fig. 3. The class *Application* at the end of Stage 1

ments into corresponding specification statements.

class *Application*
  pri $aTable : T_1 \pfun T_2$;
  meth *update* $\widehat{=}$ (**val** $n, m : T_1, T_2$; **res** rp : **string** •
    if $(n \in \mathrm{dom}\, aTable) \rightarrow$
        self.$aTable$: [**self**.$aTable$ = **self**.$aTable \oplus \{n \mapsto m\}$];
        rp := "*Updated*"
    [] $(n \notin \mathrm{dom}\, aTable) \rightarrow$ rp := "*Not_Updated*"
    fi)
  ...
end

Afterwards, a new (private) attribute is introduced in the original class *Application*. We use Law 2.3 ⟨*private attribute-coupling invariant*⟩ to add an attribute *data* whose type is seq *BasicEntity* (sequence of *BasicEntity*) to *Application*. A coupling invariant relates the new attribute with the old one. From the point of view of data refinement, the new variables are concrete variables. The coupling invariant CI_{Stage_1} is used to relate the attribute *aTable* and *data*, which is a sequence of objects of class *BasicEntity*.

$$CI_{Stage_1} \mathrel{\widehat{=}} aTable = \{i : 0 \,..\, \#data - 1 \bullet data(i).at_1 \mapsto data(i).at_2\} \land (\forall\, i, j : 0 \,..\, \#data - 1 \bullet i \neq j \Rightarrow data(i).at_1 \neq data(j).at_1)$$

This coupling invariant guarantees that *aTable* is formed by mappings relating the values in each object present in *data*. Moreover, the values for the attribute at_1 of the objects in *data* must be distinct.

The application of Law 2.3 ⟨*private attribute-coupling invariant*⟩ to class *Application* changes the methods of this class according to the laws of data refinement [21]. Specification statements and guards, as expected, must assume the coupling invariant. The class *Application* now is as follows.

class *Application*
  pri *data* : seq *BasicEntity*;
  pri *aTable* : $T_1 \pfun T_2$; ...
  meth *update* $\widehat{=}$ (**val** $n, m : T_1, T_2$; **res** *rp* : **string** •
    if $(n \in \mathrm{dom}\, aTable \wedge CI) \rightarrow$
        self.*aTable* : $[CI, \mathbf{self}.aTable = \mathbf{self}.aTable \oplus \{n \mapsto m\} \wedge CI]$;
        rp := "*Updated*"
    [] $(n \notin \mathrm{dom}\, aTable \wedge CI) \rightarrow$ *rp* := "*Not_Updated*"
    fi)
  ...
end

Now we refine the class *Application* in order to remove references to the abstract variable *aTable*. First, by applying Law 2.1 ⟨*method elimination*⟩, from right to left, we introduce the method *search* in class *Application*. The method *search* returns an object of type *BasicEntity* whose attribute at_1 has the same value as n. We proceed with algorithmic refinement of specification statements and guards. Such refinement is carried out for all methods of *Application*. The method *update*, after refinement, is as follows.

meth *update* $\widehat{=}$ (**val** $n, m : T_1, T_2$; **res** *rp* : **string** •
  var p, i : *BasicEntity*, **int** • **self**.*search*(n, p, i);
    if (p **is** *BasicEntity*) $\rightarrow$ p.*setAt2*(m); **self**.*data*$(i) := p$;
                                rp := "*Updated*"
    [] $(p = \mathbf{null}) \rightarrow$ *rp* := "*Not_Updated*"
    fi
end)

The method *update* uses two local variables: p and i (see Figure 3). First it calls the new method *search* to get, in p, the object identified by n, and in i, its index in *data*. If p is null, then there is no element identified by n, and this is reported through *rp*. If p is not null, then a call to a method of *BasicEntity* is used to set its value to that of an expression involving the parameter m of *update*, the sequence *data* is updated, and success is reported. In the development, the variables p and i are introduced along with a specification statement that is refined to introduce a call to the method *search*, which it is

```
class Application
  pri collct : BusinessCollection;
  meth update ≙ (val n, m : T1, T2; res rp : string •
                    collct.update(n, m, rp)) end
  new ≙ collct := new BusinessCollection()
...
end
```

Fig. 4. The class *Application* at the end of Stage 2

```
class BusinessCollection
  pri data : seq BasicEntity;
  meth update ≙ (val n, m : T1, T2; res rp : string •
    var p, i : BasicEntity, int • self.search(n, p, i);
      if (p is BasicEntity) → p.setAt2(m); self.data(i) := p; rp := "Updated"
      [] (p = null) → rp := "Not_Updated"
      fi
    end )
  meth search ≙ (val j : T1; res obj, pos : BasicEntity, T1 • ... )
  ...
end
```

Fig. 5. The class *BusinessCollection* at the end of Stage 2

also called by the other methods of class *Application*, those used for insertion and deletion of objects in *data*.

Calls to methods of *BasicEntity* have to replace the direct access to the (abstract) attribute *aTable* which, after that, can be removed from the class *Application*. This is done by refining the methods of *Application* using laws similar to those presented by Morgan [21] and used in [12]. By using a law for attribute elimination, we remove the attribute *aTable* from *Application*. This law is also presented in [12].

Stage 2

In this stage, we have a program in which different concepts are described in different classes. In the end, our purpose is to have the class *Application* as a facade to the system, where the bodies of its methods basically delegate responsibilities to the business collections through method calls.

We obtain the classes *Application* (Figure 4) and *BusinessCollection* (Figure 5) in two steps. The first step is the introduction of class *BusinessCollection*. Then, by using rule ⟨*Delegation Elimination*⟩ (Rule 3.1), from right to left, we make class *Application* just a delegating class.

```
class BusinessCollection
  pri rep : RepositoryClass;
  meth update ≙ (val n, m : T1, T2; res rp : string •
    var p, i : BasicEntity, int • rep.search(n, p, i);
      if (p is BasicEntity) → p.setAt2(m); rep.update(p, i); rp := "Updated"
      [] (p = null) → rp := "Not_Updated"
      fi
    end )
  new ≙ self.rep := new RepositoryClassRef ()
  ...
end
```

Fig. 6. Class *BusinessCollection* at the end of Stage 3

Stage 3

At this point, the collection class and the persistence mechanism are still interwoven. This hinders reuse and extensibility, because if the persistence mechanism is changed, part of the system must be redesigned. The business code that can be reused, when adapting the persistence mechanism, should be separated from data code. This is the purpose of this stage.

We proceed with the application of rule ⟨*Interface Clientship*⟩ (Rule 3.2), resulting in a new version of class *BusinessCollection* (Figure 6) and in the new classes *RepositoryClass*, and *RepositoryClassRef* (Figure 7). Notice that class *BusinessCollection* now is client of *RepositoryClass* and initialises attribute *rep* with an object of *RepositoryClassRef*. This attribute is target of calls to methods of *RepositoryClassRef*. Class *RepositoryClass* defines an interface between the *BusinessCollection* and the class that deals with the persistence mechanism. A class like *RepositoryClass* is similar to a Java interface. Class *RepositoryClassRef* implements the access to the data structure originally defined in class *BusinessCollection*.

The use of an interface provides independence between the collection and the repository classes. We can change the repository class, for instance, from a class that uses a list to one that uses a tree, with minimal impact on the collection class. Only the initialiser of this class needs to change to create an object of the new implementation of the repository. We now have a program structured according to the architecture described in Section 4.

The strategy described here can also be used to obtain systems structured according to a three-layer architecture [6]. The main difference between our architecture and the three-layer one is the presence of the *communication* layer.

```
class RepositoryClass
  meth update ≙ (val obj, ind : BasicEntity, T1 • abort)
  meth search ≙ (val j : T1; res obj, pos : BasicEntity, T1 • abort)
...
end
class RepositoryClassRef extends RepositoryClass
  pri data : seq BasicEntity;
  meth update ≙ (val obj, ind : BasicEntity, T1 • self.data(ind) := obj)
  meth search ≙ (val j : T1; res obj, pos : BasicEntity, T1 • ... )
...
end
```

Fig. 7. Classes *RepositoryClass* and *RepositoryClassRef*

5 Related Work

The literature related to refactoring of object-oriented programs includes work such as that of Opdyke [23], which proposes a set of seven properties that must be satisfied in order to guarantee behaviour preservation. However, there is no proof in that work that satisfying these properties preserves program semantics. Our approach to refactoring is based on laws. Each law establishes the restrictions that must be satisfied allowing the law application. The application of a law modifies a program leaving its behaviour unchanged, since the soundness of each law is proved [12] against a weakest precondition semantics of ROOL and a refinement relation defined in [8,7].

Fowler [16] suggests that before starting refactoring one should have a solid suite of tests that must be self-checking. Every change must be followed by program compilation and test. However, there are no conditions to be satisfied in order to guarantee behaviour preservation. The use of algebraic laws for refactoring, as proposed here, eliminates the need of compiling the program as the result of a law application is correct by construction, both from the syntactic and from the semantic points of view. The use of a suite of tests is optional.

Cinnéide and Nixon [11] presented a methodology for the development of design pattern transformations in a semi-formal approach to demonstrate behaviour preservation. They identified minipatterns, certain motifs that occur repeatedly across pattern catalogues. For each minipattern, a minitransformation is developed. A minitransformation comprises a set of preconditions, a sequence of transformation steps, a set of post-conditions, and an argument demonstrating behaviour preservation. Each minitransformation is defined in terms of refactorings. Their arguments lack the rigor of a fully-formal ap-

proach. There is no semantics-based proof that the transformations do not change a program behaviour, but it should rely on regression tests.

Flores, Reynoso and Moore [15] use the RAISE Specification Language to formally specify the responsibilities and collaborations of patterns participants as well as the behavioural properties. They use their model to specify any object-oriented design, so that they are able to verify that a given subset of a design corresponds to a given pattern. There is no discussion about the transformation of a design into another.

Eden [14] presents a declarative language called LePUS for specifying the structural and behavioural aspects of design patterns. He recognises that relations in LePUS specifications can be mapped to different programming constructs of different programming languages, even though he argues that the set of relations used in the description of design patterns map directly to well-defined syntactic constructs in statically typed programming languages. He does not present any systematic approach for such translation. He also argues that a prototype tool for LePUS can be developed in Prolog, allowing the manipulation of formulas representing modifications in a program. However, practical results in this area are not evident yet in his published work. In our approach, we transform a particular design with the aim of obtaining a new design according to a design or architectural pattern. The transformation is accomplished by the use of rules written using metalanguage elements, such as meta-variables for representing attributes, but using a language very similar to RoboOL, the language we use to write our programs.

Lano *et al.* [19] formally justify design patterns by relating two sets of classes, the "before" and "after" systems. The "after" system consists of a collection of classes organised according to a pattern. The proof that the "after" system is an extension of the "before" one is given via a suitable interpretation that is proved for selected axioms. Differently, we adopt a transformational approach that is constructively based on rules, and not on the verification that a system extends the original system.

6 Conclusions

This paper has illustrated how refactoring of object-oriented programs can be accomplished in a formal way by using a rule-based approach. This is based on the application of refactoring rules [12]. The proposed strategy to obtain a system with a layered architecture also involves classical data refinement [21] and algorithmic refinement.

The soundness of the transformation of a program, so that it adheres to an architectural pattern or a design pattern, relies on the use of refactoring rules,

and, eventually, on the use of programming laws and data refinement. The derivation of refactoring rules is based on the use of programming laws that deal with imperative commands and object-oriented features whose proof of soundness against the formal semantics of ROOL [8,7] is presented [12].

Notwithstanding the fact that we work with a language with a copy semantics, our experience until now reveals that this is not a hindrance to refactoring. A distinguishing feature of our research is the formal justification of design practices using a simple, uniform, and modular reasoning mechanism: a set of basic algebraic laws of ROOL.

Our general aim is formalising object-oriented design practices. We are currently working on well-established design patterns [17]. This is important for the practice of formal refactoring of object-oriented programs and also for formally justifying the validity of the design changes that aim at structuring a system according to a design pattern. In [12], we have already proposed transformations to restructure a system into one in accordance with the Facade Pattern [17], by using refactoring rules. Both the Facade Pattern and the architectural pattern presented here are obtained by the application of refactoring rules.

As a future work, we plan to build a tool to mechanise the transformations of rule applications. We intend to mechanise the application of programming laws by using systems like Elan [5]. This will serve as the basis for the mechanised derivation of refactoring rules as well as architectural and design patterns. Once programming laws and refactoring rules are mechanised, it is possible to construct an environment to reason about program transformations with a fully-formal guarantee of semantics preservation.

References

[1] Back, R. J. R., *Procedural Abstraction in the Refinement Calculus*, Technical report, Department of Computer Science, Abo - Finland (1987), ser. A. No. 55.

[2] Back, R. J. R., *Software Construction by Stepwise Feature Introduction*, in: D. B. *et. al.*, editor, *ZB 2002: Formal Specification and Development in Z and B*, Lecture Notes in Computer Science **2272**, 2002, pp. 162–183.

[3] Borba, P., A. Sampaio, A. Cavalcanti and M. Cornélio, *Algebraic Reasoning for Object-Oriented Programming*, Science of Computer Programming (2004), pp. 53–100.

[4] Borba, P., A. Sampaio and M. Cornélio, *A refinement algebra for object-oriented programming*, in: L. Cardelli, editor, *European Conference on Object-oriented Programming, ECOOP'2003*, Lecture Notes in Computer Science **2743** (2003), pp. 257–282.

[5] Borovanský, P. et al., "ELAN User Manual — ELAN V3.6," LORIA Université de Nancy 2, France (2004), available on-line http://www.loria.fr/equipes/protheo/SOFTWARES/ELAN/manual/index-manual.html. Last access 19/05/2004.

[6] Broy, M., E. Denert, K. Renzel and M. Schmidt, *Software architectures and design patterns in business applications*, Technical Report TUM-I9746, Technische Univerität München (1997).

[7] Cavalcanti, A. L. C. and D. Naumann, *A weakest precondition semantics for an object-oriented language of refinement*, in: *FM'99 - Formal Methods*, Lecture Notes in Computer Science **1709**, 1999, pp. 1439–1459.

[8] Cavalcanti, A. L. C. and D. A. Naumann, *A Weakest Precondition Semantics for Refinement of Object-oriented Programs*, IEEE Transactions on Software Engineering **26** (2000), pp. 713–728.

[9] Cavalcanti, A. L. C. and D. A. Naumann, *Forward Simulation for Data Refinement of Classes*, in: L. Eriksson and P. A. Lindsay, editors, *FME 2002: Formal Methods - Getting IT Right*, Lecture Notes in Computer Science **2391** (2002), pp. 471–490.

[10] Cavalcanti, A. L. C., A. Sampaio and J. C. P. Woodcock, *An Inconsistency in Procedures, Parameters, and Substitution in the Refinement Calculus*, Science of Computer Programming **33** (1999), pp. 87–96.

[11] Cinnéide, M. O. and P. Nixon, *A methodology for the automated introduction of design patterns*, in: H. Yang and L. White, editors, *Proceedings International Conference on Software Maintenance* (1999), pp. 463–472.

[12] Cornélio, M. L., "Refactorings as Formal Refinements," Ph.D. thesis, Centro de Informática, Universidade Federal de Pernambuco (2004).

[13] Dijkstra, E. W., "A Discipline of Programming," Prentice-Hall, 1976.

[14] Eden, A., *Formal specification of object-oriented design*, in: *International Conference on Multidisciplinary Design in Engineering*, 2001.

[15] Flores, A., L. Reynoso and R. Moore, *A Formal Model of Object-Oriented Design and GoF Patterns*, Technical report, UNU-IIST (2000).

[16] Fowler, M., "Refactoring: Improving the Design of Existing Code," Addison-Wesley, 1999.

[17] Gamma, E. et al., "Design Patterns: elements of reusable object-oriented software," Addison-Wesley Professional Computing Series, Addison-Wesley, 1994.

[18] Hoare, C. et al., *Laws of programming*, Communications of the ACM **30** (1987), pp. 672–686.

[19] Lano, K., S. Goldsack and J. Bicarregui, *Formalising Design Patterns*, in: *RBCS-FACS Northern Formal Methods Workshop*, 1996.

[20] Meyer, B., "Object-Oriented Software Construction," Prentide-Hall, 1997, second edition.

[21] Morgan, C. C., "Programming from Specifications," Prentice Hall, 1994, second edition.

[22] Morgan, C. C., K. Robinson and P. H. B. Gardiner, *On the Refinement Calculus*, Technical monograph tm-prg-70, Oxford University Computing Laboratory, Oxford - UK (1988).

[23] Opdyke, W., "Refactoring Object-Oriented Frameworks," Ph.D. thesis, University of Illinois at Urbana-Champaign (1992).

[24] Shaw, M. and D. Garlan, "Software architecture: perspectives on an emerging discipline," Prentice-Hall, 1996.

[25] Tokuda, L. A., "Evolving Object-Oriented Designs with Refactoring," Ph.D. thesis, Department of Computer Sciences, University of Texas at Austin (1999).

[26] Viana, E. and P. Borba, *Integrating Java with Relational Databases*, III Simpósio Brasileiro de Linguagens de Programação (1999), pp. 77–91, in Portuguese.