Formal Use of Design Patterns and Refactoring*

Long Quan[1], Qiu Zongyan[1], and Zhiming Liu[2]

[1] LMAM & Dept. of Informatics, School of Math. Peking University, Beijing 100871, China
[2] IIST, United Nations University, Macao, China
{longquan,qzy}@math.pku.edu.cn, lzm@iist.unu.edu

Abstract. Design patterns *has been used very effectively in object-oriented design for a long time.* Refactoring *is also widely used for producing better maintainable and reusable designs and programs. In this paper, we investigate how design patterns and refactoring rules are used in a formal method by formulating and showing them as* refinement laws *in the calculus of refinement of component and object-oriented systems, known as* rCOS. *We also combine refactoring and design patterns to provide some big-step rules of* pattern-directed refactoring.

Keywords: Object-Orientation, Design Pattern, Refactoring, *rCOS*, Refinement.

1 Introduction

Design pattern is widely accepted as a very useful OO technique to produce designs which are maintainable and reusable, as well as to reduce time and cost of development [9]. *Refactoring* is another important techniaue for improving existing designs or programs [20,8,21]. These techniques are also related that design patterns can be used to guide refactoring. In [8], Fowler collected 68 refactoring rules, where an example shows how the *State Pattern* is used to guide transformations of a program step by step using refactoring rules. Kerievsky discussed in [13] the *pattern-directed refactoring*, which can be seen as big-step refactoring rules toward patterns. However, there has been little investigation on how the techniques of design patterns and refactoring can be used in a formal method until recently when a few attempts are published.

Roberts [18] defined refactoring rules as program transformations, but there is no formal semantic-based proof to ensure that such transformations preserve the behavior of programs. M. Cornélio *et al.* [6] formalized and proved some of Fowler's refactoring rules as refinement laws in ROOL [2]. However, ROOL takes a copy semantics defined by weakest precondition transformers. Without references, some important and interesting laws of OO programs do not hold in ROOL [19]. More recently in [7], refactoring rules are defined and automated as model transformations for models at different abstraction levels including platform independent models, platform specific models and implementation specific models. Also in [17], refactoring of UML class diagrams is studied using the standard Object Constraint Language (OCL).

In this paper, we investigate how design patterns and refactoring rules can be both formulated as program refinement laws in rCOS [12,3]. The advantage of using rCOS is

* Supported by NNSF of China (No. 60718002), and Projects on HighQSofD and HTTS funded by Macao S&T Fund.

T. Margaria and B. Steffen (Eds.): ISoLA 2008, CCIS 17, pp. 323–338, 2008.

that it takes a reference semantic model with rich OO features, including *subtypes, visibility, inheritance, dynamic binding* and *polymorphism*. Also a UML profile is defined for rCOS [22,5] in a component-based model driven development process.

Similar to the idea in [6], we formulate refactoring rules as rCOS refinement laws of OO specification and programs. Take the advantage of the relational semantics of rCOS, the formulation of the refactoring rules and their proofs are better comprehensive compared with the approach of predicate transformers. We go one step further to formulate the pattern-directed refactoring rules suggested in [13]. Sketches of proofs are given in this paper, leaving the full details in a technical report [15].

We will briefly introduce rCOS in Section 2, and give the rCOS definition of the refactoring rules in Section 3, and the pattern-directed refactoring rules in Section 4. Ideas and sketches of proofs are given in Section 5. An example is used to illustrate the formal use of refactoring and pattern directed refactoring is presented in Section 6. Finally, we summarize the conclusions and discuss some future work.

2 Basics of rCOS

A program in rCOS is of the form *cdecls* • *P*, where *P* is the main program playing the role as *main method* in Java programs. P takes the form (gbv, c), where gbv denotes *global variable declarations* and c is a command. *cdecls* is a sequence of class declarations $cdecl_1; \ldots; cdecl_k$, where each $cdecl_j$ declares a class in the form :

$$
\begin{array}{l}
[\texttt{private}]\ \texttt{class}\ N\ [\texttt{extends}\ M]\ \{ \\
\quad \texttt{private} \quad U_1\ u_1 = a_1, \ldots, U_m\ u_m = a_m; \\
\quad \texttt{protected}\ V_1\ v_1 = b_1, \ldots, V_n\ v_n = b_n; \\
\quad \texttt{public} \quad W_1\ w_1 = c_1, \ldots, W_k\ w_k = c_k; \\
\quad \texttt{method} \quad m_1(\underline{T}_{11}\ \underline{x}_1, \underline{T}_{12}\ \underline{y}_1, \underline{T}_{13}\ \underline{z}_1)\{c_1\}; \cdots; m_\ell(\underline{T}_{\ell 1}\ \underline{x}_\ell, \underline{T}_{\ell 2}\ \underline{y}_\ell, \underline{T}_{\ell 3}\ \underline{z}_\ell)\{c_\ell\} \\
\}
\end{array}
$$

where we use $\underline{x}$ to denote a sequence of x. Here are some explanations:

- A class can be declared as `private`, but public by default. Only public classes and primitive types can be used in global variable declarations.
- *N* is a class name, and *M* is its direct superclass.
- Initial values of attributes are given in declarations.
- A method can have value parameters ($\underline{T}_{i1}\ \underline{x}_i$), result parameters ($\underline{T}_{i2}\ \underline{y}_i$), and value-result parameters ($\underline{T}_{i3}\ z_i$). We often use $m(\underline{paras})\{c\}$ to denote method m, with $\underline{paras}$ its parameter list and c its body command which will be defined below.
- Following the Java convention, we assume an attribute `protected` when it is not tagged with an accessible modifier.

The issues of visibility are omitted here for the simplicity of the theory. We assume that all methods in a public class are public and and accessible in the main method, and all methods in a private classes are protected.

rCOS supports typical OO constructs, as will as some special "statements" for the specification and refinement:

$$
\begin{array}{rl}
c ::= & S \mid \mathit{skip} \mid \mathit{chaos} \mid \mathbf{var}\ T\ x = e;\ c;\ \mathbf{end}\ x \mid c; c \mid c \lhd b \rhd c \mid \\
& c \sqcap c \mid b * c \mid le.m(\underline{e}, \underline{v}, \underline{u}) \mid le := e \mid \mathit{C.new}(le)[\underline{e}]
\end{array}
$$

Here S is a specification statement in the form of a *framed design* $f : p \vdash R$ in UTP, b a Boolean expression, e an expression, and le an *assignable expression* of the form $le ::= x \mid le.a$,where x is a variable name and a an attribute name. We have sequential composition $c; c$, conditional $c \lhd b \rhd c$, and iteration $b * c$, while $\mathbf{var}\ T\ x = e;\ c;\ \mathbf{end}\ x$ is a command within a local declaration scope. The nondeterministic choice $c \sqcap c$, $skip$, $chaos$ and S are introduced only for the specification and refinement.

We use $le.m(\underline{e}, \underline{v}, \underline{u})$ to denote a call to method m of the object denoted by le, with value arguments $\underline{e}$, result arguments $\underline{v}$ for return values, and value-result arguments $\underline{u}$. Command *C.new*$(x)[\underline{e}]$ creates a new object of class *C* with initial attribute values from expressions $\underline{e}$ and assigns the object to variable x, which should be of the type *C* or its supertype. The form of expressions e is standard.

rCOS adopts an observation-based relational semantics and extends UTP to deal with the OO features. As in UTP, the semantics of a program is defined as a *design* $D = (\alpha, P)$, where α is the *alphabet* of the program consisting of its input and output variables, denoted as $\underline{x}$ and $\underline{x}'$ respectively, and P is a specification in the form $pre(\underline{x}) \vdash post(\underline{x}, \underline{x}')$ with semantics defined by predicate:

$$(ok \wedge pre(\underline{x})) \Rightarrow (ok' \wedge post(\underline{x}, \underline{x}'))$$

where $pre(\underline{x})$ and $post(\underline{x}, \underline{x}')$ are predicates describing the effect of the program based on the pre- and post-states of the alphabet, ok and ok' are auxiliary Boolean variables denoting the successful startup and termination of the program, $\underline{x}, \underline{x}'$ denote values of variables $\underline{x}$ in the initial and final states, respectively. This convention will be used throughout the paper. The difference from UTP is that variables can also hold objects.

For defining semantics of a command, a *framed design* D is often used in which specification P is of the form $\beta : pre(\underline{x}) \vdash post(\underline{x}, \underline{x}')$, meaning that D only changes variables in the subset β of α.

For OO programs, the semantic definition of a command makes use of the *static structure information* given by the class declarations for dynamic binding and visibility control. These structure details are represented by a group of logical variables, where variable $\mathbf{cname}$ of the set of declared class names, $\mathbf{attr}$ of a function returning the attributes of each class, $\mathbf{op}$ of a function returning the method set of each class, $\mathbf{superclass}$ of a function returning the direct superclass. In addition to the global variables gbv in the main program, we need a variable $\mathbf{\Sigma}$ representing the set of currently existing objects during the execution. The commands in the program make changes to these variables. For example, an object creation command changes $\mathbf{\Sigma}$ by adding in the new created object; when an attribute of object *o* is modified, this change is reflected in $\mathbf{\Sigma}$ accordingly. For the details of the model we refer the reader to paper [12].

rCOS has a powerful theory of refinement developed from the refinement of design in UTP. We first recall definitions of design refinement and data refinement in UTP [11].

Definition 1 (Refinement). *Design* $D_2 = (\alpha, P_2)$ *is a* refinement *of* $D_1 = (\alpha, P_1)$*, denoted by* $D_1 \sqsubseteq D_2$*, if* P_2 *entails* P_1*, i.e.* $\forall \underline{x}, \underline{x}', ok, ok' \cdot (P_2 \Rightarrow P_1)$*, where* $\underline{x}$ *are the variables contained in* α*.* □

Definition 2 (Data refinement). *Suppose* ρ *be a mapping from alphabet* α_2 *to alphabet* α_1*. Design* $D_2 = (\alpha_2, P_2)$ *is a* data refinement *of design* $D_1 = (\alpha_1, P_1)$ *under* ρ*, denoted by* $D_1 \sqsubseteq_\rho D_2$*, if* $(P_1; \rho) \sqsubseteq (\rho; P_2)$*. Here* ρ *is called a refinement mapping.* □

Note in the the definition, the refinement mapping ρ can be specified as a design.

In rCOS, we define the notions of *OO system refinement* and *OO structure refinement* (also known as *class refinement*) [12,22].

Definition 3 (System refinement). *For $S_1 = (cdecls_1 \bullet P_1)$ and $S_2 = (cdecls_2 \bullet P_2)$ with the same global variables gbv in P_1 and P_2, we say that S_2 is a* refinement *of S_1, denoted by $S_1 \sqsubseteq_{sys} S_2$, if the behavior of S_2 is less nondeterministic than that of S_1:*

$$\forall \underline{x}, \underline{x}', ok, ok' \cdot (S_2 \Rightarrow S_1)$$

where $\underline{x}$ are the variables declared in gbv. □

$S_1 \sqsubseteq_{sys} S_2$ requires that S_2 is less nondeterministic that S_1 with regards to the states of the global variables.

Definition 4 (Structure refinement). *Let $cdecls_1$ and $cdecls_2$ be two class declaration sections, $cdecls_1$ is a* refinement *of $cdecls_2$, denoted by $cdecls_2 \sqsubseteq_{class} cdecls_1$, if the former can replace the later in any object system:*

$$cdecls_2 \sqsubseteq_{class} cdecls_1 \mathrel{\widehat{=}} (cdecls_2 \bullet P \sqsubseteq_{sys} cdecls_1 \bullet P)$$

holds for all main programs $P = (gbv, c)$. □

Intuitively, $cdecls_1$ provides the same or refined services as $cdecls_2$ can.

In [12], we studied the basic refinement laws that capture the nature of incremental development in OO programming, including the laws for adding a class into the declarations, introducing inheritance, functionality delegation, class decomposition, and encapsulating data, etc. Our work in [22] uses a graph theory for further study of the structure refinement where the soundness and completeness are established. We will study a set of special refinement laws that formalize the refactoring rules here.

3 Refactoring Rules in rCOS

In Fowler's book [8], refactoring rules are described via examples. Here we formulate them as rCOS refinement rules in order for their correctness to be provable. Fowler classifies refactoring rules into six categories. We show here the formalization for one rule from each category as a representative, leaving the others in our report [15].

In general, a refactoring rule should be formalized as a refinement law of the form $cdecls_0 \bullet P_0 \sqsubseteq cdecls_1 \bullet P_1$, where the left hand side is the original program and the right is the refactoring result. However, all rules presented here only involve structure refinement, except for rule **Parameterize Method** which needs modifications of main program. Thus, most of the rules take the form of $cdecls_0 \sqsubseteq cdecls_1$, where $cdecls_0$ and $cdecls_1$ are the class declarations involved.

We will use $N[M, pri, prot, pub, ops]$ to denote a class N with M as its direct superclass, *pri*, *prot*, and pub its private, protected and public attributes, respectively, and *ops* its methods. When no confusion, we will only give explicitly the parameters involved in a rule. For example, $N[ops]$ denotes a class with method set *ops*, and $N[prot, ops]$ for a class with protected attribute set *prot* and method set *ops*. In the rest of this section, the six refactoring rules are also presented with UML diagrams for illustration.

Rule *Extract Method* allows us to replace a piece of code in a method body with a call to a newly introduced method, which has the replaced code as its body.

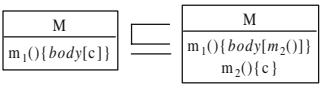

Fig. 1. Extract Method

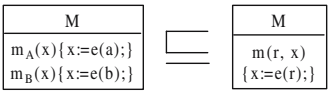

Fig. 2. Parameterize Method

Rule 1 (Extract Method). *Assume that M has method set* $\{m_1(\{body[c]\}\} \cup ops$. *If* c *is part of the body of* m_1 *that does not modify any local variables of* m_1 *outside the scope*[1] *of* c. *We have*

$$\begin{array}{l} cdecls; M[\{m_1()\{body[c]\}\} \cup ops] \\ \sqsubseteq\ cdecls; M[\{m_1()\{body[m_2()]\}, m_2()\{c\}\} \cup ops] \end{array} \quad (1)$$

where m_2 *is a new method name that is not used in cdecls or ops.*

This rule is illustrated in Fig. 1 where the program command c in the body of method m_1 and it is extracted as a method $m_2()$. Actually this is a special case of the general *Extract Method* rule. In the general case, c may refer to local variables that can be passed as arguments to the extracted method, with adequate parameter passing mechanisms.

If several methods do similar work with different values in their bodies, the following rule permits using only one method with an extra parameter for different values.

Rule 2 (Parameterize Method). *Let ops be a set of methods,* c *a command, and* m *a name not used in ops or cdecls. We have below rule where* x *is a result parameter of* m*:*

$$\begin{array}{l} cdecls; M[ops \cup \{m_a(x)\{x := e(a)\}, m_b(x)\{x := e(b)\}\}] \bullet P(c) \\ \sqsubseteq\ cdecls'; M[ops' \cup \{m(r,x)\{x := e(r)\}\}] \bullet P(c') \end{array}$$

where

$$\begin{array}{lcl} cdecls' & \widehat{=} & cdecls[m(a,x)/m_a(x), m(b,x)/m_b(x)] \\ ops' & \widehat{=} & ops[m(a,x)/m_a(x), m(b,x)/m_b(x)] \\ c' & \widehat{=} & c[m(a,x)/m_a(x), m(b,x)/m_b(x)] \end{array}$$

The idea of this rule is depicted in Fig. 2, where the two similar methods are replaced by the parameterized one. As said before, this is a system refinement rule.

Rule *Move Method* says that if a method of a class M only refers to attributes of another class N, we can move the method to class N.

Rule 3 (Move Method). *Let* N b *an attribute of class M,* $ops \cup \{m()\{c\}\}$ *the method set of* M*, where* m *is only used locally in* M. *And* ops_1 *the method set of* N *such that* $m()$ *is not in* ops_1. *If* c *only refers to an attribute* $b.x$ *of N and a method* $b.n()$ *of* b *for theoretical neatness*[2]. *Define*

$$\begin{array}{lcl} ops' & \widehat{=} & ops[b.m()/m()] - \{m()\} \\ c' & \widehat{=} & c[x/b.x, n()/b.n()] \end{array}$$

where $F[a/b]$ *stands for the substitution of all occurrences of* b. *We have*

$$\begin{array}{l} cdecls; M[N\ b, ops \cup \{m()\{c\}\}]; N[ops_1] \\ \sqsubseteq\ cdecls; M[N\ b, ops']; N[ops_1 \cup \{m()\{c'\}\}] \end{array}$$

provided that $m()$ *is not called from outside of M on the left hand side of the rule.*

[1] This means local variables declared outside c.

[2] It can be the case that c refers to a number of attributes and a number of methods of *N*.

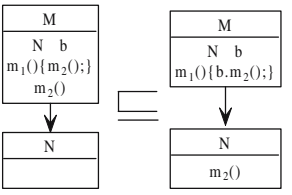

Fig. 3. Move Method

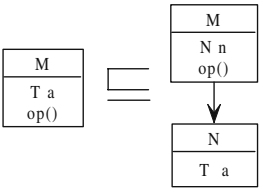

Fig. 4. Replace Data Value with Object

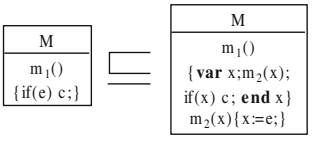

Fig. 5. Decompose Conditional

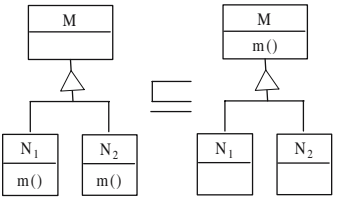

Fig. 6. Pull Up Method

A special case of this rule is shown in Fig. 3, where a arrow from M to N denotes that M has an N attribute. The combination of the rules for extracting and moving methods allows us to decompose a class for low coupling and high cohesion [14].

When we have a data item that needs additional data or behavior, the following rule allows us to define a new class to turn the data item into an object.

Rule 4 (Replace Data Value with Object). *Assume M is a class with attribute a of some primary data type T and methods ops, we have a refinement law:*

$$cdecls; M[T\ a, ops] \sqsubseteq cdecls; N[T\ a]; M[N\ n, ops']$$

Here N is a fresh class name not used in $cdecls$. If a is not public, we can define accessing methods $geta()$ and $seta()$. Further, ops' is the same as ops except the references to a are replace by suitable "get" or "set" method calls when necessary.

This rule is shown in Fig. 4. One can see that the attribute $T\ a$ is replaced by an object in the refactorred program which encapsulates the data.

When there is a complicated conditional (if-then-else) statement, we can extract a method from the condition, and use the result of the method in the conditional instead.

Rule 5 (Decompose Conditional). *Assume e is a boolean expression, then*

$$\begin{array}{l} cdecls; M[ops \cup \{m_1()\{c_1 \lhd e \rhd c_2\}\}] \\ \sqsubseteq\ cdecls; M[ops \cup \{m_1()\{\mathbf{var}\ bool\ x; m_2(x); c_1 \lhd x \rhd c_2; \mathbf{end}\ x\}, m_2(; bool\ y)\{y := e\}\}] \end{array}$$

where m_2 is a name that does not occur in ops, x is a new variable that is not in the body of m_1, and y is a result parameter of m_2.

In the general cases, the conditional is only a part of the body of m_1. This rule is illustrated in Fig. 5, where condition e is replaced by a variable which might make the program clearer.

When a method presents in each of several subclasses of a class, we can move it to the superclass. The rule below shows a special case with only two subclasses involved.

Rule 6 (Pull Up Method). *Assume M is the super class of N_1 and N_2, method $m()$ are declared with the same definition in both N_1 and N_2, and all attributes used in $m()$ are in M. Let ops be the operations declared in M which does not include $m()$. Then*

$$\begin{aligned} &\mathit{cdecls}; M[\mathit{ops}]; N_1[M, \{m()\} \cup \mathit{ops}_1]; N_2[M, \{m()\} \cup \mathit{ops}_2] \\ &\sqsubseteq \mathit{cdecls}; M[\mathit{ops} \cup \{m()\}]; N_1[M, \mathit{ops}_1]; N_2[M, \mathit{ops}_2] \end{aligned}$$

As shown in Fig. 6, the duplicated method $m()$ is pulled up to the superclass.

4 Pattern-Directed Refactoring Rules

Design Patterns [9] are widely accepted as good practice in OO development. We focus on how to transform a program to make it conforming design patterns, i.e. the transformation from a naiver program to a design pattern styled program. The formal rules for these transformations are more effective and relatively more complex. We would like to keep the term *design pattern directed refactoring rules* for these transformations.

Different from basic refactoring rules, many design patterns directed rules are contextually sensitive, which require a change in the class declarations to be related to the corresponding modification of the main program. Therefore, they need to be defined as *program refinement*. We formalize each of these rules as a refinement law of the form:

$$\mathit{Cds}; \mathit{cdecls} \bullet P[c_0] \sqsubseteq \mathit{Cds}_d; \mathit{cdecls} \bullet P[c/c_o]$$

where

- Cds are some class declarations in the original naive program, Cds_d for the corresponding classes in the result, while *cdecls* for the classes unchanged,
- $P[c_0]$ and $P[c/c_0]$ are the main programs before and after the refinement, containing c_0 and c respectively, which will be called as *protocols* of the rule.

We call touple $\langle \mathit{Cds}, \mathit{Cds}_d, c_0, c \rangle$ the *frame* of a rule, and give each rule by defining its frame.

Some pattern-directed rules require tedious side-conditions. A fully formal definition requires to specify these conditions so that the rules can be proved and the conditions can be checked when the rules are applied or automated. For simplicity of the presentation, however, we assume that all the new methods and attributes on the right hand side of a rule use fresh names. The correctness issues of the rules are discussed in the next section, and more details are left in the technical report [15].

In the Gang-of-Four book [9], design patterns are classified into three categories: *Creational*, *Structural* and *Behavioral* patterns. We will take one representative pattern from each of these categories in this paper.

Abstract Factory: It is a creational pattern providing an interface for creating a family of related objects without explicitly invoking the class-specific constructors, thus enhancing extensibility and adaptability. The following rule shows how to refactor a piece of program with constructor invocations to *Abstract Factory* pattern-style program.

The classes and their relations for this rule corresponding to this pattern is depicted in Fig. 7, where we use methods named as *constructor* in classes $\mathit{ConcreteA}_0$, $\mathit{ConcreteB}_0$,

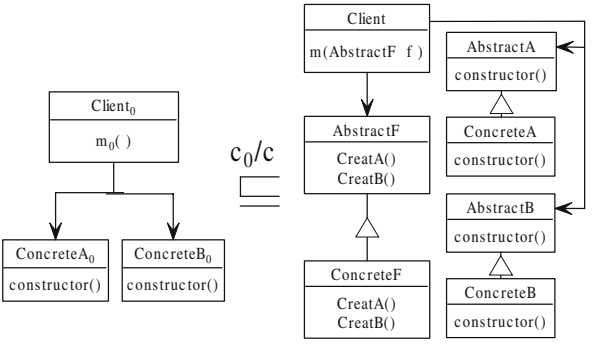

Fig. 7. Abstract Factory

ConcreteA and *ConcreteB* to explicitly show their constructor methods, that are generally assumed for all classes *C* and invoked by $C.new(x)$ in rCOS.

The class declarations *Cds* that will be refactored are shown on the left. The protocol c contains an invocation to method $m_0()$ which makes calls to the constructors of the associated two "concrete classes" *ConcreteA*$_0$ and *ConcreteB*$_0$

$$m_0()\{\mathit{ConcreteA}_0.new(a); \mathit{ConcreteB}_0.new(b)\}$$

The classes after refactoring are shown on the right of the figure that includes the new classes *AbstractF*, *AbstractA* and *AbstractB* and the *concreteF*. Also in the class declarations, the original *ConcreteA*$_0$ and *ConcreteB*$_0$ are changed to *ConcreteA* and *ConcreteB*, where the original constructor methods keep unchanged. In the new protocol c, method $m_0()$ is replaced by m in which the object creations *ConcreteA* and *ConcreteB* are carried out indirectly via its associated factory object *ConcreteF*. This is formalized in the following rule. Besides, new classes *AbstractA* and *AbstractB* are added with signatures for the constructor methods of corresponding concrete classes.

Rule 7 (Abstract Factory). *The frame of this rule is defined as:*

– *For the parts of the frame on the left hand side of the rule (in the program to be refactored)*

$$\begin{array}{lcl} \mathit{Cds} & \widehat{=} & \mathit{Client}_0[\,]; \mathit{ConcreteA}_0[\,]; \mathit{ConcreteB}_0[\,]; \\ c_0 & \widehat{=} & \{\mathbf{var}\ \mathit{Client}_0\ x; \mathit{Client}_0.new(x); x.m_0(); \gamma; \mathbf{end}\ x\} \\ m_0() & \widehat{=} & m_0()\{\mathit{ConcreteA}_0.new(a); \mathit{ConcreteB}_0.new(b)\} \end{array}$$

where γ is an arbitrary statement.

– *For the classes and protocol on the right hand side (the refactoring result)*

$$\begin{array}{lcl} Cds_d & \widehat{=} & \mathit{Client}[\,]; \mathit{AbstractF}[\,]; \mathit{ConcreteF}[\mathit{AbstractF}]; \mathit{AbstractA}[\,]; \mathit{AbstractB}[\,]; \\ & & \mathit{ConcreteA}[\mathit{AbstractA}]; \mathit{ConcreteB}[\mathit{AbstractB}]; \\ c & \widehat{=} & \{\mathbf{var}\ \mathit{Client}\ x, \mathit{ConcreteF}\ cf; \mathit{Client}.new(x); \\ & & \mathit{ConcreteF}.new(cf); x.m(cf); \gamma; \mathbf{end}\ cf, x\} \end{array}$$

where

- *class AbstractF declares two method signatures CreateA*$(, \mathit{AbstractA}\ x_a,)$ *and CreateB*$(, \mathit{AbstractA}\ x_b,)$ *where x_a and x_b are result parameters.*

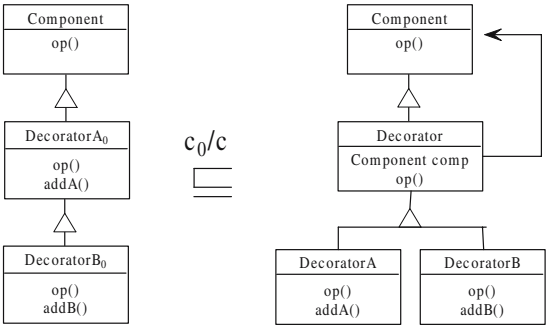

Fig. 8. Decorator

- *the two are defined in class ConcreteF:*

$$\textit{CreateA}(, \textit{AbstractA}\ x_a,)\{\textit{ConcreteA.new}(x_a)\}$$
$$\textit{CreateB}(, \textit{AbstractA}\ x_b,)\{\textit{ConcreteB.new}(x_b)\}$$

- *in the client class Client, the new method* $m()$ *is defined as*

$$m(\textit{AbstractF}\ f)\{f.\textit{CreateA}(a); f.\textit{CreateB}(b)\}$$

Concrete creation methods are encapsulated in the factory object. Because no concrete class is mentioned in the client program explicitly, the same program can be used to manipulate objects produced by other concrete factories without any modification. □

Decorator: Designers often need to add new functions to existing programs for various reasons. An example is given in Fig. 8, where the naive structures is shown on the left. Here are two classes *DecoratorA*$_0$ and *DecoratorB*$_0$ to decorate class *Component* with additional functions *addA* and *addB* injected into method *op* sequentially. This solution is not flexible enough. For example, it does not support the object with only *addB* added to *op* but not *addA*. *Decorator* is a structural pattern to provide a flexible solution for adding functionalities. The corresponding pattern style program is shown in Fig. 8 (right). The following rule describes the details of this refactoring.

Rule 8 (Decorator). *The frame of this rule is defined as:*

$$\begin{array}{lcl}
Cds & \widehat{=} & \textit{Component}[\,]; \textit{DecoratorA}_0[\textit{Component}]; \textit{DecoratorB}_0[\textit{DecoratorA}_0] \\
Cds_d & \widehat{=} & \textit{Component}[\,]; \textit{Decorator}[\textit{Component}]; \\
 & & \textit{DecoratorA}[\textit{Decorator}]; \textit{DecoratorB}[\textit{Decorator}] \\
c_0 & \widehat{=} & \{\textbf{var}\ \textit{DecoratorB}_0\ \textit{comp}; \textit{DecoratorB}_0\textit{.new}(\textit{comp}); \textit{comp.op}(); \textbf{end}\ \textit{comp}\} \\
c & \widehat{=} & \{\textbf{var}\ \textit{Component}\ \textit{comp}_1, \textit{DecoratorA}\ \textit{comp}_2, \textit{DecoratorB}\ \textit{comp}; \\
 & & \textit{Component.new}(\textit{comp}_1); \textit{DecoratorA.new}(\textit{comp}_2)[\textit{comp}_1]; \\
 & & \textit{DecoratorB.new}(\textit{comp})[\textit{comp}_2]; \textit{comp.op}(); \textbf{end}\ \textit{comp}, \textit{comp}_1, \textit{comp}_2\}
\end{array}$$

Here the actual parameters $\textit{comp}_1$ *and* $\textit{comp}_2$ *are used to initialize attribute comp, and*

– *In class DecoratorA*$_0$ *and DecoratorB*$_0$*, the definitions of method op are given as follows:*

$$\begin{array}{ll}
\textit{op}()\{\textit{Component.op}(); \textit{addA}()\} & \text{// in } \textit{DecoratorA}_0 \\
\textit{op}()\{\textit{DecoratorA}_0\textit{.op}(); \textit{addB}()\} & \text{// in } \textit{DecoratorB}_0
\end{array}$$

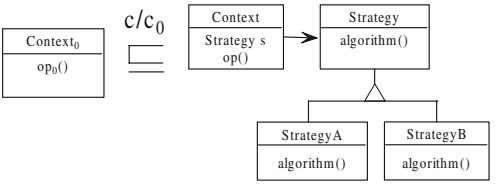

Fig. 9. Strategy

– *In Decorator, DecoratorA and DecoratorB, we have the following definitions:*

$$\begin{array}{ll} op()\{comp.op()\} & \text{// in } Decorator \\ op()\{Decorator.op(); addA()\} & \text{// in } DecoratorA \\ op()\{Decorator.op(); addB()\} & \text{// in } DecoratorB \end{array}$$

The new program supports the flexible function-extension. □

Originally, the client obtains the results of *addA*() and *addB*() by invoking *op*() of the lowest sub-class $DecoratorB_0$. After refactoring, the client still invokes *addA*() and *addB*(), in turn via the reference *comp* in *Decorator*. So the behavior is preserved.

Strategy: Sometimes we have a family of algorithms, and want to make them interchangeable during executions. *Strategy* pattern allows the context to select the algorithms dynamically without using conditionals that may make a program hard to read and maintain. We use this pattern as an example for the last category, *behavioral* patterns. This rule also shows how to refactor a program containing a conditional statement to a pattern-style program. This pattern is shown in Fig. 9.

Rule 9 (Strategy). *The frame of this rule is defined as follows:*

$$\begin{array}{lcl} Cds & \widehat{=} & Context_0[\,]; \\ Cds_d & \widehat{=} & Context[\,]; Strategy[\,]; StrategyA[Strategy]; StrategyB[Strategy]; \\ c_0 & \widehat{=} & \{\mathbf{var}\ Context_0\ con; Context_0.new(con)[\,]; con.op_0(); \mathbf{end}\ con\} \\ c & \widehat{=} & \{\mathbf{var}\ Strategy\ s; (StrategyA.new(s) \lhd b \rhd StrategyB.new(s)); \\ & & \ \ \mathbf{var}\ Context\ con; Context.new(con)[s]; con.op(); \mathbf{end}\ s, con\} \end{array}$$

where

– *In classes $Context_0$ and Context, we have the method definitions:*

$$\begin{array}{ll} op_0()\{c_A \lhd b \rhd c_B\} & \text{// in } Context_0, \text{ use conditional to make choice} \\ op()\{algorithm()\} & \text{// in } Context, \text{call an algorithmaccording to the object} \end{array}$$

where b is a global variable for the choice between specific algorithms c_A and c_B.

– *In class Strategy, StrategyA and StrategyB*

$$\begin{array}{ll} algorithm() & \text{// in } Strategy, \text{ no implementation} \\ algorithm()\{c_A\} & \text{// in } StrategyA \\ algorithm()\{c_B\} & \text{// in } StrategyB \end{array}$$

where c_A and c_B are the same sequence of commands as above. □

In the naive program, the context will execute c_A (or c_B) if b is *true* (or *false*). Correspondingly, in the pattern style program, it will create an object *StrategyA* (or *StrategyB*) if b is *true* (or *false*) for the reference s, and then execute c_A (or c_B) via the reference. So the behavior is preserved.

5 The Proofs

In this section we give the basic idea and the general proof procedure for the refinement laws, and illustrate the procedure via an example.

5.1 Class Refinement Laws

As shown above, each class refinement law is of the form

$$LHS \sqsubseteq RHS$$

According to Definition 4 (Class Refinement), to prove the law, we have to check

$$LHS \bullet P \sqsubseteq RHS \bullet P, \quad \text{For any } P = (gbv, c)$$

According to the definition, for each *class refinement* law, we need to prove that, for any $P = (gbv, c)$,

$$\forall stateV, stateV' \bullet (RHS; init; c) \Rightarrow \forall stateW, stateW' \bullet (LHS; init; c)$$

where $(stateV, stateV')$ and $(stateW, stateW')$ represent the initial and final states of the alphabets of the systems, respectively.

Using the definition of sequence composition in [11], we can prove that the following statement is equivalent to the above one,

$$\begin{array}{l} \forall stateV, stateV' \bullet (\exists stateV_0 \bullet \{RHS; init\} \\ \qquad (stateV \cup gbv, stateV_0 \cup gbv) \wedge\ c(stateV_0 \cup gbv, stateV' \cup gbv')\ ) \\ \Rightarrow \\ \forall stateW, stateW' \bullet (\exists stateW_0 \bullet \{LHS; init\} \\ \qquad (stateW \cup gbv, stateW_0 \cup gbv) \wedge\ c(stateW_0 \cup gbv, stateW' \cup gbv')\ ) \end{array}$$

As in UTP, $c(x_1, x_2)$ here stands for a predicate for the meaning (semantics) of command c, where x_1 is the state before execution of c, and x_2 the state after the execution. Further, $c_1(x, x_0) \wedge c_2(x_0, x')$ denotes that the final state after the execution of command c_1 is the same as the initial state of c_2, that is the middle state described by x_0.

We use $\mathcal{WS}$ to denote the state space of original design, and $\mathcal{VS}$ for the state space of corresponding refined design, thus *stateW*, ... $\in \mathcal{WS}$, and *stateV*, ... $\in \mathcal{VS}$. Now what we need to do is to find the corresponding states *stateW*, *stateW*$_0$, *stateW*$'$ in $\mathcal{WS}$ according to the states *stateV*, *stateV*$_0$, *stateV*$' \in \mathcal{VS}$.

To get the corresponding states, the proof goes through following steps:

1. In the first, take *stateW* = *stateV*.
2. Find a refinement mapping ρ which maps the refined state space to the original state space[3], then take *stateW*$_0 = \rho$(*stateV*$_0$).

[3] Please note that the alphabet of the state spaces is the alphabet given in section 2.2.

3. Prove the existence of *stateW′* by showing that for any command c in rCOS, the commuting diagram Fig. 10 holds. With structural induction, we need to show that the commuting diagram holds for all primitive commands. The compound ones can be proved by the structure induction easily. It is easy to check that if this commuting diagram holds, then state *stateW′* exists.

As an example, we show here the proof skeleton for **Rule 1 (Extract Method)** in Section 3. For the work, we need to give first a refinement mapping from the state space of the alphabet of our system (the seven element tuple) to itself. The mapping ρ for this rule is as follows:

$$\rho \mathrel{\widehat{=}} id \oplus \{\mathbf{op}(M) \mapsto \mathbf{op}(M) \backslash \{m_2()\}\}$$

where id stands for the identity mapping. Informally, ρ modifies only $\mathbf{op}(M)$ and keeps all the others unchanged.

Finally, we need to check that for all primitive commands, the commuting diagram in Fig. 10. holds. In this rule, we only add a method $m_2()$ and change the body of $m_1()$, so only $le.m()$ in Section 2.1 should be checked. In the proof, we need to use accessorial mappings Set, $Reset$ and ϕ. The exact definitions of these mappings can be found in [12]. Here are some informal explanations:

- $Set(\cdot)$ and $Reset$ are adjuvant *design*s for initiating and recovering the state environments when the system enters and leaves the local environment of a method call. They can be nested and act as commands to maintain the status of local variables. Although acting as "commands", they are not allowed to use explicitly in programs, but can only be used in semantic definitions and reasoning about programs.
- ϕ defines the semantics of method calls as well as any potential recursive structures. It is useful when we need to expand the semantics of a nested method call.

The proof is as follows.

- If $c = o.m_1()$ where o is an object of M, then we compute the semantics of $m_1()$. On the right hand side

$$[\![o.m_1()]\!] = \phi_M(m_1()) = Set(M); \phi_M(c); Reset$$

And on the left hand side we have

$$\begin{aligned} [\![o.m_1()]\!] &= \phi_M(m_1()) \\ &= Set(M); \phi_M(m_2()); Reset \\ &= Set(M); Set(M); \phi_M(c); Reset; Reset \\ &= Set(M); \phi_M(c); Reset \end{aligned}$$

This command does not have affect on $\mathtt{op}(M)$, so here ρ acts as id. Thus

$$\rho; c \sqsubseteq c; \rho$$

- If $c = o.m_2()$, then the right hand is not well formed. This makes the design on the right hand become *false*. Thus the law holds.

From the above example, one can see that once the refinement mapping ρ is given, the checking of the commuting diagram is straightforward by using the evaluation function. Other rules can be proved in the similar manner.

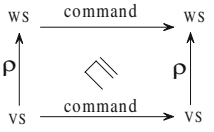

Fig. 10. Comm. Diagram for Class Ref.

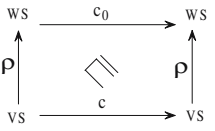

Fig. 11. Comm. Diagram of Program Ref.

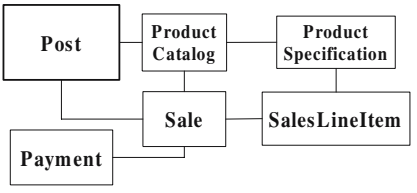

Fig. 12. Initial Structure of POST

5.2 System Refinement Laws

For *System refinement* laws, the strategy is almost the same as for the *Class refinement*. The only difference is in the third step, where we should check c and c_0 of the corresponding *protocol* rather than relative primitive commands, i.e., we have to prove

$$\rho; c_0 \sqsubseteq c; \rho$$

where c and c_0 are the given protocols in the refinement law.

The commuting diagram to be checked is illustrated in Fig. 11. Please notice that what on the arrows here are the original protocol c_0 and new protocol c rather than the primitive commands. We have given the refinement mappings and detailed proofs as well for all of our three pattern-directed rules in the report [15]. The proofs go almost the same manner as shown in Section 5.1. We omit the details here for the space.

6 A Case Study

rCOS is designed to support development of no-trivial software in an incremental way. Here we present a case study for the development of the *POST* system which is originally discussed in [14,5]. We give only an overview of this complicated case study.

POST (Point-Of-Sale Terminal) is a system used to record sales and handle payments, which is typically used in retail stores or supermarkets. It includes several hardware components such as a bar code scanner and a printer, and also a software part. We work on the software part only here. With the refinement laws for OO development, we can incrementally design the system, and finally, implement it on some platform, e.g., Java, or Visual C#.Net, etc.

In the initial design, the class structures of the program is modeled by the class diagram shown in Fig. 12.

In the refinement-based development process, supported by the relatively simple refinement laws in [12], we transform the system by steps, and finally have its *Design Model* depicted in Fig. 13, which is already very close to an executable program.

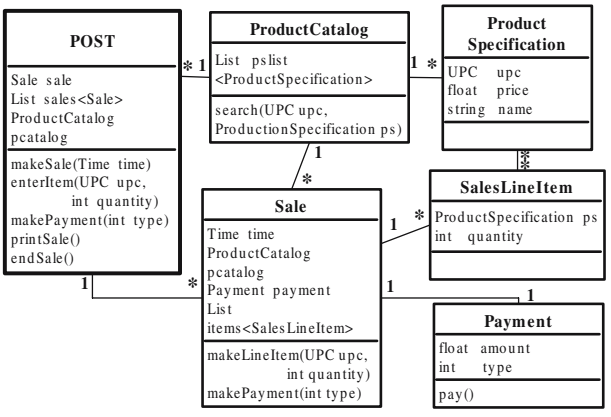

Fig. 13. Design Model of POST

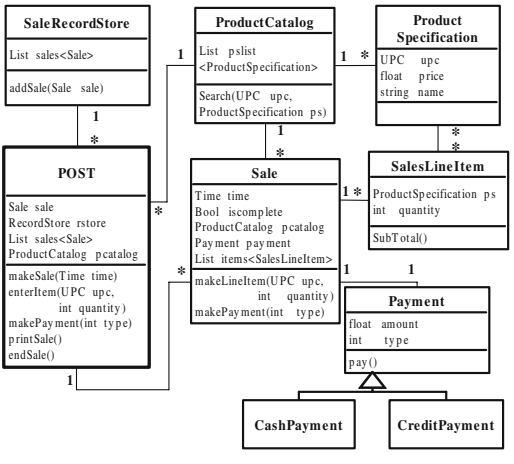

Fig. 14. Refactorred Final Model of POST

However, with a carefully analysis, we see that there are many poor-structured parts in this design model, such as:

1. Class *Post* has an attribute *sales* which is a list to record all the finished sales. It is not suitable to let the interface class maintain such a long list. In fact, this list can be considered as a database for the records. There may be several instances of *Post* working in parallel and need to share the same list. We would better use another class to maintain the list. From this consideration, we may extract a new class *RecordStore*.
2. There may be several ways to make the payment. This fact should be reflected in the design. Thus, in method *pay*() of *Payment*, we use a flag to direct the behavior of the method[4]. This kind of flag-directed code is thought a bad design for lack of flexibility and maintainability. Use the *Strategy Pattern*, we refactor it to a program with polymorphism.

[4] Please refer to [16] for the details.

3. For method *makePayment*() in class *Sale* to computer the total payment, it use now directly the attributes of *SalesLineItem*. It is better if the computation happens in the *SalesLineItem* objects to reduce the coupling between classes. So we would like to extract a method in class *Sale* and then move it to *SalesLineItem*.

Motivated by the feeling of the "bad smells", we use refactoring and pattern-directed refactoring rules to refine the system step by step. The development, or refinement, process is composed of six phases and each phase is done within several steps. The refactoring rules used include, for examples, the **Extract Method**, **Move Method**, and **Strategy** patterns etc. Here we only present the skeleton of the result of the last phase, as Fig. 14. We ignore the code of all the methods in Fig. 13 and Fig. 14..

7 Conclusions and Future Work

Refactoring and Design Pattern are both important concepts and powerful techniques in OO software development [8,13]. However, the definition of them are informal, based on intuition. Formalization of design patterns and refactoring rules is useful for scaling up the tradition refinement calculi, and it is important for tool support to correct by construction of software systems. This is the primary motivation of this work.

Based on rCOS, we present the work on formalization of refactoring rules given in [8], as well as a set of pattern-directed refactoring rules. Proving these rules and patterns precisely requires a formalism that defines object references and sharing. For example, with **Rule** 7, clients can create objects via a reference to a factory rather than invoke a constructor explicitly. This transformation improves the flexibility, reusability and maintainability of the system. In other two pattern-directed refactoring rules presented (**Rule** 8 and **Rule** 9), we also use reference type to support the implicit up-cast. This justifies the use of rCOS which is a refinement calculus with the needed features.

We include also here a detailed discussion on the skeleton of formal proofs of refactoring rules, and a proof example to illustrate the whole procedure. At last, we presented briefly a case study to show the formal development process based on these rules.

Our future work will aim at the automation of these rules and patterns and to integrate them into the rCOS tool for component-based and model driven design [4]. There exist some environments claiming to support refactoring [1]. However, a closer look at them reveals that what they do are only modifying the code rudely under people's command. There is no support to ensure that the modification is semantics-reserved.

In our current work, the proofs of the rules are based on the semantic model of rCOS. We would also like to develop an algebraic proof system for the refactoring and pattern-directed ones, so that many rules can be proved algebraically from a small set of refactoring rules. This will provide calculus of patterns and factoring rules to allows us to relate and compose patterns and rules.

References

1. Refactoringtools, `http://www.refactoring.com/tools.html`
2. Cavalcanti, A.L.C., Naumann, D.: A weakest precondition semantics for refinement of object-oriented programs. IEEE Trans. on Software Engineering 26(8), 713–728 (2000)

3. Chen, Z., He, J., Liu, Z., Zhan, N.: A model of component-based programming. In: Arbab, F., Sirjani, M. (eds.) FSEN 2007. LNCS, vol. 4767, pp. 191–206. Springer, Heidelberg (2007)
4. Chen, Z., Liu, Z., Stolz, V.: The rCOS tool. In: Cuellar, J., Maibaum, T.S.E. (eds.) FM 2008. LNCS, vol. 5014. Springer, Heidelberg (2008)
5. Chen, Z., Liu, Z., Ravn, A.P., Stolz, V., Zhan, N.: Refinement and verification in component-based model driven design. Technical Report 388, UNU/IIST, Macao SAR, China, Science of Computer Programming (submitted, 2008)
6. Cornélio, M.L., Cavalcanti, A.L.C., Sampaio, A.C.A.: Refactoring by Transformation. In: Pro. of REFINE 2002. ENTCS, vol. 70. Elsevier, Amsterdam (2002) (invited Paper)
7. Favre, L., Pereira, C.: Formalizing mda-based refactorings. In: 19th Australian Software Engineering Conference, pp. 377–386. IEEE, Los Alamitos (2008)
8. Fowler, M.: Refectoring, Improving the Design of Existing Code. Addison-Wesley, Reading (2000)
9. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns, Elements of Reusable Object-Oriented Software. Addison Wesley, Reading (1994)
10. He, J., Liu, Z., Li, X., Qin, S.: A relational model for object-oriented designs. In: Chin, W.-N. (ed.) APLAS 2004. LNCS, vol. 3302, pp. 415–436. Springer, Heidelberg (2004)
11. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Prentice-Hall, Englewood Cliffs (1998)
12. Jifeng, H., Li, X., Liu, Z.: rCOS: A refinement calculus for object systems. Theoretical Computer Science 365, 109–142 (2006)
13. Kerievsky, J.: Refactoring to Patterns. Addison-Wesley, Reading (2004)
14. Larman, C.: Applying UML and Patterns. Prentice-Hall International, Englewood Cliffs (2001)
15. Long, Q., He, J., Liu, Z.: Refactoring and pattern-directed refactoring: A formal perspective. Technical Report 318, UNU/IIST, Macao SAR, China (2005)
16. Long, Q., Qiu, Z., Liu, Z., Shao, L., He, J.: POST: A case study for an incremental development in rCOS. In: Van Hung, D., Wirsing, M. (eds.) ICTAC 2005. LNCS, vol. 3722, pp. 498–513. Springer, Heidelberg (2005)
17. Markovic, S., Baar, T.: Refactoring OCL annotated UML class diagrams. Journal Software and Systems Modeling 7, 25–47 (2008)
18. Roberts, D.B.: Practical Analysis for Refactoring. PhD thesis, University of Illinois at Urbana Champain (1999)
19. Silva, L., Sampaio, A., Liu, Z.: Laws of object-orientation with reference semantics (submitted for publication)
20. Tokuda, L.A.: Evolving Object-Oriented Designs with Refactoring. PhD thesis, University of Texas at Austin (1999)
21. Wake, W.C.: Refactoring Workbook. Pearson Education, London (2004)
22. Zhao, L., Liu, X., Liu, Z., Qiu, Z.: Graph transformations for object-oriented refinement. Formal Aspects of Computing (2008) doi:10.1007/s00165-007-0067-y