POST: A Case Study for an Incremental Development in rCOS*

Quan Long[1], Zongyan Qiu[1], Zhiming Liu[2,**], Lingshuang Shao[3], and He Jifeng[2]

[1] LMAM and Department of Informatics, School of Mathematical Sciences,
Peking University, Beijing, China
{longquan, qzy}@math.pku.edu.cn

[2] International Institute for Software Technology,
United Nations University, Macao, China
{lzm, hjf}@iist.unu.edu

[3] Software Engineering Institute, Peking University, Beijing, China
shaolsh04@sei.pku.edu.cn

Abstract. We have recently developed an object-oriented refinement calculus called rCOS to formalize the basic object-orient design principles, patterns and refactoring as refinement laws. The aim is of rCOS is to provide a formal support to the *use-cased driven, incremental* and *iterative* Rational Unified Process (RUP). In this paper, we apply rCOS to a step-wised development of a Point of Sale Terminal (POST) system, from a requirement model to a design model, and finally, to the implementation in Visual C#.

Keywords: Refinement, Software design, Object-orientation, Refactoring, UML.

1 Introduction

In the imperative paradigm, the *specification* of a problem is mainly concerned with the control and data structures of the program. The program development is the design and implementation of data structures and algorithms through a number of steps of *refinement*. *Verification* is needed to prove that each step preserves the specification of the control and data structures in the previous step. Various formal methods, especially those state-based models [5,10] such as VDM [11] and Z [4], are widely found helpful in correct and reliable construction of such a program.

The object-oriented requirement analysis, design and programming are popular recently in practical software engineering. Recent development and application of UML and the Rational Unified Process (RUP) have led to the use of design patterns and refactoring more effective.

However, the research in the formal aspects and techniques does not reflect or provide enough support to these newly developed objected-oriented engineering principles and development processes. It is still hard to obtain assurance of correctness in object-oriented developing process using old fashioned programming techniques. Model-based formalisms have been extended with object-oriented techniques, via languages such as Object-Z [1], VDM++ [6], and methods such as Syntropy [3] which

* Supported by NNSFC(No. 60173003) and NKBRPC(2004CB318000).
** Partly supported as a research task of E-Macao Project funded by the Macao Government.

D.V. Hung and M. Wirsing (Eds.): ICTAC 2005, LNCS 3722, pp. 485–500, 2005.

uses the Z notation, and Fusion [2] that is related to VDM. Whilst these formalisms are effective at modelling data structures as sets and relations between sets, they do not capture the main principles of object-oriented decomposition, including functionality delegation, class decomposition, and object-oriented refinement. Object-oriented refinement must capture the notation of substitutability of a group of associated classes by another group of associated classes. The development of rCOS is mainly motivated by these problems [8].

In this paper, we use the case study of a Point of Sale Terminal (POST) system, originally from [12] to demonstrate how a system can be formally and systematically developed supported by the rCOS based development process. A POST system is typically used in a retail store or supermarket. It includes hardware components such as a computer and a bar code scanner, and the software to control the system. The case study also shows how the techniques could be used in the development of other systems. It mainly demonstrates how functionality is decomposed in the object-oriented settings by the expert pattern, and how object-oriented structure is refined by refactoring rules.

The rest of this paper is organized as follows. We first briefly introduce rCOS and related development process in Section 2 and Section 3 respectively. And then, in Section 4, we present the development process, or refinement process of POST software system. The executable product developed from the final refined design is illustrated in Section 5. Finally, in Section 6, we conclude the paper and discuss some future research directions.

2 Overview of rCOS

In this section we give a brief introduction to the rCOS model and our earlier work based on it. We refer the readers to [8,9,15] for more details.

2.1 rCOS Syntax

rCOS is a refinement calculus of *object-oriented sequential systems*. In rCOS, a system (or program) S is of the form *cdecls* • *P*, consisting of class declaration section *cdecls* and a main method *P*. The main method P is a pair $(\mathtt{glb}, c)$ of a set `glb` of *global variables declarations* and a command c. *P* can also be understood as the *main method* in Java. The class declaration section *cdecls* is a sequence of class declarations $cdecl_1; \ldots ; cdecl_k$, where each class declaration $cdecl_i$ is of the form

$$[\mathtt{private}]\ \mathtt{class}\ N\ \mathtt{extends}\ M\ \{$$
$$\mathtt{private}\ (U_i\ u_i = a_i)_{i:1..m};\ \mathtt{protected}\ (V_i\ v_i = b_i)_{i:1..n};\ \mathtt{public}\ (W_i\ w_i = c_i)_{i:1..k};$$
$$\mathtt{method}\quad m_1(\underline{T}_{11}\ \underline{x}_1, \underline{T}_{12}\ \underline{y}_1, \underline{T}_{13}\ \underline{z}_1)\{c_1\}; \cdots ; m_\ell(\underline{T}_{\ell 1}\ \underline{x}_\ell, \underline{T}_{\ell 2}\ \underline{y}_\ell, \underline{T}_{\ell 3}\ \underline{z}_\ell)\{c_\ell\}\}$$

Note that

- A class can be declared as `private` or `public`, but by default it is assumed as `public`. Only the public classes and primitive types can be used in the global variable declarations `glb`.
- *N* and *M* are distinct names of classes, and *M* is called the direct superclass of *N*.

- Attributes annotated with `private` are private attributes of the class, and similarly, the `protected` and `public` declarations for the protected and public attributes. Types and initial values of attributes are also given in the declaration.
- The `method` declaration declares the methods, their value parameters ($\underline{T}_{i1}\ \underline{x}_i$), result parameters ($\underline{T}_{i2}\ \underline{y}_i$), value-result parameters ($\underline{T}_{i3}\ z_i$) and bodies (c_i). We sometimes denote a method by $m(\underline{paras})\{c\}$, where $\underline{paras}$ is the list of parameters of m, and c is the body command of m. The method body c_i is a command that will be defined later.

We use Java convention to write a class specification, and assume an attribute `protected` when it is not tagged with `private` or `public`. We have these different kinds of attributes to show how visibility issues can be dealt with. We can also have different kind of methods for a class, however, it is omitted here for simplicity of the theory. Instead, we assume all methods in public classes are public and can be inherited by a subclass and accessed by the main method, and all methods in private classes are protected.

When we write refinement laws, we use the following notation to denote a class declaration of class N.

$$N[M, \mathtt{pri}, \mathtt{prot}, \mathtt{pub}, \mathtt{op}]$$

where *M* is the name of the direct superclass of *N*, `pri`, `prot` and `pub` are the sets of the private, protected and public attribute declarations, and `op` is the set of the method declarations of *N*. When there is no confusion, we only explicitly give the parameters that we are concerned. For example, we use $N[\mathtt{op}]$ to denote a class with a set `op` of methods, and $N[\mathtt{prot}, \mathtt{op}]$ a class with a protected attributes `prot` and methods `op`.

Commands. rCOS supports typical object-oriented programming constructs, but it also allows some commands for the purpose of specification and refinement:

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

where b is a Boolean expression, e is an expression, and le is an expression which may appear on the left side of an assignment and is of the form $le ::= x \mid le.a$, where x is a simple variable and a an attribute of an object. We use $le.m(\underline{e}, \underline{v}, \underline{u})$ to denote a call of method m of the object denoted by le with actual value parameters $\underline{e}$ for input to the method, actual result parameters $\underline{v}$ for the return values, and value-result parameters $\underline{u}$ that can be changed during the execution of the method and with their final values as return values too. The command $C.new(x)[\underline{e}]$ creates a new object of class C with the initial values of its attributes assigned by the values of the expressions in $\underline{e}$ and assigns it to variable x. Thus, $C.new(x)[\underline{e}]$ uses x with type C to refer to the newly created object. The other commands, $c; c$, $c \lhd b \rhd c$, $c \sqcap c$ and $b * c$ denote the conventional commands of sequential composition, choice, non-determined choice, and iteration respectively.

The expressions e appear in the commands are defined in a usual way. We ignore them here.

2.2 Semantics and Refinement of Object Systems

rCOS adopts an observation-oriented and relational semantics. The model describes the behavior of an object-oriented program by a *design* containing seven logical variables as its *free variables* that form the *alphabet* "α" in [10] of the program. They are **cname**, **attr**, **op**, **superclass**, $\mathbf{\Sigma}$, **glb** and **locvar**. They record both static structure of the classes and dynamic state of the system.

Commands and class declarations, as well as an object system as a whole, are semantically defined as a *framed design* $D(\alpha, P)$ with the form $\{\alpha\} : pre(x) \vdash Post(x, x')$. That is, the effect of any piece of code are defined by the pre- and post states of the above mentioned *alphabet*. Please see [9] for details if interested.

Based on the relational model, rCOS supports refinement of object-oriented designs at different levels of abstraction during a system development. It includes design refinement, data refinement, refinement of classes and refinement of a whole system.

In [9], the *Design refinement* and *Data refinement* are defined similar to traditional ones. In this section we only present the definitions of *System refinement* and *Class refinement* as follows.

Definition 1. (**System refinement**) *Let S_1 and S_2 be object programs which have the same set global variables* **glb**. *S_1 is a* refinement *of S_2, denoted by $S_2 \sqsubseteq_{sys} S_1$, if its behavior is more controllable and predictable than that of S_2:*

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

where $\underline{x}$ are variables in **glb**.

This indicates the external behavior of S_1, that is, the pairs of pre- and post global states, is a subset of that of S_2. To prove one program S_1 refines another S_2, we require that they have the same set of global variables and the existence of a *refinement mapping* between the variables of S_1 to those of S_2 that is identical on global variables.

Definition 2. (**Class refinement**) *Let $cdecls_1$ and $cdecls_2$ be two 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:*

$$cdecls_2 \sqsubseteq_{class} cdecls_1 =_{df} \forall P \cdot (cdecls_2 \bullet P \sqsubseteq_{sys} cdecls_1 \bullet P)$$

where P stands for a main method $(\mathbf{glb}, c)$.

Intuitively, it states that $cdecls_1$ supports at least the same set of services as $cdecls_2$.

As stated in the introduction section, in our earlier work [9] and [16], we have given many useful refinement laws that capture the nature of incremental development in object-oriented programming. Please refer to them if interested.

2.3 Some Refinement Laws

We introduce some laws in [9] and [16] that will be used in the case study.

Law 1 (Law 7. in [9] Introducing a private attribute has no effect**).** *If neither N nor any of its superclasses and subclasses in* cdecls *has x as an attribute, then*

$$N[pri]; \text{cdecls} \sqsubseteq N[pri \cup \{T\ x = d\}]; \text{cdecls}.$$

Law 2 (Law 8. in [9] Changing private attributes into protected supports more services**).**

$$N[pri \cup \{T\ x = d\}, prot]; \text{cdecls} \sqsubseteq N[pri, prot \cup \{T\ x = d\}]; \text{cdecls}.$$

Law 3 (Law 9. in [9] Adding a new method refines a declaration). *If m is not in N, let $m(paras)\{c\}$ be a method with distinct parameters paras and a command c, then*

$$N[ops]; \text{cdecls} \sqsubseteq N[ops \cup \{m(paras)\{c\}\}]; \text{cdecls}$$

Law 4 (Law 10. in [9] Refining a method refines a declaration). *If $c_1 \sqsubseteq c_2$,*

$$N[ops \cup \{m(paras)\{c_1\}\}]; \text{cdecls} \sqsubseteq N[ops \cup \{m(paras)\{c_2\}\}]; \text{cdecls}$$

Law 5 (Law 1. (Extract Method) in [16]). *Assume that $m_1()\{c\}$ is a method in* op *of class M. Let* $\text{op}_1 = \text{op}\backslash\{m_1()\{c\}\}$. *Then*

$$cdecls; M[\text{op}] \sqsubseteq cdecls; M[\text{op}_1 \cup \{m_1()\{m_2()\}, m_2()\{c\}\}]$$

where m_2 is a method name that is not used in cdecls and op*.*

Law 6 ((Law 10. (Move Method) in [16]). *Let* op *and* op_1 *be sets of method declarations. Assume that $N\ b$ is an attribute of M, and $m()\{\hat{c}\}$ is a method of M, $m()$ is not in* op_1 *of N, and command c only refers attributes $b.x$ and methods $b.n()$ of class N. Define*

- $\hat{\text{op}}$ *to be the methods obtained from* op *by replacing each occurrence of $m()$ in every method with $b.m()$*
- *command c to be the command obtained from $\hat{c}$ by replacing each attribute $b.x$ with x and each method call $b.n()$ with $n()$.*

$$\begin{aligned} &cdecls; M[N\ b, \text{op} \cup \{m()\{\hat{c}\}]; N[\text{op}_1] \\ &\sqsubseteq cdecls; M[N\ b, \hat{\text{op}}]; N[\text{op}_1 \cup \{m()\{c\}\}] \end{aligned}$$

provided that $m()$ is not called from outside M on the left-hand-side of $\sqsubseteq$.

Law 7 ((Law 12. (Extract Class) in [16]). *Assume N is a fresh name which is not used in $cdecls$ and $m_2()$ does not refer any attribute of M. Then we have*

$$cdecls; M[m_1(), m_2()] \bullet P \sqsupseteq cdecls'; M[N\ n, \hat{m}_1()]; N[m_2()] \bullet P'$$

where $cdecls'$ is gain from $cdecls$ by substitute all $M.m_2()$ to $N.m_2()$, P' is gain from P by substitute all $M.m_2()$ to $N.m_2()$, and $\hat{m}_1() = m_1()[n.m_2()/m_2()]$.

Law 8 ((Law 59. (Strategy) in [16]). *Assume all the newly introduced names are fresh ones. We have*

$$\begin{aligned} &Context_0[Strategy\ s, op()]; Strategy[algorithm_0()] \\ &\sqsubseteq Context[Strategy\ s, op()]; Strategy[algorithm()]; \\ &\qquad StrategyA[algorithm()]; StrategyB[algorithm()] \end{aligned}$$

where

- $op() =_{df} \{s.algorithm()\}$.
- *In class StrategyA, algorithtm() $=_{df} \{c_A\}$, where c_A is a sequence of commands for a particular algorithm.*
- *In class StrategyB, algorithtm() $=_{df} \{c_B\}$, where c_B is a sequence of commands for another particular algorithm.*
- $algorithm_0() =_{df} \{c_A \lhd b \rhd c_B\}$, *where b is a boolean variable for making a choice between algorithm c_A and c_B.*

Finally, we have a shorthand notation $\exists o : T$ which stands for the existing of a reference o which refers to an object of type T. It can be formally defined using rCOS semantics. We use it here to replace the standard notations for simplicity and intuition. Also, we do not have *return* keyword in the syntax of rCOS. But it can be defined using local variable declaration. We will use it for intuition.

3 rCOS Support to RUP

Now we discuss how rCOS supports a step-wised, incremental and iterative development process. For more formal details, please refer to [15].

The incremental development initiates in the requirement analysis to reach the *Use Cases* of the system [13], then, the *Conceptual Model* and *Design Model* [15,14] are built sequentially. From an informal view, a *Conceptual Model* can be thought as a class diagram in which all classes have only attributes without methods, and *Design Model* a class diagram in which all classes have attributes and method specifications (not necessarily code) as well.

The process starts with the creation of a requirement model (specification) of the system. The requirement model consists of a *conceptual* class diagram and a *use-case* model. The conceptual model is specified as a rCOS class declaration section without methods. Use cases are specified as use-case controller class with the user operations as its methods. The conceptual model is created by identifying the domain concepts as classes and relations among the concepts as attributes of classes [13,15]. This can be carried out incrementally by adding more and more use cases, classes and attributes. Each incremental step is a refinement in rCOS [9]. This is called the horizontal refinements.

The design can take the use cases in turns, planned according to their significance, urgency, and risks. In the design of a use case, each use case operation is decomposed by delegation its sub-functionalities (responsibilities) to the classes which maintains the information for the realization of the functionalities. These classes are called the experts of the functionalities. This will also transform the conceptual class diagram by adding the specifications or code of these responsibilities to their expert classes, producing a design class model. This activities are also proven to be rCOS refinement.

Then implementation can also take the designs of the tasks in turns, by coding the methods of the classes. The refinement from the requirement specifications to the designs and to the implementations is called the vertical refinement.

4 A Development of POST

In this section, we present our incremental development as a sequence of refinement steps. During the development, we always denote the system as a sequence of class declarations. Initially, *POST1* stands for the first version, the *Conceptual Model*, of the system. And then, with the support of the refinement laws in Section 2, we refine it to *POST2*. Similarly, *POST2* can be refined to *POST3*. At last, the system reaches *POST7* which is the final version of the design. Intuitively, each version of the sequence of class declarations is depicted by a corresponding UML class diagram.

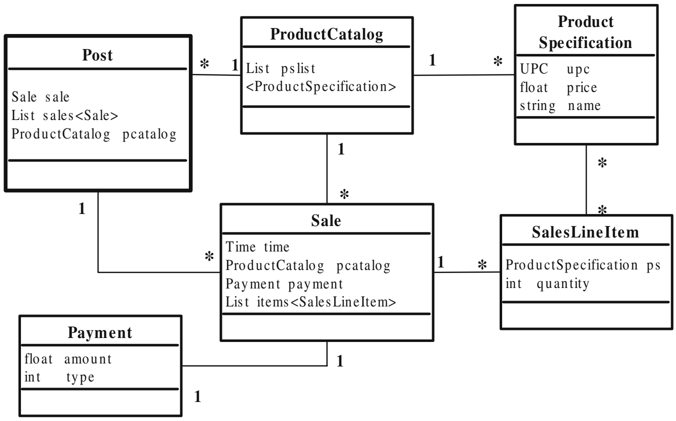

Fig. 1. Conceptual Model

4.1 Conceptual Model

At the beginning, we should determine the basic components of the system. After the requirement analysis, we decide to have the classes as follows: A *Product Catalog* as a database to store the information of all possible on sale products of the given supermarket. Each item of the database is a *Product Specification*. When a sale begins, we need to build a *Sale* object which is composed of many *Sales Line Item* to record all the products purchased. At last, the customer has to make a payment. Thus we need another object *Payment*. During the execution, we will create many instances of *Sale*, *Sales Line Item*, and *Payment*. Finally, we need a class as the user interface which is the *use case controller* of the system. We name it as *Post*.

Our next job is to add the attributes to achieve the *Conceptual Model*. During the requirement analysis, we realize that these classes should have attributes as follows:

– *Post*, which act as the interface of the system, should maintain at least three attributes: *sale* refers to the current sale object, *sales* as a list of sale objects to record all the handled sales, and a reference to the database *ProductCatalog*.
– *ProductCatalog* has a list of references to its *ProductSpecification*s.
– *ProductSpecification* should have a *name*, an attribute *upc* which stands for "Universal Product Code" as its key in the database, and another attribute *price*.
– *Sale* should have at least four attributes: a business time *time*, a reference to *ProductCatalog*, a reference to the *payment* object and a list of its *SalesLineItem*.
– *SalesLineItem* should have a reference to its corresponding *ProductSpecification* and a integer, *quantity*, to record how many products of this kind are purchased.
– The last class, *Payment* should remember how much money the customer should pay in its attribute *amount* and the payment way in *type*. Here we only deal with two kinds of payment: $type = 0$ stands for pay by cash and $type = 1$ for pay by credit card.

In our relational OO model, we can add private attributes and change a private attribute into a protected one by **Law 1** and **Law 2**. We can apply these laws repeatedly to add all the above mentioned attributes to our classes.

Thus we reach the class diagram in which all the attributes have been filled in their corresponding classes, that is, the *Conceptual Model* of the system. Fig. 1 illustrated the class diagram.

We denote the classes depicted in Fig. 1 as *POST1* which is a sequence of class declarations.

4.2 Use Case Controller Class

Having the *Conceptual Model*, *POST1*, next we consider to refine the system to the *Design Model* which includes all the method specifications. We start from the controller class *Post* in which each method specification represents a formal *use case* specification. As the result of the *use case* analysis, we realize that *Post* has to offer at least five methods: *makeSale*() to initiate a business by creating a new object *sale* of type *Sale*; *enterItem*() to add a sale line item to the *sale* object; *makePayment*() to summarize the price and create a *payment* object; *printSale*() and *endSale*() to print and end the business respectively. Further, *endSale*() has another job which is adding the reference of current *sale* object to the *sales* list.

Here we formally give the details of the method specifications as follows:

- *makeSale(Time time)*
  $pcatalog \neq nil \vdash sale'.time = time \wedge sale'.pcatalog = pcatalog$
- *enterItem(UPC upc, int quantity)*
  $pcatalog \neq nil \wedge sale \neq nil \wedge quantity \neq 0 \vdash$
  $\exists item : SalesLineItem \bullet sale.items' = sale.items \cup \{item\}$
  $\wedge item.upc = upc \wedge item.quantity = quantity$
  $\wedge(\exists ps : ProductSpecification \bullet ps \in pcatalog \wedge item.ps = ps \wedge ps.upc = upc)$
- *makePayment(int type)*
  $sale \neq nil \wedge type \in \{0, 1\} \vdash$
  $\exists payment : Payment \bullet sale.payment' = payment$
  $\wedge payment.amount = \sum_{item \in items} item.ps.price \times item.quantity$
  $\wedge((type = 0 \wedge \{\text{Paid by cash}\}) \vee (type = 1 \wedge \{\text{Paid by credit}\}))$
  where {Paid by cash} stands for customer's completion of paying by cash, and {Paid by credit} stands for customer's completion of paying by credit card.
- *printSale()*
  $sale \neq nil \wedge done(makePayment) \vdash \{\text{Print the sales line item report}\}.$
  where the predicate $done(makePayment)$ means the customer has made payment, and {Print the sales line item report} stands for printing the receipt for customer.
- *endSale()*
  $sale \neq nil \wedge done(makePayment) \vdash sale' = nil \wedge sales' = sales \cup \{sale\}$

The class diagram is depicted in Fig. 2. We denote the corresponding class declarations as *POST2*. With the support of **Law 3** we can prove that adding a method is a refinement to the system. So, trivially, applying this law five times, we have $POST1 \sqsubseteq POST2$.

4.3 Design Model

Having added the interface methods to the system, the next task we confront with is to develop all the methods of the classes to complete our *Design Model*.

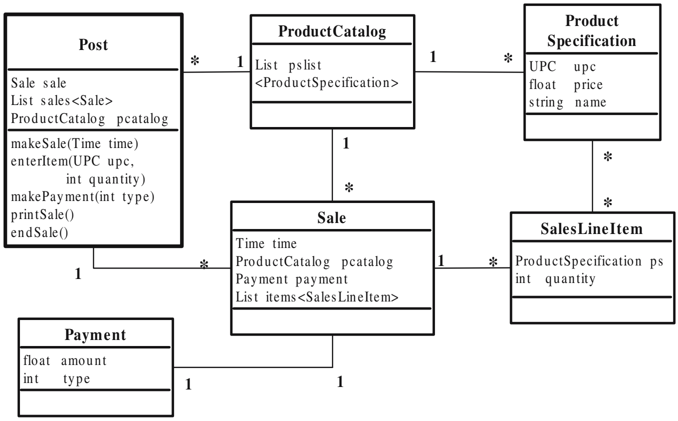

Fig. 2. Use Case Controller

Firstly, we delegate some of the tasks of *Post* to *Sale*. To achieve this, we first develop the following two methods in the class *Sale*. For the same reason to subsection 4.2, the new system added these methods refines the former version.

- *makeLineItem(UPS ups, int quantity)*
  $pcatalog \neq nil \wedge quantity \neq 0 \vdash$
  $\exists item : SalesLineItem \bullet items' = items \cup \{item\} \wedge item.upc = upc \wedge item.quantity = quantity \wedge (\exists ps : ProductSpecification \bullet ps \in pcatalog \wedge item.ps = ps \wedge ps.upc = upc)$
- *makePayment(int type)*
  $type \in \{0, 1\} \vdash \exists payment : Payment \bullet payment' = payment$
  $\wedge payment.amount = \sum_{item \in items} item.ps.price \times item.quantity$
  $\wedge((type = 0 \wedge \{\text{Paid by cash}\}) \vee (type = 1 \wedge \{\text{Paid by credit}\}))$

Secondly, we can implement, or refine, in our model, the methods *enterItem*(), and *makePayment*() in class $Post$ by invoking the above developed methods as follows:

- *enterItem′(UPC upc, int quantity)={sale.makeLineItem(upc, quantity)}*
- *makePayment′(int type)={sale.makePayment(type)}*

Now after adding the methods *makeLineItem*() and *makePayment*() to the class $Sale$, we substitute *enterItem*(), *makePayment*() with *enterItem′*(), *makePayment′*() in class $Post$. We denote the new system as *POST3*.

Using the semantic model of [9], we can prove that in class $Post$, *enterItem*() $\sqsubseteq$ *enterItem′*() and *makePayment*() $\sqsubseteq$ *makePayment′*(). By applying **Law 4** we have *POST2* $\sqsubseteq$ *POST3*.

We will still use unprimed names *enterItem*() and *makePayment*() to denote the newly refined methods in *POST3*. We make this abuse only for avoiding too many notations. In the rest of this paper we will adopt this abuse where no confusion will be made. The corresponding class diagram of *POST3* is depicted in Fig. 3.

Next, we will continue to delegate the tasks of *Sale* to *ProductCatalog* and *Payment*. Similar to the above process, we develop a new method *Search*() in class *ProductCatalog*

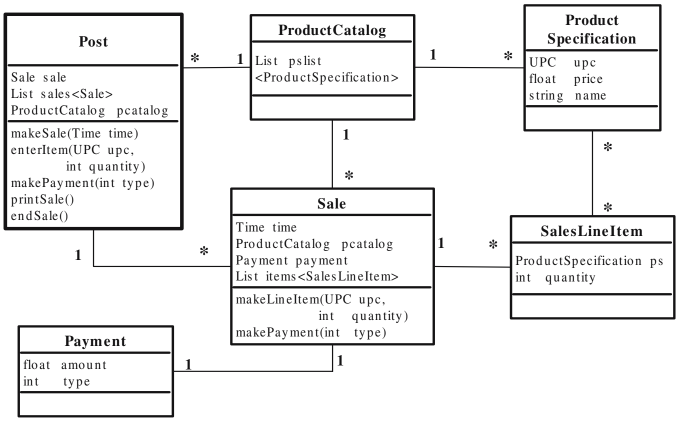

Fig. 3. Primary Design Model

and invoke it in the method $makeLineItem()$ of class *Sale*. Let us see the specification of the new method:

Search(UPC upc, ProductSpecification ps):

$pslist \neq nil \vdash (ps' = null) \lhd (\exists ps \in pslist \wedge ps.upc = upc) \rhd (ps' = ps)$

This method searches a valid *Product Specification* from *ProductCatalog* and return it when success. Supported by this method, we can implement the method *makeLineItem* in class *Sale* as follows:

$makeLineItem'(\ UPC\ upc, int\ quantity) =$
$\{\mathbf{var} ProductSpecification\quad ps;$
$Search(upc,\ ps);$
$(ps \neq null) \rhd \{$
$\quad \mathbf{var}\ SalesLineItem\quad sli;$
$\quad ProductSpecification.\mathbf{new}(sli,[ps,quantity])\}$
$items.Add(sli);$
$\mathbf{end}\ ps\}$

Also, motivated by delegating a task of class *Sale* to class *Payment*, we develop a new method $pay()$ in the class *Payment*:

$pay() = \{\{\text{Paid by cash}\} \lhd (type = 0) \rhd \{\text{Paid by credit}\}\}$

Supported by this method, we implement *makePayment(int type)* in class *Sale* as

$Sale.makePayment'(\ int\ type) =$
$\{\mathbf{skip} \lhd (type = 0 \vee type = 1) \rhd$
$\{$
$\mathbf{var}\ float\ \ amount = 0;$
$foreach\ (SalseLineItem\ item \in items)$
$\quad amount = amount + item.prise{*}item.quantity;$
$Payment.\mathbf{new}(payment,[amount,type]);$
$ayment.pay()$
$\}$
$\}$

In the semantic model we can prove in class *Sale* $makeLineItem() \sqsubseteq makeLineItem'()$ and $makePayment() \sqsubseteq makePayment'()$.

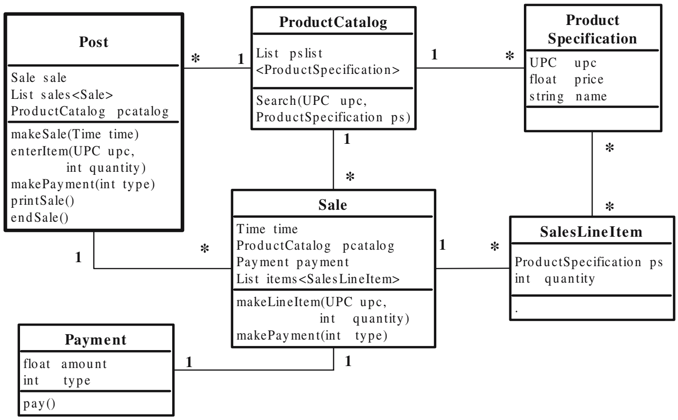

Fig. 4. Design Model

With similar process, we can get a new system as *POST4* by adding *Search*() to class *ProductCatalog* and *pay*() to class *Payment*, substituting old *makeLineItem*(), *makePayment*() with new ones in class *Sale*. And also, we have $POST3 \sqsubseteq POST4$. The corresponding class diagram of *POST4* is depicted in Fig. 4.

4.4 Refactoring: Extract Method and Move Method

After the efforts, we have reached the *Design Model*. Now we are ready to implement the system with any OO language. But there might be some K. Beck and M. Fowler's "*bad smells*" [7] existing in the design. In the rest of this section we will refactor the model to enhance the flexibility and maintainability.

After carefully reviewing of the design, we find a piece of typical code needed to be refactorred: the method *makePayment*() in class *Sale* uses the attributes of *SalesLineItem* many times. It could be better if the computation happens in *SalesLineItem* itself to reduce the coupling, or interaction, between classes. So we would like to extract a method in class *Sale* and then move it to class *SalesLineItem*.

We formally make the refactoring as follows:

Firstly, supported by the **Law 5 (Extract Method)** we have

$$Sale[makePayment()] \sqsubseteq \\ \quad Sale[makePayment()[subtotal() \backslash (item.prise * item.quantity)]]$$

where

- $subtotal() = \{\textbf{return } item.prise * item.quantity\}$
- $[a \backslash b]$ means to substitute b with a.

The right hand can be refactorred further. With the **Law 6 (Move Method)** we have

$$Sale[makePayment()]; SalesLineItem[\,] \sqsubseteq \\ \quad Sale[makePayment()[item.subtotal() \backslash subtotal()]]; SalesLineItem[subtotal()]$$

where, in the class *SalesLineItem*, $subtotal() = \{\textbf{return } prise * quantity\}$.

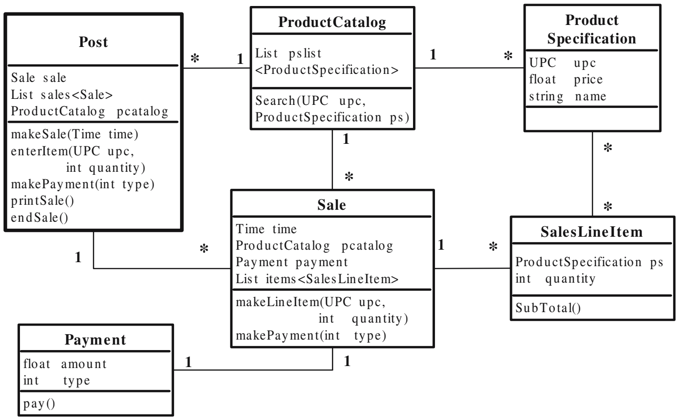

Fig. 5. Extract method and Move method

Thus we get the new class declarations *POST5* whose corresponding class diagram is depicted in Fig. 5. Again, we have *POST4* $\sqsubseteq$ *POST5*.

4.5 Refactoring: Extract Class

Next, we have a closer look at the class *Post*. It has an attribute *sales* which is a list to record all the past sales. For one thing, it is not suitable to let the interface class maintain such a long list. For another, there may be several instances of *Post* working in parallel. They should share the same list[1]. So we need another class to maintain the list. We would like to extract a new class *RecordStore* to do the job instead.

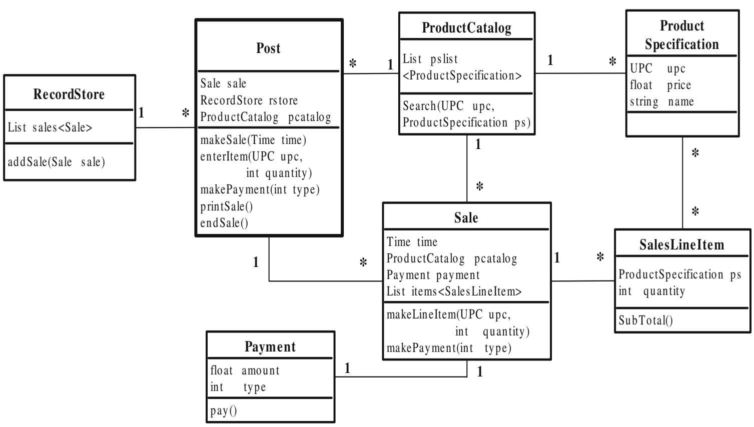

Fig. 6. Extract Class

[1] This list can be considered as a database for all the records.

Supported by the **Law 7 (Extract Class)** we have

$$Post[List\ sales\langle Sale\rangle] \sqsubseteq Post[RecordStore\ rstore]; RecordStore[List\ sales\langle Sale\rangle]$$

Similar to subsection 4.4, we can extract a method *addSale*(*Sale sale*) in class *Post*, which adds the current *sale* object to the sales list *rstore.sales*. And then, we move it to the newly developed class *RecordStore*, and have the class diagram in Fig. 6. We denote the corresponding class declarations as *POST6* and again $POST5 \sqsubseteq POST6$.

4.6 Pattern-Directed Refactoring: Strategy

Now it comes to the last phase of the refinement. This is a pattern-directed refactoring in which we introduce *Strategy* design pattern to the existing system.

In method *pay*() of class *Payment*, we have a piece of code "$c_1 \lhd type = 0 \rhd c_2$" in which the value of *type* will affect the behavior of the method. Now, directed by *Strategy Pattern,* we would like to refactor it by replacing the type code with polymorphism.

Supported by **Law 8 (Strategy)** we have

$$\begin{aligned}&Sale[makePayment(int\ type)]; Payment[int\ type, pay()] \sqsubseteq\\&\quad Sale[makePayment(int\ type)]; Payment[pay()];\\&\quad CashPayment[Payment, pay()]; CreditPayment[Payment, pay()]\end{aligned}$$

where

- The method *makePayment(int type)* on the right hand is different to the one on the left hand. We delete the command "*Payment*.**new**(*payment,[amount,type]*);" from the old method and substitute it with another command:
  $$CashPayment.\mathbf{new}(payment,[amount]) \lhd (type = 0) \rhd CreditPayment.\mathbf{new}(payment,[amount]);$$
- The method body of *pay*() in class *Payment* is empty. It is implemented by its sub-classes.
- In class *CashPayment*, method $pay() = \{\text{Paid by cash}\}$, and in class *CreditPayment*, method $pay() = \{\text{Paid by credit}\}$.

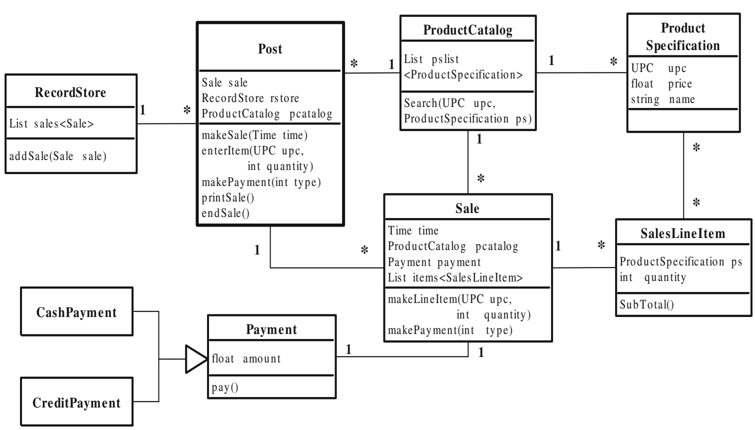

Fig. 7. Strategy Pattern-Directed Refactoring

Now the type code is replaced by polymorphism by introducing two subclasses. We denote the new class declarations as *POST7*, and have *POST6* $\sqsubseteq$ *POST7*. The class diagram is depicted in Fig. 7.

After the above refinement process, we gain the final design *POST7* from *POST0*. This ends our refinement. The classes in the final design is very near to executable code. It is easy to implement it in any OO programming languages. We have implemented it using Visual C# .Net.

5 Implementation

Supported by the $C\sharp$ and the .Net developing environment, we implement an executable software product for the final design model.

The main interface of the system, depicted in Fig. 8, is composed of five "Button"s which represent the five methods in class *Post*. Also we have two "TextBox"s to input the UPC and quantity of the current purchasing product, a "ListBox" to show the content of the current *sale*, and a pair of "RadioButton"s to choose payment ways. After the payment way is chosen, when the "Print Sale" button is pressed, the system will pop-up a form to show the receipt for customers.

An executing snapshot of our software is depicted in Fig. 9.

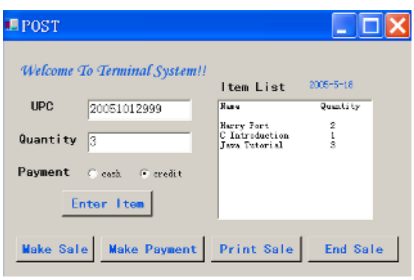

Fig. 8. Interface of POST System

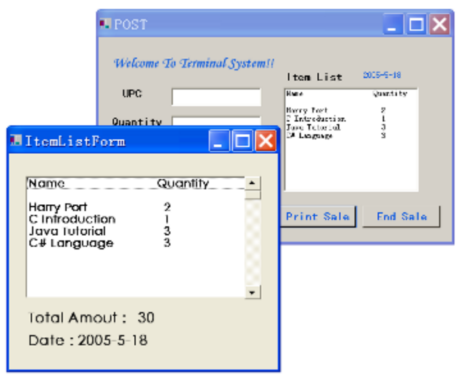

Fig. 9. POST System in Execution

6 Conclusions and Future Work

As stated in the introduction, the main motivation of this paper is to show the power of rCOS refinement calculus in incremental software development by presenting the POST case study. From this study, we could draw the conclusions about the advantages, and a tiny disadvantage as well, of rCOS.

– As we have shown in the refinement process, rCOS supports a wide range of object-oriented techniques. So it is a suitable calculus for OO development.

- In the rCOS based software development, we can prove the correctness of each developing step. So at least for highly critical systems, rCOS is a useful supporting model. Further, in teamwork of large scale software development, rCOS also offers a robust support for rigorous correctness formal proof.
- It is proven that rCOS can be used as a formal framework for the *use-cased driven, incremental* and *iterative* Rational Unified Process (RUP). And also, the rCOS based process is practical and scalable in software engineering.
- In practice, rCOS also offers a nice semantic model for correctly refactoring the existing design, and further, might give a choice for refactoring supporting tools development.
- The limitations. During the development of the POST system, we realized that there are some tiny limitations existing in the current version of rCOS. For instance, we do not have exception handling in the syntax of rCOS, making no chance to use such mechanism to deal with dynamic errors in the software development.

As for the future work, we would like to provide tool support for our refinement calculus. We hope, given the proof obligation of a refinement equation, the tool can search whether there is a refinement law syntactically matches. In rCOS, we have not yet had a result about the completeness of the laws. We will look into this problem in future work and discuss the relationship of all of our laws. Another important future work is, as mentioned above, we need to extend the current version of rCOS to support more features, such as exception handling, of OO programming languages. There, we believe, will be no essential difficulty.

References

1. D. Carrington, *et al. Object-Z: an Object-Oriented Extension to Z.* North-Halland, 1989.
2. D. Coleman, *et al. Object-Oriented Development: the FUSION Method*. Prentice-Hall, 1994.
3. S. Cook and J. Daniels. *Designing Object Systems: Object-Oriented Modelling with Syntropy*. Prentice-Hall, 1994.
4. J. Davis and J.P. Woodcock. *Using Z: Specification, Refinement and Proof.* Prentice Hall, 1996.
5. E.W. Dijkstra and C.S. Scholten. *Predicate Calculus and Program semantics*. Springer, 1989.
6. E. Dürr and E.M. Dusink. The role of VDM^{++} in the development of a real-time tracking and tracing system. In J. Woodcock and P. Larsen, editors, *Proc. of FME'93, LNCS 670*. Springer-Verlag, 1993.
7. Martin Fowler. *Refectoring, Improving the Design of Existing Code*. Addison-Wesley, 2000.
8. J. He, Z. Liu, and X. Li. rCOS: A refinement calculus for object systems. Technical Report 322, UNU/IIST, P.O. Box 3058, Macao SAR China, 2005. http://www.iist.unu.edu/newrh/III/1/page.html.
9. J. He, Z. Liu, X. Li, and S. Qin. A relational model for object-oriented designs. In *Pro. APLAS'2004, LNCS 3302*, Taiwan, 2004. Springer.
10. C.A.R. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall, 1998.
11. C.B. Jones. *Software Development: A Rigorous Approach.* Prentice Hall International, 1980.
12. C. Larman. *Applying UML and Patterns, An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice-Hall, 2001.

13. X. Li, Z. Liu, and J. He. Formal and use-case driven requirement analysis in UML. In *COMPSAC01*, pages 215–224, Illinois, USA, October 2001. IEEE Computer Society.
14. Z. Liu. Object-oriented software development with UML. Technical Report 259, UNU/IIST, P.O. Box 3058, Macao SAR China, 2002. http://www.iist.unu.edu/newrh/III/1/page.html.
15. Z. Liu, J. He, X. Li, and Y. Chen. A relational model for formal requirements analysis in UML. In J.S. Dong and J. Woodcock, editors, *Formal Methods and Software Engineering, ICFEM03, LNCS 2885*, pages 641–664. Springer, 2003.
16. Q. Long, J. He, and Z. Liu. Refactoring and pattern directed refactoring : A formal perspective. Technical Report 318, UNU/IIST, P.O. Box 3058, Macao SAR China, 2005. http://www.iist.unu.edu/newrh/III/1/page.html.