Reactive Component based Service-Oriented Design – A Case Study[1]

Jing Liu, Jifeng He
Software Engineering Institute, East China Normal University, Shanghai, China
jliu@sei.ecnu.edu.cn, jifeng@sei.ecnu.edu.cn

Abstract

Service-Oriented Design (SOD) is a software development paradigm that utilizes services as fundamental elements for developing applications/solutions. Recently, service orientation is becoming a mainstream approach for building large scale software systems. However, current software models do not take services as fundamental elements and it is very difficult for people to model from services. We consider that contract equipped reactive component model is well suited for addressing various aspects of service oriented systems including service description, delegation, as well as composition. To demonstrate this idea, a case study: e-commerce application is presented in this paper. We start from analyzing the service that the customers need, then assign the service to the interface, which in turn, delegates the service to the component. A Java-like formal language rCOS is used to describe the models. In our model, interface is used to capture the syntax information and contract is applied to capture the semantics information of the service. Component is obtained by refining the contract to designs.

1. Introduction

Service-oriented paradigm is characterized by the explicit identification and description of the externally observable behavior, or service, required by an application. For example, the World Wide Web Consortium (W3C) refers to Service Oriented Architecture (SOA) as '*A set of components which can be invoked, and whose interface descriptions can be published and discovered*'. Components can be linked, based on the description of their externally observable behavior. According to this paradigm, developers do not in principle need to have any knowledge about the internal implementation of the components being linked [1].

Fundamental to the service model is the separation between the interface and the implementation. The invoker of a service only needs to know the interface; while the implementation can evolve over time, without disturbing the clients of the service. The same interface can be offered by many implementations; several key benefits of service orientation derive from this *abstraction* of the capability.

In the context of software component models, a service is a program that communicates with others by exchanging *messages*. In other words, a service is a unit of application logic whose interface is defined purely by the messages it will receive and send. With the development of messaging standards, service orientation is quickly becoming a mainstream approach for building large scale software systems [2]. Object and component technologies are applied to service-oriented architectures. However, the inherent challenge in connecting diverse modules is the lack of [2]:

- a standard syntax, in which information from all systems could be unambiguously expressed.
- a standard semantic models that organizations could express their business practices in a consistent language.
- standard protocols that information could be passed across boundaries between operating environments and organizations.
- a standard means for binding behavior to business documents.

To address this challenge, we propose a strategy of modeling service oriented design with a contract equipped reactive component model. The services providing to clients are analyzed and assigned to the interfaces of a component. The interface can be defined to capture the syntax information of the service and contract is introduced to express the semantics of a

[1] Partially supported by National Grand Fundamental Research 973 Program (No. 2002CB312001 and 2005CB321904); National Natural Science Foundation of China (No. 60373032); Natural Science Foundation of Shanghai of China (No. 05ZR14052) and 211 Project of the Ministry of Education of China.

Proceedings of the 11th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'06)
0-7695-2530-X/06 $20.00 © 2006 IEEE

service to allow us to reason about the implementation of a component and composition of components. To show that reactive component model is effective for the specification and analysis of service oriented design, we illustrate the idea with a case study: e-Commence application. We start from analyzing the service that the customer needs, then assign the service to the interface, which in turn, delegates the service to the component. Refinement Calculus of Object Oriented System (rCOS), developed at UNU-IIST by He, Liu and Li [3], is used in model construction and model transformation.

2. Overview of the approach

We distinguish two levels of service description in service-oriented design: interface contract description and component description. An interface of a component captures the syntax information of services that the component provides and contract of the interface captures the semantics information of these services. The model of the component describes a design that is the implementation of the services to meet its interface. By model transformation, a component model can be refined step by step, even to a model ready for code generation.

In the level of an interface contract, a service is defined as a guarded design, which enables one to separate the responsibility of clients from the commitment made by the component, and identify the behavior of a component by a set of traces, a set failures and a set of divergences. Protocols are introduced to coordinate the interactions between component and the external environment [3].

A component system assumes an architectural context defined by its interfaces. We *connect* or *compose* two components by linking the operations in the provided interface of one component to the matching operations of a required interface of another. For this, we have to check whether the provided interface of one component C_1 contains the operations of a required interface of another component C_2, and whether the contract of the provided interface of C_1 meets the contract of the required interface of C_2. Further, we need to check that the provided protocol of one component is consistent with required protocol of the other so that the composed component is deadlock-free. If the two components match well, the composition gives another component C. The provided interface of C is the *merge* of the provided interfaces of C_1 and C_2. The required interfaces of C include required interfaces of C_1 and C_2, but not for the matched interfaces that are hided.

We begin with defining interface, then contract, at last component along with the composition mechanism.

2.1. Interface

As an interface of component can be defined as a specification of its access point [4], we specify an interface as a set of operation (or method) signature. It is defined as a tuple:

Definition 2.1.1 (Interface) An interface is a pair

$$I = <FDec, MDec>$$

where, *FDec* is a set fields , each with the form *T x*. *T* is the type of variable *x*; *MDec* is a set of methods, each having the form *m* (*in: U u, out: V v*). *in: U u* represents the input parameter *u* with type *U*, *out: V v* represents the output parameter *v* with type *V*. We use *I.FDec* and *I.MDec* to denote the fields and the methods of interface *I*.

Definition 2.1.2 (Composability) Two interfaces are composable if they do not use the same field name with different type.

If I and J are composable, then the notation $I \uplus J$ denotes the interface with

$$FDec =_{df} I.FDec \cup J.FDec$$
$$MDec =_{df} I.MDec \cup J.MDec$$

Definition 2.1.3 (Hiding) Let I be an interface and S a set of method names. The notation $I \setminus S$ denotes the resultant interface after removal of methods of S from I.

$$I \setminus S.FDec =_{df} I.FDec$$
$$I \setminus S.MDec =_{df} I.MDec \setminus S$$

Theorem 2.1

(1) $(I \setminus S_1) \setminus S_2 = I \setminus (S_1 \cup S_2)$

(2) $I \setminus \varnothing = I$

(3) $(I_1 \uplus I_2) \setminus S = (I_1 \setminus S) \uplus (I_2 \setminus S)$

2.2. Contract

Contract can be used to increase the reliability of software [5]. We use contracts to describe the specification of interfaces. A contract associates an interface to an abstract data model plus a set of functional specifications of its services, and as well as an interaction protocol.

Definition 2.2.1 (Contract) A contract *Ctr* is a quadruple

$$Ctr = <I, Init, Spec, Prot>$$

Proceedings of the 11th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'06)
0-7695-2530-X/06 $20.00 © 2006 IEEE

where

(1) I is an interface.

(2) *Spec* maps each method m of I to its specification

$<\alpha_m, g_m, P_m>$, where

- α_m contains the field names of the interface I and the input and output parameters of m
- g_m is the firing condition of method m, specifying the circumstances under which m can be activated. Here g_m acts as a guard and this kind of design is called a guarded design.
- P_m is a reactive design, describing the behavior of the execution of m. P_m is of the form

  $H(p \vdash R)$, where

  a. the notation $p(v, x) \vdash R(v, x, y', v', wait')$ is a design [6]

  $$p \vdash R =_{df} (ok \wedge p) \Rightarrow (ok' \wedge R)$$

  where,

  (a) the predicate p, called the pre-condition, is the assumption on the input parameters and fields on which the method can rely when it is activated.

  (b) the predicate R, called the post-condition, relates the initial observations of input parameters and fields to the final observations of output parameters and fields. And it is a commitment made by the method to its client.

  (c) v and v' represent the initial and final values of field variables. x stores the value of the input parameter and y' the value of the output parameter. ok and ok' are logic variables which represent the method has been activated or terminated respectively.

  b. H is a mapping, defined by

  $$H(Q) =_{df} (true \vdash wait') \lhd wait \rhd Q$$

  where,

  $(P \lhd b \rhd Q) =_{df} (b \wedge P) \vee (\neg b \wedge Q)$

  means, ***if*** b ***then*** P ***else*** Q. It is a conditional choice.

  The definition of H indicates that whenever a method is fired in a waiting state it will not remain idle.

(3) *Init* characterizes the initial states of the system

$$Init = true \vdash (init(v') \wedge \neg wait')$$

where v stand for the fields of interface I.

(4) *Prot* is a set of sequences of method call events $<?m_{i1}(x_{i1}), \ldots, ?m_{ik}(x_{ik})>$, standing for the interaction protocol between the contract with its environment, where the event $?m(x)$ represents a call of method m with the input x.

The *consistency* of a contact defines that the guarded designs given by the specification *Spec* and the protocol *Prot* agree with each other. For a consistent interface, if the user of the interface calls methods following the protocol allowed by *Prot*, the guarded designs given by *Spec* will ensure that method invocations can be successfully executed without deadlock. From the guarded designs given by *Spec,* we can define the *large*st (*weakest*) protocol consistent with the specification. For a formal definition of the notion of consistency, we refer the reader to [3]. Therefore, all the discussion given in the rest of this section only refers to the specification.

All the methods defined by an interface are *public*, i.e, they are directly accessible by the environment of the interface. To implement public method, we also need *internal methods* which are invisible to the environment and can only serve the public methods of the contract.

Definition 2.2.2 (General contract) A general contract *GCtr* is a contract extended with a set *PriMDec* to express the syntax information of internal methods and their specifications *PriSpec*.

$$GCtr = <Ctr, PriMDec, PriSpec>$$

where we assume that *Ctr.MDec* and *PriMDec* are disjoint.

In fact, a contract *Ctr* can be seen as a general contract with $PriMDec = \varnothing$

2.3. Component

Components are the implementation of the contracts. The designer has to ensure that of a component satisfies its contract. Its code is used to establish this satisfaction relation by a verifier.

Definition 2.3.1 (Component) A component C is a tuple

$$C = <I, Init, Code, PriMdec, PriCode, InMDec>$$

Where

- I is an interface listing all the provided services of C.
- The tuple $<I, Init, Code, PriMdec, PriCode, InMDec>$ has the same structure as a generalized contract, except that the functions *Code* and *PriCode* map a method m to its code

$$<\alpha_m, g_m, Code_m>$$

- *InMDec* denotes the set of all the required services which appear in the programs *Code(m)* or *PriCode(m)*. Clearly, the behavior of the component depends on that of these required methods which are provided by other components.

Definition 2.3.2 (Component Refinement) A component C_1 is refined by component C_2, denoted by $C_1 \sqsubseteq C_2$, if

$$C_1.MDec \subseteq C_2.MDec$$

$$C_1.InMDec \supseteq C_2.InMDec$$

and the contract refinement $C_1(InCtr) \sqsubseteq C_2(InCtr)$ holds for all input contracts *InCtr*.

2.4. Composition

In general, the most common components composing strategies are *Chaining* and *Disjoint parallel*. *Chaining* is for connecting one component's provided interface to another's required interface, whereas *Disjoint parallel* is for integrating two disjoint components in parallel. When components are composed, some of the interfaces will be hidden and some public methods will become private.

Definition 2.4.1 (Chaining) Let C_1 and C_2 be components satisfying the following conditions

(1) None of the provided or private method of C_2 appears in C_1;

(2) C_1 and C_2 have disjoint field sectors and required service sectors

In this case, the notation $C_1>>C_2$ represents the composite component which has the provided services of C_2 as its provided services, and the input services of C_1 as its input services, and connects the provided services of C_1 to the input services of C_2, and is defined by

$$FDec =_{df} C_1.FDec \cup C_2.FDec$$

$$MDec =_{df} C_1.MDec \cup C_2.MDec$$

$$Init =_{df} C_1.Init \wedge C_2.Init$$

$$Code =_{df} C_1.Code \cup C_2.Code$$

$$PriMDec =_{df} C_1.PriMDec \cup C_2.PriMDec$$

$$PriCode =_{df} C_1.PriCode \cup C_2.PriCode$$

$$InMDec =_{df} (C_2.InMDec \backslash C_1.MDec) \cup C_1.InMDec$$

Theorem 2.2

If $C_1 \sqsubseteq_{comp} C_2$, then

(1) $(C_1>>C_3) \sqsubseteq_{comp} (C_2>>C_3)$

(2) $(C_0>>C_1) \sqsubseteq_{comp} (C_0>>C_2)$

Definition 2.4.2 (Disjoint parallel) Let C_1 and C_2 be components satisfying the following conditions

(1) $FDec_1 \cap FDec_2 = \varnothing$, and

(2) C_1 and C_2 do not share any method

In this case, the notation $C_1 \otimes C_2$ represents the composite component which has the provided services of C_1 and C_2 as its provided services, and the imported services of C_1 and C_2 as its imported services, and is defined by

$$I =_{df} I_1 \uplus I_2$$

$$Init =_{df} Init_1 \cup Init_2$$

$$Code =_{df} Code_1 \cup Code_2$$

$$PriMDec =_{df} PriMDec_1 \cup PriMDec_2$$

$$PriMCode =_{df} PriMCode_1 \cup PriMCode_2$$

$$InMDec =_{df} InMDec_1 \cup InMDec_2$$

Theorem 2.3

(1) If $C_1 \sqsubseteq_{comp} C_2$ then $(C_1 \otimes C) \sqsubseteq_{comp} (C_2 \otimes C)$

(2) $(C_1 \otimes C_2) \equiv_{comp} (C_2 \otimes C_1)$

(3) $C_1 \otimes (C_2 \otimes C_3) \equiv_{comp} (C_1 \otimes C_2) \otimes C_3$

(4) $(C_1 \otimes C_2) \backslash S \equiv_{comp} (C_1 \backslash (S \cap C_1.MDec)) \otimes (C_2 \backslash (S \cap C_2.Dec))$

3. Case study: an e-Store

An e-Store is used to provide customers with an internet based shopping platform. When a customer informs the system he/she wants to take a look, or to buy something, the system should be able to list the products that are in the catalog of the store.

If the customer puts some products to a shopping cart and decides to check out, the system should check if the amount of the products in the store is enough and if the customer has registered as a member of the shop's price club. The system should calculate the rate of payment of the customer according to the shopping record and the state of price club membership. Then the system will check the customer's credit card record. At last, the system should charge the customer and dispatch the products to the customer.

Proceedings of the 11th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'06)
0-7695-2530-X/06 $20.00 © 2006 IEEE

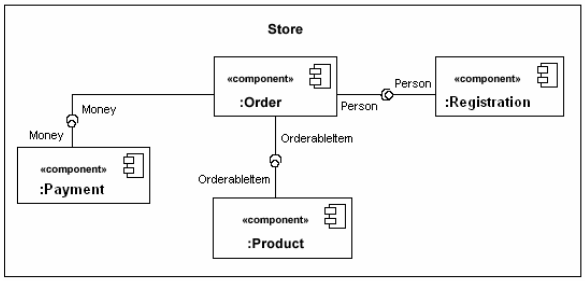

Figure 1 Structure of the e-Store

The services can be grouped according to their properties. In this system the services can be divided into four groups. One group is related to *Payment*, other groups are related to *Member Registration*, *Product* and *Order* information respectively. The group *Order* is responsible for receiving customer's request, creating order list and dispatching products. Therefore the system can be divided into four modules, namely *Payment*, *Registration*, *Product* and *Order*, which are usually treated as components in industry. The services are assigned to their interfaces.

(1) *Order*: is responsible for interacting with the customers. Services *OrderRequest*( ) and *OrderConfirm*( ) are assigned to its interface. Where *OrderRequest*( ) is responsible for getting user's request. The service *OrderConfirm* ( ) is responsible for making sure if the products in the order list are just what the customer want. The interface then delegates the two tasks to the body of component *Order* for implementation [7]. Interface I_{order} is used to capture the syntax information of the services and the contract *CO* the semantics. Component *Order* can be obtained by refining the contract *CO* [8]. To fulfill the tasks, Order also requires services from other three components to get the information about customer's deposit, registration state and product, as shown in Figure 2. ←

(2) *Registration*: is responsible for providing services related to registration. Services *MemberRegist*( ) and *MemberConfirm*( ) are assigned to its interface, as shown in Figure 3. In the same way as *Order*, the interface delegates the task to the component *Registration* for implementation. Interface $I_{registration}$ is used to capture the syntax information and the contract *RC* capture the semantics information.

(3) *Product*: responsible for recording all the information about products, such as the name, usage, amount, producer etc.. Services *LocateProduct*( ) and *TakeItem*( ) and *AddItem*( ) are assigned to it.

(4) *Payment*: keeps track of financial state of the customer. It can also be integrated with the legacy financial systems or Bank. Services *getRate*( ), *CheckDeposit*( ) and *ChangeBase*( ) are assigned to it.

These components can be constructed independently and composed together. The process of composition can be divided into three steps.

- In the first step, the interfaces are checked. We find that *Registration* just provides the services that are required by *Order*, so do Product and Payment.
- In the second step we verify the contracts of the components which are matched in interface-level, e.g. *OC* and *RC*.
- The third step is composing the components which are matched in contract-level.

There are two types of composition, chaining and disjoint parallel. As *Order* and *Registration* satisfy the conditions of chaining, we chain the two components together. In the same manner, the other components can also be composed together and the system is constructed, as shown in Figure 1.

3.1. Interface

Each component interacts with others through interfaces, providing service or requiring service. For example, component Order provides service by interfaces *Sale* and requires service by interface *Money*, *OrderableItem* and *Person*.

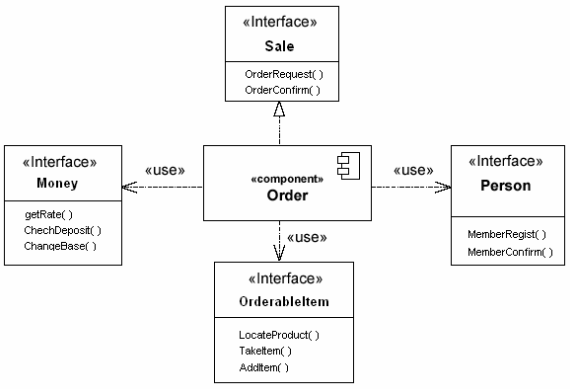

Figure 2 Interface of Order

With these interfaces the component can get services from the other three. The provided services are implemented in component *Order*, however the required services are implemented in other three components. The specification of the interface *Sale* can be described as a tuple < *MDec* >, listed as follows.

Proceedings of the 11th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'06)
0-7695-2530-X/06 $20.00 © 2006 IEEE

Interface of Order:

$\boldsymbol{I_{MOrder}}$

- ***MDec***

*OrderRequest(**in**: Bool or)* //When a customer start //shopping, *or* becomes *true*

*OrderConfirm(**in**: Cname cn, socialID sID, memberID mId, **out**: OrderID oId, Bool OrderState)* //provide to user to //confirm the products to buy

We can add the fields needed by the methods to the interface to form a general interface as follows,

General Interface of Order: *Sale*

$\boldsymbol{I_{Order}}$

- $\boldsymbol{I_{MOrder}}$
- ***Fdec***

Bool or; //indicate if shopping is started.

Bool OrderState; // indicate if an order list is // confirmed.

$\mathbb{P}$*Cname C;* $\mathbb{P}$*socialID S;* $\mathbb{P}$*memberID M;* // set of customer names, //social IDs and member IDs

$\mathbb{P}$*OrderID OID* // a set of order IDs

Here *or* represents if a customer starts to order any product; *OrderState* represents if the order process complete. *C*, *S* and *M* represent a set of customer names and a set of social IDs and a set of member IDs respectively.

3.2. Contract

A *contract OC* for interface *Sale* of *Order* assigns a specification to each method and can be written as follows, where $Spec_{Order}$ is given as the specification of each operation. The contract associates an interface to an abstract data model plus a set of functional specifications of its services, and as well as an interaction protocol. It is defined as a quadruple $<I_{order}, Init, Spec_{order}, Prot_{order}>$.

Contract of Order: <<Contract>> OC

$\boldsymbol{I_{Order}}$ // imported from the interface of Order

Init $\mathbb{P}$*OrderID OID = OID; OrderID oId = 0;*
Bool OrderState = False // order is not created

$\boldsymbol{Spec_{Order}}$

*OrderRequest(**in**: Bool or){*
(Order.OrderProduct) ◁ or ▷ (true ⊢ wait')
} //semantics of the service *OrderRequest,* where *or* is the condition of the action of order a product, if *or* is true then order a product else keep waiting.

*OrderConfirm(**in**: Cname cn, socialID sID, memberID mId, **out**:OrderID oId,Bool OrderState){*
(New(order); oId := order.id;
OID := OID ∪ oId; OrderState:=True)
◁ (cn∈C∧sId∈S∧mId∈M∧mId=locmid(sId)) ▷
(OrderState:=False)}
// semantics of the service *OrderConfirm*

$\boldsymbol{Prot_{Order}}$ *OrderRequest(**in**: Bool or);*
*OrderConfirm(**in**: Cname cn, socialID sID, memberID mId, **out**:OrderID oId,Bool OrderState)*
// the sequence of methods call

The methods defined in contract *OC* are public, which are directly accessible by other components. There are also some other existing methods that are invisible to others but can serve the public methods of the contract. Here we assume method *locmid*(*sID*) can associate a customer's member ID and his/her social ID, thus we can get a customer's member ID from the social ID. The general contract *GOC* extends *OC* by adding two private methods *OrderProduct*( ) and *DispatchProduct*( ).

General Contract of Order: <<Contract>> *GOC*

OC // imported from the Contract of Order

PriMDec

*OrderProduct(**in**:Cname cn, socialID sID, memberID mId, Pname pId, **out:** Bool orderState, OrderList olist)*

*DispatchProduct(**in**: Pname pId, Cname cn, Caddress, addr)*

PriMSpec

*OrderProduct(**in**: Cname cn, socialID sID, memberID mId, Pname pId, **out:** Bool orderState, OrderList olist){*
*Person.MemberConfirm(**in**: Cname cn, socialID sID, memberID mId, Password pw, **out**: Bool registState) ◁ mid≠0 ▷*
*Person. MemberRegist(**in**: Cname cn, socialID sID, **out**: Bool registState, memberID mId, Password pw) ;*
(OrderableItem.LocateProduct( );
OrderableItem.TakeItem( ))◁ registState ▷ skip;
*(Money.getRate(**in**: Cname cn, socialID sID, **out**: point);*
*Money.CheckDeposit(**in**: Cname cn, socialID sID, **out**: deposit);*

display(in: OrderList oList)) ◁ *price > 0* ▷ *skip*

//tagged as *price* > 0 when the product has stock

*DispatchProduct(**in**: OrderID oId, Bool OrderState, Cname cn, Caddress addr **out**:OrderID oId, OrderAddr oadd, Bool DispatchState){*
(oaddr:= addr; DispatchState:=True)
◁ **(***cn* ∈ *C* ∧ *sId* ∈ *S* ∧ *OrderState=True*)▷
(DispatchState:=False)}
//Send products to buyer.

To fulfill the tasks, component *Order* requires services from components *Registration* and *Product* and *Payment* via interface *Person* and *OrderableItem* and *Money*. Service *MemberConfirm*( ) and *MemberRegist*( ) are implemented in component Registration, as shown in Figure 2. *registState* represents the state of registration and as an output parameter, it can be obtained by calling method *MemberRegist*( ) or *MemberConfirm*( ). *DispatchState* represent the state of product dispatch and can be obtain by calling *DispatchProduct*( ). *LocateProduct*( ), *TakeItem*( ), as well as the other methods listed in the interfaces of *Person*, *Money* and *OrderableItem* in Figure 2, are defined and implemented in related components.

The contract of *Registration*,, *Product* and *Payment* can be derived the same way.

3.3. Component

Now we define a component *Order* in the *e-Store* system to provide the services to customers. The external behavior of a component is specified by the contracts of its interfaces. A design of a component has to reorganize the contract and implement it. Component *Order* is defined as a tuple <$I_{provide}$, *Init, Code,* $I_{required}$, $I_{private}$, *PriMCode* >, where

- $I_{provide}$ is the service which provided by component *Order* to the customer,
- *Init* is the initial state of the component,
- *Code* implements the specification of *Order* $Spec_{Order}$ described in its contract,
- $I_{required}$ is the required service which is required by *Order* but implemented by other component,
- $I_{private}$ is a private method of component *Order,* and
- *PriMCode* is the implementation of specification of $I_{private}$.

<<**Component**>> *Order*

I**provide** // the same as ***I*****Order** listed above in // the interface of *Order*

Code
OrderRequest(in: Bool or){
do *OrderRequest(**in**: Bool or)*//when a customer //starts to order, '*or*' becomes '*true*'
while (¬*or*); *OrderProduct*( )} // keep waiting //until *or* becomes true, start the //order process
*OrderConfirm(**in**:Cname cn, socialID sID, memberID mId, **out**:OrderID oId,Bool OrderState){*
if *cn*∈*C*∧ *sId* ∈ *S* ∧ *mId* ∈ *M* ∧ *mId* = *locmid*(*sId*)
then *New*(*order*); *oId* := *order.id*; *OID* := *OID* ∪ *oId*; *OrderState* := *True*
else *OrderState:= False}*

I**required** // provided by other component
Person, OrderableItem, Money
//*Order* get services from the three interfaces

I**private** // syntax information of the private methods
*OrderProduct(**in**: Cname cn, socialID sID, memberID mId, Pname pId, **out**: Bool orderState, OrderList olist)*
*DispatchProduct(**in**: Pname pId, Cname cn)*

PriMCode //semantics of the private methods
*OrderProduct(**in**: Cname cn, socialID sID, memberID mId, Pname pId, **out**: Bool orderState, OrderList olist){*
if *mid≠0* ***then*** *MemberConfirm*( ) ***else*** *Person.MemberRegist*( );
if *registState = True* ***then*** *{OrderableItem.LocateProduct*( );
OrderableItem.TakeItem( )};
if *price > 0* ***then*** {*Money.getRate*( );
Money.CheckDeposit( ); *display*( )}
*DispatchProduct(**in**: OrderID oId, Bool OrderState, socialID sID, Cname cn, Caddress addr, **out**:OrderID oId, OrderAddr oadd, Bool DispatchState){*
if *cn* ∈ *C* ∧ *sId* ∈ *S* ∧ *OrderState = True*
then *oaddr* := *addr*; *DispatchState* := *True*
else *DispatchState*: = *False* }

In ***PriMCode***, method *MemberConfirm*( ) is used to represent *MemberConfirm*(***in***: *Cname cn, socialID sID, memberID mId, Password pw,* ***out***: *Bool registState*) for short, so as to *MemberRegist*( ), *LocateProduct*( ) and the others, in order to keep the specification more readable. The methods with full arguments are listed in the provided interfaces of the

components that define and implement the methods, or the required interfaces of the partners that require the methods to fulfill their own tasks.

The services provided to *Order* by three interfaces *Person*, *OrderableItem* and *Money* are listed as the following, respectively,

- ***Person***

MemberConfirm(***in***: *Cname cn, socialID sID, memberID mId, Password pw,* ***out***: *Bool registState*)

MemberRegist(***in***: *Cname cn, socialID sID,* ***out***: *Bool registState, memberID mId, Password pw*)

- ***OrderableItem***

LocateProduct(***in***: *Product p, productID pID,* ***out***: *Bool OrderableState*)

TakeItem(***in***: *Product p, productID pID, Float reqamount,* ***out***: *price*)

- ***Money***

getRate(***in***: *Cname cn, socialID sID,* ***out***: *point*)

CheckDeposit(***in***: *Cname cn, socialID sID,* ***out***: *deposit*)

ChangeBase(***in***: *Cname cn, socialID sID, addpay*)

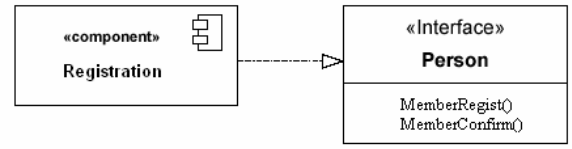

Figure 3 Interface of Registration

Component *Registration* provides service *MemberRegist*( ) and *MemberConfirm*( ) through interface *Person*, as shown in Figure 3.

The services provide by Payment via interface Money are getRate( ), CheckDeposit( ) and ChangeBase( ) is shown in Figure 4.

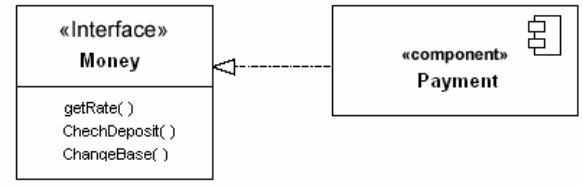

Figure 4 Interface of Payment

The services *LocateProduct*( ), *TakeItem*( ) and *AddItem*( ) are provided by Product via interface *OrderableItem*, as shown in Figure 5.

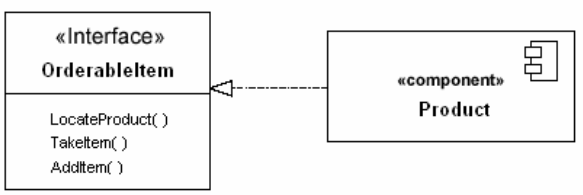

Figure 5 Interface of Product

Components *Product*, *Registration* and *Payment* can also be constructed in the same way as *Order*.

3.4. System

The system can be constructed by composing the four components together. In composing, we should check if the components to be composed meet the composing conditions described in section 2.4. If they meet the conditions, then the components can be composed together.

Now, we check component *Order* and *Registration*. Here we found that

- none of the provided or private methods of *Registration* appears in *Order*,
- *Order* and *Registration* have disjoint field sectors and required service sectors.

So we can compose components *Order* with *Registration* in the manner of *chaining*.

Order-R ::= *Order* >> *Registration*

After chaining, interface *Person* is hidden and removed from the new formed component. Thereby *Order-R* only has interfaces *Money, OrderableItem* and *Sale.*

In the same way, *Order-R* can continue compose with *Product* and *Payment* to form the system.

Store:: = *Order* >> *Registration* >>*Product*>> *Payment*

Code can be written according to the model of component and the executable system is obtained.

The components can be designed and implemented independently. The composition is dependent on the services, or the syntax and semantics of the service. For example, component *Order* can not only connect with *Registration* but also other component with that property. In addition, component *Payment* can also be shared with the finance management system of a bank.

4. Conclusion and discussion

4.1. Conclusion

This paper identifies service-oriented design as the process of designing an application service. A systematic and generic service-oriented design approach is presented based on reactive component theories. The case study has shown how to use the strategy to carry out service-oriented design, which is characterized by considering the external and internal perspective of an application. The behavior of an individual service is modeled by a guarded design. Contract is employed to describe the semantics of the services. When the contract is precise and explicit, there is no need for redundant checks [5]. In addition, various levels of abstract model for service-oriented design are defined, providing abstract and generic concepts that support the modeling of application services and their designs.

A Java-like formal language rCOS [3] is used to describe the syntax and semantics of services, and the service that a system provides can be assigned to the interfaces of a component. In our model, the contract for the provided interface of a component allows the users to check whether the component provides the services required by other components in the system, without the need to know the design details of the component. Designer of the component may decide to use services provided by other components, which services are called required services [4]. Components that provide the required services can be built independently. In assembling a system, we can compare the syntax of the provided service and the required services first, then compare the semantics of those that matched syntactically. To avoid deadlock, the assembler can check the consistency between the provided protocol of the service providing component and the required protocol of the components requiring the services. This consistency is simply set inclusion [3]. After merging the matched components, a system is created.

A system can be a composition of a family of components. The contracts of these components provide composition or decomposition of the application domain both in terms of services. Moreover, the decomposition is first done *horizontally*, i.e. decomposing according to the whole system's services. And then each group of services are decomposed *vertically* continue to assign to its subcomponents. In fact, *vertical decomposition* is more relevant to component-based development, as *horizontal decomposition* is more or less about design of independent components. Based on this discussion, we propose a service based development strategy so that components and their interfaces can be used to support service oriented design.

4.2. Discussion

Service-oriented design originated from the area of component based design (for an overview see e.g. [9]). It elaborates on this area by incorporating the principle of distinguishing between external observable behavior and internal realization of that behavior and the principle of integrating applications with business processes. Although these principles are not new, they have special status in service-oriented design methods. Our case study reflects them.

In this paper, we take some informal views of [4, 10], in which component provides services to and requires services from other components. However our models are built on service and there is a standard calculus for reasoning about and refining at different levels of abstraction. It will facilitate model checking and verification that are included in our future work.

Web service description language WSDL [14] and Business process execution language BPEL [12] are now commonly used in web service description and service composition. However the two languages are based on XML structure and can't give a precise model. Our model improved model precision.

A notion of contract can also be found in [11] that emerged in the model of action systems. It promotes the separation between the specification of what an agent can do in a system and how they need to be coordinated to achieve the required global behavior of the system. However, the architectural components are not explored.

A contract describes the coordination among a number of partners (i.e. components or objects)[13]. Its main purpose is to support system architectural evolution and to deal with changes in business rules of the system application. Our contracts here specify the services of components while we treat interaction and coordination as part of the implementation of the components.

Our goal is to support construction of software components and component-based software systems. However, it is interesting to investigate how these two notions of contracts can be combined to provide better support to both system construction and evolution.

Our future work includes combining with UML so that user can benefit the graphic environment. UML is also very suitable for model driven development. We will also explore pattern based refinement rules to facilitate the transition from model to executable.

5. References

Proceedings of the 11th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'06)
0-7695-2530-X/06 $20.00 © 2006 IEEE

[1] D.Quartel, R.Dijkman, M.Sinderen, Methodological Support for Service-oriented Design with ISDL, ICSOC'04, November 15–19, 2004, New York, USA. 2004 ACM

[2] White paper, Microsoft Corp. 2004, http://download.microsoft.com/download/d/2/5/d2513e64-0dcd-4ef6-89c4-c99ee117936f

[3] J.He, Z.Liu and X.Li, Component-Based Software Engineering: The Need to Link Methods and Their Theories, ICTAC 2005, LNCS 3722, Springer Berlin / Heidelberg, 2005

[4] C. Szyperski. Component Software: Beyond Object-Oriented Programming. Addison-Wesley, 2002.

[5] Bertrand Meyer: Applying Design by Contract, in Computer (IEEE), vol. 25, no. 10, October 1992

[6] C.A.R.Hoare and J.He, Unifying Theories of Programming, Prentice-Hall, 1998

[7] Z.Liu, J.He, X.Li and J.Liu, A Rigorious Approch to UML-Based Development, Proc. Brazilian Symposium on Formal Methods (Invited Talk), Nov. 29 – Dec. 01 2004, Recife, Brazil, Electr. Notes Theor. Comput. Sci. 101: 95-127.

[8] Z.Liu, J.He and X.Li, rCOS: refinement of component and object systems, Proc. 3rd International Symposium on Formal Methods for Components and Objects, LNCS 3657, pp. 183-221, Springer, 2005.

[9] J. Chessman and J. Daniels. UML Components: A Simple Process for Specifying Component-based Software. Addison- Wesley, 2001.

[10] G.T. Heineman and W.T. Councill. Component-Based Software Engineering, Putting the Pieces Together. Addison-Wesley, 2001.

[11] R.J.R. Back, L. Petre, and I.P. Paltor. Formalizing UML use cases in the refinement calculus. Technical Report 279, Turku Centre for Computer Science, Turku, Finland, May 1999.

[12] BEA Systems, Microsoft, IBM, and SAP. Business process execution language for web services (BPEL4WS) version 1.1, May 2003
http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/.

[13] Z.Liu, J.He and X.Li, Contract-Oriented Development of Component Software, Proc. 3rd IFIP International Conference on Theoretical Computer Science, Kluwer. 2004.

[14] W3C. Web services description language (WSDL): Part 1: Core language version 1.2. WDwsdl20-20031110, 2003, http://www.w3.org/TR/2003/

Proceedings of the 11th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'06)
0-7695-2530-X/06 $20.00 © 2006 IEEE